Empirical Software Engineering at Microsoft Research

http://research.microsoft.com/en-us/projects/esm/

Christian Bird cbird@microsoft.com
Brendan Murphy bmurphy@microsoft.com
Nachiappan Nagappan nachin@microsoft.com
Thomas Zimmermann tzimmer@microsoft.com

Microsoft Research, Redmond, USA and Cambridge, UK

(Authors are in alphabetical order.)

ABSTRACT

We describe the activities of the Empirical Software Engineering (ESE) group at Microsoft Research. We highlight our research themes and activities using examples from our research on socio technical congruence, bug reporting and triaging, and data-driven software engineering to illustrate our relationship to the CSCW community. We highlight our unique ability to leverage industrial data and developers and the ability to make near term impact on Microsoft via the results of our studies. We also present the collaborations our group has with academic researchers.

Author Keywords

Software engineering, Socio technical congruence, bug tracking and triaging, data-driven software engineering

ACM Classification Keywords

D.2.5 [Software Engineering]: Testing and Debugging; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

ACM General Terms

Human Factors, Management, Measurement

INTRODUCTION

The Empirical Software Engineering (ESE) group at Microsoft Research focuses on working in the intersection of the Software Engineering and CSCW communities.

> "Over the last decade, it has become clear that empirical studies are a fundamental component of software engineering research and practice: Software development practices and technologies must be investigated by empirical means in order to be understood, evaluated, and deployed in proper contexts. This stems from the observation that higher software quality and productivity have more chances to be achieved if well-understood, tested practices and technologies are introduced in software development. Empirical studies usually involve the collection and analysis of data and experience that can be used to characterize, evaluate and reveal relationships between software development deliverables, practices, and technologies."
>
> (Empirical Software Engineering journal, http://www.springer.com/computer/swe/journal/10664)

CSCW 2011, March 19–23, 2011, Hangzhou, China.

At a high level the goals of the ESE follows two guiding principles,

- **Empower** software development teams
- To **gain insight** from **product process, people and customers**

by employing a qualitative and quantitative approach to the software development process.

In this paper we discuss three broad themes of the ESE group,

- Socio technical congruence;
- Bug reporting and triaging; and
- Data-driven software engineering.

In each of these sections our studies leverage techniques and methods from both the Software Engineering and CSCW communities to adapt case studies in practice from the empirical domain with the CSCW aspects as all software systems which are built by teams inherently have a significant collaborative aspect. We also present our collaborations and discuss the uniqueness of our fit in the middle of these two communities.

SOCIO TECHNICAL CONGRUENCE

"Design and programming are human activities; forget that and all is lost"

– Bjarne Stroustrop

As software projects grow in size and complexity, so do the teams of engineers that develop and maintain them. Brooks, in his seminal work, "The Mythical Man Month" [1] discussed coordination as one of the key problems of running a software project with many developers. The coordination effort required to help each member of a team stay in sync and keep a project on schedule is enormous.

Socio Technical Congruence is a term that has emerged recently in the software engineering literature to describe the relationship between the "social" side of development, meaning the developers, their relationships to each other, how they communicate, work together on software, etc. and the "technical" side which encapsulates features of the software itself such as dependencies between components, component complexity, and software quality. The idea behind the term has its origins in Conway's Law, originally presented in Conway's paper "How Do Committees Invent" [2]. This is of importance to the CSCW community because

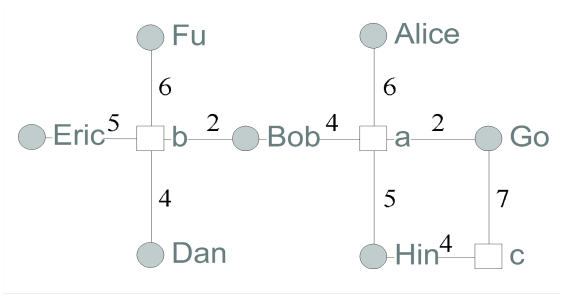

Figure 1. A developer-module network characterizes the contributions of developers within a system.

of the emphasis placed on understanding how, and under what circumstances, developers should work together on projects.

In an effort to aid the development effort at Microsoft and understand the effect of human factors, we have gathered data and investigated the relationship of software quality with developer attributes such as collaboration behavior, geographic location, position within the organization, and work assignment. Below, we describe some of our key results.

Contribution Behavior and Quality

Is a study of collaboration behavior, Nachi, in joint work with Martin Pinzger, developed a developer-module network, which characterized the contributions of developers to modules within a system [3]. Figure 1 shows an example developer-module network. Gray circles represent developers and boxes represent modules within a system. Edges connect developers to the modules that they have contributed to, with edge weights representing the number of source code repository commits. Note that the developer-module network for Windows Vista is quite large, with thousands of developers and thousands of *binaries* – executables (.exe), shared libraries (.dll) and drivers (.sys).

We found that topological properties of this network were highly related to post-release faults. For instance, modules that were more central, as defined by traditional social network analysis centrality measures, tended to have more faults than other modules. We also found that less complex measures, such as the number of distinct authors and number of distinct commits were both significant predictors for the probability of post-release failures. By using a host of social network measures (we refer the reader to the original paper for details and descriptions) in conjunction with principal component analysis, we were able to train a logistic regression module for predicting failure-prone modules that achieved an average precision of 83% and recall of 89%. We summarize our important results:

- Network centrality measures can predict failure-prone binaries in Windows Vista.
- Network centrality measures can predict the *number* of post-release failures.

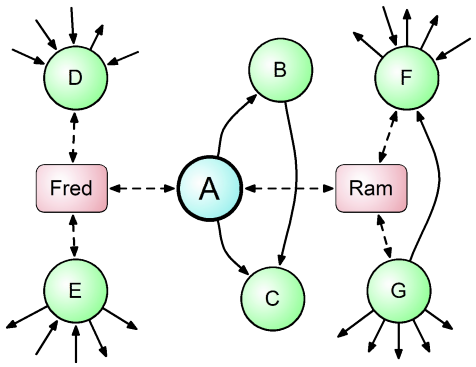

Figure 2. A socio-technical network between modules (circles) and developers (boxes).

- Advanced centrality measures can improve the prediction of number of post-release failures.

In summary, we found that a strong relationship exists between the developers' commit behavior and the software quality of modules within the system.

Adding Technical Relationships

In later work, Christian and Nachi built upon this result by adding module dependencies to the developer-module network [4]. In previous work, Tom and Nachi found that dependencies can predict failures in both modules [5] and subsystems [6]. A network that incorporates both developer contributions and dependencies is a socio-technical network because edges may represent contributions from people to modules or dependencies between modules within a system. Figure 2 depicts a portion of a socio-technical network. Circles represent modules and boxes are developers. Solid directed edges are dependencies, indicating that one module may use functions or types defined in another, may make RPC calls to another, etc. Dashed lines indicate that a developer contributed to a module (we used weights in our analysis, but do not depict them in the figure).

By combining both types of relationships into one graph, we were able to increase the power of fault predicting regression models. Using principal component analysis, we found that models based on this network had higher recall than networks with only contribution edges (developer-module networks) or only dependency edges to a statistically significant degree in Vista (recall was similar to previous models).

To see if such models are specific to Microsoft or if they are more generally applicable, we also applied the same techniques to 6 major releases of the Eclipse Java IDE (2.0 through 3.3) and achieved precision and recall rates of failure-prone plug-ins ranging from 75% to 86%. Further, we were able to train a regression model on one release of Eclipse and achieve recall and precision values ranging from 75% to 93% on the next release, showing that cross release prediction works well for network based regression models.

The key contributions of this work were:

- We found that using technical and contribution relationships together have more power than either in isolation for predicting bugs.
- We showed that such techniques are general by using them on projects that differ in size, domain, and process (commercial vs. open source).
- We demonstrated how such techniques can be used in practice to accurately predict failure prone modules in one release using data from a prior release.

In all of these models, the inclusion of developer behavior significantly improved the results over models that did not. Clearly, the human side of software engineering has a profound effect on quality.

Does Organizational Structure Affect Bugs?

One of the unique advantages of working within an organization like Microsoft is that we have access to types of data that may not be available in academia. One such form of data is the organizational structure of the teams that develop the software.

Brooks stated that product quality is strongly affected by organization structure [1]. In order to empirically evaluate this claim, Nachi and a visiting researcher, Vic Basili, developed a suite of metrics to quantify organizational complexity [7] and investigated the relationship of these measures with software quality that we summarize below. The term "owning organization" is used to denote the organization that owns the binary.

- Number of engineers that worked on a binary
- Number of engineers who worked on a binary and left the organization prior to release
- Total number of contributions to a binary
- Number of levels up the organization required to reach the person who oversees the engineers making at least 75% of the contributions to a binary
- Proportion of engineers in the owning organization who contributed to a binary
- Proportion of edits to a binary that were made by the owning organization
- Ratio of proportion of engineers reporting to the owning manager relative to the total number of engineers editing a binary
- Number of different organizations that contribute at least 10% of the edits to a binary

Each of these measures is based on a hypothesis related to software quality. For instance, a large loss of team members (2nd measure) affects knowledge retention and thus quality. The more cohesive the contributors are (organizationally, 5th measure) the higher the quality.

We gathered these metrics for Windows Vista and correlated each with post-release faults in the first six months. We also evaluated the accuracy of a predictive model based on these metrics. Our results indicate that all eight measures are important because a step-wise regression retained every measure. We also created a predictor based on principal component analysis (due to high correlation between some measures) and compared it to prior approaches that included attributes such as code churn, code complexity, dependencies, code coverage during testing, and pre-release bugs. Surprisingly, the model based purely on organizational metrics performed better, in terms of precision and recall, than all of these models to a statistically significant degree.

We were able to build a better predictor using attributes of the organization that developed the software instead of using attributes of the software itself. This finding highlights the importance of coordination and collaboration in software development, as it implies that perhaps high levels of coordination are able maintain code quality in the face of factors known to result in faults such as higher levels of complexity or code churn.

Vista is a large project, in terms of code and developers. In an attempt to determine how large a project needs to be before these organizational measures begin to have an effect, we replicated our study on a smaller data set and found that a team size of 30 engineers and three levels of organizational depth should be sufficient for a model to predict failure-proneness.

Investigating the Effects of Geographic Distribution

An additional form of information that we have related to software development is the geographic location of all developers. This enabled us to address an issue that many have wondered about and that may have consequences for Microsoft's development process, "Does distributed development affect software quality?"

In 2009, Chris and Nachi investigated this question by examining the locations of the developers that worked on each binary that shipped with Windows Vista [8]. We grouped binaries into 6 categories depending on how spread out the developers were that contributed to them. Some binaries were developed mostly by developers in the same building while others had a team that spanned multiple countries.

When we compared the defect rates for the different groups, we found that no group had more than 16% more defects than binaries developed by engineers in the same building. While this is not a trivial increase, we had expected the effect of geographic distribution to be much larger due to the barriers imposed such as lack of familiarity, time zone issues, and less-rich communication. Following a similar type of analysis to that of Herbsleb and Mockus [9], we examined the effects of distribution when controlling for team size. There was very little difference in failures, 6% at most, between distributed and collocated binaries.

We also examined attributes of the binaries in each group in order to determine if, for instance, managers only distributed binaries that were smaller, less critical to the system, or made up for distribution by testing more. In all, we exam-

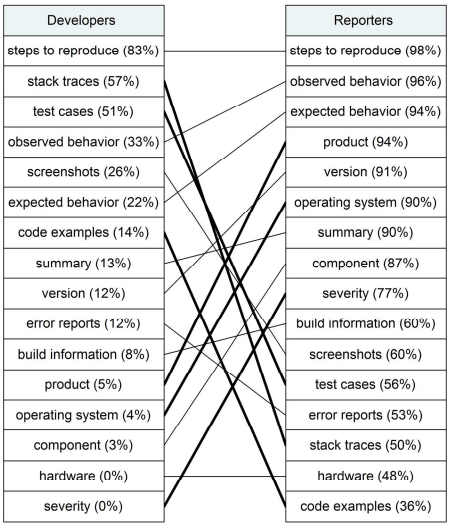

Figure 3. Information in bug reports that is considered most helpful by developers vs. information provided by reporters.

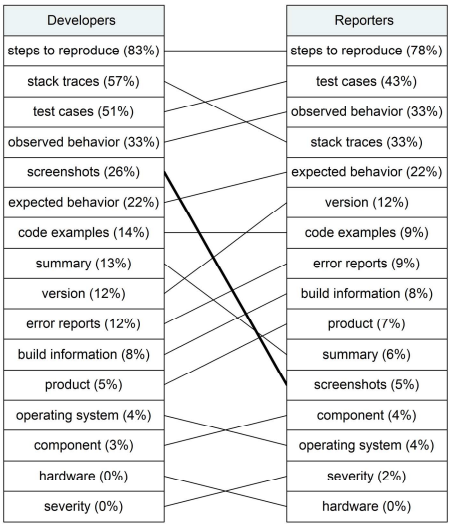

Figure 4. Information in bug reports that is considered most helpful by developers vs. what reporters believe is important.

ined over 50 measures in categories such as complexity, churn, test coverage, dependencies, and organizational metrics, and determined that there is very little difference between distributed and collocated binaries other than team size. Thus, it appears that within Microsoft, distributed development doesn't negatively affect quality. There are a number of reasons that we believe this may be (and we invite the reader to examine them in the original paper [8]), but we have yet to empirically verify them.

This result is all the more surprising in light of the findings of our study on organizational metrics, as it may seem to be at odds with those findings. The resolution of it lies in the fact that organizational structure spans geography at low levels within Microsoft. While some companies have an Asian organization, a European organization, etc., within Microsoft, it is not uncommon to have a team with developers in India and others in Beijing who report to a second line manager in Redmond. This approach *may* be one reason that geography has less of an effect, but we plan to study this further to provide more conclusive evidence.

BUG REPORTING AND TRIAGING

In the past years, Tom Zimmermann has collaborated with other researchers on studies on bug tracking systems. The advantage of these collaborations is that academic researchers can analyze open-source projects, while we can analyze projects at Microsoft. Thus findings come with a higher generality.

Bug reports are a perfect data source for CSCW research. They capture collaboration, communication, and coordination among people. Especially in open source projects, bug tracking systems directly involve the users of a software and not just the engineers. This leads to communities of several thousand people who discuss and work towards a resolution of software bugs. In our research we studied bug tracking in open source and a closed source environments..

What Makes a Good Bug Report?

Tom, in joint work with Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss, conducted a survey among developers and users of the Apache, Eclipse, and Mozilla projects [10]. The 466 responses revealed several interesting findings on how to improve bug tracking systems.

First, we observed a mismatch between the information considered most useful by developers and the information provided by reporters (see Figure 3). Developers want steps to reproduce, stack traces, and test cases in bug reports; however, this is not the information that reporters provide. Yet, when asked, the reporters' responses indicated that they know what is most helpful to developers and the rankings matched almost perfectly (see Figure 4). There are two implications for bug tracking systems: (1) Tell users while they are reporting a bug what information is important. (2) At the same time, systems should provide better tools to collect important information automatically, because often this information is difficult to obtain for users.

Next, we analyzed the comments by the survey respondents to identify additional design recommendations:

- *Support different levels of users (novice, expert*) and provide different user interfaces for each level. Inexperienced users should receive more guidance when reporting bugs.
- *Integrate bug report reputation.* Several developers pointed out that reporters who are well known, either personally or through well-written past bug reports, will get more attention. Experienced reporters could be marked in their user profiles.

- *Provide a powerful, yet simple and easy-to-use search feature.* Several respondents to our survey complaint about the limited search functionality, which is often only basic keyword search.

For a complete list of recommendations, we refer to our main publication on this work [10].

Reassignment of Bug Reports

Many people collaborate on fixing bugs and bug reports are often reassigned to other developers. Together with Gaeul Jeong and Sung Kim, Tom proposed *bug tossing graphs* [11] to capture frequent reassignment patterns. In these graphs, nodes represent developers and weighted edges represent the number of reassignments between two developers. On two large open-source projects, we showed that bug tossing graphs combined with Markov chains can reduce the number of reassignments substantially (also known as ticket routing problem [12]).

However, not all bug reassignments are necessarily bad. Sometimes reassignments are actually needed to locate the root cause for a bug and to find the right person who can fix the bug. Such "beneficial" reassignments can increase the chances of a bug report getting fixed (see next subsection). We are currently working on a characterization of bug report reassignments to identify potential improvements for bug tracking systems.

Characterizing which Bugs Get Fixed

Often, the cost or risk of fixing a bug can be too high, or the impact of a bug report can be too low (only few people affected, easy workaround). Thus not all bug reports get fixed in real software development. In joint work, with Philip Guo, we characterized which bugs get fixed in Windows Vista and Windows 7 [13]. We made several observations related to how people collaborate and coordinate:

- People who have been more successful in getting their submitted bugs fixed are more likely to get their bugs fixed in the future.
- Reassignments are not always detrimental to bug-fix likelihood; several might be needed to find the optimal bug fixer.
- Bugs assigned across teams or locations are less likely to get fixed, due to less communication and lowered trust.

Collaboration and Information Needs in Bug Reports

Especially in open source, bug tracking systems play a central role in supporting collaboration between the developers and the users of the software. To better understand this collaboration, we quantitatively and qualitatively analyzed the questions asked in a sample of 600 bug reports from the MOZILLA and ECLIPSE projects (joint work with Silvia Breu, Rahul Premraj, and Jonathan Sillito) [14].

We categorized the questions into a catalogue of frequently asked questions (eight categories, 40 subcategories) and then analyzed response rates and times by category and project. Key findings of this study include:

- *Empirical analysis of response rate and time.* Out of all questions, 67.66% were responded to. Of the questions with responses, 79.4% received responses within the day.
- *Evolving information needs.* We learned that the kind of questions and thus the information needs change over a bug's life cycle.
- *Community-based bug tracking.* Bug reporting and tracking should be understood as a social activity within a community, supported by the bug tracking system.

Our results showed that the *role of users goes beyond simply reporting bugs:* their active and ongoing participation is important for making progress on the bugs they report. Based on the results, we suggested four ways in which bug tracking systems can be improved (see the main publication on this work [14]).

DATA-DRIVEN SOFTWARE ENGINEERING

A significant proportion of empirical research is done via case studies which collect and analyze data from software artifacts and the associated processes and variables to quantify, characterize and explore the relationship between different variables to deliver high quality secure software on time and within budget. *Data-Driven Software Engineering* forms a crucial part of empirical software engineering as it can be used to understand the successful development of software systems. Nachi Nagappan and Brendan Murphy were some of the first at Microsoft to begin collecting and analyzing software engineering artifact data for this purpose.

In this section we will explain three of our projects at a very high level that involve data driven software engineering. They range from software product to software practice issues. The three projects are,

1. **Software product** – Failure-prediction/Risk analysis: Using software development data obtained during the development process to predict failures and identify the best predictors.
2. **Software practice** – Does test-driven development work? If so is there any supporting data for teams to make decisions to use test-driven development.
3. **Software practice** – Is there data available on how effective Unit testing is? What is the cost associated with unit testing and do developers offer a resistance to unit testing.

Failure-Prediction/Risk Analysis

An important application of data-driven software engineering is in the field of failure-prediction. Failure prediction can be used to understand the overall success of the development process and plan for maintenance activities. Software organizations can benefit greatly from an early estimation regarding the quality of their product. Because product quality information is available late in the process, corrective actions tend to be expensive [15].

During the development cycle different metrics can be collected that can be related to product quality. The goal is to use such metrics to make estimates of post-release failures early in the software development cycle, during the implementation and testing phases. Such estimates can for example help focus testing, code and design reviews and affordably guide corrective actions. Across a span of several years, Nachi and Brendan (in collaboration with others) have used different metrics for failure prediction: code coverage [16]; code churn [17]; code complexity [18]; code dependencies [19]; people and organizational metrics [7].

Based on the results from using these various metrics either individually or in as a composite model effective failure prediction models have been built and is used in a wide variety of products at Microsoft. These failure-prediction models are built as services which allow engineers to predict risk; identify other engineers who share dependencies with their code which might be affected by changes; prioritize testing; identify ownership to have the best person fix bugs and plan for staffing up for maintenance activities.

Test-Driven Development

Test-driven development (TDD) [20] is an "opportunistic" software development practice that has been used sporadically for decades. With this practice, a software engineer cycles minute-by-minute between writing failing unit tests and writing implementation code to pass those tests. Test-driven development has recently re-emerged as a critical enabling practice of agile software development methodologies [21], in particular Extreme Programming (XP) [22]. However, little empirical evidence supports or refutes the utility of this practice in an industrial context.

For this purpose, Nachi collected and analyzed [23] data from three different teams at Microsoft (in Windows, MSN and Visual Studio) to build up an empirical body of knowledge on the efficacy of TDD. This has enabled teams to decide on the utility of TDD as a development practice. Further, by documenting the contextual information about the human factors about the engineers involved (their experience, programming expertise, whether collocated or distributed) team can make a data-driven decision on their move to following a TDD for software development.

Software Unit Testing

Unit testing is the testing of individual hardware or software units or groups of related units (IEEE [24]) and has been widely used in commercial software development for decades. But academic research has produced little empirical evidence via a large scale industrial case study on the experiences, costs, and benefits of unit testing. Does automated unit testing produce higher quality code?

To help other teams make a data-driven decision, Nachi, Laurie Williams, and Gunnar Kudrjavets observed [25] one large Microsoft team consisting of 32 developers transitioned from ad hoc and individualized unit testing practices to the utilization of the NUnit automated unit testing framework by all members of the team. We quantified the quality and effort required to transition from the ad-hoc testing to a more formal unit testing process. Also to further quantify developer perceptions we conducted a survey and interviews with the team to determine the tradeoffs of doing unit testing. These results can help other teams decide on the cost and overhead to transition towards a more formal unit testing process.

In general the three projects in the data –drive software engineering domain are more focused towards the empirical data analysis with making the results accessible to engineers via tools, techniques and processes.

Analytics for Software Development

The previous subsection presented studies where the ESE group collaborated with product teams at Microsoft. Our future work will focus on making data-driven software engineering accessible to a wider audience of engineers and managers.

We plan to *build tools that allow an easy access to data to simplify data-driven decision making***.** For example, existing development environments such Microsoft's Team Foundation Server and IBM's Jazz provide dashboards to inform engineers of the status of various events. However, while showing status and indicators is fairly straightforward, it is unclear what are the most important factors are for development teams to make data-driven decisions. What do we need to surface so that development data becomes actionable for teams so that they can improve how they work together?

Furthermore, we plan to evangelize empirical methods in software development and will provide analytics tools to empower development teams to run studies that go beyond the use of simple dashboards. In particular, we foresee the role of a *software development analyst* who combines the expertise in collecting and analyzing data with the knowledge of processes specific to the product team. Right now, this expertise is often split across Microsoft Research (who have the analytics knowledge) and product teams (who have the domain knowledge).

WHAT MAKES EMPIRICAL SOFTWARE ENGINEERING RESEARCH AT MICROSOFT UNIQUE?

An industrial research lab such as Microsoft Research has several advantages to conduct research.

Easy access to industrial data. During software development a large amount of data is generated and recorded in software repositories. Being inside Microsoft simplifies the access to such data and enables empirical studies as the ones presented in this paper.

Easy access to developers. Not only is the access to data easier, but also the access to engineers. This allows validation of empirical findings, user studies of prototypes, interviews, surveys, etc. and makes an ideal environment to study collaboration in software development.

Near term impact. Since Microsoft's core business is developing software, findings that result from our studies can

Main locations of the ESM group (black pins): Redmond (USA), Cambridge (UK)

Collaborations with other Microsoft Research Labs: Microsoft Research India (Bangalore), Microsoft Research Asia (Beijing), European Microsoft Innovation Center (Aachen, Germany)

Interns 2007-2010: University of Virginia (Ray Buse); University of California, Santa Cruz (Ken Hullett); National Institute of Technology, Tiruchirappalli, India (Kalaikumaran Ramamurthy); Stanford University (Philip Guo); Boğaziçi University, Turkey (Ayse Tosun); Darmstadt University of Technology (Andreas Johansson); North Carolina State University (Lucas Layman, Meiyappan Nagappan)

Visitors 2007-2010: Hong Kong University of Science and Technology (Sung Kim); University of Zurich (Harald Gall, Martin Pinzger); Saarland University (Andreas Zeller); North Carolina State University (Laurie Williams); University of Maryland (Victor Basili); Darmstadt University of Technology (Neeraj Suri).

Figure 5. Collaborations of the Empirical Software Engineering and Measurement (ESM) Group at Microsoft Research.

be put into practice immediately within the company. This serves to validate the findings and results in higher levels of quality and productivity.

Collaboration with other Microsoft researchers. There are plenty opportunities to collaborate with other researchers inside Microsoft. At the moment Microsoft Research has more than 800 researchers, working in eight locations around the world. For most areas, experts are easily accessible and allow for multidisciplinary research when needed.

Collaboration with external researchers. There are many opportunities for our group to collaborate with researchers outside Microsoft. Often we conduct research in tandem: we analyze data from Microsoft projects, while an academic researcher analyzes open-source projects. This increases the generality of our empirical findings. Selected academic researchers also get the opportunity to access to Microsoft data either as interns (typically PhD students) or visiting researchers (for example, professors during a sabbatical).

Figure 5 shows a Bing map with the locations of the Empirical Software Engineering Group and the collaborator's locations. In the past years we have worked with 8 interns and 7 professors from all over the world. We are always looking for outstanding visitors and interns**. To learn more about visits or internships visit our web-site and/or contact one of the authors of this paper.** In fact, three ESE group members interned before they joined Microsoft full-time (Nachi Nagappan in 2005, Tom Zimmermann in 2006, and Christian Bird in 2008 and 2009)

CONCLUSION

In this paper, we presented three main themes that showcase the research of the ESE group at Microsoft: the analysis of *socio technical congruence* and *bug tracking systems* allows us to understand how development teams collaborate and to build tools that help them collaborate with each other. With *data-driven software engineering*, we want to take this one step further: development teams should have the knowledge and tools to analyze their collaboration patterns themselves and monitor their improvement over time.

Being located inside Microsoft offers us with many unique opportunities to pursue our goals. Microsoft has many large software projects and the "cooperative" aspect is omnipresent. Rather than just coming in after the fact and investigating (like many studies in the mining software repositories field do), we can watch software development while it happens. We can also test tools related to helping to support cooperative work and understand when cooperation is needed and when engineers can work on their own.

For more information about the ESE group at Microsoft and/or to apply for an internship, logon to

http://research.microsoft.com/en-us/projects/esm/

ACKNOWLEDGMENTS

We thank Tom Ball, Robin Moeur, Wolfram Schulte; our *visitors* Sung Kim (2010), Harald Gall (2008, 2009), Laurie Williams (2009), Andreas Zeller (2005, 2009), Victor R. Basili (2007), Neeraj Suri (2007), Martin Pinzger (2007); our *interns* Ray Buse, Ken Hullett, Meiyappan Nagappan Kalaikumaran Ramamurthy (all 2010), Philip Guo, Ayse Tosun (both 2009), Lucas Layman, Andreas Johansson (both 2007); and we thank our *collaborators*. Thanks for the great work!

We also thank our colleagues at Microsoft Research and the many product groups at Microsoft who have worked with us and helped with studies. You rock!

REFERENCES

1. Brooks Jr., F.P. *The mythical man-month*. Addison-Wesley, 1975.

2. Conway, M.E. How do committees invent. *Datamation*, 14, 4 (1968), 28-31.

3. Pinzger, M., Nagappan, N., and Murphy, B. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2008), 2-12.

4. Bird, C., Nagappan, N., Devanbu, P., Gall, H., and Murphy, B. Putting it All Together: Using Socio-Technical Networks to Predict Failures. In *Proceedings of the 17th International Symposium on Software Reliability Engineering* (2009), 109-119.

5. Zimmermann, T. and Nagappan, N. Predicting Defects using Social Network Analysis on Dependency Graphs. In *Proceedings of the 30th International Conference on Software Engineering* (2008), 531-540.

6. Zimmermann, T. and Nagappan, N. Predicting subsystem failures using dependency graph complexities. In *Predicting subsystem failures using dependency graph complexities* (2007), 227-236.

7. Nagappan, N., Murphy, B., and Basili, V. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th International Conference on Software Engineering* (2008), 521-530.

8. Bird, C., Nagappan, N., Devanbu, P., Gall, H., and Murphy, B. Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista. In *Proceedings of the International Conference on Software Engineering* (2009), 518-528.

9. Herbsleb, J.D. and Mockus, A. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29, 6 (2003), 481 - 494.

10. Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schröter, A., and Weiss, C. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering*. To appear. http://doi.ieeecomputersociety.org/10.1109/TSE.2010.63.

11. Jeong, G., Kim, S., and Zimmermann, T. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2009), 111-120.

12. Shao, Q., Chen, Y., Tao, S., Yan, X., and Anerousis, N. Efficient ticket routing by resolution sequence mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2008), 605-613.

13. Guo, P.J., Zimmermann, T., Nagappan, N., and Murphy, B. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (2010), 495-504.

14. Breu, S., Premraj, R., Sillito, J., and Zimmermann, T. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work* (2010), 301-310.

15. Boehm, B.W. *Software Engineering Economics*. Prentice Hall, 1981.

16. Mockus, A., Nagappan, N., and Dinh-Trong, T.T. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement* (2009), 291-301.

17. Nagappan, N. and Ball, T. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering* (2005), 284-292.

18. Bhat, T. and Nagappan, N. Building Scalable Failure-proneness Models Using Complexity Metrics for Large Scale Software Systems. In *Proc. of the Asia Pacific Software Engineering Conference* (2006), 361-366.

19. Nagappan, N. and Ball, T. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement* (2007), 364-373.

20. Beck, K. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

21. Cockburn, A. *Agile Software Development*. Addison-Wesley Professional, 2001.

22. Beck, K. and Andres, C. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004.

23. Bhat, T. and Nagappan, N. Evaluating the efficacy of test-driven development: industrial case studies. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering* (2006), 356-363.

24. IEEE. *IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology*. , 1990.

25. Williams, L., Kudrjavets, G., and Nagappan, N. On the Effectiveness of Unit Test Automation at Microsoft. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering* (2009).