Distributed Collective Practices and Free/Open-Source Software Problem Management: Perspectives and Methods

Les Gasser[1,2] — Gabriel Ripoche[1,3]

[1] *GSLIS, University of Illinois at Urbana-Champaign, USA*

{gasser,gripoche}@uiuc.edu

[2] *ISR, University of California at Irvine, USA*

[3] *LIMSI-CNRS, Université Paris-Sud, France*

ABSTRACT. *This paper presents the state of our research on Distributed Collective Practices (DCPs) in Free/Open-Source Software (F/OSS) projects, focusing on sensemaking and resolution of software problems. We are exploring the hypothesis that variations in the content and in the articulation of these socio-technical processes have an impact on the outcome of the activity of F/OSS collectives, and more specifically on problem resolution. Our preliminary techniques for combining qualitative data analysis with automated process extraction result in a scalable analysis method called "Computational Amplification" (CA). We are applying CA to 128,000 problem reports from the Mozilla F/OSS project. The paper illustrates how CA is used to create multidimensional process models and shows types of conclusions we can reach.*

RÉSUMÉ. *Cet article présente l'état d'avancement de nos recherches sur les Pratiques Collectives Distribuées (DCP) du Logiciel Libre et de l'Open Source (F/OSS) ; et en particulier sur les pratiques liées à la création de sens et à la gestion de problèmes logiciels. Nous partons de l'hypothèse que les variations dans le contenu et l'articulation de ces processus socio-techniques ont une influence sur les résultats de l'activité de ces collectifs F/OSS, et en particulier sur la résolution des problèmes logiciels. En combinant analyses qualitatives et techniques d'extraction automatique, et en appliquant cette méthode à 128 000 rapports de problèmes du projet Mozilla, nous décrivons le type de modèles multi-dimensionnels que nous développons et reportons plusieurs observations obtenues à l'aide de cette méthode.*

KEYWORDS: *Software problem management, Collective knowledge management, Automated process extraction, Information extraction from natural language texts.*

MOTS-CLÉS : *Gestion de problèmes logiciels, Gestion collective des connaissances, Extraction automatique de processus, Extraction d'information à partir de textes en langue naturelle.*

1. Distributed Collective Practices

We are interested in how large-scale socio-technical systems work: what processes and technologies are used, how work is coordinated, what patterns of information use emerge, and what methods seem more and less effective. We give the name "Distributed Collective Practices" (DCPs) to the phenomena we study. Briefly, we construe DCPs by examining four elements: *activities*, *objects*, *social and technical infrastructures*, and *common problems and methods*.

The basic research issues for DCP are how to understand, organize, and support DCPs to achieve objectives such as:

– Collective sensemaking: How can people reach common collective interpretations of events and objects?

– Collective construction, selection, and stabilization of objects of analysis: How can people jointly delimit and stabilize events, documents, interpretations, technologies, etc.?

– Collective sustainability of objects and processes: How can DCPs be sustained over long periods of time in dynamic socio-technical environments?

– Collective innovation: How can DCPs innovate the very objects and processes that constitute them?

– Calculability: How can participants in DCPs establish the inter-calculability of objects, placing them into joint discourse, relating them, establishing exchange values for them? [CAL 98]

– Coordination: How can DCPs construct and use objects (rather than markets or hierarchies) as arenas for, and means of, coordination.

In addition, our work is driven by two principal theoretical issues:

– *Sensemaking*: How are "meanings and artifacts produced and reproduced in [the] complex nets of collective action" [CZA 92] that characterize DCPs?

– *Information and activity*: How does information shape activity and how does activity shape information in ongoing DCPs?

2. A Specific DCP: Problem Management in F/OSS Projects

Free/Open Source Software (F/OSS) is software built by widely distributed groups of people who agree to share the products of their work, including the source code, for free. A number of large-scale[1] F/OSS efforts have been in place with lifespans of several years or more, and these have developed products of significant functionality and sophistication that are in widespread use. These efforts cross a number of application domains, including basic Internet and computing infrastructure (such as the

1. By large-scale we mean on the order of several hundred developers and tens of thousands to millions of lines of source code.

Mozilla Web Browser suite, the Apache Web Server, and the Linux kernel); office productivity tools (such as the OpenOffice.org word processor/spreadsheet/presentation system); scientific analysis software (such as data-analysis software for the Chandra X-Ray Observatory); and "mods" for Massive Multi-Player Online Role-playing Games (MMPORGs). We have been studying each of these to some degree.

We have been examining software development and maintenance processes in a number of these projects, to understand how DCPs unfold in a specific kind of collective effort. Software development, and specifically "bug fixing", is of great interest for several reasons. First, all software contain bugs, and both users and developers would like to reduce or eliminate them as sources of failure and misfit with user's tasks. Thus, we can assume at least some level of collective interest and motivation that serves to glue the community together, and render many of the distributed collective practices mutually intelligible from the outset. Second, problem management is a delimited kind of technical work which exhibits constrained patterns of activity and information use. These constraints make it somewhat easier to generate basic insights into DCPs.

We approach F/OSS bug fixing and DCPs with a set of basic, practical questions to guide our research. Primarily, we want to know how large online communities continuously (re)design and (re)develop software over long periods of time; how they manage *continuous streams* of software errors and mistakes; and how they capture, represent, and use collective knowledge in the repair and redesign of software.

3. How Do DCP Communities Build Software and Manage Problems?

The standard model of software development for tightly organized (e.g., common commercial and industrial) projects comprises a set of stages carried out in a basically linear fashion [SCA 01]. Each earlier stage sets the context of decisions and actions for the following stage. A typical staged development model includes activities of analysis, specification, design, implementation, testing, release, and use. These stages of activity occur in the order given, with minimal feedback and return to earlier stages. Each stage is supported and enacted using specialized, stage-specific tools and infrastructure (e.g. specification languages for capturing specifications, language editors and compilers for implementation, automated test and debugging tools for test phases, etc.). This linear model relies on the ability to fully specify the functionality of software before the software is implemented. With this constraint, specification provides a natural guide for implementation, and implementation can be carried out in a focused (i.e., less experimental) way. Similarly, a fully functional specification provides a standard against which testing can be done. This model of testing to a pre-existing standard provides a more-or-less unambiguous definition of software quality: high quality software is software which meets its specifications. Finally, cost studies of software built under the linear model have repeatedly shown ten- to fifty-fold cost reductions for software problems that emerge during specification phases, over those that emerge in release phases. Because of this tremendous cost difference, there is

great pressure to enhance the detail and quality of specification activities, as a way of pushing costs earlier and thereby reducing them.

However, with this linear development model, two central questions arise. First, are there classes of systems whose functionality cannot be specified before they are implemented and used? Second, what if linear models don't scale up to ever-larger, more modular systems, built by widely distributed communities such as F/OSS efforts built under regimes of DCP? What new models are necessary for large-scale, distributed, modular systems? [MOC 02, BRO 02]

3.1. *Software Refinement*

As mentioned above, all software contain "bugs", and to some degree most users find problems putting software to work in their specific contexts of use [GAS 86]. In general, we are interested in understanding the process through which users and developers address these problems. A simple model of this process is as follows:

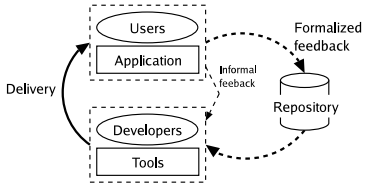

Figure 1. *Feedback process in F/OSS development.*

Developers use tools to build a version of their artifact, and then deliver it to users. When a user has a problem, one common approach is to make the problem *explicit* by converting the experience of the problem into some communicable form and addressing it to developers. We call this a *user-integrated* development process, and it appears frequently in F/OSS projects[2]. A key step here is the user's transformation of a problem from an *unrepresented local phenomenological event* into an *explicit representation* which can be communicated—whether that representation be a transient spoken description (e.g. in a telephone call) or a more persistent written or diagrammed document.

In a small-scale user-integrated project, where users may have easy access to developers, feedback on artifact use and problems may be informal, non-persistent, and direct (represented by the small dashed line in Figure 1). However, as development processes become larger-scale and as relationships between users and developers become more complex, both users and developers must employ complexity reduction

2. For example, Mockus and colleagues note the large proportion of "user-developer/developer-user" participants in F/OSS communities as a feature which makes user involvement quite high [MOC 00, MOC 02].

strategies to manage the volume of information and to sustain their relationships. One typical way to do this is to create more formalized, persistent feedback systems to capture and manage information about problems. Such feedback systems typically take the form of a collection, database, or repository of reports on problems and issues, sometimes accompanied by accumulated information and discussion about each issue and what steps are being taken to deal with it (represented by the larger dashed arrows and the repository in Figure 1). In practice, communities organize formalized feedback systems in several ways. More *unstructured* repositories use easy-to-capture, searchable information lists supported by very widespread technologies such as hypermail archives or newsgroup managers (e.g., Chandra X-ray software, Apple's problem-management newsgroups). More *structured* repositories employ specialized, less widely available database tools designed specially for problem management, such as Bugzilla, Scarab, and many others.

3.2. *Repository Structure*

All the user-integrated problem-tracking repositories that we have examined—informal repositories and formal, structured ones—contain at least two kinds of information: 1) individual informal items of information such as emails describing issues, workarounds, and repair strategies, and 2) some way of capturing, interrelating and structuring ongoing *streams* of such informal items and activity records, creating collections. Moreover, structured repositories also contain formalized information such as statuses and timelines for tracking activities, controlled vocabularies for specifying issue attributes, lists of people involved, votes for importance, module in which the problem appears, and so on. Figure 2 presents a general view of repository structure. It specifically represents the Bugzilla approach, but the idea is quite general.

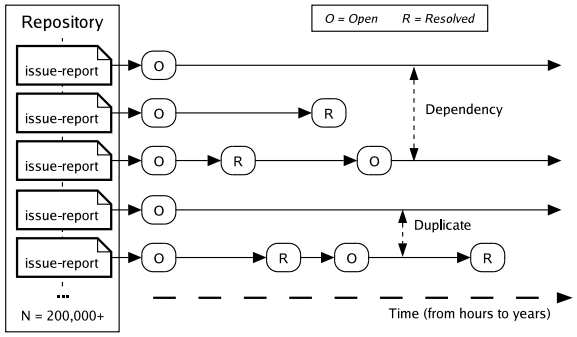

Figure 2. *Structure of problem-tracking repositories.*

Using a problem-tracking repository as part of a user-integrated development process leads to several key transformations in how the artifact's community manages knowledge. The following section discusses some of these transformations.

4. Transformations of Knowledge

4.1. *Accumulation of Community Knowledge*

The creation of a community's knowledge of artifacts becomes transformed from a *linear/learned* process to a *continuous/constructed* one. When software is built under the "specify-design-build-test-release-maintain" linear model, users are necessarily isolated from direct involvement in the development process as it unfolds. Because of this, users can only learn about their artifacts by being told by developers—for example, via documentation, manuals, and training. With a persistent user-integrated process, we find that users learn about their software artifacts quite differently. Under this model, the developer-user community builds on an initial seed artifact, continuously and collectively transforming the community's knowledge of what the target artifact *is*, how it *works*, what it *should be*, and how it *should work*. That is, the community engages in continuous collective (re)design. What is not yet well understood are the specific processes of this development and transformation. However, we have observed the following points:

– Community knowledge is constructed from a multiplicity of viewpoints, representations, and experiences. Because of the wide variety of people participating in F/OSS projects (users, "power" users, developers, testers, etc.), the knowledge development/transformation process is characterized by extreme multiplicity of viewpoints, representations, experiences, and usages.

– Community knowledge development is enacted through knowledge management environments. F/OSS design/development processes are made possible and enacted by shared, structured, and (more or less) complex knowledge management environments that persistently capture traces of design and analysis. These systems allow the community to accumulate, organize, make sense of, and use the wide variety of knowledge contributed by the many people involved.

– The artifact (and knowledge about it) is constructed through a dynamic network of social processes. The "linear with feedback loops" structure of the more standard software development process is replaced by a network of social processes arranged in a highly dynamic topology, including: design, construction, testing, releasing, work coordination, critiquing, use, suggesting, specification, tool-building, triage, negotiation, evaluation, etc. Compared to processes represented in a rational design scenario, processes constituting F/OSS design/development are varied and numerous, and dynamically and loosely articulated.

4.2. *Representation of Community Knowledge*

Persistent repositories used in continuous distributed collective practices of F/OSS design give us new ways of thinking about representation. Indeed, we believe understanding them may require new models and theories of representation that *function at a collective, heterogeneous level*. This is a transformation from a representation model

in which problem reports are taken to be *bounded, faithful reflections* of phenomena experienced by users and of conditions in source code, to a conception of problem reports as *massive, contentious assemblages* of information subject to *ongoing reinterpretation*.

Standard models of knowledge representation underlying problem report systems assume a conventional relation between problem reports on the one hand, and the use and testing experiences they represent on the other hand. In this view, documents bear a relationship to an experiential life-world, and documents represent things, events, and experiences through this relationship. Empirical examination of the actual processes of continuous collective software design is instead revealing a quite different underlying representation model.

– Documents record viewpoints, and these viewpoints are subject to multiple interpretations. Many comments are typically made on each problem report (an average of 10 comments per report in Bugzilla). These comments often contradict or conflict with each other, and some comments are not interpretable by some readers and respondents. Moreover, problems are reported more than once in different ways, and it is often difficult to correlate or link these reports and their underlying manifestations.

– The scope of documentation is unknowable. Duplicate bug reports are not always linked, index terms differ, and since the documentation system itself is a large, open system, its state is continuously being revised. Therefore, knowledge is incomplete in the sense that there might be "more out there", but that this potential knowledge is not accessible to the community as a whole. For example, separate groups might work for months on related problems (even identical problems sometimes) before realizing that their work is related and creating the "missing link".

– The fidelity of relationship between documents and experiences is uncertain. Problem reports are textual discourses that purport to describe aspects of problems; they are not the problems themselves. Numerous debates in the Bugzilla corpus involve negotiation over the "proper" interpretation or referent of a problem report (e.g.: Which underlying cause or manifestation does it refer to? Which experience is "real" and which one is incorrect?) Thus, it is in general not a routine matter to directly and unambiguously match a problem description with an experience or a cause.

– The relevant life-world is unknowable. Users have widely varying styles of use and functional requirements, which are in general impossible to fully describe in a distributed context. This multiplicity, diversity, and distribution together mean that the scope of experience underlying problem descriptions is not knowable.

Taken together, these four observations mean that standard notions of representation fail to capture what is really going on in collective continuous design contexts such as the F/OSS projects we are studying, and new models are needed. We believe that conventional ideas of representation may need to be modified or reinvented to account for how large, distributed, multi-party information repositories are used in DCPs.

5. Specific DCP F/OSS Issues

We have begun studying in detail the general issues raised above using a large collection of research data from the Mozilla project. Mozilla, a successor to the well-known Netscape Communicator, is a modern, open-source web browser suite containing a standards-based HTML rendering engine, a web browser, a mail/news reader, an HTML page composer, an IRC tool, and several other components. The users and developers in the Mozilla community manage software problems using a (now widely disseminated) repository tool called Bugzilla, which itself was built as part of the Mozilla project. Our Bugzilla snapshot contains over 128,000 problem reports, of which about 88,000 have been resolved. The average report contains approximately ten unstructured comments, though some contain as many as two hundred fifty or more. Using these data, we have initially been focusing on four pragmatic research questions:

– How do participants collectively "translate" collections of problem *reports* into collectively bounded and actionable *problems*, and then into *resolutions*?

– How does a community recognize duplication, redundancy and interdependence among problems?

– Why do some problems persist for long periods while others get resolved quickly? Can this *time-to-resolution* be predicted on the basis of early-stage characteristics of the problem-resolution process?

– With a very large number of simultaneous open problems (e.g., 50,000+ in our Bugzilla snapshot), how does a community decide what to do next and how to do it?

Below we sketch our progress on several of these questions.

5.1. *Translating Problem Reports into Problems and Resolutions*

Figure 3 illustrates our current model of a community translating issue reports into resolved problems. Problem reports are texts that contain formal information, examples, and unstructured comments. Despite the fact that these texts are called "bugs" and are referred to using formal nametags of the form "Bug <number>", it is very clear from the discussion in the reports themselves that these reports are *localized descriptions of problems*. That is, they are partial and context-sensitive representations, and are not in themselves actual problems. We say this for reasons given above: a) reports are representations, not actual causes or phenomena; b) many reports sometimes refer to the same underlying problem; c) a single report may contain information on many problems; d) there are linkages and interdependencies across both problems and reports; and e) there is often considerable uncertainty and contention over the relationships between reports and problem(s) to which they refer. Hence the principal aim of the community is to construct, arrange, and manage the relationships between these dynamic concrete representations, and the virtual, abstract, shared, *immanent* entities they know as problems.

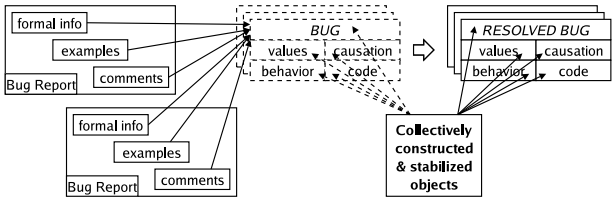

Figure 3. *Translation process: from problem reports, to reports, to resolutions.*

This first requires a model of what a problem is. Our initial model comprises four elements, as shown in Figure 3: behavior, values, causation, and code. *Behavior* refers the manifested phenomenon that the community believes constitutes or illustrates the problem. *Values* refers to a set of criteria the community uses to determine whether the behavior is problematic or not (e.g., some reports are considered not to reflect actual problems, others are enhancement requests, etc., and these determinations reflect some criteria of evaluation). *Causation* refers to the community's model of how code, infrastructure (such as the hardware running the code) and user behavior, in concert, lead to manifestations of the problematic behavior. *Code* refers to the actual software source code implicated by the causal model, as well as to the patched, repaired code meant to solve the problem.

We see a problem as a schema with these four elements. It is important to stress that this schema is virtual, distributed community knowledge. This knowledge is never explicitly represented, and the schema itself is not a first-class representational object in Bugzilla (or in any similar repository we have seen), though parts and traces of this knowledge are often written down in comment fields of specific component reports. When a problem is fully resolved, the community's knowledge includes some articulation of each element of this schema, or else contains some rationale for why that element is irrelevant.

At various times through the life-cycle of a problem, each of these four problem elements is potentially missing or contentious. Thus, the principal work comprising the DCPs of the community is to fill in the gaps and to resolve these contentions, settling on a collectively complete and accepted interpretation of the problem as constituted by its elements. If we view each of the elements as a collective object, we can see that the work is *stabilization work*: making these elements unchanging and non-contentious so the community can move beyond them and accept them as background facts. For example, in a number of bugs that require code fixes (the *Code* element), we see community members suggesting alternative fixes, arguing about their respective merits, modifying the fixes, and eventually settling on a specific code configuration. Once implemented and verified, the final fix becomes highly stabilized, to the point where no further change is made to any of the elements in the problem's schema. The complexity and duration of this stabilization work can vary greatly, from being trivial to being very complex or even endless.

5.2. *Recognizing Duplication, Redundancy and Interdependence*

Identifying duplicate problem reports is a critical part of the DCP of problem management. In our snapshot, out of 128,823 total bug reports, 49,765 (38.6%) are marked with some kind of duplication relationship. Thus, at the date when our snapshot was taken, a new problem report had a 38% prior probability of duplicating an existing report. Unrecognized duplication leads to redundant work, and we have found numerous instances in which duplication relationships were not recognized until late in the problem-resolution process, meaning that: a) much redundant work was expended by different participants before the duplication was recognized, and b) accumulating knowledge that could have been used by all was compartmentalized.

Defining duplication is a progressive and uncertain process; it is by no means inherently clear when two reports refer to the same issue. We conclude this for the following reasons. First, we already mentioned above the late-stage discovery of duplicates. Second, we know that human duplicate recognition is incomplete because we have used automated means to find duplicates not yet marked directly by people. Third, we believe that in a corpus of this size, full human discovery of all duplicates is mathematically implausible. Fourth, we have found that early bug reports are sometimes marked as duplicates of later reports—a kind of strange backward representation. Finally, our data also shows that duplicate-marking decisions are negotiated, as in this example:

88481, comments 1-3:

```
*** This bug has been marked as a duplicate of 69167 ***

This is _not_ a dupe!  The effects are very similar but the
underlying conditions are very different.  I was specifically
asked to file this separately.

Ok my bad reopening
```

The identification of interdependencies among different problems is similar. In our corpus, known interdependencies are formally represented with two directed relations: *depends-on* and *blocks*. For example here is dependency data from bug report 28586, with related but already-resolved bug reports shown in strike-through text:

```
Bug 28586 depends on:  39098 ~~156997~~ 157004 ~~157102~~ 157527 157531
159071 ~~160548~~ 161276 188795 189204
Bug 28586 blocks:  8345 61685 ~~74987~~ 84128 88810 ~~92997~~ 96887 ~~104166~~
121209 ~~126906~~ 154414 159324 ~~160423~~ 163993 181083
```

We have not yet quantitatively analyzed the specifics of this process. Nonetheless, it is clear that explicitly represented knowledge of interdependencies both reflects and contributes to the analysis of the causal properties of problems, and helps participants refine causal accounts—a central element of the translation process discussed above. Knowledge of interdependencies also helps participants to order and sequence their work, as shown by the following excerpt:

88810, comment 14:

```
I see the following ways of reducing this problem:
1) Fix bug 77675 (this is something everybody seems to agree on)
2) Fix bug 28586 "use error page, not dialog for inaccessible pages".
When that bug is fixed, most error messages would be displayed
in an existing window and nothing will have to be raised anywhere.
3) I would imagine that these two bugs do _not_ cover all the cases
where windows are raised unnecessary.  I believe that we should
try to document all of those (we can start in this bug and then
file additional bug reports as needed).  This way we can have a
meaningful discussion of concrete issues instead of just "Mozilla
raises windows to often" (which is something I completely agree
with, but it's not getting us anywhere).
```

Like duplicate identification, finding and reasoning about interdependencies is also a progressive and uncertain process, and the formal data on interdependencies represented in reports is often incomplete. For example, comment 38 of report 28586, whose formally-marked dependencies are shown above, notes that "bug 91632 was filed as a dependent on this one in order to...." However, despite this comment, bug 91632 does not appear in the dependencies list shown above. In fact, this dependency was asserted, then retracted, with much discussion in the comments of bug 91632. While the assertion was recorded, the retraction was buried in an activity log, and went unremarked in the dialogs. So, like duplication decisions, dependency decisions are also argued and negotiated, as shown in this comment:

89939, comment 16:

```
I don't think that this bug blocks bug 61521, because in that bug,
we can use nntp:  URIs with *hostnames* and msgids, which work
already (I think).
```

Dependency and duplication relationships among problems and reports are but two classes of relationships through which problem information is clustered and arranged. There are many other implicit relationships, including having common developers and/or reporters, common signifiers and language, common locations within source code, and so on. All of these provide analyzable axes for understanding how participants structure these DCPs.

5.3. *Persistence of Problems*

We would like to explain, predict, and, ultimately, to minimize the time it takes to move a problem from its initial reported state to its resolution. To study this issue comparatively, we first extracted all 88,000 resolved reports in our corpus, sorted them by report number, and computed their duration to final resolution. Figure 4 shows a histogram of these data, with durations on the horizontal axis (linear scale) and the frequency of reports taking that duration on the vertical axis (log scale). Clearly, most reports get resolved quickly, but there is a significant fraction that take considerable time.

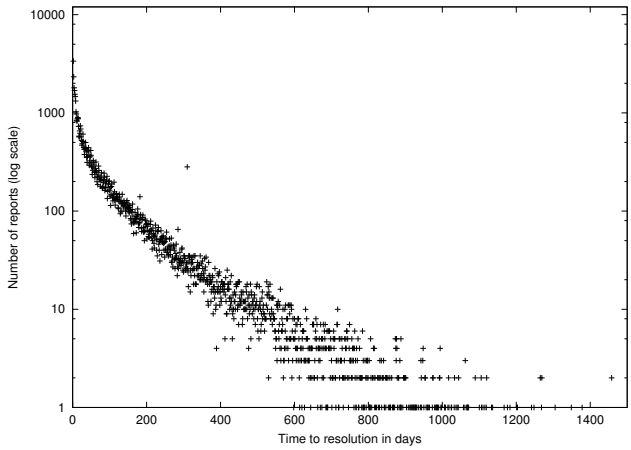

Figure 4. *Frequencies of bug reports given time-to-resolution.*

Our aim is to use information distilled during early stages of a problem's life to predict the duration, outcome, and specific trajectories of activities of its later life. Using machine learning techniques, we want to establish correlations between patterns in the "known conditions" and trajectories in the "conditions to be predicted", using these correlations to predict durations for new problems (see Figure 5).

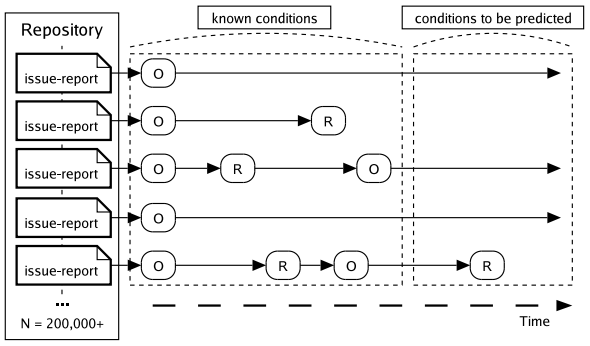

Figure 5. *Predicting time-to-resolution from initial conditions.*

In addition, some of our prior research identified mis-alignment in distributed activity networks as a predictor of difficulties in resolving problems [GAS 00]. To build on this, we comparatively analyzed two samples from the 88,000 resolved reports in the Mozilla corpus. One sample comprised all 59 bug reports with durations longer than 1000 days, and the other sample comprised 100 randomly sampled reports whose duration was exactly 30 days. These reports were all examined by hand and

coded roughly for information about what kinds of basic social processes (see Section 5.4) were occurring in the analysis and resolution activities. Preliminary analysis has shown more conflict, ambiguity, uncertainty in long resolution processes. This observation has led us to hypothesize that duration depends on the time it takes for the community to reach consensus [MUL 03] on each problem element discussed above, namely values, behavior, causes, and code. If time-to-consensus does seem to be a variance-explaining factor, then several other questions are of interest:

– What factors explain this *time-to-consensus*?
– Through what processes do people reach consensus?
– How do people initiate and develop objects around which to develop consensus?
– How do consensus and objects of consensus co-construct each other?
– What other "time-to-" dimensions could we consider in similar ways?
– Is consensus necessary for action? Is time to act a more critical issue?

5.4. *Understanding How a Community Selects a Course of Action*

As we demonstrated with the few issues discussed above, we are interested in understanding how participants in DCPs collectively choose and arrange their courses of action. This issue is of particular interest in situations such as F/OSS problem management, in which large numbers of problems are concurrently open and are candidates for being worked on.

The data collected in the Mozilla corpus lends itself remarkably well to such studies, as it contains longitudinal data on a large number of problem resolution processes. We can use these data to make both qualitative and statistical examinations of how processes unfold, triangulating the qualitative and statistical perspectives to refine the overall picture.

With over 128,000 bug reports and a total of more than 1.2 million comments contributed by over 45,000 registered users, a central question becomes how to reliably and scalably extract, annotate, and model the unfolding processes and their relationships. Clearly, detailed human-based qualitative analysis of individual problem reports will not possibly scale to data collections as large as those we have assembled, and we are investigating automated techniques that will complement—or "amplify"—these manual analyses. In the following paragraphs we briefly present the framework we have developed for this purpose.

5.4.1. *Process Data Extraction*

The first step in our framework consists in extracting from the corpus some process data that will be used to build exploratory models. Some aspects of processes are already formally represented in the corpus. For example, the trajectory of a given bug report through different types of "Status" and "Resolution" over time are recorded in the Bugzilla database. Other process aspects have to be filtered out from time-tagged

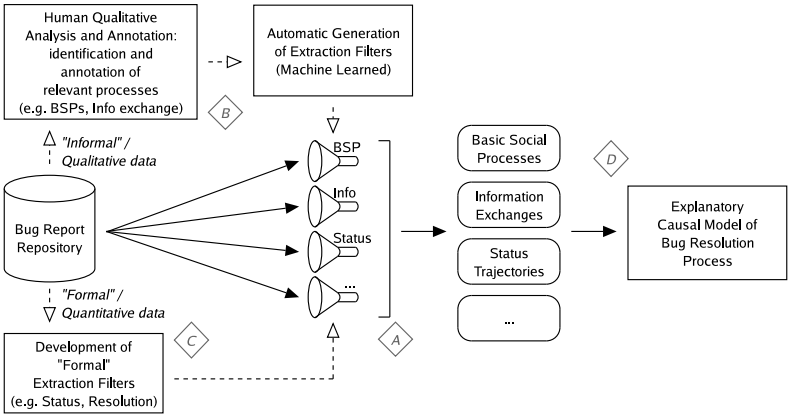

Figure 6. *Our "computational amplification" framework.*

"informal" data[3]. For example, such "informal" process aspects might include the kinds of information used, the kinds of conflict and negotiations that appear, and the various collective activities that can be observed in the data.

We rely on the notion of *filters* as means of extracting specific processes from the data corpus (see Figure 6.A). Consequently, we will have filters that will extract processes such as the "Status" and "Resolution" trajectories from a bug report, and other filters for more informal process aspects. While some of these filters will be as straightforward as getting a list of state changes for a formally represented characteristic of a bug report (see Figure 6.B), others will require more elaborate techniques primarily dealing with the textual parts of bug reports (see Figure 6.C). We have initiated work in this direction in order to identify instances of "Basic Social Processes" (BSPs) that are represented in the comments of a bug report. Examples of BSPs we are currently investigating include *evaluation*, *conflict*, *negotiation*, *articulation*, *description*, *building rationale*, etc. We are using language processing and information extraction techniques to discover linguistic signatures for these BSPs in the comments, in order to automatically extract candidate instances from the entire corpus. The automated learning techniques rely initially on seed samples that have been manually annotated with these specific concepts using traditional qualitative methods. After automated extraction trials, the results are checked by human annotators and possibly refined. Extraction results also provide additional proposed instances of concepts that can help refine and extend our original understanding of them (and therefore improve the way they are annotated).

3. By informal data we mean information that has not been formalized in the bug report database with standardized fields and vocabularies. This is independent of the actual formality of the process itself.

Using these various filters to extract specific process aspects, we can then build: 1) complete process models from each individual process aspect, and 2) models that integrate multiple process aspects and express the relationships between them.

5.4.2. *Process Modeling*

Process modeling can provide a foundation for understanding bug repair processes, and for linking bug repair process decisions to process structures and outcomes. Using techniques such as finite state machines and Markov models [COO 95], we are able to infer statistical models of the various processes we extract from our corpus.

For example, we have generated a probabilistic bug resolution process model based on the "Status" and "Resolution" process trajectories of over 88,000 resolved problem reports. Figure 7 visualizes this probabilistic representation of the model (the figure 7 only shows "Status" for the sake of readability). Each node represents one status in which a bug report can be, and the edges indicate the probability of a transition from a given status to the next, as represented in the data. The probabilistic nature of these networks provides useful heuristics for suggesting what underlying data should be examined for concepts that explain the differentials in these probabilities, and so provides a sampling frame.

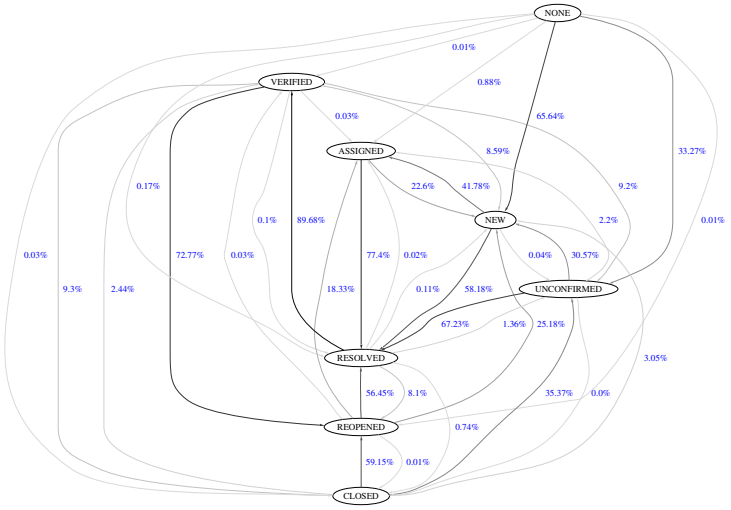

Figure 7. *Visualization of the "bug resolution" process model ("Status" only).*

Currently we are building models that integrate more aspects of the bug report resolution process. We are first focusing on integrating social process aspects (BSPs). We believe that such a model will allow for a better analysis of how community choices affect the trajectories of problem resolution, and more specifically the time it takes to

solve a problem (see the preliminary results of this approach presented in Section 5.3 above).

More generally, the ability to integrate different process aspects in a single causal model (see Figure 6.D) provides a flexible framework to explore the potential relationships between specific process aspects and specific outcome variables of interest.

6. Conclusions

In this paper we have discussed our approach to investigating large-scale, distributed decision-making and activity viewed under a new framework called *Distributed Collective Practices*. We have reported on the current state of our investigations into basic issues of DCPs, in the context of software problem management in Free/Open Source Software development communities. Our investigations into representation, problem duplication and interdependency, and scalable models of community action selection and decision-making have begun to show interesting features of these phenomena. As we and others make progress on new, scalable methods such as statistical process modeling and Computational Amplification, we hope to unlock many more insights into Distributed Collective Practices.

Acknowledgements

We gratefully acknowledge the collaboration of our colleagues Dave Dubin, Bryan Penne, and Bob Sandusky of UIUC, Walt Scacchi of U.C. Irvine, and Jean-Paul Sansonnet and Bill Turner of LIMSI-CNRS. This material is based upon work supported by the National Science Foundation ITR (Digital Society and Technologies) Program under Grant No. 0205346. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. References

[BRO 02] BROWN A. W., BOOCH G., "Reusing Open-Source Software and Practices: The Impact of Open-Source on Commercial Vendors", GACEK C., Ed., *7th International Conference on Software Reuse*, SpringerVerlag, 2002, p. 123-136.

[CAL 98] CALLON M., Ed., *The Laws of the Markets*, Blackwell Publishers, 1998.

[COO 95] COOK J. E., WOLF A. L., "Automating Process Discovery Through Event-Data Analysis", *17th International Conference on Software Engineering*, Seattle, WA, USA, 1995, p. 73-82.

[CZA 92] CZARNIAWSKA-JOERGES, *Exploring Complex Organizations: A Cultural Perspective*, Sage Publications, 1992.

[GAS 86] GASSER L., "The Integration of Computing and Routine Work", *ACM Transactions on Information Systems*, vol. 4, num. 3, 1986, p. 205-225.

[GAS 00] GASSER L., "The Social Organization of Errors in Computing Work", Working Paper LG-2000-04, Graduate School of Library and Information Science, University of Illinois at Urbana-Champaign, 2000.

[MOC 00] MOCKUS A., FIELDING R., HERBSLEB J., "A Case Study of Open Source Development: The Apache Server", *22nd International Conference on Software Engineering*, Limerick, Ireland, June 2000, p. 263-272.

[MOC 02] MOCKUS A., J. H., "Why Not Improve Coordination in Distributed Software Development by Stealing Good Ideas from Open Source?", *2nd Workshop on Open Source Software Engineering*, Orlando, FL, USA, May 2002.

[MUL 03] MULLER J.-P., Private Communication, May 2003.

[SCA 01] SCACCHI W., "Process Models in Software Engineering", MARCINIAK J. J., Ed., *Encyclopedia of Software Engineering*, John Wiley and Sons, New York, 2nd edition, December 2001.