Indirect Conflicts: An Exploration and Discussion of Tools, Process, and Developer Insight

Jordan Ell and Daniela Damian
University of Victoria, Victoria, Canada
jell@uvic.ca, danielad@csc.uvic.ca

***Abstract*—Awareness techniques have been proposed and studied to aid in developer understanding, efficiency, and quality of software produced. Some of these techniques have focused on either *direct* or *indirect conflicts* in regards to prevent, catching, or debugging these conflicts as they arise through source code changes. While the techniques and tools for direct conflicts have had large success, tools either proposed or studied for indirect conflicts have had many common struggles and little developer interest. To better understand common indirect conflict tool issues, we performed a grounded theory study with 97 developers involved in either interviews or a web survey. We found that: indirect conflicts are a real and frequent problem, developers already posses a large suite of tools and processes which can prevent and catch indirect conflicts at a manageable scale, and that developers have a work until something breaks philosophy which leads to the lesser want of prevention tool as to the want of quick and perhaps automatic debugging tools for indirect conflicts.**

I. Introduction

As Software Configuration Management (SCM) has grown over the years, the maturity and norm of parallel development has become the standard development process instead of the exception. With this parallel development comes the need for larger awareness among developers to have "an understanding of the activities of others which provides a context for one's own activities" [1]. This added awareness mitigates some downsides of parallel development which include the cost of conflict prevention and resolution, however, in practice we see these mitigated losses continue to appear quite frequently. Not only do these conflicts still occur, but at times they can prove to be a significant and time-consuming chore for developers [2].

Two types of conflicts have attracted the attention of academics over the years, *direct* and *indirect conflicts*. Direct conflicts involve immediate workspace concerns such as developers editing the same artifact, or finding expert knowledge about a particular file. Tools have been created and studied for direct conflicts [3], [4], [5], [6] with relatively good success and positive developer feedback. However, indirect conflict tools have not shared the same success. Indirect conflicts can arise in source code for such reasons as having one's own code negatively affected by a library upgrade, making a code change only to find out it had negative effects on unit or integration test, or having negative effects on the way other developer's have interpreted the use of your code. Tools that have attempted to either help developers prevent or solve indirect conflicts have either attracted little developer interest or have had some common struggles which remain to be studied or solved.

Difficulties regarding indirect conflicts are to determine when software dependencies change and more crucially, when those changes will create negative consequences in a software project. Sarma et al. [7] built Palantir, which can both detect potential indirect conflicts, at the class signature level, and alert developers to these conflicts. Holmes et al. [8] take it one step further with their tool YooHoo by detecting fine grained source code changes, such as method return type changes, and create a taxonomy for different types of changes and their proneness to cause indirect conflicts. The tool Ariadne [9] creates an environment where developers can see how source code changes will affect other areas of a project at the method level and thus where indirect conflicts may occur. Another indirect conflict tool, CASI [10], utilizes dependency slicing [11] instead of method call graphs to provide an environment to see what areas of a project are being affected by a source code change. Most of these tools have all shown to have common difficulties of: scalability, false positives, information overload, or providing redundant information to preexisting tools. Our own tool Impact! [1] also suffered these same fates.

Since Cataldo et al. [12] have shown that socio-technical congruence can be leveraged to improve task completion times, many indirect conflict tools support the idea of a socio-technical congruence [13] in order to help developers solve their indirect conflict issues through social means [14] [15]. However, socio-technical congruence is largely unproven in regards to its correlation to software quality [16] and again the problems of scalability and information overload become a factor.

We are interested in exploring three questions in the hopes of resolving some of the issues that are occurring with indirect conflict tools:

- RQ1 *What is the nature of indirect conflicts?*
- RQ2 *What types of process or tools are being used by developers in regards to indirect conflicts?*
- RQ3 *What do developers want from future indirect conflict tools?*

We interviewed 19 developers from across 12 companies and open source projects as well as surveyed 78 developers in order to answer the aforementioned questions. Our findings indicate that while indirect conflicts are very much a real and

[1] https://github.com/jordanell/Impact

frequent problem in software development, developers already posses the tools to prevent and catch indirect conflicts to a manageable degree and have the mindset of working until something breaks, then attempting a fix. Thus, developers would rather spend their time developing instead of taking or reading preventative measures, resulting in the need of tools to debug and solve indirect conflicts quickly as opposed to tools which attempt to prevent or catch indirect conflicts early.

We continue this paper by first presenting our summarized methodology we used to perform this study in Section II. We then explain and discuss the results of our study in regards to our 3 research questions in Section III. Next, an evaluation of how our study was performed in terms of the credibility and quality of our results is described in Section V. We then cover related work in Section VI and conclude in Section VII.

II. Methodology

Our grounded theory study was performed in two parts. First, a round of semi-structured interviews were conducted which addressed the 3 research questions as mentioned above. Secondly, a survey was conducted which was used to confirm and test what was theorized from the interviews on a larger sample size as well as obtain larger developer opinion of the subject.

A. Grounded Theory

We approached our study based on grounded theory as described by Corbin and Strauss [17]. Grounded theory is a qualitative research methodology that utilizes *theoretical sampling* and *open coding* to create a theory "grounded" in the empirical data. For an exploratory study such as the ours, grounded theory is well suited because it involves starting from very broad and abstract type questions and making refinements along the way as the study progresses and hypotheses begin to take shape. Grounded theory involves realigning the sampling criteria throughout the course of the study to ensure that participants are able to answer new questions that have been formulated in regards to forming hypotheses. In our study being presented, data collected from both interviews and surveys (when open ended questions were involved) was analyzed using open coding. Open coding involves assigning codes to what participants said at a low sentence level or abstractly at a paragraph or full answer level. These codes were defined as the study progressed and different hypotheses began to grow. We finally use axial coding it order to link our defined codes through categories of grounded theory such as context and consequences. The ultimate goal of grounded theory is to produce a theory that is tied to the empirical data collected; our final theories can be seen in findings **??** - **??**. In Section V, we give a brief evaluation of our studying using 3 criteria that are commonly used in evaluating grounded theory studies.

B. Interview Participants

We selected our interview participants from a large breadth of both open and closed source software development companies and projects. The population which participated in our interview came from the following software groups: IBM, Mozilla, The GNOME Project, Microsoft Corporation, Subnet Solutions, Ruboss Technology Corporation, Amazon, Exporq Oy, Kano Apps, Fireworks Design, James Evans and Associates, and Frost Tree Games. Our participants we chosen based on their direct involvement in the actual writing of software for their respective companies or projects. These participants' software development experience ranged from 3-25 years of experience with an average of 8 years of experience. In addition to software development, some participants were also chosen based on both their experience of software development as well as their experience with project management at some capacity.

C. Interview Procedure

Participants were invited to participate in the interviews by email and were sent a single reminder email one week after the initial invitation if no response had been made. We directly emailed 22 participants and ended up conducting 19 interviews. Interviews were conducted in person when possible and recorded for audio content only. When in person interviews were not possible, one of Skype or Google Hangout was used with audio and video being recorded but only audio being stored for future use. We were pleased with the response rate to our initial invitation emails as our breadth of participants was rather large, spanning multiple open and closed source, Agile and Waterfall based projects, and our interview answer saturation [18] was quite high.

Interview participants first answered a number of structured demographic items. Next, participants were asked to describe various software development experiences regarding our three research questions. Our three questions were studied by having the interview participants talk about their experiences and opinions in the following 10 semi-structured research topics:

- Examples of indirect conflicts from real world experiences.
- Software artifact dependency levels and where conflicts can arise.
- Software development tools for dependency tracking and awareness.
- Software development process for preventing indirect conflicts.
- How developers find internal or external software dependencies.
- How indirect conflicts are detected and found.
- How indirect conflict issues are solved or dealt with.
- Developer opinion of preemptive measures to prevent indirect conflicts.
- Developer opinion on what types of changes are worth a preemptive action.
- Developer opinion on who is responsible for fixing or preventing indirect conflicts.

While each of the 10 question categories had a number of starter questions, interviews largely became discussions of developer experience and opinion as opposed to direct answers to any specific question. However, not all participants

had strong opinions or any experience on every category mentioned. For these participants, answers to the specific categories were not required or pressed upon. We attribute any non answer by a participant to either lack of knowledge in their current project pertaining to the category or lack of experience in terms of being apart of any one software project for extended periods of time. We account for these non answers in our analysis and results as seen in Section III. Interviews lasted from 15 minutes up to 75 minutes and were very much dependent on the experience of the participant.

D. Survey Participants

We selected our survey participants from a similar breadth of open and closed source software development companies and projects as the interviews participants with two large exceptions. The software organizations that remained the same between interview and survey were: Mozilla, The GNOME Project, Microsoft Corporation, Subnet Solutions, and Amazon. However, participants who took part in the round of interviews were asked to act as a contact point for other developers in their team, project, or organization who may be interested in completing the survey. Aside from this aforementioned list, two groups of developers were asked to participate as well, these being GitHub users as well as Apache Software Foundation (Apache) developers. The GitHub users were selected based on large amounts of development activity on GitHub and the Apache developers were selected based on their software development contributions on specific projects known to be used heavily utilized by other organizations and projects.

E. Survey Procedure

Survey participants were invited to participate in the survey by email. No reminder email was sent as the survey responses were not connected with the invitation email addresses and thus participants who did respond could not be identified. We directly emailed 1300 participants and ended with 78 responses giving a response rate of 6%. We attribute the low response rate with: the surveys were conducted during the months of July and August while many participants may be away from their regular positions. and our GitHub and Apache participants could not be verified as to whether or not they actively support the email addresses used in the invitations. In addition, the survey was considered by some to be long and require more development experience than may have been typical of some of those invited to participate.

The created the survey was based off of categorical hypotheses created by the round of interviews. The survey was designed to test these hypotheses and to acquire a larger sample size of developers who may have similar or different opinions from those already acquired from the interviews. The survey went through two rounds of piloting. Each pilot round consisted of five participants, who were previously interviewed, completing the survey with feedback given at the end. Not only did this allow us to create a more polished survey, but it also allowed the previously interviewed developers to examine what hypotheses were formed and what we would be moving forward with. The final survey consisted of: 2 multiple choice questions for demographic information; 3 level of agreement questions for an indication of what types of environments indirect conflicts are more likely to occur; 6 level of use questions to indicate what types of software changed developers find trouble some with respect to indirect conflicts; and 9 short answer questions for indication of when indirect conflicts occur, what types of processes are used to prevent or react to indirect conflict, and to provide an outlet for general opinion on the matter. Each non demographic question was made optional as it was shown through the interviews that some questions require more experience from participants than may be provided.

F. Evaluation

Following our data collection and analysis, we re-interviewed some of our initial interview participants in order to validate our findings. We confirmed our hypotheses as to whether or not they resonate with industry participants' opinions and experiences regarding indirect conflicts and as to their industrial applicability. Due to limited time constraints of the interviewed participants, we could only re-interview five participants. Those that were re-interviewed came from the range of 5-10 years of software development experience. Re-interviewed participants were given our 3 research questions along with results and 9 findings, and asked open ended questions regarding their opinions and experiences to validate our findings. The five participants found FILL IN HERE WHEN COMPLETE.

III. Results

We now present our results of both the interviews and surveys conducted in regards to our 3 research questions outlined in Section I. We restate each research question, followed by our quantitative and qualitative results from which we draw our discussion to be seen in Section IV.

A. What is the nature of indirect conflicts?

From the interviewed participants, 63% of developers believe that a large contributing factor to the cause of indirect conflicts comes from the changing of a software object's contract. Object contracts are, in a sense, what a software object guarantees, meaning how the input, output, or how any aspect of the object is guaranteed to work; made famous by Eiffel Software's [2] "Design by Contract"™. In light of object contracts, 73% of interviewed developers gave examples of indirect conflicts they had experienced which stemmed from not understanding the far reaching ramifications of a change being made to an object contract towards the rest of the project. Of those 73%, 21% dealt with the changing of legacy code, with one developer saying "legacy code does not change because developers are afraid of the long range issues that may arise from that change". Another developer, in regards to changing object contacts stated "there are no changes in the

[2] http://www.eiffel.com/

input or changes in the output, but the behavior is different". Developers also noted that the conflicts that do occur tend to be quite unique from each other and do not necessarily follow common patterns.

In regards to object contract changes, 9 developers currently working with large scale database applications listed database schemas as a large source of indirect conflicts while 5 developers that work on either software library projects or are in test said that methods or functions were the root of their indirect conflict issues. 36% of interviewed developers mentioned that indirect conflicts occur when a major update to an external project, library, or service occurs with one developer noting "their build never breaks, but it breaks ours". Some other notable indirect conflict artifacts were user interfaces in web development and full components in component base game architecture.

From the interviewed participants, 58% of developers explained that indirect conflicts occur "all the time" in their development life cycle with a minimum occurrence of once a week, with more serious issues tending to occur once a month. 65% of surveyed developers answered that indirect conflicts occur on minimum bi-weekly, with the majority of developers saying that weekly occurrences are most common.

63% of developers interviewed said that when a project is in the early stages of development, indirect conflicts tend to occur far more frequently than once a stable point is reached. Developers said "At a stable point we decided we are not going to change [this feature] anymore. We will only add new code instead of changing it." and "the beginning of a project changes a lot, especially with agile". Surveyed developers also added "indirect conflicts after a release depend on how well the project was built at first", "[indirect conflicts] tend to slow down a bit after a major release, unless the next release is a major rework.", and "[indirect conflicts have] spikes during large revamps or the implementation of cross-cutting new features.". Surveyed participants also answered that indirect conflicts are more likely to occur before the first major release rather than after at the daily and weekly occurrence intervals as seen in Table I.

In terms of organizational structure, surveyed participants answered that as a project becomes larger and more developers are added, even to the point that multiple teams are formed, indirect conflicts become more likely to occur. However, indirect conflicts still occur at a lower number of developers as well with even 43% of developers saying they are like to occur in single developer projects.

B. What types of process or tools are being used by developers in regards to indirect conflicts?

In terms of preventative processes used for indirect conflicts, 3 major components were found. First, design by contract is heavily used by interviewed developers as a means to avoid indirect conflicts or understand when they are likely to occur. The use of design by contract here means that developers tend not to change an object's contract when possible, and that an object's contract is used as a type of documentation towards awareness of the software object. One developer stated that "design by contract was invented to solve this problem and it does it quite well", while another noted that software object contracts do solve the problem in theory, but that doesn't mean that problems don't still occur in practice. Second, 21% of interview developers mentioned that the add and deprecate model is used to prevent indirect conflicts once the project, feature, or component has reached a stable or mature point. Add and deprecate meaning instead of editing code, the developer simply clones old code (if needed), and edits the clone while slowly phasing out the out the old code in subsequent releases or ad needed. This allows other software to use the older versions of software objects which remain unchanged, thus avoiding indirect conflicts. Lastly, pure developer experience was mentioned with 37% of developers mentioning that when planning code changes, either a very experienced member of the project was involved in the planning and has duties to foresee any indirect conflicts that may arise, or that developers must use their personal knowledge to predict where indirect conflicts will occur while implementing.

Of the the 37 developers surveyed who could give positive identification of preventative processes for indirect conflicts, 27% said that individual knowledge of the code base and their impact of code change was used, 59% mentioned some form of design by contract or the testing of a methods contract, and 14% said that add and deprecate was used in their projects to avoid indirect conflicts.

In regards to catching indirect conflicts, 69% of interviewed developers mentioned forms of testing (unit, and integration) as the major component of catching indirect conflict issues, subscribing to the idea of "run the regression and integration tests and just see if anything breaks". The words "use case coverage" were constantly being used by developers when expressing how proper unit and integration tests should be written. Developers expressed that with proper use case coverage, most if not all indirect conflicts should be caught. 31% of developers said build processes (either nightly builds or building the project themselves), and others mentioned code reviews while those dealing with a user interface mentioned user complaints from run time testing. The surveyed developers confirmed these results with 49% mentioning forms of testing as the major tool used to catch indirect conflicts, 33% said build processes, while 31% used work their IDE or IDE plug-ins to catch indirect conflicts. Surveyed developers also mentioned review process and personal expertise as factors of catching indirect conflicts.

Once an indirect conflict has occurred and developers need to resolve it, 75% of developers interviewed said they checked historical logs to help narrow down where the problem could originate from. Most developers had the mindset of "Look at the change log and see what could possibly be breaking the feature.". The log most commonly referred to was the source code change log to see what code has been changed, followed by build failure or test failure logs to examine errors messages and get time frames of when events occurred. Of the developers surveyed, 23% said they used native IDE tools

TABLE I: Results of survey questions to how often indirect conflicts occur, in terms of percentage of developers surveyed.

Occurrences	Daily	Weekly	Bi-Weekly	Monthly	Bi-Monthly	Yearly	Unknown
Early stages of a development	32%	18%	4%	5%	0%	5%	36%
Before the first release	13%	29%	6%	8%	1%	3%	40%
After the first release	6%	18%	8%	18%	1%	5%	44%
Late stages of development	6%	5%	5%	18%	8%	12%	46%

and 21% said they use features of the language's compiler and debugger in order to solve indirect conflicts. Interestingly, only 13% of developers mentioned a form of communication with other developers in aid to solving these conflicts and only 4% mentioned the reading of formal documentation.

Through the processes and tools of prevention, detection, and resolution of indirect conflicts, it is important to note that most developers ascribe to the idea of "I work until something breaks", or taking a curative rather than preventative approach. This means that while developers do have processes and tool for prevention, they would rather spend their time at the detection and resolutions stages. One developer noted that preventative processes are "too much of a burden" while a project manager said "[with preventative process] you will spend all you time reviewing instead of implementing".

C. What do developers want from future indirect conflict tools?

When asked about preventative tools, interviewed developers had major concerns that the amount of false positives provided by the tool which may render the tool useless. Developers said "this would be a real challenge with the number of dependencies", "it depends on how good the results are in regards to false positives", and "I only want to know if it will break me", meaning that developers seem to care mostly about negative impacts of code changes as opposed to all impacts in order to reduce false positives and to keep preventative measures focused on real resulting issues as opposed to preventing potential issues. Overall, developers had little interest in preventative tools or processes.

In terms of catching indirect conflicts, interviewed developers suggested that proper software development processes are already in place to catch potential issues such as testing, code review, individual knowledge, or static language analysis tools. 68% of surveyed developers said that they would always want to be notified about method signature changes as they have a high chance to break the code as opposed to only 23% who always wanted to be notified on a pre or post condition change and 27% who want to be notified for a user interface change. Other change types varied from never being notified to most times being notified, showcasing the complexity of change types which may or may not negatively affect a project.

When asked about curative tools, developers could only suggest that resolution times be decreased by different means. Surveyed developers suggested the following improvements to curative tools:

- Aid in debugging by finding which recent code changes are breaking a particular area of code or a test.
- Automatically write new tests to compensate for changes.
- IDE plug-ins to show how current changes will affect other components of the current project.
- Analysis of library releases to show how upgrading to a new release will affect your project.
- Built in language features to either the source code architecture (i.e. Eiffle or Java Modeling Language) or the compile time tools to display warning messages towards potential issues.
- A code review time tool which allows deeper analysis of a new patch to the project allowing the reviewer to see potential indirect conflicts before actually merging the code in.
- A tool which is non-obtrusive and integrates into their preexisting development styles without them having to take extra steps.

IV. Discussion

V. Evaluation

As per grounded theory research, Corbin and Strauss list ten criteria to evaluate quality and credibility [17]. We have chosen three of these criteria and explain how we fulfill them.

Fit. "Do the findings fit/resonate with the professional for whom the research was intended and the participants?" This criterion is used to verify the correctness of our finding and to ensure they resonate and fit with participant opinion. It is also required that the results are generalizable to all participants but not so much as to dilute meaning. To ensure fit, during interviews after participants gave their own opinions on a topic, we presented them with previous participant opinions and asked them to comment on and potentially agree with what the majority has been on the topic. Often the developers own opinions already matched those of the majority before them and did not necessarily have to directly verify it themselves.

To ensure the correctness of the results, we also linked all findings in Section III to either a majority of agreeing responses on a topic or to a large amount of direct quotes presented by participants.

Applicability or Usefulness. "Do the findings offer new insights? Can they be used to develop policy or change practice?" Although our findings **??** - **??** may not be entirely novel or even surprising, the combination of these results allow us to discover the insightful findings of **??** and **??** regarding indirect conflict tools. Given how many indirect conflict tools are left with the same common issues, we believe that these findings will help researchers focus on what developers want and need moving into the future more than has been possible

in the past. These finding set a course of action for where effort should be spent in academia to better benefit industry.

10 of the 78 participants who were surveyed sent direct responses to us asking for any results of the research to be sent directly to them in order to improve their indirect conflict work flows. 7 of the 19 participants survey expressed interest concerning any possible tools or plans for tools to come out of this research as well. The combination of academia relatability and direct industry interest in our results help us fulfill this criterion.

Variation. "Has variation been built into the findings?" Variation shows that an event is complex and that any findings made accurately demonstrate that complexity. Since those participants interviewed came from such a diverse set of organizations, software language knowledge, and experience the variation naturally reflected the complexity. Often in interviews and surveys, participants expressed unique situations that did not fully meet our generalized findings or on going theories. In these cases, we worked in the specific cases which were presented as boundary cases and can be seen in quotations in Section III. These quotations add to the variation to show how the complexity of the situation also resides in a significant number of unique boundary situations as well as the complexity in the generalized theories and findings.

VI. Related Work

Since this paper has covered a wide spectrum in regards to indirect conflicts (preventing, catching, solving, process, and tools), there exists a large body of work in which to draw from regarding indirect conflicts. While some of the previous literature may not deal explicitly in the notion of indirect conflicts, lessons learned from topic in awareness, preemptive direct conflict detection, and debugging can be used.

Kim conducted several initial focus groups as well as web surveys to determine what developers are interested in with regards to software modifications [19]. The top two interests found were that developers wanted to know whose recent code changes semantically interfere with their own code changes, and whether their code is impacted by a code change. These areas of interest resonate with what was found in this paper. Developers want to know when a code change is going to interfere with their work in a potentially negative way. Kim also found that developers are concerned with interfaces of objects and when those interfaces change, similar to the object contracts that were found in this paper. Finally, Kim also identified the same issues towards information overload through false positives with developers noting "I get a big laundry list... I see the email and I delete it".

In a case study of impact awareness de Souza et. al. [20] found that developers use their personal knowledge of the code base to determine the impact of their code changes on fellow developers, teams, and projects. This corresponds with the findings of preventative measures we have found in that some human interaction or knowledge is required in preventing indirect conflicts along with development processes.

In a study regarding static analysis tools, Johnson et. al [21] found similar conclusions towards false positive output as well as developer process integration. Developers complained that static analysis tools usually produce a large amount of false positives to the point where output is ignored, as well as the tool not being designed properly to fit their current development work flow. In our case we found that the current development work flow is "work until something breaks" as opposed to try and find where things might break before making code changes.

Hattori et. al. [22] found, through qualitative user studies, that developers tend to use the bare amount of communication towards other developers in order to solve direct conflicts, and take the same approach of only communicating once a conflict has arose. This is the same mentality found in this paper as "I work until something breaks". Hattori et. al. also show that with direct conflicts, the sooner preemptive information if available to developers the more they will communicate and either avoid or easily solve their code merges. This is the situation many indirect conflict tools have strived for. In terms of communication, Bolici et. al [23] give the possibility of developer stigmergy as the reason that developers do not need to explicitly communicate with each other about issues. The idea that developers use artifacts left behind by other developers to solve their issues could play a large role in a lack of formal communication.

While not specifically developed with indirect conflicts in mind, Zeller's delta debugging techniques [24], can be, and should be, applied to solving indirect conflict issues. Zeller gives automated techniques for identifying failure-inducing changes (as well as other failure causing components) of the software project in order to isolate and debug some issue. These practices fit with the idea that developers wait for something to go wrong, then wish to fix the issue as fast as possible. Program slicing [25] has also been extensively studied in order to aid in debugging in manual or automated techniques.

VII. Conclusions

Indirect conflicts are a significant issue with real world developers, however, many proposed techniques and tools to mitigate losses in this realm have been unsuccessful in attracting major support from developers. Based on our qualitative study involving 19 interviewed developers from 12 organizations as well as 78 surveyed developers, we have provided characterization of indirect conflicts, current developer work flow surrounding indirect conflicts, and what direction any future tools should follow in order to aid developers in their work flow with indirect conflicts.

We have shown through findings **??** - **??** why indirect conflicts occur, when indirect conflicts are more likely to occur, as well as what types of software objects are susceptible to these conflicts. Findings **??** - **??** have shown how developers in industry currently handle the prevention detection and solving of indirect conflicts. And Lastly, findings **??** and **??** have provided a foundation as to why past techniques and tools have

had low adoption rates and where researchers should focus their current and future efforts in handling indirect conflicts. We hope that this study and its findings will inspire future techniques and tools for dealing with indirect conflicts that will aid developers in industry as well as test and validate our theories put forth in this paper as well as spark a potential investigation into preventative versus curative approaches to software defects.

VIII. Acknowledgments

We would sincerely like to thank all participants who were willing to be interviewed or who participated in completing our survey. We thank these people for taking time out of their day to participate and for sharing their developer experience with us. Without these people our research could not have been possible.

References

[1] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, ser. CSCW '92. New York, NY, USA: ACM, 1992, pp. 107–114.

[2] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel changes in large-scale software development: an observational case study," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 3, pp. 308–337, Jul. 2001.

[3] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, and S. X. Yang, "Ensemble: a recommendation tool for promoting communication in software teams," in *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, ser. RSSE '08. New York, NY, USA: ACM, 2008, pp. 2:1–2:1.

[4] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, "Fastdash: a visual dashboard for fostering awareness in software teams," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '07. New York, NY, USA: ACM, 2007, pp. 1313–1322.

[5] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: Interactive visual exploration of socio-technical relationships in software development," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 23–33.

[6] H. Khurana, J. Basney, M. Bakht, M. Freemon, V. Welch, and R. Butler, "Palantir: a framework for collaborative incident response and investigation," in *Proceedings of the 8th Symposium on Identity and Trust on the Internet*, ser. IDtrust '09. New York, NY, USA: ACM, 2009, pp. 38–51.

[7] A. Sarma, G. Bortis, and A. van der Hoek, "Towards supporting awareness of indirect conflicts across software configuration management workspaces," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 94–103.

[8] R. Holmes and R. J. Walker, "Customized awareness: recommending relevant external change events," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 465–474.

[9] E. Trainer, S. Quirk, C. de Souza, and D. Redmiles, "Bridging the gap between technical and social dependencies with ariadne," in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, ser. eclipse '05. New York, NY, USA: ACM, 2005, pp. 26–30.

[10] F. Servant, J. A. Jones, and A. van der Hoek, "Casi: preventing indirect conflicts through a live visualization," in *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE '10. New York, NY, USA: ACM, 2010, pp. 39–46.

[11] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An internet-scale software repository," in *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, ser. SUITE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–4.

[12] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: implications for the design of collaboration and awareness tools," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, ser. CSCW '06. New York, NY, USA: ACM, 2006, pp. 353–362.

[13] I. Kwan and D. Damian, "Extending socio-technical congruence with awareness relationships," in *Proceedings of the 4th international workshop on Social software engineering*, ser. SSE '11. New York, NY, USA: ACM, 2011, pp. 23–30.

[14] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 125–134.

[15] A. Borici, K. Blincoe, A. Schrter, G. Valetto, , and D. Damian, "Proxiscientia: Toward real-time visualization of task and developer dependencies in collaborating software development teams," in *In Proc*, ser. CHASE 2012.

[16] I. Kwan, A. Schroter, and D. Damian, "Does socio-technical congruence have an effect on software build success? a study of coordination in a software project," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 307–324, May 2011.

[17] J. Corbin and A. C. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, 3rd ed. Sage Publications, 2007.

[18] G. Guest, A. Bunce, and L. Johnson, "How many interviews are enough?: An experiment with data saturation and variability," *Field Methods*, vol. 18, no. 1, pp. 59–82, 2006.

[19] M. Kim, "An exploratory study of awareness interests about software modifications," in *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE '11. New York, NY, USA: ACM, 2011, pp. 80–83.

[20] C. R. B. de Souza and D. F. Redmiles, "An empirical study of software developers' management of dependencies and changes," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 241–250.

[21] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681.

[22] L. Hattori, M. Lanza, and M. D'Ambros, "A qualitative user study on preemptive conflict detection," in *Global Software Engineering (ICGSE), 2012 IEEE Seventh International Conference on*, 2012, pp. 159–163.

[23] F. Bolici, J. Howison, and K. Crowston, "Coordination without discussion? socio-technical congruence and stigmergy in free and open source software projects," in *2nd International Workshop on Socio-Technical Congruence, ICSE*, Vancouver, Canada, 19 May 2009. [Online]. Available: http://docs.google.com/View?id=dhncd3jd_405fzt842gv

[24] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[25] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, Mar. 2005.