State of The Practice in Indirect Conflicts: Causes, Mitigation Strategies, and Future Directions

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

***Abstract*—Awareness techniques have been proposed and studied to aid developer understanding, efficiency, and quality of software produced. Some of these techniques have focused on either *direct* or *indirect conflicts* in order to prevent, detect, or resolve these conflicts as they arise from a result of 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 common issues of information overload, false positives, and scalability. To better understand these issues, we performed a mixed method study involving interviews with 19 developers from 12 commercial and open source projects, and a questionnaire with 78 respondents different from our interviewees. Using a grounded theory approach to data analysis, we obtained insights about events and conditions in which indirect conflicts are likely to occur in software development, that developers prefer to use detection and resolution processes or tools over those of prevention, that they do not want awareness mechanisms which provide non actionable results, and that there exists a gap in software evolution analytical tools.**

I. Introduction

As Software Configuration Management (SCM) has grown, 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, empirical evidence shows that despite of recent research has proposed mitigation strategies, these losses continue to appear quite frequently and can prove to be a significant and time-consuming chore for developers [2], [3].

Two types of conflicts have attracted the attention of researchers, *direct* and *indirect conflicts*. Direct conflicts involve immediate workspace concerns such as developers editing the same artifact. Tools have been created and studied for direct conflicts [4], [5], [3], [6] with relatively good success and positive developer feedback. Indirect conflicts (ICs) are caused by source code changes that negatively impact another location in the software system such as when libraries are upgraded or when a method is changed and invoking methods are influenced negatively. Indirect conflict tools however, have not shared the same success as direct conflict tools [7], [8], [9], [10], [11].

While indirect conflict tools have shown potential from studies of developers, some of the same problems continue to arise throughout most, if not all tools. The most prevalent issue is that of false positives and information overload, which causes tools to eventually be ignored [7], [10]. A second primary issue is that of code dependency identification and tracking. Many different dependencies have been proposed and used in indirect conflict tools such as method invocation [9], and class signatures [7] with varying success; however, the identification of failure inducing changes, other than those which are already identifiable by other means such as compilers or unit tests, to these dependencies still remains an issue. Some tools [12], [13], [11] have also leveraged the notion of socio-technical congruence [14] to help developers pursue dependencies through social means. However, the same aforementioned problems still arise. Evaluations of these tools or approaches have largely been small in scale, and involved a limited number of industrial participants or student evaluators. As a result, the problems of scalability, information overload and false positives in tackling indirect conflicts remains an outstanding research problem.

In this paper we report from a field study of software practitioners that sought to characterize the state-of-the-practice in indirect conflicts, particularly their causes and mitigation strategies. Our intention has been to uncover the root causes of information overload, false positives, and scalability issues, in order to better understand why indirect conflict tools and methods fail to fully achieve their intended success. Three research questions that guided our study:

- RQ1 *What are the types, factors, and frequencies of indirect conflicts?*
- RQ2 *What mitigation techniques are used by developers in regards to indirect conflicts?*
- RQ3 *What do developers want from future indirect conflict tools?*

To answer these questions, we interviewed 19 developers from across 12 commercial and open source projects, sought confirmation of the themes we identified in our interviews through a questionnaire with 78 developers, as well as 5 follow-up interviews. Our findings indicate that:

- Indirect conflicts occur frequently and are likely caused by software contract changes and a lack of understanding of such changes
- Developers tend to prefer to use detection and resolution processes or tools over those of prevention
- Developers do not want awareness mechanisms which

provide non actionable results
- There exists a gap in software evolution analytical tools from the reliance on static analysis resulting in missed context of indirect conflicts

II. Related Work

Many indirect conflict tools have been developed, tested, and published. Sarma et al. [7] created Palantir, which can both detect potential indirect conflicts, at the class signature level, and alert developers to these conflicts. Palantir represented one of the first serious attempts at aiding developers towards indirect 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. This tool and its taxonomy had a severely reduced false positive rate, however, the true positives detected may already be detectable by current tools such as compilers and unit testing. 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, using method call graphs, showing where indirect conflicts may occur. This type of exploratory design has been used often in the visualization of indirect conflict tools, allowing developers a type of search area for their development needs. Another indirect conflict tool, CASI [10], utilizes dependency slicing [15] instead of method call graphs to provide an environment to see what areas of a project are being affected by a source code change. The evaluations of these tools (including our own prototype Impact! [16]) indicate that they have the same common difficulties of scalability, false positives, and information overload.

Following Cataldo et al. [14]'s work that indicated that socio-technical congruence can be leveraged to improve task completion times, many indirect conflict tools support the idea of a socio-technical congruence [12] in order to help developers solve their indirect conflict issues through social means [13] [11]. Begel et al. [13] created Codebook, a type of social developer network related directly to source code artifacts, which can be used to identify developers and expert knowledge of the code base. Borici et al. [11] created ProxiScientia which used technical dependencies between developers to create a network of coordination needs. However, again the problems of scalability and information overload become a factor.

For developer interest in regards to software modifications, Kim [17] found 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. Kim found that developers are concerned with interfaces of objects and when those interfaces change. 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". Kim's field study does however fall short in actually creating a resolution for indirect conflicts, or finding new concerns of developers which are not already detected by compilation or other static analysis tools. In studying awareness of change impact, deSouza et. al. [18] 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. However, their work does not study which types of changes (types, frequencies, compounding factors) developers encounter in their work, nor provides concrete information on formal mitigation strategies or resolutions of indirect conflicts.

The insights of our study intends to fill the gap. In our field study we seek to understand not only why information overload, false positives, and scalability are such difficult problems to tackle in indirect conflict tools, but also how developers currently deal with indirect conflicts in practice through their mitigation strategies, what are their major concerns as well as suggestions for future research and tools to tackle these concerns.

III. Methodology

We performed a mixed method study in three parts. First, a round of semi-structured interviews were conducted which addressed our 3 main research questions. Second, a questionnaire 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. Third, 5 confirmatory interviews were conducted by re-interviewing original interview participants to once again confirm our insights. We used grounded theory techniques to analyze the information provided from all three data gathering stages.

A. Interview Participants

Our interview participants came from a wide range of both open and closed source software development companies and projects, using both agile and waterfall based methodologies, and from a wide spectrum of organizations, as shown in Table I. The participants had software development experience in average of 8 years, as well as some project management experience.

B. Interview Procedure

Participants were invited based on their software engineering and project management experience. We recruited participants through an email message to a number of software organizations in our working business network, and followed up in a single reminder email one week after the initial invitation if no response was made. We directly emailed 22 participants and conducted 19 interviews. Interviews were conducted in person when possible, or over video chat when not possible, and recorded for audio content only.

Interview participants first answered a number of demographic questions and were then asked to describe various software development experiences and practices. Specifically, ten semi-structured questions related to our research questions where ask and guided our interview:

- What tools are used for dependency tracking?
- What processes are used for preventing ICs?

TABLE I: Demographic information of interview participants.

Company	# of Participants	Software Development Experience (years)	Development Process	Software Access	Current Language Focuses
Amazon	2	5, 7	Agile	Closed source	C++
Exporq Oy	1	8	Agile	Closed source	Ruby, JavaScript
Fireworks Design	1	6	Agile	Closed source	JavaScript
Frost Tree Games	1	4	Agile	Closed source	C#
GNOME	1	13	Agile	Open source	C
James Evans and Associates	5	3, 3, 3, 4, 13	Waterfall	Closed source	Oracle Forms
Kano Apps	1	10	Agile	Closed source	JavaScript, PHP
IBM	2	5, 18	Agile	Open and closed source	Java, JavaScript
Microsoft	2	6, 10	Agile	Closed source	C#
Mozilla	1	25	Agile	Open source	C++, JavaScript
Ruboss	1	5	Agile	Closed source	JavaScript
Subnet Solutions	1	5	Agile	Closed source	C++

- What artifact dependency levels are analyzed and where can conflicts arise?
- How are software dependencies found?
- Give examples of ICs from real world experiences.
- How are ICs detected or found?
- How are IC issues solved or dealt with?
- Opinion of preemptive measures to prevent ICs.
- What types of changes are worth a preemptive action?
- Who is responsible for fixing or preventing ICs?

Not all participants had strong opinions or any experience on every category listed above. For these participants, answers to the specific categories were not required. 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 part of any one software project for extended periods of time.

C. Questionnaire Participants

Our questionnaire respondents were different from our interviewees. We recruited the respondents to our questionnaire from a similar breadth of open and closed source software development companies and projects as the interviews participants with two main exceptions. The GitHub users were recruited from the Google BigQuery database; the Apache respondents were recruited from the list of tops contributors to the Apache Software Foundation. The software organizations that remained the same between interview and questionnaire were: Mozilla, The GNOME Project, Microsoft Corporation, Subnet Solutions, and Amazon. 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.

D. Questionnaire Procedure

Questionnaire participants were invited to participate in the questionnaire by email. We directly emailed 1300 participants and obtained 78 responses giving a response rate of 6%. We attribute the low response rate with the questionnaires being sent out during the months of July and August when many participants may be away from their regular positions.

We designed the questionnaire [1] to follow the interviews. We intended to confirm some of the insights we obtained from the interviews with a larger sample size of developers who may have similar or different opinions from those already acquired from the interviews. The question topics asked in the questionnaire were:

- What frequency do ICs occur at around different project milestones?
- How does team size affect IC frequency?
- What software change types do developers care about?
- What processes are used for preventing ICs?
- What tools are used for detecting ICs?
- What tools are used for debugging ICs?

E. Data Analysis

To analyze both the interview and questionnaire data we used grounded theory techniques as described by Corbin and Strauss [19]. Grounded theory is a qualitative research methodology that utilizes *theoretical sampling* and *open coding* to create a theory "grounded" in the empirical data. In this study we used grounded theory techniques for analysis and not to

[1] http://thesegalgroup.org/projects/indirect-conflicts/

develop a theory per say. For an exploratory study such as 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 questionnaires (when open ended questions were involved) was analyzed using open and axial coding (relating code and concepts to each other). In Section VI, we give a brief evaluation of our studying using 3 criteria that are commonly used in evaluating grounded theory studies.

F. Validation

Following our data collection and analysis, we re-interviewed 5 of our initial interview participants in order to validate our findings. We confirmed our findings 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 main discussion points, and asked open ended questions regarding their opinions and experiences to validate our findings. We also evaluated our grounded theory approach as per Corbin and Strauss' [19] list of criteria to evaluate quality and credibility, described in Section VI.

IV. Results

In this section we present our results of both the interviews and questionnaires conducted in regards to our 3 research questions outlined in Section I. For each research question we summarize the quantitative and qualitative evidence we obtained. In each subsection, quantitative data given refers to interviews conducted unless explicitly said otherwise.

RQ1. *What are the types, factors, and frequencies of indirect conflicts?*

The most common occurrence of indirect conflict is when a software object's contract changes. The frequency of indirect conflicts, while usually high, decreases as a stable point is reached in the development cycle. The frequency of indirect conflicts is compounded by the number of developers working on a project.

We found that a large contributing factor to the cause of indirect conflicts comes from the changing of a software object's contract (reported by 12 developers directly). Object contracts, or "Design by Contract"™, a concept made well-known through the work on Eiffel Software's [2] are, in a sense, what a software object guarantees, meaning how the input, output, or how any aspect of the object is guaranteed. In our interviews, 14 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 14, 3 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 contracts stated "there are no changes in the 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. 7 developers mentioned that indirect conflicts occur when a major update to an external project, library, or service occurs with one developer noting that "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.

11 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. The questionnaire showed that, 64% of indirect conflicts occur bi-weekly or more frequent, with 25% saying that weekly occurrences are most common (seen in Table II). 5 questionnaire participants also stated that the stage of the development cycle can greatly cause the frequency of indirect conflicts to differ.

12 developers reported 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. For example, "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". From the survey respondents we also learned that "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." which corroborate our interview results. Questionnaire data also showed 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 II.

In terms of organizational structure, questionnaire data shows 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, as shown in Table III, indirect conflicts still occur in small teams, with 43% of respondents indicating they are likely to occur in single developer projects.

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

TABLE II: Results of questionnaire as to how often indirect conflicts occur, in terms of percentage of questionnaire participants.

Occurrences	Daily	Weekly	Bi-Weekly	Monthly	Bi-Monthly	Yearly	Unknown
In general	18%	25%	21%	16%	3%	5%	11%
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%

TABLE III: Questionnaire results about development environments in which indirect conflicts are likely to occur, in terms of percentage of questionnaire participants.

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%

RQ2. *What mitigation techniques are used by developers in regards to indirect conflicts?*

Three preventative techniques are used to mitigate indirect conflicts: design by contract, add and deprecate, and personal experience. For catching indirect conflicts, developers use: testing (unit and integration) with proper use case coverage, scheduled build processes, and static analysis tools built into IDEs. For resolving indirect conflicts, developers use: historical error and information logs, compiler and debugger tools, as well as static analysis tools built into IDEs.

Our analysis indicates indicates three major preventative processes used for indirect conflicts. First, design by contract is heavily used by interviewed developers as a means to avoid indirect conflicts or to 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 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 7 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 questionnaire participants 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. This is a confirmation of what was found in the interviews.

In regards to catching indirect conflicts, 13 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. In corroboration, questionnaire participants corroborated 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. Questionnaire participants 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, 14 developers 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. Questionnaire data show that 23% of developers use 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

TABLE IV: Questionnaire results about source code changes that developers deem notification worthy, in terms of percentage of questionnaire participants.

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%

mentioned a form of communication with other developers in aid to resolving 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 tools 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".

RQ3. *What do developers want from future indirect conflict tools?*

Because developers believe "good enough" solutions exist for preventing and detecting indirect conflicts, we identified a strong preference for improvements to debugging processes. Developers prefer automated debugging tools and better source code analysis tool for impact management and supporting cross team and cross sub-project projects.

When asked about preventative tools, 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, 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. Further, developers said that having strict change type monitoring for indirect conflicts does not work due to the inherent complexities of indirect conflicts. This can be seen corroborated in that questionnaire participants had very few cases in which they always wanted to be alerted to a change type as seen in Table IV. Only method signature changes caused 68% of questionnaire participants to want to be always notified as they have a high chance to break the code. This is opposed to other change types listed in Table IV which have 27% or lower of developers always wanting to be notified. This again showcases the complexity of indirect conflicts through 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. Interview and questionnaire participants 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. Eiffel or Java Modeling Language [3]) 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.

V. Discussion

We now discuss these results in relation to the outstanding issues of tool-based solutions regarding indirect conflicts and which are information overload, false positives, and scalability. As per our findings, three main discussion points emerge: (1) What developers really want and don't want from indirect conflict awareness tools, (2) Whether indirect conflicts are better handled by extensively trying to prevent them or by simply fixing them as they occur with less prevention, and (3) A gap in analytical tools for software dependencies found in current industry settings.

A. Unwanted Awareness

Developers tend to only care about the awareness of other's activities, if it causes a negative influence on their own

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

work. Developers only want to hear about another developer's actions if it forces them to take some action because of it. This limited awareness is quite different that what literature suggests, which is larger awareness about most, if not all actions, and it also suggests why developers see a high amount of what they believe to be false positives, as the changes being reported are not causing them to take action.

According to our study participants, indirect conflicts represent a serious problem that occur frequently, sporadically, and differ greatly from case to case. These conditions pose large issues for the creation of generalizable theories or tools in regards to indirect conflicts. These underlying complexities are the probable cause of disinterest of software developers to proposed or implemented tools as discussed in Section II. This inability to generalize is what we believe to be the leading cause of information overload and false positives awareness systems, causing developers to eventually ignore information being presented to them, rendering the system useless.

These false positives are caused by a difference in what developers consider to be false positives versus what awareness literature considers they are. This disjoint, we argue, is caused by developers only considering events which cause some action to be taken on their part, to be true positives, where as current awareness understanding would state any event which is related to an individual's work [20] [21], to be true positives. "You need a good understanding of what the code change is or else you will have a lot of false positives" said one developer, showing that not all changes around an object should be reported for awareness.

Developers have been found to have a great understanding of what they need to know about and more importantly what they don't in their project awareness. To be able to fully understand a developer's awareness intuition, we can see from the results of this paper that developers only want awareness of an event if the event forces the developer to take some action.

Developers don't care about what they don't need to account for. "We would want a high probability of the [reported] change being an issue". This sense of unwanted or limited awareness is crucial to understand why generalized awareness techniques of difficult to generalize problems, such as indirect conflicts through generalized changes to classes [7], or methods [9], [10], often encounter difficulties of false positives and information overload. Developers simply do not want to know about events which effect them, but require no action on their part.

This unwanted awareness, or limited awareness, seems counter intuitive to current awareness understanding which would state that being aware of all events in ones surrounding leads to higher productivity or other quality aspects. In theory this is correct, but as it was found in practice, this is an incorrect assumption. In regards to this full awareness, one developer said "There is no room for this in [our company], as tools are already in place for analysis of change[s] and code review takes care of the rest". Since software developers have limits on their time, awareness of all events surrounding a developer's work or project is not possible. In terms of developer communication, 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. **Developers prefer to spend their limited time dealing with the awareness of events which cause them to take some action (changing code, communication, etc) rather than simply being aware of events occurring around their work which pose no direct threat to the consumption of their time.**

Of course, whether or not this unwanted awareness is the correct path for developers to take is an open question to consider. When developers encounter a problem which could have been solved by having greater awareness of events which did not directly affect them initially, we must consider the positive and negative influences of adding this, for now, unwanted awareness. A positive influence of total, or near total awareness of events at the developer level, would be the full understanding of a developer's work and environment which comes with higher quality or understanding of the product. A negative influence would be that valuable developer time is spent understanding events which may not directly apply to themselves as opposed to producing more output of their own work. This balance between awareness and productivity is found to be a fine line in practice, however, when given the choice, developers tend to lean towards less awareness in order to be more productive in their eyes.

B. Prevention versus Cure

Developers would rather spend their time on curative measures (fixing problems after they arise) than preventative measures (preventing problems from happening). Developers see the task of fixing real problems to be less of a burden than preventing potential problems because of available tools and their perceived notions of time management. Focusing on cure is a direct response to the issue of scalability found in indirect conflict tools. Trying to prevent all conflicts is too large a problem to handle, while curing existing conflicts is manageable by a developer.

Our findings suggest that developers possess both tools and practices for the prevention, detection, and resolution of indirect conflicts.

Through unwanted awareness, and the use of developer time, developers tend to prefer working on real issues that have already occurred as opposed to preventing issues of the future which may never arise. Related to the popular adage "I work until something breaks" taken by most developers, this mindset is a clear example of developers taking a curative approach to software development opposed to a preventative approach. Prevention here refers to taking precautionary steps to stop potential or some probability of issues, or indirect conflicts, form occurring in the first place, while cure refers to fixing non probable (have happened) issues as they arise which includes not attempting, or putting little attempt, into preventing them in the first place.

Two out of the three identified prevention methods taken by developers are simple blanket risk mitigation strategies accomplished essentially by not changing code (design by contract, and add and deprecate), while the third is simply developer experience and knowledge. Clearly, developers are spending little to no time in prevention (or what the perceive as prevention). Developers do however spend a large amount of time in detection and cure through the writing of tests and the debugging of issues (although writing tests can be viewed as prevention while their utility is only see at detection). In fact, most improvements mentioned by developers in regard to dealing with indirect conflicts occurred at either the detection or cure levels. "You're reaction time is much more crucial" said one developer in that resolution tools are believed to be or larger importance as once issues have occurred, it doesn't matter what prevention was taken, the issue must simply be resolves as quick as possible now.

The lack of prevention process and tools being used is believed to be due to 2 factors as discovered in the interview process. The first being the identification of dependencies. Even with an experienced system architect, identifying dependencies and notifying those involved is a daunting task which is ignored more than dealt with. The second being the knowledge of when a dependency will fail, also requires vast knowledge of the product, more than anyone may have. This is compounded by the unique and sporadic nature of indirect conflicts. Hattori et. al. [22] 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, which is the situation many indirect conflict tools have strived for, however the 2 factors of dependencies and knowledge of failure are what ultimately have led to the amount of false positives and information overload seen in previous tools. **The abundance of detection and curative process and tools shows once again developers willingness to debug real issues, that maybe even could have been prevented, rather than prevent the issues in the first place. The question ultimately presented in this area is if curative measures are more productive than preventative measures.**

Dromey [23] has raised the debate of prevention versus cure in software development and how difficult a problem it is to measure. The pros and cons of prevention versus cure we have identified, are similar to those of unwanted awareness, in that they result in a trade off of where time is spent and how productive each side of the argument truly is. If prevention can be shown to be more effective in that it reduces the number of indirect conflicts or time spent debugging them compared to the time taken to prevent them, should we then not be thinking of how to move developers into a more preventative mindset. If the inverse is true, should we not be putting more time into automatic debugging, such as Zeller's Delta Debugging [24] or program slicing techniques [25], or automatic test case creation.

A last interesting observation about current curative measures being taken in industry, is that developers view testing of software as curative, when one could easily make the argument that it is preventative in that most tests are written to pass originally and are kept in place to ensure future changes do not cause issues. This mindset may come from the notion that writing tests is originally thought of as part of normal code writing, meaning that developers see the extra task of test coverage as part of feature implementation. This may suggest that if, given the right tools, developers may no longer view preventative measures as a "burden" and may be more inclined to take a more balanced preventative approach rather than mostly curative, as "prevention is the goal" was commonly said by developers.

This prevention versus cure discussion resides purely on the developer level and should be noted that it may not apply to system designers, architects, or managerial stakeholders. From the project manager's interviewed it was shown that they are heavily favored towards planning and prevention (even though their prevention may be on more abstract levels than developers actually need) while leaving the curative approaches to their developers.

C. Gap in Software Evolution Analysis

Due to a lack of productive software analysis tools, caused by project infrastructures and unfriendly analysis languages, contextual indications of indirect conflicts are often missed. This lack of analysis also causes some software project configurations and software languages to be quite prone to indirect conflicts. This is yet another issue related to scalability of existing indirect conflict tools.

Indirect conflicts are more likely to occur, from developer opinion, before a mature point is reached in the project's evolution. "Once you get the API stable, people are better at communicating changes in regards to dependency concerns."(This mature point may stem from a major release, end of an iteration, a new feature being released or any point considered stable and reliable) However, this context of a mature point, as well as any other potential contextual attributes, are rarely identified or accounted for in regards to indirect conflicts. Developers have said that different change types may occur at different rates throughout a project's life time and that this may drastically effect the outcomes of indirect conflict tools or processes.

A deeper understanding of what context indirect conflicts occur in seems to be more of a success factor to indirect conflict research than may have been previously thought. Static analysis may be useful in regards to indirect conflict context in that we could identify trends surrounding the mature points of previous projects in order to give a better understanding of what it means for a project to be pre or post mature point, which could affect the outcome of indirect conflict understanding. Fluri et al. [26] uses static analysis and abstract syntax trees to be able to determine what types of source code changes occur at a given point in time which could be used for determining such a context of mature points.

This added level of context, would add to the prevention versus cure debate as previously discussed. Prevention may be a better choice once a project has reached a mature point

as the code base becomes more stable and source code changes become more dangerous to the quality of the project. Curative measures may be a better choice before a mature point as code churn can be higher which causes more bugs than can be prevented. It may be imperative to discover more on project stability and the mature point in order to fully understand the nature of indirect conflicts and their context.

While this added sense of understanding is important to indirect conflicts, an understanding of a project's evolution, a mature point being reached as an example, may pose as a more useful tool for project management rather than developers. This more abstract tendency of indirect conflict occurrences may add even more power to project management for evaluation of progress, code stability, and code reviews.

Aside from context, dependency identification and tracking is a key missing component of indirect conflict analysis due to the weaknesses of static analysis. The gap in this identification comes from software and organizational structures of software teams. **Between increased modularization (multiple sub projects or repositories), cross language dependencies, and languages which do not lend themselves to static analysis [27], static analysis tools have become quite limited in identifying and tracking dependencies where they were once strong.** In a study regarding static analysis tools, Johnson et. al [28] found similar conclusions towards the ill fit of current static analysis to current development practices..

Relational database schemas are one of the highest sources of indirect conflicts found in their projects, "it breaks stuff all over the place", yet we know from Maule et al. [29] that relational database schemas have been scarcely researched in terms of indirect conflicts. This falls in with our previous understanding of cross language dependency tracking in that database schemas are independent of languages which may be on the receiving end of their output.

These increases of complexity in software products has left quite a gap in dependency identification and tracking which has lead to some of the deficiencies of indirect conflict research.

D. Implications for Future Research Work

Prevention versus Cure *There is a need for continued study of the open question "Which of prevention or cure is more effective for software development and indirect conflicts?". We believe that this simple question will undoubtedly produce extremely complex answers. Developers are currently favoring cure, but we should be mindful that what developers want may not always be the best choice.*

The fact that software developers focus their current efforts on curative measures, for indirect conflicts, suggests that while it may not be the most effective, it may be the easiest road for developers to take. This may be the path of least resistance, but it may not be optimal. Software engineering as a whole should strive to answer this question or provide more insight into possibilities, as its answer may determine many future actions taken by the research community.

Recommendations towards the study of prevention versus cure involve the examination of formal processes and tools used by industry professionals with measurements of efficiency and effectiveness, similar to the work of Tiwana [30] in coordination tools. With these studies, we may find insight into the correct balance of prevention versus cure, thus being able to increase developer productivity as well as identify more gaps in theory versus practice which may lead to improved tools or the abandoning of existing ones as was shown with UML [31].

Theories of Awareness to Model Real World *There is a need to further characterize the mismatch between awareness approaches and tools as developed in the research community and awareness needs as perceived by industry. Theories of awareness should consider modeling real development environments where interruptions, time management, and full development life cycles are often large factors.*

As was identified in the previous discussion, while current understanding suggests that awareness of all events related to ones work produces a more coherent understanding of a person's environment, developers find this to be overly time consuming to the point where they only want to be aware of events which require action on their part. This difference, of what should be and what is, should be further understood to combat future potential failures in tool development or theories attempted to be used in practice.

E. Implications for Future Tool Work

Give Developers What They Want *Developers have their own notions of how "good" something is for their productivity. We should pay attention to what they want, in order to shape our models and better address their needs.*

The direct implication this research has on tool development is that of current adoption among developers. As was stated, developers are more keen to invest their time at the detection and cure / resolution stages of indirect conflicts. We believe that focus at these two stages for tool developer will lead to larger adoption among developers. Similarly, detection must come with an almost zero rate of false positives as current tools (such as unit and integration tests) only report real problems with almost 100% precision (even though their recall may not be 100%).

Stronger Source Code Analysis *With languages like JavaScript becoming popular, having projects with multiple languages, and more modular structures, we should focus on better techniques for static analysis based on what industry standards.*

The more indirect implication of this research on tool creation is that of improving existing tools. While not all existing tools are used for indirect conflicts alone (automatic debugging [32], unit tests, etc), most of these tools have a need for rapid expansion, according to developers, for dealing with indirect conflicts. The ability to have unit tests automatically written for a given software object's contract, the ability to find a change in an external project which has broken a developer's own project, or any automation of the existing detection and

resolution stages of indirect conflicts are what developers currently seek. But of course, most of these implications rely on the improvement of static analysis tools.

These tool implications themselves imply the need of further development of static analysis tools. Static analysis lays at the heart of most if not all stages of indirect conflict research. We must be able to track and manage software dependencies across the new landscape of software development that is multiple projects, repositories, and cross language support. These improvements will allow the further development of both current and future indirect conflict tools.

VI. Evaluation and Threats to Validity

From the re-interviewed participants, we found extremely positive feedback regarding both our results and major discussion points. Participants often had new stories and experiences to share once they had heard the results of this paper which confirmed the findings and often were quite shocked to hear the results as they did not usually think about their actions as such but then realized the results held true to their daily actions for better or worse.

As per grounded theory research, Corbin and Strauss list ten criteria to evaluate quality and credibility [19]. We have chosen three of these criteria and explain how we fulfill them. The three evaluation criteria selected are more relevant to a qualitative study than the usual threats to validity associated with qualitative work. These selected criteria can also be viewed as the threats to validity of this paper.

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.

As added insurance, we conducted 5 confirmatory interviews after we analyzed our data and drafted themes in our findings. We describe this procedure in Section III.

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 results may not be entirely novel or even surprising, our findings do (1) suggest a disjoint between the current research understanding of developers' awareness needs and the these awareness needs related to indirect conflict tools in practice, and (2) provide some insight into the debate of prevention versus cure in software development. Given that many indirect conflict tools still suffer from these 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.

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 interviewed participants came from such a diverse set of organizations, software language knowledge, and experience the variation naturally reflected the complexity. Often in interviews and questionnaires, 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 Section IV as some unique findings or highly specialized cases. These cases 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.

VII. Conclusions

Indirect conflicts are a significant issue with real world development, however, many of our research proposed techniques and tools 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 questionnaires respondents, we sought to characterize indirect conflicts, current developer work flows surrounding indirect conflicts, and frame directions for future tool development that support developers in their tackling of indirect conflicts.

Our findings indicate that indirect conflicts occur frequently and are likely caused by software contract changes; while design by contract, add and deprecate, and personal experience help prevent indirect conflicts, developers tend to prefer to use detection and resolution processes or tools over those of prevention, and developers want indirect conflict tools to focus on automatic debugging and better source code analysis.

Our study of practitioners intends to fill a gap in the software engineering research with respect to the outstanding problems in current tool-based solutions to indirect conflicts, namely the issues of information overload, false positives and scalability. Overall, we described a disjoint between the current research understanding of developers' awareness needs and these awareness needs related to indirect conflict tools in practice. More specifically, our findings suggest that developers do not want awareness mechanisms which provide non actionable results; developers would rather focus on curing existing problems rather than preventing potential issues; and that there exists a gap in software evolution analytical tools between what is available and what practitioners need. We hope that these insights will be found valuable in future research endeavours and development of tools to assist developers in tackling indirect conflicts.

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] 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.

[4] 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.

[5] 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.

[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] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] J. Ell and D. Damian, "Supporting awareness of indirect conflicts with impact," University of Victoria, Tech. Rep. DCS-351-IR, 2013.

[17] 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.

[18] 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.

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

[20] J. D. Herbsleb, A. Mockus, and J. A. Roberts, "Collaboration in software engineering projects: A theory of coordination," in *In Proceedings of the International Conference on Information Systems (ICIS 06*, 2006.

[21] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ser. ESEM '08. New York, NY, USA: ACM, 2008, pp. 2–11.

[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] R. Dromey, "Software qualityprevention versus cure?" *Software Quality Journal*, vol. 11, no. 3, pp. 197–210, 2003.

[24] A. Zeller, "Isolating cause-effect chains from computer programs," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 6, pp. 1–10, Nov. 2002.

[25] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982.

[26] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[27] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the html dom and browser api in static analysis of javascript web applications," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 59–69.

[28] 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.

[29] 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.

[30] A. Tiwana, "Impact of classes of development coordination tools on software development performance: A multinational empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, pp. 11:1–11:47, May 2008.

[31] M. Petre, "Uml in practice," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 722–731.

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