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*—Developer awareness techniques have been proposed and studied to aid in developer understanding and efficient developer output in terms of source code produced. These techniques involve the sharing of information regarding source code changes in a specific project in order to either catch potential issues before they are fully integrated or to help developers debug pre-existing issues. These techniques have largely focused on either *direct* or *indirect conflict* issues. While the techniques and tools for direct conflicts have had large success, tools either proposed or studied which focus on indirect conflicts have had many common struggles and poor developer interest. To better understand why indirect conflict tools either fail or have little developer investment, we performed a grounded theory study with 97 developers involved in either interviews or a web survey. We identified the main features that characterize indirect conflict issues, what processes and techniques developers use to aid in preventing and solving indirect conflict issues, and what developers would look for in a tool to aid them in these 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 in theory mitigates the downside of parallel development which is the cost of conflict prevention and resolution, however, in practice we see these mitigated losses 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, finding expert knowledge about a particular file, or knowing when another developer is working on the same file as you. Tools have been created and studied for these types of issues [3], [4], [5], [6] with relatively good success and positive developer feedback. However, indirect conflict tools have not had 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 or 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.

The inherent difficulty regarding indirect conflicts is one, to determine which software dependencies are causing indirect conflicts, and two, to be able to analyze these dependencies, specifically as to when they change and whether or not they will induce an indirect conflict or otherwise negative consequences of a software project. Sarma et al. [7] have devised a tool (Palantir) which can both detect potential indirect conflicts and alert developers to these conflicts arising in their software projects at the class signature level. Ariadne [8], another indirect conflict tool, creates an exploratory environment where developers can see what areas of a project they are affecting with source code changes at the method level and thus where indirect conflicts may occur. Another tool developer for indirect conflicts is CASI [9] which utilizes dependency slicing [10] instead of the more traditional method call graphs to provide again an environment to see what areas of a project are being affected by a source code change. These tools have all shown to have the same common difficulty of scalability in the face of information overload. Too many dependencies with the potential for indirect conflicts are being reported to developers to the point where the number of false positives reported becomes unbearable. Our own unpublished tool Impact! [1] also suffered this same fate.

Many indirect conflict tools support the idea of a socio-technical congruence [11] in order to help developers solve their indirect conflict issues through the means of other developers [12] [13]. Common issues arise here in that developers do not always utilize others to solve issues, socio-technical congruence is largely unproven in regards to its correlation to software quality [14] and again the problem with scalability in that many many developers may be linked to anyone particular persons work. Again the notion of confirmation overload becomes apparent and false positives become a large concern.

We are interested in exploring three main areas in the hopes of resolving some of the issues that are occurring with indirect conflict tools: the nature of these conflicts in the how, what, and where, the processes that developers use prevent, catch, and solve indirect conflicts, and what developer opinions are on preventative indirect conflicts tools as well as what they would look for in any tool for indirect conflicts. We broke these categories down into the following 8 research questions:

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

RQ1 *How do indirect conflicts occur?*
RQ2 *How often do indirect conflicts occur and at what times in a project do they occur more frequently?*
RQ3 *What software artifacts and environments are susceptible to indirect conflicts?*
RQ4 *How do developer prevent indirect conflicts?*
RQ5 *How do developers catch indirect conflicts?*
RQ6 *How to developers react to and solve indirect conflicts?*
RQ7 *When are preemptive indirect conflict tools best used and what would developers want from them?*
RQ8 *What kinds of tools do developers want to help them handle and solve indirect conflicts?*

We interviewed 19 developers from across 12 companies and open source projects as well as surveyed 78 developers in order to answer these questions. Our findings indicate that while indirect conflicts are very much a real problem in software development, developers already poses the tools to prevent and catch indirect conflicts to a certain 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 (reactionary tools) as opposed to tools which attempt to prevent or catch indirect conflicts early (preemptive tools).

We continue this paper by first presenting our summarized methodology we used to perform this study in Section II. We continue with the results of our study in regards to our 8 research questions in Section III followed by a discussion in Section IV of what was found and how it can be applied to future study and implementation of indirect tools or processes. We next present how evaluation was performed in terms of the credibility and quality of our results in Section V, 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 larger who, what, when, where, and how about indirect conflicts. Secondly, a survey was given out which was used to confirm what was theorized from the interviews on a larger sample size as well as obtain larger developer opinion.

A. Grounded Theory

We approached our study based on grounded theory as described by Corbin and Strauss [15]. Grounded theory is a qualitative research methodology that utilizes *theoretical sampling* and *open code* to create a social theory "grounded" in the empirical data. For an exploratory study such as the ours, grounded theory is well suited because 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 regining 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 to 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 F1 - F9. 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 titles included: senior developer, lead developer, software developer, software developer in test, and junior developer. In addition to software development, some participants were also chosen based on their previous experience of software development as well as their current experience with project management at some capacity. These management type participants involved in project planning to some capacity between technical architecture, project maintainer, chief technical officer, or program manager.

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 quite pleased with the response rate to our initial invitation emails as our breadth of participants was rather large spanning multiple open and closed source project and our interview answer saturation [16] was quite high (to be seen in Section III).

Interview participants first answered a number of structured demographic items. Next, participants were asked to describe various software development experiences regarding our three main areas of focus: the nature of indirect conflicts, current processes used regarding indirect conflicts, opinions on future and the use of current tools. processes they use or encounter in their current working environment. These three areas of focus were studied by having the interview participants talk about their experiences and opinions in the following 10 semi-structured research topics which were asked of:

- Software development tools for dependency tracking and awareness.
- Software development process for preventing indirect conflicts.
- Software artifact dependency levels and where conflicts can arise.
- How developers find internal or external software dependencies.
- Examples of indirect conflicts from real world experiences.
- 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. In general, once the category was explained to the participant, little prompting was necessary as the participant would jump right into an experience, or issue he or she had with the question or category. 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. The length of the interview largely depended on the amount of experience the participant had in software development as they were able to give more experiences and more informed opinions.

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 not asked to participate in the surveys but rather 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 large 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 participation on specific projects known to be used heavily utilized by other organizations and projects.

E. Survey Procedure

The created survey was based off of the 10 categorical hypotheses given 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. The previous interview participants were selected based on their domain knowledge expertise with indirect conflicts in order to provide what we think would be amount to better feedback and resulting in a more polished survey. This also allowed the previously interviewed developers to examine what hypotheses were formed and we would be moving forward with. The final survey consisted of the following: 2 multiple choice questions for developer experience and 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.

Survey participants were invited to participate in the survey by email. No reminder email was ever sent as the survey responses were not connected with the invitation email addresses and thus participants who did respond could not be removed from a reminder list. We directly emailed 1300 participants and ended with 78 responses giving a response rate of 6%. We attribute the low response rate with a couple of factors. One, the surveys were conducted during the months of July and August while many participants may be away from their regular positions. Two, our GitHub and Apache participants could not be verified as to whether or not they actively support the email addresses used in the invitations. And finally, 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.

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 8 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 research questions outlined in Section I. We first present and discuss the general nature of indirect conflicts in terms of when, what types, and how often indirect conflicts occur. We next show the findings for the process related research questions in regards to how developers react to, processes to prevent, and reactionary steps to indirect conflicts.

A. The Nature of Indirect Conflicts

Although a detailed manual inspection of a project's lifetime and the indirect conflicts it experiences throughout its development is beyond the scope of this paper, we first wanted to explore the general nature of indirect conflicts. This being the case we explored, through our interviews and surveys, what software developers believed to be the case when it came to how indirect conflicts occur, when and how often they occur, and the types of software object's that incur indirect conflicts.

RQ1 *How do indirect conflicts occur?*

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 garunteed to work; made famous by Eiffle 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%, examples were broken down as follows. 21% of examples 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". 36% of interviewed developers mentioned that indirect conflicts occur when an update to an external project, library, or service occurs and the resulting integration fails noting that major releases of libraries can be a large issue with one developer saying "their build never breaks, but it breaks ours". Finally, 29% of developers interviewed explained that changes to databases are often a large source of indirect conflicts saying "[schema changes] break stuff all over the place".

RQ2 *How often do indirect conflicts occur and at what times in a project do they occur more frequently?*

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. Of those 58%, developers noted that a lot of their development time is put into understanding code to avoid indirect conflicts, more serious issue tend to happen once a month, and that the conflicts that do occur tend to be quite unique from each other.

In terms of when in a project indirect conflicts are more likely to occur 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".

For those developers interviewed, we asked how often indirect conflicts occur in their software development experience, how often they occur before and after a project's first release, as well as how often indirect conflicts occur late in a project's lifetime. Table I shows the results of these questions.

Of those surveyed that could not give direct answers to the questions in Table I, the more common answers were as follows: "People rushing to meet [the release date] is where most of my indirect conflicts occur.", "Indirect conflicts after a release depends on how well the project was built at first", "They tend to slow down a bit after a major release, unless the next release is a major rework.", and "Conflicts continue to occur at roughly the same rate after the first release, with spikes during large revamps or the implementation of cross-cutting new features.".

RQ3 *What software artifacts and environments are susceptible to indirect conflicts?*

From the developers interviews, 9 were currently working with large scale database applications and all 9 listed database schema as a large source of indirect conflicts. Out of the 5 developers interviewed that currently work on either software library projects or in test, all 5 said that methods or functions were the root of their indirect conflict issues. Some other notable artifacts were user interface displays for web development, and full components in component base game architecture.

As per environments, Table II shows which development team environment developers believe to be the most prone to indirect conflicts.

B. Processes of Indirect Conflicts

The results that follow pertain to software development process and its relation to indirect conflicts. We explore how developers react to indirect conflicts, which stakeholders are responsible for their fixes, and if any development processes can be used to prevent or catch indirect conflicts.

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

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

Question	Daily	Weekly	Bi-Weekly	Monthly	Bi-Monthly	Yearly	Unknown
How often do you experience indirect conflicts in your software development?	17%	27%	21%	17%	3%	5%	10%
How often do indirect conflicts occur in the early stages of a projects development?	32%	18%	4%	5%	0%	5%	36%
How often do indirect conflicts occur before the first release of a project?	13%	29%	6%	8%	1%	3%	40%
How often do indirect conflicts occur after the first release of a project?	6%	18%	8%	18%	1%	5%	44%
How often do indirect conflicts occur late in a projects lifetime?	6%	5%	5%	18%	8%	12%	46%

TABLE II: Results of survey questions to development environments in which indirect conflicts are likely to occur, in terms of percentage of developers surveyed.

Environment of conflicts	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Developing alone (conflicts in own code)	18%	20%	19%	24%	19%
Developing in a team between 2 - 5 developers (Inter-developers conflicts)	3%	8%	22%	49%	18%
Developing in a multi team environment (Inter-team conflicts)	1%	11%	14%	39%	35%

RQ4 *How do developer prevent indirect conflicts?*

Of the developers interviewed, 3 main points of interest were found in preventing indirect conflicts. The first was the use of "Design by Contract"^{TM}, meaning that software objects have certain guarantees about their behavior in terms of input, output, and internal workings. This methodology can be used as a type of documentation towards awareness of a 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. The second point was the add and deprecate methodology. 21% if developers interviewed mentioned that the development process changes once a stable point of a project is met and the "add and deprecate model [prevents] a lot of these issues". Add and deprecate meaning instead of editing code, the developer simply clones code (if needed) and edits the clone while slowly phasing out the out code in subsequent releases or other development cycle needs. This allows a stoppage of indirect conflicts by allowing developers to use the older versions of software objects which remain unchanged. Lastly, pure developer experience was mentioned as a source to prevent indirect conflicts. 37% of developers mentioned 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 while implementing a code change developers must use their personal knowledge to predict where indirect conflicts will occur.

Of the developers surveyed who could give positive identification to preventative measures taken to avoid indirect conflicts (37 developers total), 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 (which could be viewed as catching indirect conflicts), and 14% said that add and deprecate was used in their projects to avoid indirect conflicts.

RQ5 *How do developers catch indirect conflicts?*

Of the 19 developers interviewed, 69% mentioned forms of testing (unit, integration, etc) 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". 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.

Of the 78 developers survey, 49% mentioned 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. Other developers surveyed also mention code review process (14%) and project expertise (6%) as factors of catching indirect conflicts.

RQ6 *How to developers react to and solve indirect conflicts?*

Once an indirect conflict has occurred, usually alerted to the developer by test or build failures, 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. Developers also mentioned their personal expertise in the project as a large component of solving indirect conflicts as well as issue or work item trackers as many tests and builds can be linked to these management tools.

Of the developers surveyed, 37% confirmed that they used some type of source control management tools to help them solve their indirect conflicts while 23% said they used native IDE tools 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.

C. Tools for indirect Conflicts

RQ7 *When are preemptive indirect conflict tools best used and what would developers want from them?*

The developers who were interviewed had a major concern with any preemptive tool in that the amount of false positives provided by the tool may render the tool useless. Developers said "this could relate to information overload", "this would be a real challenge with the number of dependencies", "I only want to know if it will break me", and "it depends on how good the results are in regards to false positives". These concerns combined with RQ2 suggest that preemptive tools are best used in unstable phases on the project when indirect conflicts are more likely to happen, thus reducing the number of false positives. One developer in particular noted "You will spend all your time getting notified and reviewing instead of implementing". Interviewed developers also suggested that their proper software development process are already in place to catch potential issues before they arise such as code review, individual knowledge, IDE call hierarchies (static language analysis), or communication within the project. Finally, most interviewed developers mentioned that their current work flow does not subscribe to the ideas of preemptive prevention and that they just "work until something breaks".

As per what developers would want from such preemptive tools, Table III shows how developers feel towards being alerted in certain types of functional changes within code which they are using.

RQ8 *What kinds of tools do developers want to help them handle and solve indirect conflicts?*

The developers who were surveyed, the following list represents the more common suggestions for how an indirect conflict resolution tool should be implemented.

- Aid in debugging indirect conflicts by finding which recent code changes are breaking a particular area of code or a test.
- Automated testing to show how code changes affect old tests and the ability to automatically write new tests to compensate for the change.
- 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 or pull request 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

We discuss the results of Section III as follows: first we look at and discuss the nature of indirect conflicts by addressing RQ1 - RQ3, we proceed to talk about how developers structure their work flow around indirect conflicts currently by examining the results of RQ4 - RQ6, and we lastly investigate why developers have not had much interest in tools built previously for indirect conflicts and what we can do moving forward to accommodate developer work flow with potential tools by looking at RQ7 and RQ8.

A. The Nature of Indirect Conflicts

In order to build solution tools for indirect conflicts, we must first understand their nature; what causes them and how often do they occur. From the results we see in RQ1, a common theme of awareness is obvious. Developers cause indirect conflicts because they do not fully understand the ramifications of their source code change. This was especially apparent when developers were dealing with legacy code. Developers identified the changing of older code to be quite prone to indirect conflicts as they were not sure where the code was being used anymore. We would think this can be a simple fix with static analysis tools of source code to allow developers to identify which parts of the project they are affecting with a change. However, as we found out this is not always the easy fix. For starters, not all programming languages are subject to strong static analysis if any, while some projects involve many cross language dependencies which also breaks down any potential for static analysis on these software dependencies. Instances of upgrading libraries to newer versions was a common cause of indirect conflicts, and since these may be binary files, static analysis of the changing code is not always possible. The other side of upgrading libraries also posed a problem. Developers who were often tasked with updating older libraries had no knowledge of what other outside projects used these libraries and were therefore unable to understand how they may affect these outside projects with their changes.

We see from RQ3 that developers have shown that the human factor involved with indirect conflicts cannot be ignored. From Table II, is can clearly be seen that as a project becomes larger and more and more developers are added, even to the point that multiple teams have to be formed, indirect conflicts

TABLE III: Results of survey questions to source code changes that developers deem notification worthy, in terms of percentage of developers surveyed.

Code change type	Never	Occasionally	Most Times	Always	I Don't Care
Method signature change	5%	8%	12%	68%	7%
Pre-condition change	5%	27%	37%	23%	7%
Main algorithm change	11%	45%	19%	15%	11%
User interface change	12%	32%	20%	27%	9%
Run time change	13%	29%	25%	20%	12%
Post-condition change	7%	28%	32%	23%	11%

become more likely to occur. However, this is not to say that indirect conflicts do not occur at the lower number of developer level. Even when developing individual projects, 43% of developers agreed or strongly agreed that indirect conflicts are likely to occur. Once more than one developer is associated with a project, or a project has multiple teams, 67% and 74% of developers agreed or strongly agreed that indirect conflicts are likely to occur. So even when developers are working alone, they do not have enough awareness of their own code as to how a software change may affect another portion of their project. This can be due to any of the reasons mentioned above. The problem also seems to be compounded once multiple developers and teams are involved in parallel development schemes. The lack of awareness as to other developers work in a project again comes to light. This lacking is increased as project size trends up. It would appear possible that the addition of software developers to a project, which seems to compound the likely hood of indirect conflicts, developers would be able to leverage other developer expertise of the project to help avoid or solve indirect conflicts. However, we will see in Section IV-B that most developers do not leverage their social connections within a project to help solve indirect conflicts.

From RQ2, we can make the deduction that indirect conflicts are more likely to occur early in a projects development life cycle as opposed to later. We used to notion of the first major release to make the distinction between early and late in a project but the same hypothesis could be applied to iterative development where during the earlier stages of the current phase could be considered more probable for indirect conflict to occur. In Table I, we see that as the project becomes more mature, a drop off in the frequency of indirect conflicts occurs. Developers surveyed answered that daily and weekly indirect conflict frequencies trail off past the project's first release and those developers interviewed mentioned a point in the project, the "mature point", in which indirect conflicts become less frequent after. This would suggest that, as will be seen in Section IV-C, preemptive indirect conflict tools would be more useful before this point as the number of false positives will have a higher chance of being true positives. This "mature point" may be cause for one of two reasons. The first would be that the project's source code has stabilized quite a bit in terms of large changes not occurring as much. This could be a result of a project being released to a customer or because it is now being used as part of many other projects. The second would be that a change in development process has occurred in that source code is not being changed so much as newer code is being added or old code is being deleted. This would cause a slow in indirect conflicts as developers interviewed said that changes to code, not the addition of or deletion of code is prone to indirect conflicts. This two possibilities are not mutually exclusive. In the end, developers survey pointed out that before the "mature point" indirect conflicts are like to occur daily and weekly (32% and 18%) as opposed to after the "mature point" when developers said they are more likely to occur on a monthly or even yearly frequency (18% and 12%).

From the results of RQ3 we can also see evidence that the problemed areas of software architecture in regards to indirect conflicts are the interfaces between structures. As was seen, developers listed both methods/functions and database schema's as the largest contributors to indirect conflict structures. Since methods are often the interface which developers use most, it seems natural that they are identified as the main software structure in regards to indirect conflicts. It appears that when interfaces of methods change through natural software evolution, indirect conflicts are more prone to occurring. Most tools built so far have focused on the method level, and those that attempt to prevent indirect conflicts often look for a change in interface to a method, either in signature or in internal body composition. The more interesting result was that of database schema based indirect conflicts. There has been little focus of indirect conflicts at the relational database schema level [17], possibly due to the inherit difficult nature of analysis a database schema in relation to source code structure. Project often use unique libraries and procedures when dealing with database operations which causes a generic tool for this analysis to be difficult at best to implement. However, developers have noted that this is often a very problemed area and can affect large portions of the software project.

We theorize the nature of indirect conflicts with the following findings:

- F1 Indirect conflicts are caused by a lack of understanding and awareness of source code changes which are brought on by the natural evolution of software. This lack of awareness and understanding is compounded by having a multi developer or multi team environment.
- F2 Indirect conflicts occur more often before a project

reaches its “mature point” rather than after for each development cycle.

F3 Indirect conflicts are brought on by the changing of interfaces to software structures. The most common type of structures to cause indirect conflicts are methods/functions and database schemas.

B. Processes of Indirect Conflicts

To be able to accommodate developers in the prevention and solving of indirect conflicts, it is important to understand the current work flows of developers in regards to how they prevent, catch, and solve indirect conflicts. RQ4 shows us that most developers rely on two items in order to prevent indirect conflicts. The first of these items are development processes known as “Design by Contract”™and the add and deprecate methodology. These processes are implementation of software artifacts based on what guarantees should be made about their behavior, and instead of changing old code, the code is simply copied into a new method/function and then modified to make a new version while the old version gets deprecated and phased out later in the project’s life, respectively. “Design by Contract”™ensures that software artifacts, while they can be changed, should always behave in a way that is understood by other developers (the guaranteed aspects of the artifact), and should be documented in a way to reflect this behavior in order to avoid misinterpretations of the artifact. Most project’s which are past their “mature point” use add and deprecate development process as a way of preventing indirect conflicts of legacy code by not changing it. The second item is to have highly experienced team members be involved in the project planning in order to foresee where problems may arise and act accordingly. (Developers also mentioned their personal experience in preventing indirect conflicts.) These three tools are really the only measures that developers use right now in order to prevent indirect conflicts. This is probably due to the inherit difficulties in predicting indirect conflicts, as will be seen in Section IV-C. Even with proper process and expert knowledge, it is still improbable to predict every indirect conflict, and thus, conflicts continue to occur. However, developers did not seem overly concerned that these three items do not totally prevent indirect conflicts. This is because most developers ascribe to the idea of “I work until something breaks”. This shows that developers are somewhat content with the level of prevention that happens for indirect conflicts and would explain why preemptive indirect conflict tools have failed to catch on, as will be seen in Section IV-C. This again is probably due to the difficult nature of predicting indirect conflicts. RQ4 has mostly shown that developers would rather spend their time actually writing the software than trying to prevent only potential issues. Time is more effectively spent developing and fixing problems as needed than putting excess time into the prevention of potential errors.

The results of RQ5 show us how developers currently catch indirect conflicts. It appears that most indirect conflicts are currently detected by using what some would call proper software practices. The use of unit and integration testing, and the use of nightly or weekly project builds are the main contributing factors of such discovery. 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. This leads us to believe that as long as the contract of a software structure is understood, proper tests should be able to be written which will catch most if not all indirect conflicts. The problems occur when the contract is not properly understood and use cases are not tested for or examined as explained in previous discussion. Once an object is not properly tested, the negative consequences may not be caught until later in a project with the potential for more damage to be caused. Some languages such as Eiffle and the Java Modeling Language [3] enforce object contracts to be well understood through the ideas of pre and post conditions of structures. Aside from testing was the build process. Developers said that the build process is simply there to ensure a, at least, semi-functioning product. They use the build process more to catch compile time errors witch prevent the product from being used. These types of build breaking indirect conflicts do occur but for the most part developers posses tools capable to build their solutions in order to catch these problems before the code change is added to the project. Therefore, unit and integration testing appears to be the largest factor in catching indirect conflicts.

Finally, once the indirect conflict is caught, the results of RQ6 show us how developers tend to solve the issues. Here, the generic approach brought forth by developers was that of history checking. Developers would look into source code change logs and build or test failure logs in order to see what is broken and what might have changed recently that could be affecting the break. This gives the impression that developers already have the tools to solve indirect conflicts. However, this leads to the question: how quickly and efficiently are they able to solve them? While each developer gave some examples on solving indirect conflicts and the times needed, they all agreed that to be able to solve them quicker would be a huge benefit. A possible solutions as to solving indirect conflicts quicker are the automatic analysis of source code change logs to detect what changes will affect a certain part of a project’s code. Luckily, some of this solution has already been implemented by several researchers in both delta debugging [18] and program slicing [19]. The interest that developer’s have in fixing issues as they arise more quickly again shows that developer’s prefer fixing real problems directly as opposed to trying to prevent potential problems. Their time is more effectively spent developing than speculating.

We sum up the developer process with indirect conflicts with the following findings:

F4 Developers use a combination of object contract understanding, the add and deprecate model, as well as expert knowledge in order to prevent indirect conflicts.

[3]http://www.eecs.ucf.edu/ leavens/JML//index.shtml

F5 Developers use unit and integration testing to examine use case coverage in order to detect indirect conflicts.

F6 Developers prefer to spend time debugging real problems opposed to preventing potential problems.

C. Tools for indirect Conflicts

Ultimately, the goal of this research would be to produce any sort of tool that could aid developers in any stage (prevention, detection, or solving) of indirect conflicts. In order to address what types of tools would be best, we analyze the results from our previous research questions along with RQ7 and RQ8. We first examined the prevention / detection side of indirect conflicts through the use of preemptive indirect conflict tools. Preemptive indirect conflict tools are used to find potential code changes that could result in indirect conflicts. As was found through RQ7, developers major concern with preemptive indirect conflict tools was the number of false positives put forth by such a tool and the information overload that was sure to follow. Developers were very quick to identify this problem which has been a common occurrence among any preemptive indirect conflict tool built. The main issue with false positives is the detection of changes which frequently cause indirect conflicts. Change types are very possible to detect through software static analysis, but even those that are prone to causing indirect conflicts do not cause them all the time. This is quite evident in Table III where developers identified that there is an even distribution of software changes that could cause indirect conflicts and as to if they would want to be made aware of those changes. The only two exceptions which stand out are that of method signature changes and main algorithm changes. The problem here however, is that method signature changes will be caught by IDE errors, nightly builds, unit testing, integration testing and so on, while main algorithm changes are very difficult to detect. This all leads to the hypothesis that preemptive indirect conflict tools cannot be built without a very large number of false positives being reported at any given time. Even if developers, as per Table III, want to be alerted to changed preconditions in methods most times if not always, there is still a large chance that that changed precondition will actually not introduce an indirect conflict in the source code. This leads to a large number of false positive being reported, which leads to information overload, and eventually to developers abandoning the tool.

In order to prevent or catch indirect conflicts with an automated tool without introducing a large amount of false positives, would be to have an automatic understanding of how the project's code is suppose to work. From the halting problem [20], we know it is an impossible task to write one program that can verify the correctness of another. This leaves us with the only option of some kind of human intervention in the matter which leads us back to use case coverage. As was stated before, if the developer has a complete understand of the use cases of a particular code artifact, then they can write proper tests to ensure those use cases are always met and not broken which would be the best form of preventing and catching indirect conflicts.

Given the above challenges, some indirect conflict tools have had some general success. The key difference with these tools is that they contain the idea of developer exploration. The idea of only presenting information to users on request is the complete opposite of the preemptive notion of notifying developers to potential upcoming problems. The tools which provide information on user request usually involve taking some artifact of interest and presenting information which is dependent on the artifact. Here, the idea that a user can provide some sort of input, some sort of context, is a powerful tool for reducing false positives. We have really just moved the false positives of the preemptive tool to the "search results" of the exploratory tool, however, with the user input the results can be narrowed down to provide a larger precision of results. These types of tools are in direct comparison with the results of RQ8 in that developers were more interested in the debugging of issues quickly and being able to narrow down where problems emanate from. This also sticks with the findings found in Section IV-B being that developers would rather spend their time fixing and exploring a real issue when it does occur than trying to prevent potential issues moving forward. As researchers, we should be focusing our efforts on the solving as opposed to prevention and the detection of indirect conflicts. Prevention does not appear to worth developer time as indirect conflicts and software failures will always occur, detection will always require some form of human intervention to determine the correctness of a program, so this only leaves us with the time and effort spent on debugging.

We sum up our conclusions for future indirect conflict tools with the following findings:

F7 It is extremely difficult, if not impossible, to create a preemptive indirect conflict tool which can detect indirect conflicts with a low rate of false positives.

F8 Indirect conflict tools are best used by providing exploratory environments to aid in the solving of indirect conflicts.

F9 Indirect conflict tools should be reducing debugging time of indirect conflicts as opposed to preventative time in order to better fit developer activities.

V. Evaluation

As per grounded theory research, Corbin and Strauss list ten criteria to evaluate quality and credibility [15]. 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 IV 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 F1 - F6 may not be entirely novel or even surprising, the combination of these results allow us to discover the insightful findings of F7 - F9 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 [21]. 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. [22] 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 [23] 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. [24] 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.

While not specifically developed with indirect conflicts in mind, Zeller's delta debugging techniques [25], 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 [26] 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 F1 - F3 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 F4 - F6 have shown how developers in industry currently handle the prevention detection and solving of indirect conflicts. And Lastly, findings F7 - F9 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.

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. [Online]. Available: http://doi.acm.org/10.1145/383876.383878

[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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] A. Maule, W. Emmerich, and D. S. Rosenblum, "Impact analysis of database schema changes," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 451–460. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368150

[18] A. Zeller, "Isolating cause-effect chains from computer programs," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 6, pp. 1–10, Nov. 2002. [Online]. Available: http://doi.acm.org/10.1145/605466.605468

[19] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/358557.358577

[20] L. Burkholder, "The halting problem," *SIGACT News*, vol. 18, no. 3, pp. 48–60, Apr. 1987. [Online]. Available: http://doi.acm.org/10.1145/24658.24665

[21] 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. [Online]. Available: http://doi.acm.org/10.1145/1984642.1984662

[22] 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. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368122

[23] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486877

[24] 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.

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

[26] 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. [Online]. Available: http://doi.acm.org/10.1145/1050849.1050865