Supporting Awareness of Indirect Conflicts with Impact

Jordan Ell
University of Victoria
Victoria, British Columbia
jell@uvic.ca

Daniela Damian
University of Victoria
Victoria, British Columbia
danielad@cs.uvic.ca

***Abstract*—Awareness has been largely studied in the field of CSCW research within software engineering. Many tools and techniques have been proposed and built in order to provide software developers and stakeholders with a greater sense of workspace and task awareness within their software projects. These techniques and tools have been largely focused on detecting *direct* conflicts which arise over a project's life time or have created an exploratory ground for stakeholders to use as a means of resolving self discovered direct or indirect conflicts. However, detecting and providing pertinent information regarding *indirect* conflicts has been largely ignored partially due to their inherently larger complexity than direct conflicts. Indirect conflicts arise when changes in one software artifact affect another. In this paper, we present *Impact*, a new task awareness tool directly aimed at both detecting and presenting indirect conflicts which arise inside of a software project. *Impact* represents a first step towards the design and implementation of awareness tools that specifically address indirect conflicts in software projects. To describe *Impact*, we introduce previous indirect conflict awareness attempts, the existing problems in the field, our design and implementation for a potential solution, as well as *Impact's* future work on a proposed solution to information overload which arises with indirect conflict awareness.**

I. Introduction

Awareness is characterized as "an understanding of the activities of others which provides a context for one's own activities" [1]. The study of awareness and its tools has become an important topic of research in software engineering especially with the new importance of distributed work and collaboration. Awareness is generally associated with both technical and social dependencies that are created and evolve over a software project's life time. The study of these dependencies has become the primary focus of most awareness related research. Task awareness has become the most prevalent field of awareness research to understand how developers cope with these technical and social dependencies.

Tools have been created to attempt to solve task awareness related issues with some success [2], [3], [4], [5]. However, these tools have been designed to solve task awareness related issues at the direct conflict level. Examples of direct conflict awareness include knowing when two or more developers are editing same artifact, finding expert knowledge of a particular file, and knowing which developers are working on which files. Meanwhile, task awareness related issues at the indirect conflict level continue to be an issue which is largely unsolved by most coordination mechanisms. Examples of indirect conflict awareness include having one's own code effected by another developer's source code change or finding out who might be indirectly effected by one's own code change. Previous interviews and surveys conducted with software developers have shown a pattern that developers of a software project view indirect conflict awareness as a high priority issue in their development [6], [7], [8], [9].

Indirect conflicts arising in source code are inherently difficult to resolve as most of the time, source code analysis or program slicing [**?**] must be performed in order to find relationships between technical objects which are harmed by changes. While some awareness tools have been created with these indirect conflicts primarily in mind [8], [10], most have only created an exploratory environment which is used by developers to solve conflicts which may arise. These tools were not designed to detect indirect conflicts that arise and alert developers to their presence inside the software system. Sarma et al. [11] has started to work directly on solving indirect conflicts, however, these products are not designed to handle internal structures of technical objects.

While indirect conflicts remain to be a large problem area in task awareness, some attempts have been made to both detect and provide developer awareness of indirect conflicts. Servant et al. [12], have devised a tool which can both detect and alert developers to indirect conflicts arising in their software projects. However, a main issue of information overload continues to arise from this and many other indirect conflict tools. As software systems grow larger and more complex, indirect dependencies tend to become more numerous. This causes small changes in source code to have a large impact on the software system which these tools often report exhaustively. Information overload must be addressed in any feasible attempt made on indirect conflicts.

In this paper, we report on our research into supporting indirect conflicts and present the design, implementation, and future evaluation of the tool *Impact*, a web based tool that aims at detecting indirect conflicts among developers and notifying the appropriate members involved in these conflicts. By leveraging technical relationships inherent of software projects with method call graphs [13] as well as detecting changes to these technical relationship through software configuration management (SCM) systems, *Impact* is able to detect indirect conflicts as well as alert developers involved in such conflicts

in task awareness while limiting information overload by using design by contract [14] solutions to method design. While this paper outlines *Impact's* specific implementation, its design is rather generic and can be implemented in similar indirect conflict awareness tools. *Impact* represents a first step towards the design and implementation of awareness tools which address indirect conflicts in software development.

The rest of this paper is organized as follows. First, we begin by discussing similar indirect conflict awareness tools which have partially solved the issues presented by this paper and how their designs can be applied to *Impact*. In the following section we describe a generic design and implementation of *Impact* as an awareness tool. We then discuss future work in terms of evaluation and further understanding of the problems at hand.

II. Related Work

Although there is an abundance of awareness tools developed in research today, only a handful have made an attempt to examine indirect conflicts. Here, we will outline four of the forefront projects in indirect conflicts and how these projects have influenced the decision making process in the design and implementation of *Impact*.

We first start with both Codebook [8] and Ariadne [10]. These projects produce an exploratory environment for developers to handle indirect conflicts. Exploratory pertains to the ability to solve self determined conflicts, meaning that once a developer discovers a conflict, they can use the tool as a type of lookup table to solve their issue. Codebook is a type of social developer network that relates developers to source code, issue repositories and other social media while Ariadne only looks at source code for developer to source code association. Through Codebook, developers become owners of source code artifacts. Both projects also use program dependency graphs [15] in order to relate technical artifacts to each other. These projects make use of method call graphs in order to determine which methods invoke others which forms the basis for linking source code artifacts creating a directed graph. While these projects can be great tools for solving indirect conflicts which may arise, by querying such directed graphs to view impacts of conflict creating code, they lack the ability to detect potential conflicts on their own.

A serious attempt at both detecting and informing developers of indirect conflicts is the tool Palantir [11]. Palantir monitors developers activities in files with regards to class signatures. Once a developer changes the signature of a class, such as by modifying changes in name, parameters, or return values of public methods, any workspace of other developers which are using that class will be notified. Palantir utilizes a push-based event model [16] which seems to be a favored collection system among awareness tools. Sarma et al. [11] also developed a generic design for future indirect conflict awareness tools. However, Palantir falls short in its collection and distribution mechanisms. First, Palantir only considers "outside" appearance of technical objects, being their return types, parameters, etc. Secondly, Palantir only delivers detected conflicts to developers who are presently viewing or editing the indirect object while other developers who have used the modified class previously are not notified.

We will lastly examine the tool CASI [12] which uses a sphere of influence for each developer to determine how source code changes are indirectly related to other components of a software project. CASI uses dependency slicing [17] instead of the call graphs in Ariadne [10] which gives dependencies among all source code entities. This provides a verbose output of dependencies when source code is changed. CASI also implements a visualization where a developer can see what parts of a software projects he or she may be effecting with the source code change. This allows the developer themselves to go and fix potential issues elsewhere in the project before the code change is committed to the software repository. While CASI covers great ground in its approach, it still leaves the issue of information overload, although attempts were made to solve this by having severity levels of indirect conflicts presented to the user.

Impact is designed to address these limitations by extending this work in two directions: (1) the detection of high importance indirect conflicts at an internal level of technical objects, (2) the dissemination of awareness information to all appropriate developers regardless of their current workspace activities, and (3) potential solutions to information overload. *Impact* is also designed around the successes these projects have had in the past with dependency graphs as well as elements of collection, ownership, and distribution functionality.

III. Impact

This section will proceed to give an outlined detail of *Impact* in both its design and implementation. The design of *Impact* was created to be a generic construct which can be applied to other indirect conflict awareness tools while the implementation is specific to the technical goals of *Impact*.

A. Design

Compared to tool design for direct conflicts, the major concern of indirect conflict tools is to relate technical objects to one another with a "uses" relationship. To say that object 1 uses object 2 is to infer a technical relationship between the two objects which can be used in part to detect indirect conflict that arise from modifying object 2. This kind of relationship is modeled based on directed graphs [15]. Each technical object is represented by node while each "uses" relationship is represented by a directed edge. This representation is used to indicate all indirect relationships within a software project.

While technical object relationships form the basis of indirect conflicts, communication between developers is our ultimate goal of resolving such conflicts. This being the case, developer ownership must be placed on the identified technical objects. With this ownership, we now infer relationships among developers based on their technical objects "uses" relationship. Developer A, who owns object 1, which uses object 2 owned by developer B, may be notified by a change

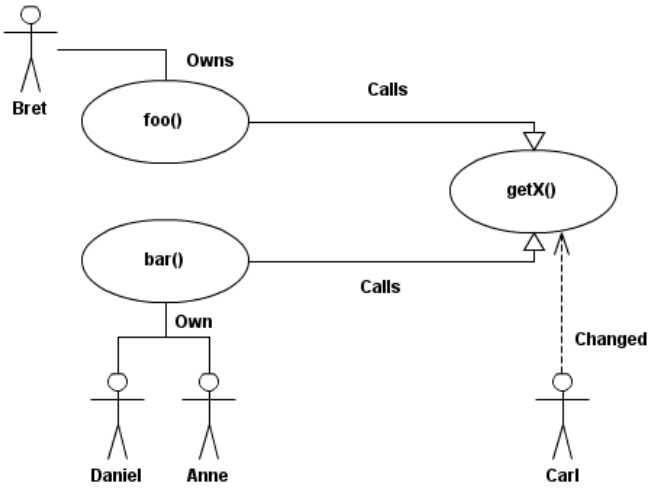

Fig. 1. Technical object directed graph with ownership

to object 2's internal workings. Most, if not all, ownership information of technical objects can be extracted from a project's source code repository (CVS, Git, SVN, etc.).

Finally, the indirect conflict tool must be able to detect changes to the technical objects defined above and notify the appropriate owners to the conflict. Two approaches have been proposed for change gathering techniques: real time and commit time [16]. We propose the use of commit time information gathering as it avoids the issue of developers overwriting previous work or deleting modifications which would produce information for changes that no longer exist. However, the trade off is that indirect conflicts must be committed before detected, which results in conflicts being apart of the system before being able to be dealt with as opposed to catching conflicts before they happen. At commit time, the tool must parse changed source code in relation to technical artifacts in the created directed graph detailed above. Where *Impact's* design differs from that of Palantir's is that the object's entire body (method definition and internal body) is parsed, similar to that of CASI [12], at commit time, as opposed to real time, to detect changes anywhere in the technical object. This is a first design step towards avoiding information overload. Once technical objects are found to be changed, appropriate owners of objects which use the changed object should be notified. However, to avoid information overload at this step, analyses of the changed object must be preformed to determine whether or not it is a "notification worthy" change. Our proposed solution to this is explained in Section IV. In Figure 1, Carl changes method (technical object) 1, which effects methods 2 and 3 resulting in the alerting of developers Bret, Daniel, and Anne. We have opted to alert the invoking developers rather than the developer making the change to potential solutions as our conflicts are detected at commit time and this supports the idea of a socio-technical congruence [18] from software structure to communication patterns in awareness systems.

With this three step design: (i) creating directed graphs of technical objects, (ii) assigning ownership to those technical objects, and (iii) detecting "notification worthy" changes at commit time and the dissemination of conflict information to appropriate owners, we believe a wide variety of indirect conflict awareness tools can be created or extended. The on going implementation of *Impact* described in the following section follows these three design guidelines.

B. Implementation

For *Impact's* implementation, we decided to focus on methods as our selected technical objects to infer a directed graph from. The "uses" relationship described above for methods is method invocation. Thus, in our constructed dependency graph, methods represent nodes and method invocations represent our directed edges. In order to construct this directed graph, abstract syntax trees (ASTs) are constructed from source files in the project.

Once the directed graph is constructed, we must now assign ownership to our technical objects (methods) as per our design. To do this, we simply query the source code repository. In our case we used Git as the source code repository, so the command *git blame* is used for querying ownership information. (Most source code repositories have similar commands and functionality.) This command returns the source code authors per line which can be used to assign ownership to methods.

To detect changes to our technical objects (methods), we simply use a commit's *diff* which is a representation of all changes made inside a commit. We can use the lines changed in the *diff* to find methods that have been changed. This gives cause of potential indirect conflicts. Here is where information overload can occur and must be dealt with. Impact's current state does not address this as our potential solution is still under investigation, but is the main focus of our next steps as detailed in Section IV. We now find all methods in our directed graphs which invoke these changed methods. These are the final indirect conflicts.

Once the indirect conflicts have been found, we use the ownership information of our technical objects to send notifications to those developers involved in the indirect conflict. All owners of methods which invoke those that have been changed are alerted to the newly changed method. Impact can been seen in Figure 2, the user interface of *Impact*. Here, in an RSS type feed, the changing developer, time of change, changed method, invoking methods, and commit message are all displayed. The weight provided will be an indication of importance based on the future work to be outline in Section IV. This again helps solve the issue of information overload.

IV. Proposed Solution to Information Overload

As was previously stated, the main issue to be addressed with Impact is information overload. Our potential solution comes from the ideas of Meyer [14] on "design by contract". In this methodology, changes to method preconditions and postconditions are considered to be the most harmful. This includes adding conditions that must be met by both input and output parameters such as limitations to input and expected

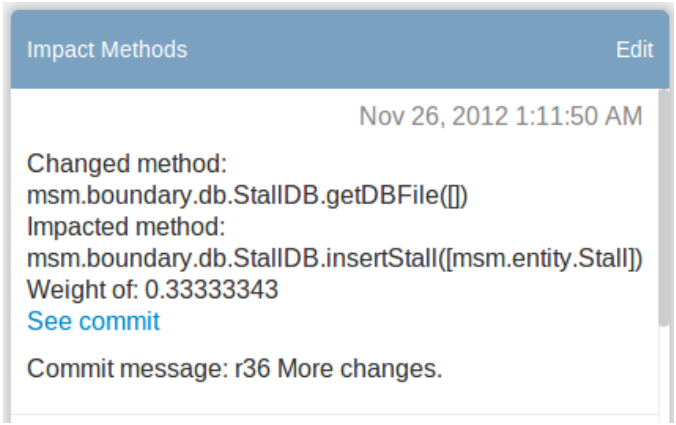

Fig. 2. *Impact's* RSS type information feed.

output. To achieve this level of source code analysis, the ideas of Fluri et al. [19] can be used on the previously generated ASTs for high granularity source code change extraction when determining if preconditions or postconditions have changed. We plan to bring these ideas into the realm of what makes a change "notification worthy", as to deal with information overload for developers using Impact. By only examining "notification worthy" changes, Impact would cut down the number of notifications issued to just those which developers care about. To investigate this solution, we plan to interview and survey industry software developers at a wide range of experience to determine what makes a change in software important. We believe by only monitoring what developers and Meyer [14] find important to indirect conflict awareness, we will be able to eliminate information overload in Impact and other dependency awareness tools. We believe that these preconditions and postconditions will serve as the main tool for dealing with information overload.

V. Conclusion and Future Work

In this paper, we have presented the issues that arise from indirect conflicts in present awareness tools. We have proposed a generic design for the future development of awareness tools in regards to handling indirect conflicts. We have presented a prototype awareness tool, *Impact*, which was designed around our generic technical object approach. We have also proposed a solution to information overload which occurs in most indirect conflict awareness tools. We plan on investigating our solution further through developer interview and surveys, and by finally implementing the solution into Impact.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] D. Damian, L. Izquierdo, J. Singer, and I. Kwan, "Awareness in the wild: Why communication breakdowns occur," in *Global Software Engineering, 2007. ICGSE 2007. Second IEEE International Conference on*, aug. 2007, pp. 81 –90.

[7] C. A. Halverson, J. B. Ellis, C. Danis, and W. A. Kellogg, "Designing task visualizations to support the coordination of work in software development," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, ser. CSCW '06. New York, NY, USA: ACM, 2006, pp. 39–48.

[8] 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.

[9] A. Schröter, J. Aranda, D. Damian, and I. Kwan, "To talk or not to talk: factors that influence communication around changesets," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, ser. CSCW '12. New York, NY, USA: ACM, 2012, pp. 1317–1326.

[10] 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.

[11] 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.

[12] 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.

[13] A. Lakhotia, "Constructing call multigraphs using dependence graphs," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '93. New York, NY, USA: ACM, 1993, pp. 273–284.

[14] B. Meyer, *Object-Oriented Software Construction*, 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[15] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proceedings of the 14th international conference on Software engineering*, ser. ICSE '92. New York, NY, USA: ACM, 1992, pp. 392–411.

[16] G. Fitzpatrick, S. Kaplan, T. Mansfield, A. David, and B. Segall, "Supporting public availability and accessibility with elvin: Experiences and reflections," *Comput. Supported Coop. Work*, vol. 11, no. 3, pp. 447–474, Nov. 2002.

[17] 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.

[18] 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.

[19] 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.