Mining Software Repositories for Social Norms

Hoa Khanh Dam
University of Wollongong
Australia
hoa@uow.edu.au

Bastin Tony Roy Savarimuthu
University of Otago
New Zealand
tony.savarimuthu@otago.ac.nz

Daniel Avery
University of Wollongong
Australia
davery@uow.edu.au

Aditya Ghose
University of Wollongong
Australia
aditya@uow.edu.au

***Abstract*—Social norms facilitate coordination and cooperation among individuals, thus enable smoother functioning of social groups such as the highly distributed and diverse open source software development (OSSD) communities. In these communities, norms are mostly implicit and hidden in huge records of human-interaction information such as emails, discussions threads, bug reports, commit messages and even source code. This paper aims to introduce a new line of research on extracting social norms from the rich data available in software repositories. Initial results include a study of coding convention violations in JEdit, ArgoUML and Glassfish projects. It also presents a new life-cycle model for norms in OSSD communities and demonstrates how a number of norms extracted from the Python development community follow this life-cycle model.**

I. Introduction

Social norms plays an important role in governing individuals' behavior in group settings (e.g. societies and communities), and regulating the interactions between those individuals. Norms can emerge formally where behavioral expectations are explicitly described and implemented by groups, e.g. overspeeding prohibited by laws. Norms can also arise informally, representing generally accepted and universally sanctioned routines in our daily life, e.g. people obliged to queue at a counter. Both formal and informal norms are crucial to the smooth functioning of social groups. Norms are dynamic, evolve or devolve over time, can spread from one community to another, but can also vary between communities. Social norms have long been an important research topic in sociology (e.g. [1]), psychology (e.g. [2]), economics (e.g. [3]) and recently computer science (e.g. in the area of multi-agent systems [4] or mining social media [5]).

There are, however, very little work on norms in the fast emerging, highly diversified open source software development (OSSD) communities. Large OSSD communities like Linux and Android OS have hundreds of contributors and millions of users around the world. OSSD communities seem to be largely governed by norms, and in many cases emerged without an initial formal organizational structure with regulations (i.e. norms and policies) explicitly stated and enforced. Hence, an in-depth study on norms and extracting them to make them explicit to stakeholders (*norm mining*) would significantly benefit the software engineering community in general and the OSSD communities in particular.

An important research is to study how the functioning of OSSD communities is shaped by norms, in particular the decision-making process about how rules are initially suggested, supported and ratified to become a policy (i.e. the *normative process*). For example, members of a newly formed open source project have to decide the policy governing who is authorized to commit code to the system. The policy may be initially suggested by some members, which upon gaining support, may be codified. These decision making processes are not captured formally in many open source projects. Understanding such underlying normative processes is very important since the construction of (open source) software is a complex group activity involving highly autonomous, volunteering contributors who have different levels of experience and are from different background, cultures and geographical regions.

A huge amount of human-interaction (e.g. between developers, users, admins, and other contributors) information on software development have been recorded and stored in software repositories such as mailing lists, and discussion threads, bug reports, source code, and commit messages (e.g. in 2012 Mozilla Firefox had 800,000 bug reports [6]). Hidden in such rich data are norms which were *implicitly* created, discussed and enforced. This offers an excellent opportunity for doing an in-depth study on norms in OSSD communities. Although many research (e.g. [7]) in mining software repositories have leveraged those kinds of data, to the best of our knowledge none of those work dedicated to norm mining – extracting norms and making them available to community members.

Our current research aims to fill that gap by addressing the following topics:

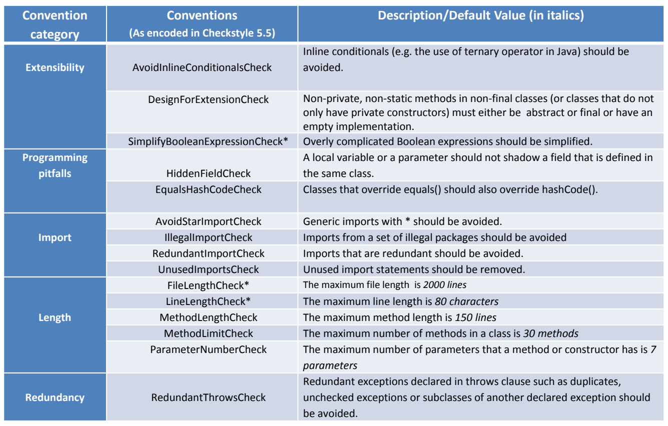
1) Developing a norm life-cycle model that describes various phases a norm goes through, from its formation to enforcement, in OSSD communities. This life-cycle model is then used to extract norms, particularly those that emerge (without planning or design) from individuals' interactions, and analyze the conditions under which such norms come into existence.
2) Understanding how norms evolve, vary and spread from one OSSD community to another. An interesting focus is identifying the difference in the type of norms and the nature of normative processes in successful and failed software projects, the impact of norm compliance on the success of projects, and the impact of different stakeholders' roles in the norm life-cycle.
3) Developing norm "databases" for OSSD communities such that the extracted norms are explicitly recorded, updated and reasoned about (e.g. checking for inconsis-

tencies between norms). This provides a basis for more advanced support such as making (new) members be aware of norms related to the context of their work and contribution.

The remaining parts of the paper discuss some initial results of our work towards the above research directions.

II. Norms and Conventions in Software Development Communities

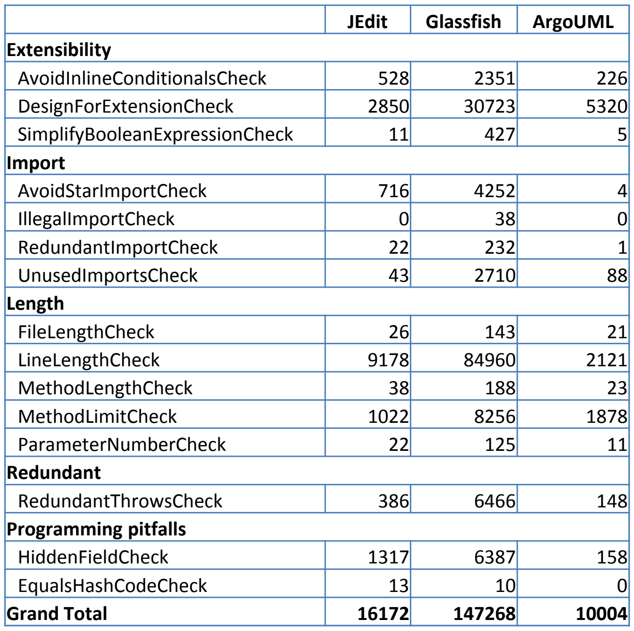
Convention is a common expectation amongst (most) others that an agent should adopt a particular action or behaviour (e.g. the convention in ancient Rome was to drive on the left). Coding standards of an OSSD community is an example of conventions. The specifications of these conventions may be explicitly available from the project websites or can be inferred implicitly. For example, many open source Java projects (e.g. JEdit, ArgoUML and Glassfish) assume that their developers should use the Java coding conventions. Figure 1 shows 15 examples of Java coding conventions which are divided into five groups: extensibility, import, length, redundancy and programming pitfalls. Note that some of these are conventions that were originally specified in the Java coding conventions (e.g. the length of a line in Java should be less than 80 – these are marked with *), while others have informally emerged over time as good practices, but have not been updated in the Java convention specification (e.g. avoiding star imports).



Convention category	Conventions (As encoded in Checkstyle 5.5)	Description/Default Value (in italics)
Extensibility	AvoidInlineConditionalsCheck	Inline conditionals (e.g. the use of ternary operator in Java) should be avoided.
	DesignForExtensionCheck	Non-private, non-static methods in non-final classes (or classes that do not only have private constructors) must either be abstract or final or have an empty implementation.
	SimplifyBooleanExpressionCheck*	Overly complicated Boolean expressions should be simplified.
Programming pitfalls	HiddenFieldCheck	A local variable or a parameter should not shadow a field that is defined in the same class.
	EqualsHashCodeCheck	Classes that override equals() should also override hashCode().
Import	AvoidStarImportCheck	Generic imports with * should be avoided.
	IllegalImportCheck	Imports from a set of illegal packages should be avoided
	RedundantImportCheck	Imports that are redundant should be avoided.
	UnusedImportsCheck	Unused import statements should be removed.
Length	FileLengthCheck*	The maximum file length is *2000 lines*
	LineLengthCheck*	The maximum line length is *80 characters*
	MethodLengthCheck	The maximum method length is *150 lines*
	MethodLimitCheck	The maximum number of methods in a class is *30 methods*
	ParameterNumberCheck	The maximum number of parameters that a method or constructor has is *7 parameters*
Redundancy	RedundantThrowsCheck	Redundant exceptions declared in throws clause such as duplicates, unchecked exceptions or subclasses of another declared exception should be avoided.

Fig. 1. Examples of Java coding conventions

Coding conventions were however not strictly enforced in a range of OSSD projects. We have performed an empirical study using CheckStyle[1], a coding standard analyzer for Java, to analyze how those 15 coding conventions in Figure 1 were honoured in three well-known Java-based open source projects: JEdit, ArgoUML and Glassfish. Figure 2 presents our findings, which show a substantial number of instances (16,172 instances in JEdit, 147,268 in Glassfish, and 10,004 in ArgoUML) where the 15 Java coding conventions were violated. Some of those conventions (e.g. DesignForExtensionCheck) were frequently violated while there are fewer instances of violation for others (e.g. IllegalImportCheck), which may reflect the degree of enforcement from the community towards each of those conventions.



	JEdit	Glassfish	ArgoUML
Extensibility			
AvoidInlineConditionalsCheck	528	2351	226
DesignForExtensionCheck	2850	30723	5320
SimplifyBooleanExpressionCheck	11	427	5
Import			
AvoidStarImportCheck	716	4252	4
IllegalImportCheck	0	38	0
RedundantImportCheck	22	232	1
UnusedImportsCheck	43	2710	88
Length			
FileLengthCheck	26	143	21
LineLengthCheck	9178	84960	2121
MethodLengthCheck	38	188	23
MethodLimitCheck	1022	8256	1878
ParameterNumberCheck	22	125	11
Redundant			
RedundantThrowsCheck	386	6466	148
Programming pitfalls			
HiddenFieldCheck	1317	6387	158
EqualsHashCodeCheck	13	10	0
Grand Total	**16172**	**147268**	**10004**

Fig. 2. The number of coding convention violations in some OSSD projects

As conventions gain force, the violations of conventions may be sanctioned at which point a social norm comes into existence. For example, if driving on the right is sanctioned, the left-hand driving becomes a norm. 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. Norms in OSSD communities may generally be classified into three categories: prohibition, obligation and permission (as in deontic norms [8]). *Prohibition norms* prohibit members of a project group from performing certain actions. 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. 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 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 or forks.

III. Norm life-cycle in OSSD communities

Understanding how norms are created, codified, monitored and enforced is crucial to the extraction of norms in OSSD

[1]http://checkstyle.sourceforge.net

communities. We propose the following four-stage life-cycle model of norms in OSSD communities. An instance of the four-stage model is shown in Figure 3.

Phase 1 – Convention formation: At this initial stage, the members of the community discuss what the conventions of the software project should be. Conventions not only concern coding conventions but also could govern other aspects of the community such as the deadline for attending to a reported issue and the number of members that should verify a supplied patch before it can be accepted. The proposal for conventions could be put forward by leaders of the project (e.g. Linus Torvalds in the Linux project), or they could come from the other members. The agreement on the conventions are based on consensus for those issues that are known *a-priori*. However, conventions can also be emergent. Once the project is well underway, there could be a convention that all version changes should be accompanied with a non-trivial explanation or comment on the change that was made. Thus, the pool of conventions is amenable to change.

Phase 2 – Convention codification: Once the conventions 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[2] and the Android development community[3] provide guidelines on conventions including coding conventions. Once the convention has been codified, that forms the basis of expectations, and hence *norms*. It is expected that the members of the project community adhere to these norms.

It should be noted that not all the conventions might be codified. In many communities, the conventions are assumed to be common knowledge and they exist only in the minds of the individuals (usually in the tacit form). For example, in the Java-based project communities there are uncodified conventions, such as not using star imports or overriding the *hashCode* method whenever overriding the *equals* method, that exist in various forums outside the community pages associated with a OSS project (blogs, discussion boards, etc.). However, there are issues with these uncodified (informal) conventions. It is difficult to assess how often violations are sanctioned (i.e. the salience of the norm) and there are also ambiguities surrounding whether the project development infrastructure (e.g. project submission system) can automatically check for the violations of the conventions and potentially sanction violators.

Phase 3 – Norms monitoring: 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[4] and StyleCop[5] 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 on any violations. Though centralized monitors facilitate tighter control, they can be used only for simple "static" checks. More involved inspections are possible when humans are involved in the loop.

Phase 4 – Norm enforcement: While using one of the convention checking tools, the enforcement could be 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. Thus, the process enables a feedback loop to the norm formation phase.

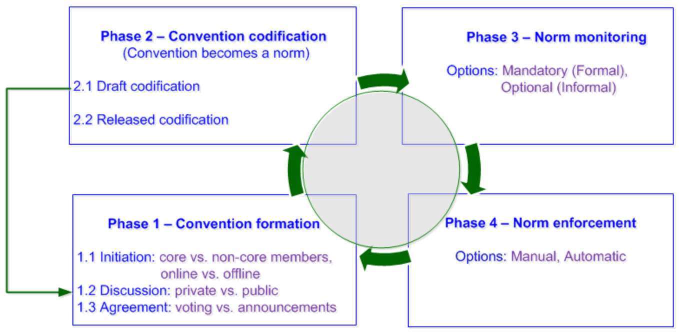

Phase 2 – Convention codification
(Convention becomes a norm)
2.1 Draft codification
2.2 Released codification
Phase 3 – Norm monitoring
Options: Mandatory (Formal), Optional (Informal)
Phase 1 – Convention formation
1.1 Initiation: core vs. non-core members, online vs. offline
1.2 Discussion: private vs. public
1.3 Agreement: voting vs. announcements
Phase 4 – Norm enforcement
Options: Manual, Automatic

Fig. 3. Norm life-cycle in the Python project community

IV. A CASE STUDY OF NORMS IN THE PYTHON PROJECT

The norm life-cycle discussed above was used as a basis for our norm mining in the well-known Python[6] project community. Python contributors use mailing lists to propose, discuss and select norms for the community. We mined the email messages during two different phases of the project: (1) early 2000s when the norms of collaboration using versioning systems were first used and several norms were proposed; and (2) 2011-2013 where some changes to norms were proposed. The first phase was manually done and the second phase was semi-automated by a norm miner tool. Figure 3 shows a lifecycle of norms in the Python project community. We found that any new proposal or change proposed to the Python language is initiated by a member (or members) of the core member group (who has more privilege in code submission) or the non-core member group. The initiation could happen over meetings (e.g. physical meetings or in a teleconference or using an online messaging service) or other channels (e.g. emails). If the proposal is from a core-member group, the proposal goes for discussion amongst the non-core members. The proposal may be subject to some modifications based on ensuing discussions and would be accepted for implementation.

[2]http://portals.apache.org/development/code-standards.html
[3]https://sites.google.com/a/android.com/opensource/submit-patches/code-style-guide
[4]http://checkstyle.sourceforge.net/
[5]http://archive.msdn.microsoft.com/sourceanalysis
[6]http://www.python.org

ID	Norm description	Discussion	Agreement	Enforcement	Norm type
1	Core developers can submit small fixes without going through the mandatory code review stage	Medium	Agreement	Optional	Permission
2	Security issues and fixes should not be discussed in public channels until a fix has been announced on an official channel	Medium	Agreement	Mandatory	Prohibition
3	The Python mentor program should be a gated community, requiring registration and approval by an admin.	Large	Agreement	Mandatory	Obligation
4	Documenting public API's are encouraged but not mandatory	Medium	Vote	Optional	Permission
5	The API Code should be frozen after Python 3.2 release	Small	Declare	Mandatory	Obligation

Fig. 4. Example of norms mined from Python development mailing list during 2011–2013

However, if a non-core member initiates a proposal, then a voting is conducted. The voting process used is documented as a separate process[7]. The proposal may or may not be accepted (by the core member group and the non-core member group). This is different from proposals that come from the core member group which always tend to get accepted. When the proposal from non-core member group is accepted it might go for another round of discussions. The announcement of acceptance or rejection is made by one of the members of the core member group. Once the proposal is accepted, it can then be monitored and enforced. Monitoring can be mandatory or optional. Enforcement can be automatic (e.g. creating a bug entry when an automatic build process fails) or can be manual (i.e. a developer sanctions another in an email).

We have developed a norm miner tool (leveraging data mining and natural language process techniques) and used it to further mine 31,574 email messages from Python development mailing list from 2011–2013. Approximately 15% of those emails concerned with norms creation (as either voting or agreement messages) and there were 3,751 norms extracted from them. Due to space limitations, we present only 5 representatives in Figure 4. We found that the overwhelming majority of norms were initiated by core members. This may suggest that norm emergence in Python community occurred centrally and were initiated by a relatively small part of the community. The majority of discussions ended democratically (by voting or agreement). Norms that were declared (such norm #5 in Figure 4) often had very little discussion, suggesting that these norms were not resisted by the community. Some of the norms (e.g. norms #2, #3, and #5) are enforced mandatorily which could suggest that these norms were a response to critical issues. Obligation norms (e.g. norms #3 and #4) were dominant in Python, which may imply that the community is more concerned about discussing what they should be doing as opposed to what they should not be doing.

V. Conclusions and Future Work

Social norms are critical to the functioning of distributed and diverse social groups like OSSD communities. This paper has outlined a number research directions in the study of norms in OSSD communities, especially leveraging the huge amounts of human-interaction data generated from the software development activities. Our empirical study on the degree of coding convention compliance in a number of large open source software projects highlighted the differences between conventions and norms. We have also proposed a life-cycle model of norms in OSSD communities, and demonstrated how extracted norms from the Python project community follow this life-cycle.

Our initial work here has opened up the possibility of answering a number of norm-related questions which have seldom been answered in the context of open source software development. Future work involves addressing the remaining topics we raised earlier in the paper, particularly studying the impact of norms on the success/failure of a software project and building an explicit norm "database". These topics would interest both social researchers and computer scientists. We believe a synergy between the two is required for addressing these broader questions. Additionally, the knowledge obtained through answering these questions will advance our understanding of human interaction norms in the open source domain.

References

[1] B. Herrmann, C. Thöni, and S. Gächter, "Antisocial punishment across societies," *Science*, vol. 319, no. 5868, pp. 1362–1367, 2008.

[2] R. B. Cialdini, "Descriptive social norms as underappreciated sources of social control," *Psychometrika*, vol. 72, no. 2, pp. 263–268, 2007.

[3] J. Elster, "Social norms and economic theory," *The Journal of Economic Perspectives*, vol. 3, no. 4, pp. 99–117, 1989.

[4] 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*, vol. 7, no. 1, pp. 21–54, 2011.

[5] 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.

[6] T. Menzies and T. Zimmermann, "Software analytics: So what?" *Software, IEEE*, vol. 30, no. 4, pp. 31–37, 2013.

[7] *MSR 2014: Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2014.

[8] 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*. New York, USA: John Wiley & Sons, Inc., 1994, pp. 17–40.

[7]Refer to Python Enhancement Proposal 10 (PEP 10) - http://www.python.org/dev/peps/pep-0010