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

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

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

I. Introduction

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

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

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

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

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

RQ1 *What is the nature of indirect conflicts?*
RQ2 *What types of process and tools are being used by developers in regards to indirect conflicts?*
RQ3 *How should indirect conflict tools work to best benefit developers?*

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

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

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

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

II. Methodology

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

A. Grounded Theory

We approached our study based on grounded theory as described by Corbin and Strauss [17]. Grounded theory is a qualitative research methodology that utilizes *theoretical sampling* and *open coding* to create a theory "grounded" in the empirical data. For an exploratory study such as the ours, grounded theory is well suited because it involves starting from very broad and abstract type questions and making refinements along the way as the study progresses and hypotheses begin to take shape. Grounded theory involves realigning the sampling criteria throughout the course of the study to ensure that participants are able to answer new questions that have been formulated in regards to forming hypotheses. In our study being presented, data collected from both interviews and surveys (when open ended questions were involved) was analyzed using open coding. Open coding involves assigning codes to what participants said at a low sentence level or abstractly at a paragraph or full answer level. These codes were defined as the study progressed and different hypotheses began to grow. We finally use axial coding it order to link our defined codes through categories of grounded theory such as context and consequences. The ultimate goal of grounded theory is to produce a theory that is tied to the empirical data collected; our final theories can be seen in findings F1 - F8. In Section IV, we give a brief evaluation of our studying using 3 criteria that are commonly used in evaluating grounded theory studies.

B. Interview Participants

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

C. Interview Procedure

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

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

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

While each of the 10 question categories had a number of starter questions, interviews largely became discussions of developer experience and opinion as opposed to direct answers to any specific question. However, not all participants had strong opinions or any experience on every category mentioned. For these participants, answers to the specific categories were not required or pressed upon. We attribute any non answer by a participant to either lack of knowledge in their current project pertaining to the category or lack of experience in terms of being apart of any one software project for extended periods of time. We account for these non answers in our analysis and results as seen in Section III. Interviews lasted from 15 minutes up to 75 minutes and were very much dependent on the experience of the participant.

D. Survey Participants

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

E. Survey Procedure

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

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

F. Evaluation

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

III. Results and Discussion

We now present our results of both the interviews and surveys conducted in regards to our 3 research questions outlined in Section I. We list each finding followed by our qualitative and quantitative results from which we draw the finding which are then followed by discussion.

A. What is 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 towards to how indirect conflicts occur, when and how often they occur, and the types of software object's that incur indirect conflicts.

F1 Indirect conflicts are caused by a lack of understanding and awareness of source code changes which are brought on by the manipulation of software contracts. This lack of awareness and understanding is compounded by having a multi developer or multi team environment.

From the interviewed participants, 63% of developers believe that a large contributing factor to the cause of indirect conflicts comes from the changing of a software object's contract. Object contracts are, in a sense, what a software object guarantees, meaning how the input, output, or how any aspect of the object is guaranteed to work; made famous by 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%, 21% dealt with the changing of legacy code, with one developer saying "legacy code does not change because developers are afraid of the long range issues that may arise from that change".

From these listed results, we see a common theme of awareness is obvious. Developers cause indirect conflicts because they do not fully understand the ramifications of their source code change, especially when a software object's contract is being changed. This was largely apparent when developers were dealing with legacy code, or another developer's code in general. The non-understanding of how implementation decisions were previously made is a large factor and that changing older code was quite prone to indirect conflicts as they were unsure of where the code was being used. This presents an immediate problem regarding indirect conflicts in software maintenance and we would think the simple fix would be static analysis tools for 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. 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. Another potential solution to this issue could be formal documentation such as UML. However, from Petre [19] we see that UML is not used so much in practice. These short comings leave the door open for other type of pro-developer documentation or tools to allow quick understanding of code changes and how they may impact cross team, cross project, or other dependencies. At any rate, it is clear that developers do not posses the tools needed to quickly evaluate a software change (especially when contracts change) from a wide spectrum of dependencies that may not have the ability to be analyzed by traditional means. This finding lays the foundation for the reality of indirect conflicts occurring in industry,

F2 Indirect conflicts, while occurring quite frequently throughout a development cycle, occur more often before a project reaches its "mature point" rather than after for each development cycle.

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, with more serious issues tending to occur once a month, and that the conflicts that do occur tend to be quite unique from each other. Right away we can see a serious issue with the standard development work flow. Developers are not actually spending their time developing, rather they are spending large amounts of time trying to avoid indirect conflicts. This shows that indirect conflicts are a serious threat to developer productivity and are a problem worth study and investigation. This also raises the point of how developers should be spending their time: preventing, or solving indirect conflicts.

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 surveyed, we used this notion of a mature or stable point to ask how often indirect conflicts occur in their software development experience, how often they occur before and after a project's first release (mature point), as well as how often indirect conflicts occur late in a project's lifetime. Table I shows the results of these questions.

From the interviews and the surveys combined we can make the deduction that indirect conflicts are more likely to occur early in a projects development cycle as opposed to later. This must be a major concern for any indirect conflict tools moving forward. Since many of the tools listen in Section I implement some sort of filtering system or taxonomy to avoid information overload, this finding should be included in future workings. Knowing that indirect conflicts occur more frequently early in a development cycle will let these tools know that those reported later in the development life cycle have a higher probability (compared to without this knowledge) of being a true positive. This is a major concern when dealing with information overload and to provide developers with what they may find relevant to their work. The results in in Table I also suggest that preemptive indirect conflict tools may be more accurate towards the mature point of a project, but have potentially more use earlier where the indirect conflicts are more likely to occur. While what the "mature point" of a project is could be speculated upon, we suggest it is caused 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. 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

[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%

to occur on a monthly or even yearly frequency (18% and 12%).

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.”. These results can be argued to show that the occurrence of indirect conflicts can largely depend on the management, and deadlines of projects. These are important factors to consider when trying to define a “mature point”. A large refactoring or rewrite can reset the “mature point” of a project and should be considered for any taxonomies created for preemptive indirect conflict tools. The “mature point” of a project (or multiple mature points) should be considered when building a taxonomy for indirect conflict tools.

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

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 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. These results show that indirect conflicts seem to be compounded by a multiple human factor found in parallel development. The lack of awareness as to other developers work in a project again comes to light. We would think that developers would leverage the other developer’s knowledge in their projects to avoid indirect conflicts but as we will see most developers tend not leverage their social connections within a project to help solve indirect conflicts. This could be a potential issue of social structure in that developers simply do not have the means or will to talk to their colleagues, or it could be that developers heavily leverage the notion of software stigmergy as shown by Bolici et. al [20]. Creating effective communication around technical dependencies between developers and teams through either explicit or implicit means appears to be a large task to accomplish but a carsickness task as shown by these result being that static analysis breaks down at these levels.

This finding characterizes that indirect conflicts are a frequent problem in industry. It also shows what environments are more prone to indirect conflicts and should be handled with more delicate care than others.

F3 Indirect conflicts are largely 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.

From the developers interviewed, 9 were currently working with large scale database applications and all 9 listed database schema as a large source of indirect conflicts with developers saying “[schema changes] break stuff all over the place”. 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. 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”. Some other notable artifacts were user interface displays for web development, and full components in component base game architecture.

From these results, we can see the evidence that the problemed areas of software architecture in regards to indirect conflicts are the interfaces between structures, mainly being methods and database schemas. This is good news for preexisting indirect conflict tools as many of them have focused at the signature level of methods as a spot for technical dependency changes. However, there has been little focus on indirect conflicts at the relation database schema level [21], possibly due to the inherit difficult nature of analysis of a database schema in relation to source code structure. Developers have

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%

given researchers an area of indirect conflicts that have been over looked and we should be making an effort to provide answers in this area. Also, note here that developers while in F1 showed that changes to software object contracts were the concern, the focus here was shifted to the interface. While the interface does make up a large component of a software object contract, it does not fully encompass it. Precondition and postcondition checks, which are part of the object contract are performed in the body of a method and are largley overlooked by indirect conflict tools. This may be a source of many false negatives from these tools.

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. Developers who were also 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. However, it appears that when interfaces of methods change through software evolution, indirect conflicts are more prone to occur. This result again shows that even at the library or binary level, any type of support tools for indirect conflicts should focus at the interface level of change.

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

The findings 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.

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.

Of the developers interviewed, 3 main points of interest were found in preventing indirect conflicts. The first was the use of "Design by Contract"$^{\text{TM}}$, 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. Since "Design by Contract"$^{\text{TM}}$ensures that software artifacts, while they can be changed, should always behave in a way that is understood by other developers, languages should just directly support "Design by Contract"$^{\text{TM}}$, however, this is not the case. One developer said that "is it too much of a burden" in regards to strictly making the contract apart of the source code. But why is it not such a burden is write separate unit tests that are normally designed around testing the contract? There is obviously a trade off between having rules built into the source versus having external tests to check if rules were implemented correctly. The question should become which is more effective for software development in terms of bugs produced from contract breaks or bugs produced from convoluted code. This paper cannot begin to attempt to answer this question.

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. From the interviews, it seems this is a largely effective way of preventing indirect conflicts, however, it was mostly used for libraries and methods only. In some cases such as development in a large database project, the add and deprecate model simply does not work as all code must run off the newest changes. So while the add and deprecate model is certainly a good fix for heavily version-ed software, especially when given to 3rd party developers, it does not necessarily work on internal projects which need to cooperate on which versions of methods or schemas they use.

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. While most interviewed developers had such an expert in their software projects, they admitted to not being able to foresee all indirect conflicts that may occur, especially when talking about external teams or projects who may or may not use their final product. This raises the question of how much planning by expert developers should really go into avoiding indirect conflicts? Dromey [22] raises the debate of prevention versus cure of software defects and how difficult a problem it is to measure. Should developers be using certain tools to plan their changes or should they be even planning at all? Which method provides the best software quality for the minimum effort. These are not easy questions to answer but may be key in improving developer productivity when concerning indirect conflicts.

To confirm our findings, 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.

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", or taking the curative approach. This shows that developers are somewhat content with the level of prevention that happens for indirect conflicts, however, this could be an issue of developers simply following old standards. It seems that the cure approach to indirect conflicts is heavily used in industry, but is it better than prevention? The lack of preventative process being used is probably due to the difficult nature of predicting indirect conflicts. These results have 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, in the developer's eyes, developing and fixing problems as needed than putting excess time into the prevention of potential errors. This is an important consideration for moving forward with indirect conflict tools and research.

F5 Developers use unit and integration testing to examine use case coverage in order to detect 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. It appears that most indirect conflicts are currently detected by using what some would call proper software development practices. 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. This is another example of developers subscribing to the work until something breaks philosophy in that if proper tests are written, the errors will be caught after they break something. However, problems occur when a software contract is not properly understood and use cases are not tested for or examined. 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. This is the major negative result of work until something breaks. The break detectors need to be properly written. Some languages such as Eiffle and the Java Modeling Language [3] enforce object contracts to be well understood through the ideas of preconditions and postconditions of structures which is an example of preemptive conflict detection. However, the trade off between external tests and in-line contracts is still in place as previously mentioned.

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. 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 as long as they are properly written and contracts are understood.

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

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.

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

Here, the generic approach brought forth by developers was seen to be 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 and is the time spent debugging outweighing the potential time spent preventing such errors? 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. This again shows the developer preference towards work until something breaks as opposed to prevent the error in the first place. While the debate of prevention versus cure goes on, 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 [23] and program slicing [24]. The fact that researchers have had such success in the cure or debugging side of indirect conflicts could be due to the lack of indirect conflict. Once a problem has occurred, there is no chance of it being a false positive. Academics do not have to create taxonomies to filter results and developers do not need to spend time working on only potential issues. Their time is more effectively, in their eyes, spent developing than speculating.

Of the developers surveyed, 23% also 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. This shows is the serious lack of static analysis tools available for developers in debugging as many projects and languages simply do not support it (adding cause for non static analysis tools moving forward), but more importantly, that developers tend to not communicate or use formal documentation for debugging indirect conflicts. Petre [19], as already mentioned, showed that it is not industry practice to use UML, which could open to floor to more informal means such as online forums. The none developer communication, as again already mentioned, could be addressed by either more formal communication structures or by developer stigmergy in their projects if needed.

C. How should indirect conflict tools work to best benefit developers?

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 the proceeding results.

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. Preemptive tools should be exploratory in nature.

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 RQ1 suggest that preemptive tools are best used in stable phases on the project when indirect conflicts are less likely to happen, thus reducing the number of false positives reported. 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. All of this again shows that developers would prefer to subscribe to the "work until something breaks" philosophy. Again, it is up for debate whether or not this is the correct approach the developers should be taking (effectiveness of prevention versus cure), but it is the one they currently prefer.

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.

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 [25] (when possible), 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.

Given the above challenges, some indirect conflict tools have had some general success. The key difference with these

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%

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. Here, the idea that a user can provide some sort of input, a 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 given the context. These types of tools stick to previous findings being developers working until something breaks, then requesting information to fix it.

F8 Indirect conflict tools should take a curative approach towards indirect conflicts as opposed to preventative in order to better fit developer activities.

The developers who were surveyed, the following list represents the more common suggestions for how an indirect conflict curative tools 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.

From the above list, we can see three main areas of focus that researchers of indirect conflicts should spend their time in. The first is the debugging process through utilizing logs. Showing which recent code changes affect an area of code or how a commit and break a specific unit test are key methodologies that developers want in order to aid their preferred curative approaches. Developers also want to be able to understand their software object contracts better or even automatically. Detections of contract changes will play a large part in indirect conflict and dependency awareness moving forward, especially at the method body and database schema levels. These contract detections also play a role in the automatic generation of unit tests in order to comply with use case coverage. Finally, and most importantly, developers want tools that integrate into their current development styles, that being "work until something breaks". While a complete evaluation of the curative approach is beyond the scope of this paper, it is how developers act today and in the end, tools should be created to aid developers.

IV. Evaluation

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

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

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

Applicability or Usefulness. "Do the findings offer new insights? Can they be used to develop policy or change practice?" Although our findings F1 - F6 may not be entirely novel or even surprising, the combination of these results allow us to discover the insightful findings of F7 and F8 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.

V. 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 [26]. 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. [27] 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 [28] 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. [29] found, through qualitative user studies, that developers tend to use the bare amount of communication towards other developers in order to solve direct conflicts, and take the same approach of only communicating once a conflict has arose. This is the same mentality found in this paper as "I work until something breaks". Hattori et. al. also show that with direct conflicts, the sooner preemptive information if available to developers the more they will communicate and either avoid or easily solve their code merges. This is the situation many indirect conflict tools have strived for. In terms of communication, Bolici et. al [20] give the possibility of developer stigmergy as the reason that developers do not need to explicitly communicate with each other about issues. The idea that developers use artifacts left behind by other developers to solve their issues could play a large role in a lack of formal communication.

While not specifically developed with indirect conflicts in mind, Zeller's delta debugging techniques [30], 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 [31] has also been extensively studied in order to aid in debugging in manual or automated techniques.

VI. 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 and F8 have provided a foundation as to why past techniques and tools have had low adoption rates and where researchers should focus their current and future efforts in handling indirect conflicts. We hope that this study and its findings will inspire future techniques and tools for dealing with indirect conflicts that will aid developers in industry as well as test and validate our theories put forth in this paper as well as spark a potential investigation into preventative versus curative approaches to software defects.

VII. Acknowledgments

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[19] M. 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.

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

[21] 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.

[22] R. Dromey, "Software qualityprevention versus cure?" *Software Quality Journal*, vol. 11, no. 3, pp. 197–210, 2003. [Online]. Available: http://dx.doi.org/10.1023/A%3A1025162610079

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

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

[25] 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.

[26] 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.

[27] 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.

[28] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don&#039;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] 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.

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

[31] 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.