# Towards mining norms in open source software repositories

Bastin Tony Roy Savarimuthu[1], and Hoa Khanh Dam[2]

[1] University of Otago, Dunedin, P O Box 56, Dunedin, New Zealand
[2] School of Computer Science and Software Engineering, University of Wollongong
New South Wales 2522, Australia
tonyr@infoscience.otago.ac.nz, hoa@uow.edu.au

**Abstract.** Extracting norms from computer-mediated human interactions is gaining popularity since huge volume of data is available from which norms can be extracted. Open source communities offer exciting new application opportunities for extracting norms since such communities involve developers from different geographical regions, background and cultures. Investigating the types of norms that exist in open source projects and their efficacy (i.e. *the usage of norms*) in enabling smoother functioning however has not received much attention from the normative multi-agent systems (NorMAS) community. This paper makes two contributions in this regard. First, it presents norm compliance results from a case study involving three open source Java projects. Second, it presents an architecture for *mining* norms from open source projects. It also discusses the opportunities presented by the domain of software repositories for the study of norms. In particular, it points towards how norms can be mined by leveraging and extending prior work in the areas of Normative Multi-Agent Systems (NorMAS) and mining software repositories[3].

## 1 Introduction

A good example of a large, real-world multi-agent organization involving a huge number of (human) agents is open source software development (OSSD). In fact, large-scale open source systems such as Linux and Android OS, used and tested by millions are developed by hundreds of contributors over extended period of time. Many large open source projects are highly successful although they did not *initially* have formal organizational structure with regulations (i.e. norms and policies) explicitly stated and enforced (as in traditional commercial software projects). Therefore, it is important and interesting to explore the role norms play in the success (or failure) of a particular open source project, and how these norms have emerged and are enforced in such large organizations of volunteering contributors who have different experience and are from different background, cultures and geographical regions.

Mining norms in open source projects is facilitated by the rich, extensive and readily available data from the software repositories of those projects. Software repositories

[3] An early (short) draft of this paper appears as a working paper (an informal publication) at Otago (http://otago.ourarchive.ac.nz/handle/10523/2101) and was orally presented at the 2012 Dagstuhl seminar on NorMAS.

can be in various forms such as historical repositories (e.g. SVN or CVS repositories, archived communication records, bug repositories), code repositories (e.g. Sourceforge.net and Google code), and run-time repositories (e.g. deployment and/or execution logs). Since these repositories contain information about human actions and their interactions with one another, these repositories contain explicit or implicit information on *norms* relevant to the communities involved in the process of software development. These repositories can be *mined* to uncover useful and important patterns and information about the normative processes associated with the development of software systems. For instance, we can directly observe developer discussions, identify their contents (e.g. patches, bugs, reviews) on mailing lists or forums. We can build social networks, and cross-check associated discussion and programming activity. In addition, we can leverage existing mining software repositories (MSR) technologies [7] such as data pre-processing, cross-linking data from different sources for mining norms.

Although OSSD offers a real, large-scale platform for norm mining, to the best of our knowledge, there has been no existing work in the study of norms in open source software projects. Thus, this line of work is a promising candidate for NorMAS researchers to investigate the processes associated with real norms such as norm formation, spreading and enforcement. Insights gained from this study can be used to inform both research and practice of norms. Moreover, this research can leverage existing techniques developed by NorMAS researchers, data mining researchers and other computer science disciplines such as information retrieval and natural language processing.

The work in this paper specifically contributes to call for identifying challenges and future research directions on the synergy between agents and data mining. Towards that goal, this paper explores how norms govern the smoother functioning of large, distributed OSSD projects (which are real-life, large-scale multi-agent organizations). Towards the understanding of normative processes in OSSD, this paper makes two contributions. First, it discusses a case study involving three open source projects to investigate norm compliance. Second, it presents an architecture for mining norms in open source software repositories and identifies research challenges that can be addressed using the proposed architecture.

## 2 Background

Norms are expectations of behaviour in an agent society. Researchers in normative multi-agent systems (NorMAS) study how norms can play a role in the design and the operationalization of socio-technical systems. Research in normative multi-agent systems can be categorized into two branches. The first branch focuses on normative system architectures, norm representations, norm adherence and the associated punitive or incentive measures. The second branch is concerned with the emergence of norms. For an overview of the study of norms in agent systems we refer the reader to [5, 13]. In this section, we provide a brief overview of the relatively new domain of mining software repositories, and particularly how norms can be mined from huge volumes of data.

### 2.1 Mining Software Repositories

Mining Software Repositories (MSR) [7] is an emerging research area that focuses on mining the rich and large quantities of data available in software repositories to uncover useful and important patterns and information about software systems and projects. Such information assists developers, managers, testers, etc. working on those systems by offering insights into the nature of open source software development (OSSD) through the development techniques and tools.

Efforts in MSR[4] research have been mainly on providing techniques to extract and cross-link important information from different software repositories. Such information can be used in various activities during the development of software systems. For instance, a range of work (e.g. [18]) have proposed to make use of historical co-changes (e.g. entities or artefacts that were changed together) in a software project to propagate changes across the software system during maintenance and evolution. A large number of existing MSR work (e.g. [12]) also mine bug reports and historical changes to predict the occurrence of bugs, e.g. parts of the code that likely to contain errors. Such predictions are useful for managers in allocating and prioritizing testing resources. Information mined from reported bugs and execution logs can also be used to improve the user experience, e.g. warning the user when they are about to perform a buggy action, and suggesting when an existing piece of code can be re-used. Empirical software engineering also substantially benefits from MSR since many empirical studies can be done (and repeated) on a large number of subjects, i.e. OSS repositories enable the verification of generality of prior findings (e.g. the study in [8] confirms that cloning seems to be a reasonable or even beneficial design option in some situations). Recent MSR work (e.g. [1]) has also attempted to mine discussions from archived mailing lists, forums, and instant messaging to understand the dynamics of large software development teams, including how and when team members get invited, detecting team morale at a particular point in time, and understanding the process of bug triage.

### 2.2 Mining norms from large repositories

Based on a large Twitter dataset of 1.7 billion tweets [10], researchers have investigated how two out of seven independently proposed re-tweeting conventions became widely adopted. Their main finding was that social conventions are more likely to arise in the active and densely connected part of the community. However, the study was unable to ascertain why some conventions were widely adopted and why some were not.

Another research [3] has investigated the naming conventions used for Java classes to check whether the names correspond to the actual recommendation provided by the Java naming convention (i.e. they should be noun phrases) and have proposed whether the class names have to be changed to adhere to this recommendation or whether the class itself has to be refactored. The work of Boogerd and Moonan [2] notes that following coding conventions can increase the chance of faults occurring since any modification of the code to adhere to a convention has a non trivial probability of introducing

[4] A extensive review of the work in MSR can be found from the "Bibliography on Mining Software Engineering Data" available at http://ase.csc.ncsu.edu/dmse

errors. This finding is interesting, however, the result is based on investigating data from one project only. To our knowledge, there has been no prior work on mining different categories of conventions in open source projects. Also, none of the prior works have proposed an architecture for extracting norms from open source repositories. The rest of this paper contributes towards addressing these issues.

## 3 Classifications of norms in open source software repositories

We note that several types of norms might exist in open source software development communities. We briefly discuss the distinction between norms and conventions using some examples in the context of OSSD. Also, we provide a brief overview of the norm life-cycle that can be observed in OSS projects.

**Conventions** - Conventions of a community are the behavioural regularities that can be observed. Coding standards of a project community is an example of a convention. The specifications of these conventions may be explicitly available from the project websites[5] or can be inferred implicitly (e.g. a wide spread convention that may not be explicitly specified in project websites).

**Norms** - Norms are conventions that are enforced. A community is said to have a particular norm, if a behaviour is expected of the individual members of the community and there are approvals and disapprovals for norm abidance and violation respectively. There have been several categorizations of norms proposed by researchers (cf. [13]). We believe that deontic norms - the norms describing prohibitions, obligations and permissions studied by the NorMAS community [17] is an appropriate categorization for investigating different types of norms that may be present in OSSD communities.

*Prohibition norms* prohibit members of a project group from performing certain actions. However, when those actions are performed, the members may be subjected to sanctions. For example, the members of an open source project may be prohibited to check-in code that does not compile, and they may be prohibited to check-in a revised file without providing a comment describing the change that has been made. *Obligation norms* on the other hand describe activities that are expected to be performed by the members of a project community. When the members of a community fail to perform those, they may be subjected to sanctions. For example, the members may be expected to follow the coding convention that has been agreed upon. Failure to adhere to this convention may result in the code not being accepted by the repository (e.g. based on automatic checking) or a ticket may be issued by a quality assurance personnel. *Permission norms* describe the permissions provided to the members (e.g. actions they can perform). For example, an user playing the role of the project manager is permitted to create code branches.

[5] Refer to http://source.android.com/source/code-style.html for the coding guidelines for Android development.

In NorMAS, researchers have proposed a life-cycle for norms (e.g. [13]). The norm life-cycle in the context of open source development consists of four phase, convention creation, codification, monitoring and enforcement. In open source repositories, an initial phase is the convention formation (or creation) phase where the members of the community discuss what the conventions of the software project should be. Once the coventions have been agreed upon, they might be codified into written rules. There are several examples of codified conventions in several open source communities. For example, the open source Apache project[6] and the Android development community[7] provide guidelines on conventions including coding conventions. Once the convention has been codified, that forms the basis of norms through the creation of *normative expectations*. It is expected that the members of the project community adhere to these norms. Upon the codificiation of these norms, projects choose to monitor norms either through centralized or distributed mechanisms. Some projects have integrated convention checking tools such as CheckStyle[8] and StyleCop[9] in their project submission systems and any violations of norms are by default prohibited. Another option is for projects to facilitate a distributed monitioring mechanism which is primarily manual where individual contributors report any violations. The fourth stage is the enforcement stage. While using one of the convention checking tools, the enforcement is instantaneous. However, in a distributed approach to sanctioning, there could be several types of penalties. For example, a ticket could be issued for breaking a norm. There could be email exchanges between individuals discussing the importance of honouring conventions. Also, there might be invisible penalties for the violator such as the decrease of reputation and trust. Based on the discussions generated on a particular norm, there may be re-evaluations leading to the adjustment of norms (change of norms). Thus, the process enables a feedback loop to the norm formation phase. Norms can also be formed through an emergent approach. Once the project is well underway, there could be a new convention[10] that advocates all version changes should be accompanied with a non-trivial explanation or comment on the change that was made. Thus, the emergent behaviour can be encoded as a convention (as a part of the norm formation stage).

## 4 A case study on norm mining

We conducted a case study to examine certain categories of conventions to study whether they are adhered in large software repositories (phase 3, the monitoring phase of the norm life-cycle). In order to conduct this study, we chose three representative open source projects based on Java which follow the Java coding conventions. The first two projects were Apache Ant[11] and Apache Struts[12] which explicitly advocate their par-

[6] http://portals.apache.org/development/code-standards.html
[7] https://sites.google.com/a/android.com/opensource/submit-patches/code-style-guide
[8] http://checkstyle.sourceforge.net/
[9] http://archive.msdn.microsoft.com/sourceanalysis
[10] The new convention could emerge based on discussions.
[11] ant.apache.org, version 1.8.4
[12] struts.apache.org, version 2.3.4

ticipants to follow Java coding conventions[13]. The third project, Apache ODE[14] did not explicitly state it follows Java coding convention. We included this for comparison purposes to see if the honouring of conventions in this project are different from the ones that had explicit statements about honouring conventions. Basic details of these projects can be found in Figure 4.

For these three projects, we identified the five categories of conventions given below (both explicit and non-explicit) and checked whether these conformed to the conventions. These five categories were chosen to broadly represent different aspects of software development (e.g. extensibility, redundancy, and smaller footprint). We used CheckStyle 5.5, a coding standard analyzer for Java to check whether conventions are honoured.

1. *Extensibility* refers to a set of criteria that tests whether the code that has been developed is amenable to easier modification in the future.
2. *Programming pitfall* refers to a set of criteria that tests whether the developed code has avoided some common programming pitfalls that are experienced by developers. These pitfalls affect the run-time behaviour of the system.
3. *Import* refers to a set of criteria on the proper use of file imports that are included in a Java file. For example, a Java file should not have import statements with * because importing all classes from a package may lead to tight coupling between packages which might lead to problems when a new version of a library introduces name clashes.
4. *Length* refers to a set of criteria where length related attributes are measured and compared with some default values. For example, when the length of a file or a method is over certain limit, it impacts readability and maintainability of code.
5. *Redundancy* refers to redundant use of code. For example, there might be redundant exceptions that are caught or redundant import statements that need to be removed. Redundancy results in code bloat thus increasing both storage and transmission costs.

The table given in Figure 1 shows convention categories in column one and convention names as given by CheckStyle 5.5 in column two and the description of the conventions considered for this case study (or default values corresponding to certain conventions in italics) in column three.

The table shown in Figure 2 presents the results that were obtained by running the CheckStyle checker based on the standard checks template[15] on these three projects. We note again that some of these are conventions that were originally specified in the coding conventions (e.g. the length of a line in Java should be less than 80), while others have informally emerged over time as good practices, but have not been updated in the Java convention specification (e.g. avoid star imports). We have specified the codified conventions using * in column two of Figure 1.

In terms of extensibility, all the three projects, seem to have substantial issues. In line conditionals are predominantly used which obscure the readability of the code and

[13] Phases 1 and 2 of the norm life-cycle are complete at this stage.

[14] ode.apache.org, version 1.3.5

[15] http://checkstyle.sourceforge.net/availablechecks.html

| Convention category | Conventions (As encoded in Checkstyle 5.5) | Description/Default Value (in italics) |
|---|---|---|
| Extensibility | AvoidInlineConditionalsCheck | In line conditionals are hard to read (e.g. the use of ternary operator in Java), which limits the ability to understand the code. |
| | DesignForExtensionCheck | Checks whether classes are designed for inheritance. |
| | SimplifyBooleanExpressionCheck* | Checks for overly complicated boolean expressions. |
| Programming pitfalls | HiddenFieldCheck | Checks that a local variable or a parameter does not shadow a field that is defined in the same class. |
| | EqualsHashCodeCheck | Checks that classes that override equals() also override hashCode(). |
| Import | AvoidStarImportCheck | Avoid generic imports with *. |
| | IllegalImportCheck | Checks for imports from a set of illegal packages. |
| | RedundantImportCheck | Checks for imports those are redundant. |
| | UnusedImportsCheck | Checks for unused import statements. |
| Length | FileLengthCheck* | *2000 lines* |
| | LineLengthCheck* | *80 characters* |
| | MethodLengthCheck | *150 lines* |
| | MethodLimitCheck | *30 methods* |
| | ParameterNumberCheck | *7 parameters* |
| Redundancy | RedundantThrowsCheck | Checks for redundant exceptions declared in throws clause such as duplicates, unchecked exceptions or subclasses of another declared exception. |

**Fig. 1.** Convention categories and description

it was observed that many classes were not designed for extension. There were not many issues with regards to simplifying the complex boolean expression.

There were not a huge number of issues on imports (as a percentage of errors). However, what was interesting was that most of the import errors in the ODE project were on the files in the testing package (implying testers were not following conventions). However, this was not observed in the other two projects.

There were substantial number of errors across all the projects on exceeding the line length (80 characters). The number of classes exceeding the method limits check (30 methods) were high in Ant project, but there were not any in the other two projects. There were more than 10 instances in all the three projects that had more than 7 methods in a class. The method limit check, method length check and the parameter number check are indicators of complexity of the system. Higher values for these indicate that there may be a need for refactoring (e.g reducing the number of methods in the classes of the Ant project).

It is interesting to note that there are many instances in all the three projects where a global field was hidden by a local field (hiddenFieldCheck). This is an issue because this may result in erroneous run-time behaviour of the system. In both ODE and Struts projects there were a few instances where *equals* method was overridden while the *hashCode* method was not overridden. Ant did not have any *equalsHashCodeCheck* violation. All the three projects had substantial number of redundant throws classes which causes code bloating.

Overall, our observation is that all the three projects do not adhere to a number of conventions. However, there were not any substantial differences between the projects

| Row Labels | Sum of ODE errors | Sum of Ant errors | Sum of Struts errors |
|---|---|---|---|
| **Extensibility** | | | |
| AvoidInlineConditionalsCheck | 272 | 755 | 545 |
| DesignForExtensionCheck | 4313 | 9124 | 10199 |
| SimplifyBooleanExpressionCheck | 2 | 9 | 9 |
| **Import** | | | |
| AvoidStarImportCheck | 71 | 14 | 157 |
| IllegalImportCheck | 0 | 1 | 0 |
| RedundantImportCheck | 17 | 11 | 8 |
| UnusedImportsCheck | 0 | 14 | 0 |
| **Length** | | | |
| FileLengthCheck | 6 | 7 | 9 |
| LineLengthCheck | 9853 | 3837 | 15378 |
| MethodLengthCheck | 13 | 18 | 19 |
| MethodLimitCheck | 0 | 1526 | 0 |
| ParameterNumberCheck | 10 | 14 | 28 |
| **Redundant** | | | |
| RedundantThrowsCheck | 959 | 915 | 1425 |
| **Programming pitfalls** | | | |
| HiddenFieldCheck | 359 | 1845 | 2977 |
| EqualsHashCodeCheck | 3 | 0 | 7 |
| **Grand Total** | **15878** | **18090** | **30761** |

**Fig. 2.** Comparing conventions across projects

that explicitly following conventions and not-following conventions in our case study[16]. This of course needs to be evaluated in a larger context involving a large number of projects. Additionally, analyzing projects on these five types of conventions reveal some information that warrant further investigation. For example, it could be the case that the same (small) group of programmers may have introduced all the import errors or the programming pitfall errors. Such analysis can be undertaken in the future. Our objective in this work was to undertake initial investigation of convention adherence in some well-known projects.

## 5 Agent-based architecture for norm mining

The preliminary case study that we have conducted on convention monitoring is a first step towards the grand challenge of recognizing normative processes in the software repository setting. In this section, we present our second step, where we propose an agent-based architecture for identifying existing and emergent norms and the level of support (i.e. conformance) for these norms in OSS repositories.

Agent mining is an emerging interdisciplinary area that integrates several disciplines such as multi-agent systems, data mining and knowledge discovery and machine learning [4]. Agent-based mining architectures have been proposed by researchers for massively parallel data processing and integration [9] and also mining banking domain specific information [11]. Here, we propose an agent-based architecture for norm min-

[16] There could be reasons such as internalized (non-explicit) norms that could have operated in the ODE project which are similar to the explicit norms.

ing from OSS repositories. To our knowledge no other work has considered a similar approach.

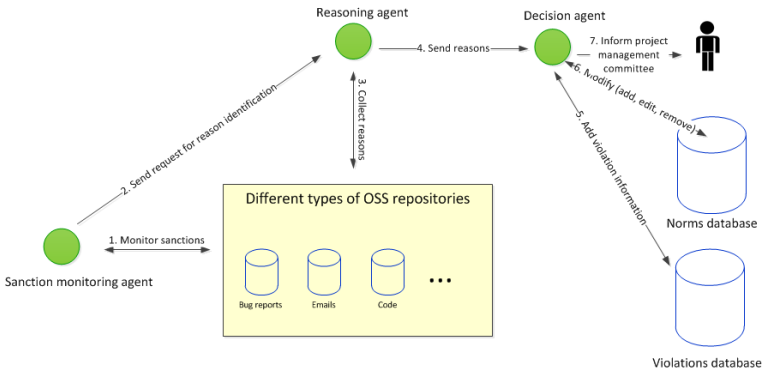


**Fig. 3.** Architecture for norm identification in open source software repositories

The architecture for the purpose of identifying norms in OSS repositories is given in Figure 3. The framework makes use of software agents to store, manipulate and disseminate normative information. There are three types of agents, sanction monitoring agent, reasoning agent and decision-making agent. The rectangular box represents different types of information that may be stored in different repositories (e.g. code repository and bug repository) which are mined for appropriate information. The process associated with identifying two types of norms, the pre-specified norms and the emergent norms is described below.

- Step 1: A sanction monitoring agent monitors for sanction information (e.g. messages containing sanctions) that are added to any of the monitored repositories. For example, a message may contain information where agent A informs B that it has violated a norm. However, the veracity of the sanctions need to verified by the system (i.e. the sanction was for the right reason - an actual violation).
- Step 2: Upon sanction identification, the monitoring agent sends a request to the reasoning agent that identifies the reason for the sanction to occur.
- Step 3: The reasoning agents identifies the reason from the appropriate repositories. Note that the reason for the sanction to occur can be found in one or more of the repositories.
- Step 4: Upon identifying the reason for the sanction, the reasoning agent sends this information to the decision making agent.
- Step 5: The decision making agent checks if the reason for the violation is because one of the existing norms in the norms database has been violated. If that is the case, the norm violation is recorded in the violation database (which can be used to identify norm uptake).
- Step 6: On the other hand, if the norm is a new norm (potential norm), it is added to the norms database and the violations database. Norm change (modification and deletion) is also facilitated at this stage.

– Step 7: When a new norm is added, the project management committee is informed about the new norm (who have the ability to modify the norms if need be).

The processes associated with sanction monitoring, reasoning and decision making are elaborated further in the following sub-sections.

### 5.1 Sanction monitoring

Enforcement of a pre-specified norm typically involves the delivery of appropriate sanctions. Such sanctions can be easily seen in a proprietary production process, e.g. if a developer working in a company is not following the rules, she may be at a risk of being warned or even fired. In contrast, it can be challenging to find evidence of sanctions in the OSSD setting since reliance on informal authority to accept/reject contributions tend to play a key role in the organization of OSSD. However, there are some good candidates that can be identified as sanctions.

The most visible forms of sanctioning in open source are "flaming" (public condemnation of developers that violate norms) and "shunning" (refusing to cooperate with the developers who have broken a norm) [16]. Evidences of such sanctions can be found in various archived sources (repositories) such as email lists, bug/ticket reports and forums. For example, a bug report on a module that does not deliver the specified functional requirements can be viewed as a sanction. Additionally, tickets issued for not resolving a bug completely can be considered as a sanction. Sanctions that follow violations act as triggers to infer norms. Frequency of norm violations over time may provide evidence for the uptake of a norm in a society.

We note that identifying and categorizing different types of sanctions from different types of artifacts is a challenge since the extraction of sanctions involves natural language processing. Verbose text may be used in the construction of sanction messages. For example, the messages may involve terms that are well beyond the deontic terms such as "should not", "must not", "ought not" in the case of prohibitions. One way to address this problem is to use existing ontological tools (e.g. WordNet [6]) to extract synonyms of terms used in the text to infer deontic terms and also use information retrieval tools that offer data manipulation functions such as cleaning and disambiguating the verbose text in order to extract sanctions. Suitability of tools such as OpenCalais[17] and AlchemyAPI[18] for this purpose can be investigated. We believe recognizing sanctions is indeed a huge challenge. At the same time, it presents opportunities such as the construction of normative ontologies that can be used across projects for recognizing sanctions. We envisage the Natural Language Processing (NLP) features discussed here will be built-in to the sanction monitoring agent.

### 5.2 Norm Identification

The process of recognizing norms (pre-specified and emergent) consists of two phases. In the first phase, the reasons for violations will be identified. In the second phase, the decision agent will decide whether a norm is pre-specified or emergent and also make other decisions accordingly.

[17] `http://www.opencalais.com`
[18] `http://www.alchemyapi.com`

**Reason identification** - The machinery proposed by Savarimuthu et al. [14, 15] can be used as a starting point to infer prohibition and obligation norms. In their work, prohibition norms are identified by extracting sequence of action (or actions) that could have caused the sanction by using a data mining approach [15]. Sanctions form the starting point for norm identification. In the case of obligation norms, missing event sequence (or sequences) that was responsible for the occurrence of a sanction, is identified [14]. The result of this process will provide a reason for the occurrence of violations (i.e. the reason for the sanction is the violation of a prohibition or an obligation norm).

While these work on norm identification can be used as a starting point for the extraction of norms in simple cases, the domain of OSSD poses non-trivial challenges. For example, correlating or linking different types of documents containing relevant information is required before a sequence of actions can be constructed. For example, an email message may contain the sanction message exchanged between developers A and B. Let us assume that A sanctions B for not adding a valid comment to the latest version of the file he or she uploaded. The problem in extracting the norm in this case is that, first, the verbose message sent from A to B should be understood as a normative statement which involves natural language processing. Second, a cross-check should be conducted to evaluate whether the normative statement is indeed true (i.e. checking whether the comment entered by B is invalid by investigating the log)[19]. We envisage these tasks will be performed by the reasoning agent.

**Decision making** - A decision making agent will identify whether the reason for the sanction is because of the violation of an existing norm or an emergent norm (not yet known, but could potentially be identified as a norm). If it is a violation of an existing norm, it records this information in the violation database (which can be used to study norm uptake, i.e. how common are violations of a particular norm). Otherwise, if it is a new norm and if it meets certain threshold (for reporting), it adds the new norm to the norm database and also sends this information to the project management committee who then may decide to approve the new norm.

### 5.3 Discussion

When the architecture is implemented and run, we envisage that there will be a number of instances of sanction monitoring agents and reasoning agents. For example, a monitoring agent can monitor certain type of norm violation and a reasoning agent can be responsible for mining the reason for the violation from different repositories.

We believe, the field of computer science now is in the cusp of transformation where the challenges posed by big data can only be solved by employing techniques from a variety of disciplines. The framework that is developed based on the above proposed architecture should be equipped with appropriate libraries for (a) information retrieval techniques (including natural language processing) in order to identify sanctions; (b)

[19] In this example only two artifacts, the email message and the log are involved. But in practice, several different types of documents may need to be traversed to find the relevant information. Techniques developed in the field of MSR (e.g. [1], [12]) can be employed for cross-linking documents.

mining software repositories (e.g. cross-linking different sources); and (c) norm extraction (e.g. inferring norms from sequences of events).

We also believe that the architecture proposed in this paper can be the basis of studying interesting cross-disciplinary questions such as the ones given below.

*Q1. How different are norms in large projects (e.g measured based on total number of members or size of the project in kilo-lines of code) than the smaller projects? Are norm violation and enforcement patterns different in these projects?*

*Q2. What are the relationships between roles of individuals in software development and norms (e.g. contributor vs. reviewer vs. module administrator)?*

*Q3. Are there cultural differences within members of a project with regards to norms (inter- and intra- project comparisons) since individuals from different cultures may have different norms?*

*Q4. Is there a difference between norm adoption and compliance between open-source and closed-source projects?*

These questions may interest both social researchers and computer scientists. Synergy between the two is required for addressing these questions. As computer scientists we can employ our expertise in several areas (i.e. information retrieval, MSR and normative multi-agent systems) to help answering these questions.

## 6 Conclusions

Open source projects are real-life, large-scale, multi-agent organizations whose data repositories are ripe for mining normative information. In this paper, we have discussed how this important application area can be leveraged by employing the techniques developed by researchers in Normative Multi-agent Systems (NorMAS), mining software repositories and also other research disciplines in computer science towards the goal of understanding normative processes that operate in human societies. Responding to the call for challenging domains to apply agent-based data mining techniques, the main motivation of the paper has been on bringing the issues forward to the agent community in general and also discussing the initial work we have undertaken. Towards that goal, first, we have presented the results on norm compliance on three open source software projects. Second, we have presented a high-level, agent-based architecture for mining norms in open source software repositories. We have also highlighted the research challenges that need to be addressed in the future.

## 7 Appendix

Basic information about the three projects are provided in Table 4. Columns 3 and 4 show the version numbers of the projects considered and the total number of violations observed (including the five categories of violations presented in the case study).

| Projects | Description | Version | Convention violations |
|---|---|---|---|
| Apache ODE | Business process execution engine for processes written in WS-BPEL standard. | 1.3.5 | 60608 |
| Apache Ant | Ant is a Java library, mainly used to compile, assemble, test and run Java applications. | 1.8.4 | 48082 |
| Apache Struts | Struts is framework for creating Java web applications | 2.3.4 | 97457 |

**Fig. 4.** Basic project information of the projects considered in the case study

## References


1. C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 137–143, New York, NY, USA, 2006. ACM.
2. C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. In *ICSM*, pages 277–286, 2008.
3. S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Mining java class naming conventions. In *ICSM*, pages 93–102, 2011.
4. L. Cao, G. Weiss, and P. Yu. A brief introduction to agent mining. *Autonomous Agents and Multi-Agent Systems*, 25:419–424, 2012.
5. N. Criado, E. Argente, and V. J. Botti. Open issues for normative multi-agent systems. *AI Communications*, 24(3):233–264, 2011.
6. C. Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.
7. A. Hassan. The road ahead for mining software repositories. In *The 24th IEEE International Conference on Software Maintenance, Frontiers of Software Maintenance.*, pages 48 –57, October 2008.
8. C. Kapser and M. W. Godfrey. Cloning considered harmful considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.
9. H. Kargupta, I. Hamzaoglu, and B. Stafford. Scalable, distributed data mining-an agent architecture. In *Proceedings Third International Conference on Knowledge Discovery and Data Mining*, pages 211–214, 1997.
10. F. Kooti, H. Yang, M. Cha, P. K. Gummadi, and W. A. Mason. The emergence of conventions in online social networks. In *Proceedings of the Sixth International Conference on Weblogs and Social Media, Dublin, Ireland, June 4-7, 2012*, 2012.
11. H. X. Li and R. Chosler. Application of multilayered multi-agent data mining architecture to bank domain. In *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*, pages 6721 –6724, sept. 2007.
12. N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.
13. B. T. R. Savarimuthu and S. Cranefield. Norm creation, spreading and emergence: A survey of simulation models of norms in multi-agent systems. *Multiagent and Grid Systems*, 7(1):21–54, 2011.
14. B. T. R. Savarimuthu, S. Cranefield, M. A. Purvis, and M. K. Purvis. Obligation norm identification in agent societies. *Journal of Artificial Societies and Social Simulation*, 13(4):3, 2010.

15. B. T. R. Savarimuthu, S. Cranefield, M. A. Purvis, and M. K. Purvis. Identifying prohibition norms in agent societies. *Artificial Intelligence and Law*, pages 1–46, 2012.
16. S. Weber. *The Success of Open Source*. Harvard University Press, Apr. 2004.
17. R. J. Wieringa and J.-J. C. Meyer. Applications of deontic logic in computer science: a concise overview. In *Deontic logic in computer science: Normative system specification*, pages 17–40. John Wiley & Sons, Inc., New York, USA, 1994.
18. T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.