Cassandra: Proactive Conflict Minimization through Optimized Task Scheduling

Bakhtiar Khan Kasi and Anita Sarma
Computer Science and Engineering Department
University of Nebraska-Lincoln
Lincoln, NE, USA
{bkasi, asarma}@cse.unl.edu

***Abstract*—Software conflicts arising because of conflicting changes are a regular occurrence and delay projects. The main precept of workspace awareness tools has been to identify potential conflicts early, while changes are still small and easier to resolve. However, in this approach conflicts still occur and require developer time and effort to resolve. We present a novel conflict minimization technique that proactively identifies potential conflicts, encodes them as constraints, and solves the constraint space to recommend a set of conflict-minimal development paths for the team. Here we present a study of four open source projects to characterize the distribution of conflicts and their resolution efforts. We then explain our conflict minimization technique and the design and implementation of this technique in our prototype, Cassandra. We show that Cassandra would have successfully avoided a majority of conflicts in the four open source test subjects. We demonstrate the efficiency of our approach by applying the technique to a simulated set of scenarios with higher than normal incidence of conflicts.**

***Index Terms*—Collaborative development, coordination, collaboration conflicts, task scheduling**

I. INTRODUCTION

Conflicting changes in parallel software development occur frequently despite advances in communication and coordination environments [1]–[3]. Conflicting changes typically occur because of breakdowns in an understanding of how one's work fits with others' changes. For example, two developers can edit the same file concurrently (direct conflict) or an interface that was presumed to be stable is changed without appropriate notifications to developers using it (indirect conflict) [4]

The state of practice aims for conflict resolution. Configuration management (CM) systems allow each individual to check out files and work on their changes in local workspaces that are periodically synchronized with the repository. While such a loose synchronization protocol enables rapid parallel development, it also allows developers to inadvertently make conflicting changes. Automated diff and merge techniques[5]–[7] help in resolving direct conflicts, but often require manual intervention [3], [4], [8]. Resolution of indirect conflicts is not currently supported.

Our analysis of four popular open source projects reveals that conflicts are a regular occurrence. In the projects analyzed merge conflicts ranged from 7.6% to 19.3%. Of the clean merges 2.1% to 14.7% had build failures, and 5.6% to 35% of correct builds incurred test failures. Resolving these conflicts took substantial effort, typically spanning multiple days. In sum, conflict resolution is costly [1], [9]–[13]: it delays the project while developers backtrack to determine the reason(s) for the conflict and find a resolution.

Research in coordination has focused on conflict mitigation, that is, identifying and notifying developers of potential conflicts as they emerge. This allows developers to coordinate to resolve potential conflicts early, while conflicts are still small and easier to resolve. Workspace awareness tools monitor changes at real time to notify developers of emerging direct and indirect conflicts [4]. Tools such as FastDash [14], Syde [15], and others [8], [16] largely support direct conflicts, whereas tools like Palantír [4], Crystal [3], and CollabVS[17] support both conflict types. While these tools are proactive in identifying conflicts early, they are still reactive since conflict resolution can only be performed *post hoc* - after the conflict has already developed.

To alleviate this situation, we present a novel conflict minimization technique that evaluates task constraints in a project to recommend optimum task orders for each developer. A key goal is to proactively determine conflicting tasks – tasks that will conflict when performed in parallel and appropriately schedule them to recommend conflict-free development paths.

Constructing a conflict avoidance task scheduler raises many questions including: what are the different types of constraints in the context of parallel software development; how can we prioritize constraints if all of them cannot be satisfied; how can individual and overall team goals be reconciled; how to provide guidance without overly restricting or overloading users; and questions about scalability, and general effectiveness of the approach.

In this paper, we provide a first exploration in answering these questions through the implementation of our research prototype, Cassandra. The specific contributions of this paper include:

- Empirical analysis of four open source projects to identify the distribution of different types of conflicts and the effort needed to resolve them. Contrary to common belief that higher order conflicts are harder to resolve, we found that the resolution effort for direct and indirect conflict to be comparable in some projects. Further, each project had a different conflict profile, suggesting that there is no "one-size fits all" analysis. To the best of our knowledge, ours is the first study to characterize the resolution effort needed for different types of conflicts.
- An implementation of a novel conflict minimization technique that formalizes a parallel software development context – task dependencies, file dependencies, and developer

preferences into constraints, and then solves these constraints to determine conflict-minimal development paths for the team.
- Design and implementation of Cassandra, our research prototype that implements the conflict minimization technique to provide developers a recommended list of (conflict free) tasks and the rationale for these recommendations in a contextualized manner.
- An extensive evaluation of Cassandra using data from several open source projects, as well, as simulated data with a much higher distribution of conflicts than typically found.

The rest of the paper is organized as follows: We present background on related work in Section II, followed by our empirical analysis of open source projects in Section III. Section IV presents a motivating example followed by a discussion of our approach in Section V. Implementation details are presented in Section VI and our evaluation in Section VII. We discuss the threats to validity in Section VIII and conclude with a brief outlook into future work in Section IX.

II. Related Work: Change Awareness Tools

Many different kinds of conflicts can arise in parallel software development when it is performed in distributed workspaces. The impact of each type of conflict and the effort to resolve them vary.

Three major categories of conflicts exist. First, merge conflicts arising because of parallel changes to the same artifact. We refer to them as *direct conflicts*. This type of conflict is typically identified when a developer attempts to check-in their changes while a newer version already exists in the repository. Second, build failures arising because of parallel changes to two different artifacts that cause syntactic mismatches and ensuing compilation errors. Such conflicts are typically identified during a system-wide build. Finally, test failures arising because of parallel changes to two different artifacts that cause mismatches in program behavior. Such conflicts are only detected during testing (integration) or may remain as defects. Conflicts arising because of changes in one file affecting changes in another are classified as *indirect conflicts*.

Research on change awareness attempts to alleviate the impact of conflicts by informing developer of ongoing changes that can potentially conflict. The primary goal is to provide awareness: "an understanding of the activities of others, which provides a context for your own activity [18]", so that developers can proactively coordinate their work while conflicts are still small and easier to resolve. Some examples tools are:

FastDash [14] provides a dashboard visualization that spatially represents the files each developer in a team is editing. Syde [15] also identifies merge conflicts, but reduces false positives via a fine-grained analysis of the abstract syntax trees (ASTs) modifications. In addition to direct conflicts, Palantír and CollabVS detect indirect conflicts when a user starts editing a program element that has a dependency on another program element that is being edited in parallel [4], [17]. These techniques all identify conflicts at a syntactic level (either AST differences or static dependency changes).

SafeCommit [19] performs deeper program dependence analysis by identifying changes that pass a given set of test cases. The proposed technique identifies changes that are covered by original and edited test suites (either pass or fail) as well as changes that do not have coverage, to identify changes that may fail a given test suite. Crystal [3] leverages the infrastructure provided by decentralized version control systems (e.g., Git) and integrates (commits) local changes in a workspace into a shadow repository and executes build and test scripts to identify potential changes sets that will cause merge conflicts or test or build failures. The paper also presents data on the number of days it took to resolve merge conflicts. Guimaraes et al [8] also use a shadow repository for merging changes that are then analyzed, compiled, and tested to detect (merge) conflicts. The above techniques depend on the existence of robust build scripts and test cases in the project.

III. Conflicts in Practice

Coordination breakdowns and ensuing conflicts occur frequently [20]–[22]. Case studies have found that: parallel changes are a leading cause of defects (one study in a telecommunication system found 98 developers to be working in parallel on the same artifact [21]), coordination issues in distributed development lead to code integration problems [23], developers have difficulty in identifying their impact network – changes that may impact them or changes that may be impacted [24], and developers spend significant portion of their time in coordinating their efforts [9], [20].

Studies of coordination failures have been largely qualitative thus far [22], [25]–[27]. To the best of our knowledge there are only two studies ([3], [13]) that quantitatively characterize the distribution of conflicts: these studies found the frequency of direct conflicts that required manual intervention to range from 16% to 47% in several open source projects. Of the two studies, one (Brun et al. [3]) investigated indirect conflicts. They found that in the three open source projects studied, 33% of the 399 merges that the version control system reported as being a clean merge, resulted in an indirect (build or test) conflict. These studies do not report on the time taken to resolve indirect conflicts

Empirical Study: We analyzed four open source projects (see Table I) using similar techniques as Brun et al. [3]. Our study is different since our primary goal is to analyze the distribution of conflicts to characterize the constraint space. To the best of our knowledge we are the first to present resolution effort data for each type of conflict across multiple projects.

We chose projects that are hosted in GitHub [28], based on the following criteria: (1) popularity of the projects in GitHub, (2) project activity: at least 20 developers and more than 500 changesets, (3) inclusion of built and test scripts in the repository, and finally (5) the project is not a Git mirror of another CM system repository, such as SVN.

We use version histories of the following projects:
- *Perl*: Programming language: Jun-2002 to Feb-2010
- *Storm*: Distributed real time computation system: Sep-2011 to Jun-2012
- *Jenkins*: Continuous integration server: Mar-2009 to Jun-2012
- *Voldemort*: Distributed key-value storage system: Jun-2009 to Jun-2012

We downloaded the version histories of the above projects in the specified time periods using our tool GitMiner [29], which converts the version histories into a graph database (Neo4J [30]). We use Gremlin[31] for traversal of the data.

TABLE I. SUBJECT (OPEN SOURCE) PROJECTS ANALYZED FROM GITHUB

Project	KLOC	#Developers	#Changesets	#Merges	#Conflicts	Merge		Build		Test	
						#Conflicts	Resolution Days Average (Med)	#Failures	Resolution Days Average (Med)	# Failures	Resolution Days Average (Med)
Perl	2,213	51	23,079	185	74 (40%)	14 (7.6%)	22.93 (10)	4 (2.1%)	0.75 (0.75)	56 (30.2%)	30.5 (14)
Storm	60	24	975	88	39 (44%)	17 (19.3%)	6 (2)	9 (10.2%)	5 (8)	13 (14.7%)	7.9 (3)
Jenkins	565	100	14,627	505	204 (54%)	68 (13.5%)	26.51 (4)	74 (14.7%)	4.98 (2)	28 (5.6%)	6.9 (2)
Voldemort	171	33	3,026	380	170 (34%)	55 (14.5%)	20.14 (4)	16 (4.2%)	2.25 (0.75)	133 (35%)	6.01 (4)

We determine the following information: (1) the three basic kinds of conflicts: merge, build, and test failures, (2) their frequency, and (3) the number of days the conflict existed in the repository, which serves as a measure of its resolution effort.

We identify conflicts in each project by recursively integrating developer changes into a shadow master repository. When using Git, developers fork the main repository to create their own repository, which contains their working copy of the code. Commits by each developer are logged as local commits in their respective repositories. When a developer is ready to share her changes with the team she can either send a "pull request" or commit to the master repository. In performing our analysis we make the following simplifying assumptions: (1) each developer has only one repository, (2) developers create local commits on finishing their tasks, and (3) developers fork from; and commit to the master repository.

We recursively integrate changes into our shadow repository based on the order in which these local commits appear in the master repository, and progressively perform conflict analysis. That is, we first integrate local commits and if the Git merge fails then we flag that merge as a failure. If the merge is successful then we run build scripts on those "clean merges"; if the build is successful then we run test cases.

Conflict resolution times are approximated as the number of days between when a conflict first occurred and until when it was resolved (i.e. the number of days the conflict persisted in the master repository). We calculate this by tracking the number of changesets between a failed and successful event in the repository. For example, if a build conflict is detected when a developer merges her changeset, which she resolved in a subsequent clean merge. We consider all changesets between these two merges as the resolution effort. Here, we are assuming that the set of changesets reflect the efforts of the developer in exclusively resolving the conflict, which might not always be the case in an open source project. This might make our effort estimation results an over-approximation.

Our analyses show that conflict occurrences are a norm even in open source projects and occur irrespective of the size of the project (KLOC) or the numbers of developers. Each project exhibited different distribution of each type of conflict. Merge conflicts ranged from 4.2% to 19.3%; build failures: from 2.1% to 14.7%; and test failures: from 5.6% to 35%.

Further analysis of the conflict distribution shows activity spurts: distinct periods of high levels of parallel activity and conflicts followed by lower activity and conflicts (see Fig. 3).

The resolution times for different kinds of conflicts also vary significantly across projects. For example, Perl experienced the least number of merge conflicts (7.6% of total merges of change sets), but these required long resolution times (22.93 days average and a 10 days median). In comparison, Storm had a high incidence of direct conflicts (19.3%) but required less resolution times (6 days average).

Our results provide two key insights: (1) project structures, team policies, development practices play a role in conflicts and their resolution, the interplay of which needs to be further investigated, and (2) there is no 'one-size-fits all' conflict mitigation technique; analyses need to be fine-tuned per project.

We use the distribution of conflicts and the severity of each type (based on resolution efforts) to guide our UnSAT heuristics (see Section VI). They are also used to guide the task simulation that we use for our evaluation (see Section VII)

IV. MOTIVATING EXAMPLE

In this section, we provide a highly simplified scenario that we will use throughout the paper. Let us assume that Alice and Bob are working on a hypothetical project (see Table II) involving polygons, where classes Square.java, Rectangle.java, and Triangle.java inherit from class Shape.java.

To plan for future additions of new shapes to the code base Alice in task T_{A1} refactors Shape.java to combine two methods for calculating areas into a single method by using a parameter for the type of shape (s_type). She then updates existing methods in the project to reflect this change. Rectangle.java is the only class affected, which she updates and commits.

TABLE II. TASK LIST FOR BOB AND ALICE

Alice's Workspace		Bob's Workspace	
T_{A1}	Shape.java - area(float l, float w) - area(float l) + area (float l, float w, s_type) Rectangle.java - shape.area (l, w) + shape.area (l, w, s_type)	T_{B1}	Square.java + innerArea (float b) + shape.area(l) Rectangle.java + innerArea(float b)
T_{A2}	+ Canvas.java	T_{B2}	+ Triangle.java
T_{A3}	+ Panel.java	T_{B3}	+ Plot.java

Meanwhile, Bob in T_{B1} adds the functionality of calculating the area without its border (innerArea) to Square.java and Rectangle.java. These methods in turn call the respective methods in Shape.area(). When committing his changes he faces a *merge conflict* and realizes that his copy of Rectangle.java is out of date and needs to be reconciled with changes in the repository. He also faces a *build failure* for Square.java since he used the earlier version of Shape.area(float*),* which now also incudes the shape_type parameter.

In $T_{B2,}$ Bob creates a new class Triangle.java. Bob ensures that he is calling the new Shape.area (float,float,Type) with the shape_type parameter set as 'T'. However, Alice did not create functionalities for the area of a triangle in Shape.java, which defaults the shape to a rectangle. Bob's changes therefore lead to a *test failure*.

Alice's task T_{A3} depends on task T_{A2}, where class Panel extends from superclass Canvas. Thus there is a *precedence* of task T_{A2} over T_{A3}. For simplicity, we assume that Alice's tasks (T_{A2}, T_{A3}) are independent of Bob's task (T_{B3}).

Our goal is to schedule tasks for Alice and Bob such that the conflicts in T_{A1}, T_{B1} and T_{B2} can be avoided. For example if Alice works on T_{A1} then Bob must work on T_{B3}. However if Bob decides to work on T_{B1} first, then Alice must work on T_{A2} followed by T_{A3}.

V. CASSANDRA

We designed Cassandra, a novel task scheduling system that aims to minimize conflicts by recommending task orders that restrict dependent tasks or tasks that share common files from being concurrently edited. When implementing Cassandra we make the following design decisions:

- *Proactive*: A primary goal of Cassandra is to move from reactive conflict mitigation to proactive conflict avoidance. Conflict mitigation techniques help in identifying conflicts early thereby allowing developers to resolve these conflicts while they are still small. However, effort is still needed for resolution. In prior work, we found that users of our workspace awareness tool consistently took longer than the control group (without workspace awareness) [4]
- *Granularity*: Awareness tools typically provide change information at the file level. It is the responsibility of the developer to keep track of this information and relate it to their current or future tasks. We, on the other hand, provide information at the task level that parallels the developers logical unit of work. Cassandra presents contextualized information of the rationale for its recommended task orders.
- *Timeliness*: A key principle of workspace awareness is to keep users abreast of ongoing changes. Tools, therefore, continuously "push" change information to the users' workspace, albeit in an unobtrusive manner. However, user experiments have shown that developers actually note change information at very specific points (e.g., check-in of change, starting a new task, or taking a break) [32]. We, therefore, update the user's task view and re-evaluate the constraint space when a developer completes her work, so that the information is ready when she is about to start her new task.
- *Individual versus team strategy*: Conflict resolution as well as mitigation tools make the user responsible for coordinating and resolving conflicts. This can lead to individualistic strategies, such as racing to finish one's work or checking-in unfinished code to avoid having to perform conflict resolution [11]. We aim to reconcile these conflicting individual and team goals by providing different heuristics for task scheduling that can be guided by team policies.

The architecture of Cassandra is shown in Fig. 1. Grey components represent generically available version control system, issue tracker, and development editor that are used in a project. Other components are implemented by Cassandra, and are architecturally separated to allow extensibility. The *Context Generator* and *Visualization* components are implemented at the client side, whereas the *Task Scheduler* and *Internal Storage* component are centralized. We describe each of these components below.

Context Generator: is implemented as a set of workspace wrappers that track activities in the Eclipse workspace, events

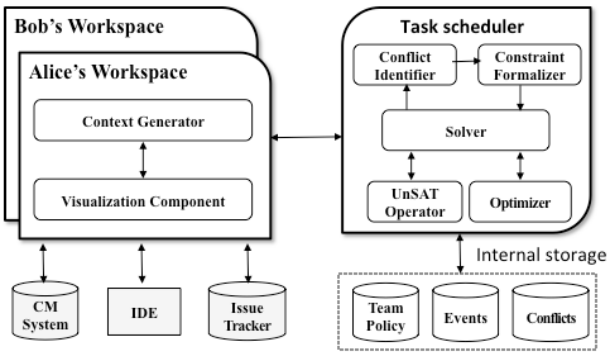

Fig. 1. Cassandra Architecture.

from the Mylyn plugin [33], and CM operations. More specifically it logs the following activities: (1) workspace activities such as save, open, etc., (2) Mylyn events, such as creating or activating a task, selecting files for a particular task, and edit and propagation events, (3) CM operations such as check-in, check-out, update.

A primarily responsible of this component is to track the development context of a task. We track three main context factors: (1) *task precedence* – task dependencies created as a result of functional dependencies, for example, in our scenario class Triangle depends on the existence of parent class Shape, (2) *task dependencies* – dependencies across tasks because of the underlying program dependencies among files that are to be modified in these tasks; and (3) developer preferences of the ordering of their tasks. Currently, we track task precedence information from the issue tracker (e.g., Bugzilla), and monitor the IDE for the other two pieces of information.

Currently, we rely on the developers' interactions with Mylyn to identify the set of files (F_e) that they are going to edit for a task. Once we have the F_e set, we perform a simple call-graph analysis using a third party tool (Dependency Finder [34]) to identify the set of dependent files (F_d). Note that here we assume a team policy where developers when assigned a task use Mylyn to identify the set of files that they intend to modify (F_e). We note that this requires upfront developer effort and might not be feasible in all development contexts. In the future, we plan to use automated data mining and machine learning techniques to mine similar past issues to predict resources that are likely to be used for a current task. Such techniques have been successfully applied in automated bug triaging [35].

Any changes to the task context as a result of ongoing development (new files added to the F_e set) as well as CM operations are communicated to the scheduler component and stored in the Internal Storage component.

Visualization Component: is created as an Eclipse plugin that modifies the task view of Mylyn to present the order of our task recommendation. Our goal is not to restrict, but to guide the user in choosing their next (optimum) task. We present the rationale for our task recommendation as a popup linked from the recommended task order number.

Task Scheduler: formalizes the task precedence, task dependencies, and developer preferences into constraints. It preprocesses the data to identify conflicting tasks, that is, if the tasks share a common file (direct conflict) or require modification of dependent files (indirect conflict), which are then formalized. If a constraint free solution exists, then it is optimized

to match developer preferences to the extent possible. If no solution exists, then constraints are progressively relaxed until a solution is achieved. We explain this component in detail in the following section.

Internal Storage: maintains an overview of development activities – recommended task orders, the particular task that are being edited in a particular workspace and their resources.

It also tracks the number and type of conflicts that have been predicted and those that have occurred in the project. Finally, it keeps track of other Cassandra events for bootstrapping new or returning clients.

The architecture of Cassandra is explicitly designed to allow:

- *Extensibility*: Cassandra architecturally separates its main components to enable easy plug and play. For example, supporting a different IDE (e.g., Visual Studio) is possible by simply changing the context generator. Similarly, incorporating a more sophisticated dependency analysis will only require changes to the "Conflict Identifier" component of the Task Scheduler.
- *Flexibility*: Different projects incur different frequencies of conflicts and have different team policies. Cassandra therefore provides different heuristics that a team can choose for relaxing constraints when conflict-free development paths are not possible. These heuristics also help in reconciling individual and team preferences.

VI. Constraint Solving and Task Scheduling

Constraint satisfaction is the process of finding a solution to a set of constraints that impose conditions that a set of variables must satisfy in a given domain. A constraint space can be expressed as a triplet $CSP = (V, D, C)$, where values are selected from a given finite domain (D) and assigned to each variable (V), while ensuring that the given set of constraints (C) are satisfied [36], [37]. Constraints can be of two types: (1) hard constraints that must always hold true when arriving at a solution, and (2) soft constraints that can be relaxed if necessary. Next, we discuss our formalization of the constraint space and how we solve it.

Formalizing Constraints: Our goal is to identify a set of task ordering from a given set of tasks in the project that *satisfies* the constraint sets in task assignments. For example, given n developers and m tasks per developer in a team, $\mathrm{T_a} = \{t_1, t_2, \ldots, t_m\}$ is a set of tasks for developer a. The set of all tasks is then $\mathrm{T} = \cup_{1 \le d \le n} T_d$. The tasks for each developer are ordered as a sequence, $\mathrm{O}_a = \langle t_a^1, t_a^2, \ldots, t_a^m \rangle$, which encodes a permutation of $\mathrm{T_a}$. We write $\mathrm{O_a}[i]$ to denote the i^{th} task in the sequence.

```
1  s = Solver() # initialize Z3 solver with appropriate config's
2
3  A = IntVector('A', 3) # 3 tasks assigned to Alice
4  B = IntVector('B', 3) # 3 tasks assigned to Bob
5
6  # Task's are assigned within valid range e.g. (T1 to T3)
7  s.add([1 <= A[0], A[0] <= 3,1 <= A[1], A[1] <= 3,1 <= A[2], A[2] <= 3,\
8         1 <= B[0], B[0] <= 3,1 <= B[1], B[1] <= 3,1 <= B[2], B[2] <= 3])
9
10 s.add([Distinct(A), Distinct(B)]) # Assignment must be unique for each task
11
12 DC = BoolVector('DC', 1) # 1 Direct Conflicts
13 IC = BoolVector('IC', 3) # 3 Indirect Conflicts
14
15 # Adding conflicts as assertions
16 s.add(Implies(DC[0], A[0] != B[0])) # TA1:Rectangle.java <-> TB1:Rectangle.java
17
18 s.add(Implies(IC[0], A[0] != B[0])) # TA1:Shape.java <-> TB1:Square.java
19 s.add(Implies(IC[1], A[0] != B[1])) # TA1:Shape.java <-> TB2:Traiangle.java
20 s.add(Implies(IC[2], A[1] < A[2]))  # TA2:Canvas.java <-> TA3:Panel.java
21
22 isSat = s.check(DC+IC) # Check if all constraints are Satisfiable
23 if isSat == sat:
24     print getOptimizedSolution() # function for optimizing solution
25 else:
26     relaxConstraints(s,DC,IC) # function for relaxing constraints
27     m = s.model()
28     print printResult(m)
```

Fig. 2. Z3Py Constraint Encoding of Example Scenario.

The constraints in the development context are formalized as follows (see Fig. 2, where our example scenario is formalized using Z3Py [38] notations). First, we formalize task precedence relation, $<$, as hard constraints, since the implementation of a task is functionally dependent on the completion of another: $\forall_{1 \le i \le m} O_a[i] < O_b[i]$. In our example, Alice's task $\mathrm{T_{A2}}$ (creating parent class Canvas) precedes $\mathrm{T_{A3}}$ (child class Panel) (see Fig. 2, line #20)

Second, we need to identify those task dependencies that will lead to a conflict if a set of tasks are concurrently executed. The $\mathrm{F_e}$ (files to be edited) and $\mathrm{F_d}$ (dependent files) sets provided by the Context Generator are used by the *Conflict Identifier* component (see Fig. 1) to detect potential conflicts. Potential direct conflicts occur when task pairs include files that are common in their $\mathrm{F_e}$ sets ($F_e[i] \cap F_e[j] \neq \emptyset$). Potential indirect conflicts occur when dependent files of a task are modified by another developer ($\mathrm{F}_d[i] \cap \mathrm{F}_e[j] \neq \emptyset$). We note that these are currently simplistic measures. We plan to next use program analysis techniques to create a more sophisticated conflict detection algorithm. Also note, that at this stage, before tasks have started we cannot not differentiate between tests and build failures. Therefore, both are grouped as indirect conflicts.

In our example Alice and Bob's tasks will result in direct (Rectangle) and indirect (Square) conflicts:

$\mathrm{T_{A1}}$: F_e = {Rectangle, Shape}, F_d = {Square}
$\mathrm{T_{B1}}$: F_e = {Rectangle, Square}, F_d = {Shape}

After we have identified the conflicts, the *Constraint Formalizer* component (see Fig. 1) encodes these constraints. We use the relation: $c = \mathrm{T} \times \mathrm{T} \rightarrow \{0,1\}$ to define the existence of a conflict between a pair of tasks. Given two developers, a and b, the ordering of their tasks must then be conflict free: $\forall_{1 \le i \le m} \neg c(O_a[i], O_b[i])$. We formalize conflicts (direct or indirect) between a pair of tasks as a set of soft constraints that discourages two tasks from being concurrently performed. In our example scenario, Alice and Bob faced one direct and two indirect conflicts (Fig. 2, line# 12-13), which are formalized as soft constraints (lines# 15-20).

Next, developer preferences are formalized[1], so that they can be used to optimize the solution set (explained later). Other implicit assumptions about the development context also need to be explicitly encoded. For example, our recommended task assignments should respect the actual developer task assignments (i.e., Alice can only be assigned to her tasks: $\mathrm{T_{A1}}$, $\mathrm{T_{A2}}$ or $\mathrm{T_{A3}}$, (Fig. 2: lines# 6-8). Similarly, we assume that a developer performs only one task at a time (line 10)

Solving Constraints: Constraint satisfaction problems on finite domains are typically solved using a form of search. Popular techniques include variants of backtracking [39], constraint propagation [40], and local search [41]. Constraint satisfaction problems (CSP) arise in different application areas including software engineering (e.g., static program analysis,

[1] formalization not shown here because of space constraints. See source code: http://interaction.unl.edu/cassandra/resources

test-case generation, symbolic execution). A well-known CSP is propositional satisfiability (SAT) that aims to determine whether a formula comprising Boolean variables that are formed by using logical connectives can be solved by choosing true/false values for its variables. Often richer languages (arithmetic or linear inequalities) better describe a given problem such as ours. Solvers for such formulations are called "satisfiable modulo theories" or SMT solvers. Research on SMT solvers has produced several robust tools (e.g. Barcelogic [42], Yices [43] and Z3 [38]). These solvers have made checking formulas with hundreds of thousands of variables tractable.

We decided to use Z3 as our SMT solver of choice, since it is a generalized framework, is well supported, and well documented. Z3 allows constraints to be encoded via different programming languages such as SCALA, C++, and Python. We chose to use Python (Z3Py). The *Constraint Formalizer* component encodes the development context into a Z3Py script (see Fig. 2) to create the "SMT formula" where logical connectives are combined with atomic formulas in the form of linear arithmetic inequalities. The Z3 solver then solves this formula (see Fig. 2, line# 22) to check for a satisfiable assignment, a mapping of task variables to the orders in which they should be performed.

Optimizing Solution Space*:* Finding a conflict free ordering is not our only goal. We aim to determine the *optimum* schedule, that is, if multiple conflict free development paths exist we match our recommended task orders with developer preferences to the extent possible. For an average problem there may be multiple solutions. Selecting a solution that is best suited to a problem requires traversal of the solution space [44].

We optimize the solution by first restricting the solution space by implementing a cost function that evaluates the quality of the solution to an integer value (e.g., number of preferences violated). We progressively add these values as a set of tighter constraints until we find the least cost solution (we use half-interval search). In our example, our optimized solution assigns Alice the following task order: (T_{A1}, T_{A2}, T_{A3}), whereas Bob is assigned: (T_{B3}, T_{B2}, T_{B1}). This indicates that Alice can follow her preferred sequence, whereas only one of Bob's "preferences" (T_{B2}) is satisfied. This gives us an optimization level of 4/6. An alternate satisfiable solution (among others) is – Alice: (T_{A3}, T_{A1}, T_{A2}) and Bob: (T_{B2}, T_{B1}, T_{B3}) – satisfying only1/6 preferences. Other sophisticated cost functions can be easily implemented and plugged into our *Optimizer* component (see Fig. 1).

Relaxing Constraints: If the development context is over constrained, then the SMT formula is unsatisfiable (UnSAT). This requires the identification of an UnSAT core—that is, a small unsatisfiable subset of the formula's clauses. Z3 returns unsatisfiable constraints as an "unsat" core. However, the unsat core is not minimal. An UnSAT core is minimal, if it becomes satisfiable whenever any one of its clauses is removed. It is always desirable to find a minimal UnSAT core because this will ensure that relaxing the least number of constraints provides a solution. Algorithms for finding minimal sets are domain and context specific.

In the context of parallel, collaborative software development, we identify four different approaches for relaxing constraint: (1) *conflict focus* that focuses on minimizing certain types of conflict (e.g., avoid test failures at all costs), (2) *team focus* that focuses on minimizing constraints for the entire team as compared to per developer, (3) *task focus* that attempts to relax constraints on the most constrained task or the reverse, that is, not relax constraints on certain critical tasks, and (4) *file focus* that prohibits relaxing constraints for critical files or encourages relaxing constraints based on files that are not "prone" to defects. These heuristics are necessarily team specific and will be influenced by the conflict distribution and resolution efforts in a project. Note, Cassandra is explicitly designed to allow teams the flexibility to choose the heuristics that best matches their need (or create new ones).

Here, we have implemented and evaluated two *conflict-focused* UnSAT heuristics. We decided to first implement this class of UnSAT heuristic since we only have conflict related data from the version histories of our subject programs, which allowed us to design and evaluate these heuristics. Information needed for the other classes of heuristics (information on critical files, team policies, or high-priority tasks) were not readily available to us.

The first conflict-focus heuristics (*basic*) favors indirect conflicts over direct conflicts, since direct conflicts are always flagged by the CM systems and can accurately localize the conflict. On the other hand, indirect conflicts are identified at a much later time (either during a system build, integration testing, or as defect), which makes localizing the cause of the conflict more difficult. Therefore, in this heuristic, if the UnSAT core contains constraints related to direct conflicts they are first relaxed.

Our second *empirically guided* approach prioritizes the different types of conflicts based on the effort it takes to resolve each. For example, we found that in a project like Perl, test failures are frequent (30.2%) and take longer time to fix (median of 14 days, see Table I). In this case, test-failure constraints should be the last to be relaxed. However, in the case of Jenkins, direct conflicts occur frequently (13.5%) and take the longest to resolve (median of 4 days). Therefore, in Jenkins direct conflicts should be relaxed last.

Reevaluating Constraints*:* As development progresses new constraints might be added, changing the satisfiability and soundness of our solution. For example, a developer might edit a different file than what we initially predicted. As changes take place in the project, we can also better determine the seriousness and probability of a conflict, which can guide constraint relaxation. Therefore, we need to reevaluate the constraint space periodically to ensure that satisfiability of the solution is up-to-date. A primary factor for deciding when to revaluate the constraint space, is that developers cannot be asked to change their current task because of changes in the development context. Therefore, to avoid interruptions or disruption of a developer's task, we re-evaluate the constraint space, when a developer has finished her changes. That is, we track commit operations from the workspace, which triggers a reevaluation of the constraint space. Therefore, by the time a developer is ready to work on her new task we have reevaluated and updated the recommended task order, if needed.

The reevaluation of the constraint space takes place in a matter of seconds. This reevaluation is computationally cheaper, since now we have to find a non-conflicting task for only one developer, while the tasks for other developers are fixed. Also (new) constraints are incrementally added so that the solver can reuse its constraint decisions.

TABLE III. EVALUATION RESULTS FOR OPEN SOURCE PROJECTS

	Jenkins								Perl		Voldemort		Storm
	Week		Month		Quarter		6 Months		6 Months		6 Months		Complete
	Avg.	High	Avg.	High	Avg.	High	Avg.	High	Avg.	High	Avg.	High	
Date Range	3/28/11 4/3/11	12/6/10 12/12/10	06/11	02/12	04/12 06/12	10/10 12/10	01/11 06/11	01/12 06/12	04/02 09/02	10/09 03/10	01/10 06/10	04/11 09/11	09/11 06/12
# Developers	24	8	8	16	24	12	38	48	59	22	14	8	20
Avg. # Tasks	7	1	4	2	3	4	5	5	7	2	6	10	8
Avg. # Files	3	5	4	3	4	9	4	4	4	3	4	8	6
# Changesets	11	30	32	75	49	203	258	384	50	135	78	167	171
Merge Failures	1	6	1	8	6	21	12	34	1	2	9	15	18
Build Failures	1	5	3	7	3	20	17	37	1	2	0	4	9
Test Failures	0	1	1	2	2	7	6	12	2	22	21	29	13
Direct Conflicts	1	6	1	8	6	21	12	34	1	2	9	15	18
Indirect Conflicts	1	6	4	9	5	27	23	49	3	24	21	33	22
Is SAT?	UnSAT	SAT	UnSAT	UnSAT	SAT	UnSAT	UnSAT	UnSAT	SAT	UnSAT	SAT	SAT	UnSAT
# DC Relaxed	1	-	0	0	-	0	0	0	-	0	-	-	0
# IC Relaxed	0	-	1	1	-	2	2	2	-	1	-	-	4
Conflicts Avoided	1/2	-	4/5	16/17	-	46/48	33/35	81/83	-	25/26	-	-	36/40
% Pref. Matched	63.64	36.67	46.88	33.33	22.45	23.15	15.12	13.80	44.00	51.11	7.69	10.78	12.87
CPU Time	0.211	0.219	0.224	0.484	0.304	2.881	5.617	16.651	0.307	0.64	0.532	1.162	9.076
Optimization													
% Pref. Matched	90.91	93.33	87.50	84.00	85.71	87.19	Time out		96.00	94.07	94.87	80.84	78.95
CPU Time (sec)	0.235	0.773	0.698	3.052	1.821	1.827			1.084	15.06	6.684	111.593	126.228

VII. EVALUATION

Here we aim to determine the feasibility of constructing a conflict minimizing, task scheduler such as Cassandra. Our evaluation goal, therefore, is to assess Cassandra's success in minimizing conflicts when given a development context with software conflicts. We measure success as the number of conflicts avoided (effectiveness) and the time taken to arrive at an optimized solution (efficiency).

We perform a set of controlled experiments by running Cassandra on data from a set of four open source projects (see Section III). We chose a range of time slices for our analysis: weekly (w), monthly (m), quarterly (q) and 6-monthly (6m). The development context from these time periods were then encoded as constraints and solved. Note that in real life, such analysis is likely to be performed at a daily or weekly basis.

We found that the changesets in these time periods were functionally dependent and retrospectively changing their order when integrating them into the master repository led to a different set of conflicts. These functional dependencies could have arisen because of hidden task precedence conditions or situations where developers, in addition to resolving conflicts also changed code functionality. Since we do not have control over these variables in our retrospective analysis, we stop after arriving at optimized, satisfiable task orders and do not integrate the changes (see Table III).

Our analysis shows that while the selected projects had substantial number of conflicts in total, their distribution is relatively low when analyzed at weekly to quarterly basis. Fig. 3 shows conflict distribution data for Jenkins on a quarterly basis. Therefore, to stress test the efficiency of our approach we use data from one of the projects (Storm) and mutate it to induce additional constraints (see Table IV).

All evaluation scenarios were evaluated on Z3 version 4.0 installed on a MacBook Pro 2.4 Ghz (Intel Core 2 Duo) with 4Gb of memory and running OSX 10.6.8.

A. Open Source Data Evaluation

We analyzed data from our test projects across different time slices (weekly, monthly, quarterly, and 6-monthly). Table III reports our evaluation results for all time intervals in Jenkins data and a single time interval for the other projects. Jenkins had the highest number of merges (505 merges, 100 developers) so we started our analysis with this project. We found that the conflict scenarios for shorter time intervals (w, m, q) were relatively trivial for Z3 to solve; therefore, here we only show 6-month time period analyses for Perl (185 merges, 51 developers) and Voldemort (380 merges, 33 developers). We analyzed the entire project history of Storm since it had only 88 merges and 24 developers.

For each time interval (w, m, q, 6m) in the project, we analyze the data and select a time period that is representative of that time interval. For each time interval, we select two time periods: one with an average number of conflicts and the other with a high number of conflicts (75% quartile). From each of these chosen periods, we extract the task scenarios: (1) the number of developers, (2) the number of changesets (tasks) and the files involved in that changeset, (3) the developers who committed these tasks, and (4) the number and types of conflicts along with affected resources. An example scenario is: a build conflict developed between changesets T_{A1} and T_{B1}, which included resources Shape.java and Square.java, and were committed by Alice and Bob respectively. Since conflicts are identified through a retrospective analysis, and for the build failure to occur changes by Alice had to precede that of Bob, we encode this conflict as a $<$ precedence relation. Test failures are similar and are treated as such.

Table III presents an overview of our results. We find that the "average" conflict periods in Jenkins had low conflict instances (ranging from 1-12) for all time intervals (w, m, q, 6m). Finding a satisfiable (unoptimized) solution was trivial for these scenarios (5.6s was the longest time for 6m). For the "high" periods, conflict numbers ranged from 6-34 and the longest

TABLE IV. EVALUATION RESULTS FOR TASK GENERATOR

	Task Generator Scenarios											
	Week			Month			Quarter			6 Months		
	Low	High	Avg.	Low	High	Avg.	Low	High	Avg.	Low	High	Avg.
# Developers	3	3	3	2	8	6	6	5	3	5	13	5
Avg. # Tasks	4	5	4	6	6	6	8	8	9	8	8	8
Avg. # Files	4	3	3	3	5	3	3	3	3	3	4	3
# Changesets	14	15	12	12	50	36	50	41	27	42	105	42
Merge Failures	0	1	0	0	21	10	8	8	4	7	79	9
Build Failures	0	4	0	2	21	4	9	31	11	6	14	38
Test Failures	2	2	2	3	2	1	26	44	15	60	142	54
Direct Conflicts	0	1	0	0	21	10	8	8	4	7	79	9
Indirect Conflicts	2	6	2	5	23	5	35	75	26	66	156	92
Is SAT?	SAT	SAT	SAT	SAT	SAT	SAT	SAT	UnSAT	UnSAT	UnSAT	UnSAT	UnSAT
Basic												
# DC Relaxed	-	-	-	-	-	-	-	2	2	1	23	1
# IC Relaxed	-	-	-	-	-	-	-	58	23	40	121	66
# Runs								3	2	3	11	3
Conflicts Avoided	-	-	-	-	-	-	-	23/83	5/30	32/73	91/235	34/101
Empirically Guided												
# DC Relaxed	-	-	-	-	-	-	-	3	2	25	57	5
# IC Relaxed	-	-	-	-	-	-	-	28	15	35	82	27
# Runs								8	3	6	5	8
Conflicts Avoided	-	-	-	-	-	-	-	52/83	13/30	13/73	96/235	69/101
% Pref. Matched	7.1	20.0	33.3	16.7	8.0	11.1	12.0	7.3	18.5	9.5	16.2	11.9
CPU Time	0.133	0.143	0.125	0.141	0.326	0.231	0.342	1.104	1.658	0.99	10.792	1.495
Optimization												
% Pref. Matched	85.7	66.7	66.7	66.7	60.0	83.3	48.0	82.9	85.2	54.8	16.4	50.0
CPU Time (sec)	0.355	0.346	0.259	0.354	4.466	1.158	12.961	11.618	51.973	10.644	156.33	13.339

time to reach a solution was 16.65s, indicating that solving conflict constraints using our approach encoded in Z3 is trivial.

Cassandra also avoided a majority of conflicts in these scenarios. Perl, Voldemort, and Storm have similar characteristics.

When we attempt to optimize task orders to match developer preferences (the order in which tasks appeared per developer was treated as their preference order), we find that the time required to attain a solution increases. In fact, finding an optimized task order for the 6m periods in Jenkins took longer than 3 minutes. We used a threshold of 3 minutes to terminate our analysis, since a delay of 3 minutes or more is sufficiently long to be considered disruptive by end users and can potentially frustrate them. However, note that this time period had a large changeset (258 and 384 for "average" and "high" activity periods). It is highly unlikely that a team will need to schedule such a large number of tasks. We note that our optimization matches developer preferences to a large extent (78.95% - 96%).

In situations where the development context was unsatisfiable, the UnSAT core was small (largest set being 4 for Storm). Interestingly, all UnSAT cores only had a single type of conflict. Since these UnSAT cores were small and had a single conflict type, both heuristics for relaxing constraints evaluated to be the same. We only show results of the *basic* UnSAT heuristic in Table III.

B. Task Simulation

Here we test Cassandra and its efficiency in simulated task contexts with high numbers of constraints. We do so by simulating constrained tasks by mutating the data of one of our test subjects. We choose Storm as our subject since it is a small project and had the least skew in its development activity (see Fig. 3 (b)). The mutation is performed by a Task Generator component that first creates a distribution of the changesets and conflicts for the entire project history. It then identifies the statistics (mean, sd) of the conflict distribution for each time interval (w, m, q, 6m). It then uses this data to create a normal distribution from where it randomly creates three scenarios for each time interval. These scenarios include low (25% quartile), average, and high (75% quartile) numbers of conflicts.

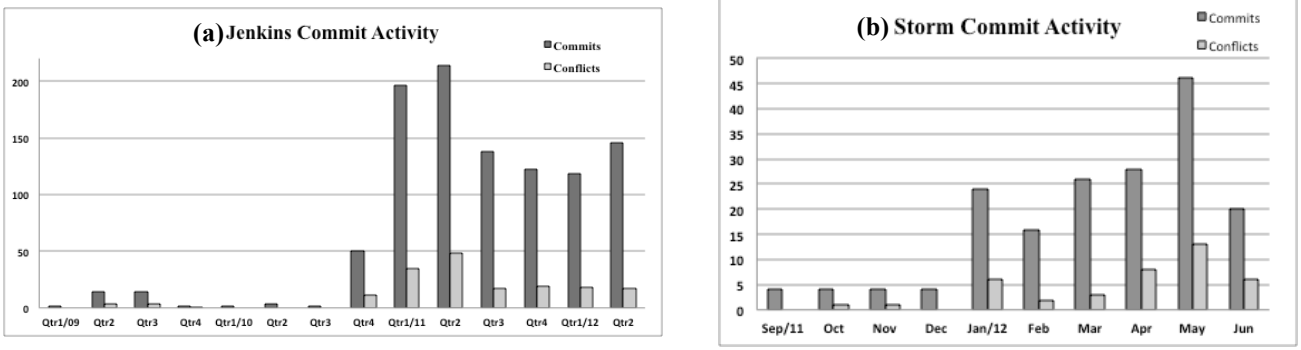

Fig. 3. Conflict Distribution in projects (a) Jenkins - Quarterly, (b) Storm - Monthly.

The scenarios are encoded by the Constraint Formalizer and solved (see Table IV). As we can see the number of conflicts in each scenario is much higher than previously observed, especially for the "high" conditions. However, despite these high numbers of conflicts, the weekly and monthly data were SAT, taking only seconds to complete; 0.33s was the longest for the "high" monthly condition. Most scenarios for the quarterly and 6-monthly period had UnSAT cores. Cassandra identified optimized solutions much quicker than in the previous evaluation. However, because of the higher number of conflicts finding task orders that match developer preferences was more difficult (16.4% to 85.7%). Note that the high, 6m periods has the lowest preference matching (16.4%), but faces 235 conflicts.

As expected, the large number of conflicts in these scenarios causes Z3 to return larger UnSAT cores. Table III presents the results from our two UnSAT heuristics. Note that finding the minimal UnSAT core takes several iterations (#Runs). A set of conflicts is released at each iteration based on the UnSAT heuristic used, after which the solver reevaluates the rest of the constraint space. This process continues until a satisfiable solution is reached.

Note that of the two approaches, the Empirically Guided approach performs better when we compare the number of conflicts avoided. This is so, since in Storm merge conflicts had the highest incidence, followed by test failures and then by build failures. In the Empirically Guided approach constraints are relaxed in the reverse order. Whereas, in the Basic approach direct conflicts are first relaxed, followed by build and then by test failures. This shows that different heuristics can affect the quality of a solution. Also note, that currently we relax all constraints of a particular type at every iteration; we can further refine the process to select a subset of a type of conflict from the UnSAT core. Despite, the high number of conflicts and higher number of iterations required for relaxing constraints, solutions were still found quickly (156.33s was the longest time needed for the "high" 6m period).

VIII. Threats to Validity

Our empirical analyses and evaluations naturally leave open a set of potential threats to validity, which we explain here:

Construct: We use version histories of open source projects to identify conflicts and when they were resolved. When performing this analysis we have assumed: (1) each developer only has a single line of development that is regularly synchronized with the master repository, (2) if a developer faced a conflict then she exclusively worked to resolve that conflict (merge, build, test) in subsequent merges until the conflict was resolved, and (3) build and test scripts available in the versioning history repository are robust and have good coverage.

Internal: When scheduling tasks with Cassandra we assumed that all tasks were of equal length and functionally independent, allowing them to be reordered. Based on this information, we predicted the number of conflicts Cassandra could have avoided. However, this was clearly not the case, which precluded us from being able to retrospectively integrate changes based on our (reordered) task orders. The main goal of this paper was to determine the feasibility of constructing Cassandra; future studies will analyze current task contexts of teams.

External: Our retrospective studies only focused on four open source projects hosted on GitHub. While, we ensured that we chose projects that were popular, had high parallel activity, and included conflicts, they might not be representative of other projects. Moreover, these are open source projects where contribution is voluntary and a small group performs the majority of work. Commercial projects will have different characteristics. However, studies have shown that such projects have higher number of parallel changes and conflicts, which suggests that Cassandra will be even more useful in such settings.

IX. Conclusions

Collaborative software development allows developers to work in parallel, which can result in software conflicts. Such conflicts are a norm rather an exception; even in open source projects where developers often contribute in their spare time and the majority of work is performed by a small core group. We analyzed four popular open-source projects and more than one year's worth of data per project. We found that all projects faced substantial number of conflicts (ranging from 34% to 54%) and required resolution times spanning multiple days. Moreover, each project had different distributions of different types of conflicts and different resolution times for each conflict type.

Given that conflicts are bound to occur in any collaborative development scenario and their resolution takes time, even when detected early, we present a novel conflict minimization technique. This technique implemented in our research prototype Cassandra proactively identifies conflicts and other constraints in a development context to determine task orders that will avoid the incidence of conflicts. We evaluated the feasibility of constructing a system such as Cassandra by evaluating its scheduler on data from four open source projects. Our results show that Cassandra is able to solve a large set of constraints and able to avoid the majority of conflicts (that were identified in our retrospective analysis).

This work was a first exploration in constructing a system such as Cassandra. There are several possible directions for enhancements. First, we will explore automated data mining and machine learning techniques to automate the context generation, so that we can provide an initial set of resources to be edited per task, which the user can then refine. Second, we will explore program dependency analyses to refine our conflict identification technique. Finally, we will implement other UnSAT heuristics (task, file, team focused). We will perform qualitative studies, including surveys of development teams to guide the heuristic implementation.

Acknowledgment

We thank Matthew Dwyer for his guidance on constraint solving. We thank Patrick Wagstrom and Corey Jergensen for the GitMiner tool. This research is supported by grants NSF-0414698, 1016134 and AFSOR FA9550-09-1-0129.

References

[1] 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, pp. 308–337, 2001.

[2] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *26th International Conf. on Software Eng.*, pp. 563–572, 2004.

[3] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," *19th ACM SIGSOFT symposium*, pp. 168–178, 2011.

[4] A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantír: Raising Awareness among Configuration Management Workspaces," *25th International Conf. on Soft. Eng.*, pp. 444–454, 2003.

[5] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys*, vol. 30, pp. 232–282, 1998.

[6] J. Estublier and S. Garcia, "Process Model and Awareness in SCM," *12th International Workshop on Software Configuration Management*, pp. 69–84, 2005.

[7] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Trans. on Software Eng.*, vol. 28, pp. 449–462, 2002.

[8] M. L. Guimaraes, "Improving Early Detection of Software Merge Conflicts," *2012 International Conference on Software Engineering*, pp. 342–352, 2012.

[9] C. R. B. de Souza, D. Redmiles, and P. Dourish, "'Breaking the Code', Moving between Private and Public Work in Collaborative Software Development," *International ACM SIGGROUP Conf. Supporting Group Work*, pp. 105–114, 2003.

[10] P. Dewan, "Dimensions of Tools for Detecting Software Conflicts," *2008 International Workshop on Recommendation Systems for Software Eng.*, pp. 21–25, 2008.

[11] R. E. Grinter, "Using a Configuration Management Tool to Coordinate Software Development," *Conference on Organizational Computing Sys,* pp. 168–177, 1995.

[12] D. K. Shao, "Evaluation of Semantic Interference Detection in Parallel Changes: an Exploratory Experiment," *23rd IEEE International Conf. on Soft. Mainten.*, pp. 74–83, 2007.

[13] T. Zimmermann, "Mining Workspace Updates in CVS," *4th International Workshop on Mining Soft. Repos.*, p. 11, 2007.

[14] J. Biehl, M. Czerwinski, G. Smith, and G. Robertson, "FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams," *SIGCHI Conference on Human Factors in Computing Systems*, pp. 1313–1322, 2007.

[15] L. Hattori and M. Lanza, "Syde: A Tool for Collaborative Software Development," *32nd ACM/IEEE International Conf. on Software Eng.*, vol. 2, pp. 235 –238, 2010.

[16] C. O'Reilly, P. Morrow, and D. Bustard, "Improving Conflict Detection in Optimistic Concurrency Control Models," *2001 ICSE Workshops on SCM*, pp. 191–205, 2003.

[17] P. Dewan and R. Hegde, "Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development," *10th European Conference on Computer-Supported Cooperative Work*, pp. 159–178, 2007.

[18] P. Dourish and V. Bellotti, "Awareness and Coordination in Shared Workspaces," *ACM Conference on Computer-Supported Cooperative Work*, pp. 107–114, 1992.

[19] J. Wloka, B. Ryder, F. Tip, and X. Ren, "Safe-commit analysis to facilitate team software development," *31st International Conf. on Software Eng.*, pp. 507–517, 2009.

[20] J. M. Costa, M. Cataldo, and C. R. de Souza, "The Scale and Evolution of Coordination Needs in Large-Scale Distributed Projects: Implications for the Future Generation of Collaborative Tools," *SIGCHI Conference on Human Factors in Computing Systems*, pp. 3151–3160, 2011.

[21] 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.

[22] C. R. B. de Souza and D. Redmiles, "An Empirical Study of Software Developers' Management of Dependencies and Changes," *30th Intern. conf. on Soft Eng.*, pp. 241–250, 2008.

[23] R. E. Grinter, "Recomposition: Putting It All Back Together Again," *1998 ACM Conference on Computer Supported Cooperative Work*, pp. 393–402, 1998.

[24] C. R. B. de Souza and D. Redmiles, "The Awareness Network: Should I Display My Actions to Whom? and, Whose Actions Should I Monitor?," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 325–340, 2011.

[25] B. Curtis, "Insights from Empirical Studies of the Software Design Process," *Future Generation Computer Systems*, vol. 7, pp. 139–149, 1992.

[26] J. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter, "An Empirical Study of Global Software Development: Distance and Speed," *23rd Intern. Conf. on Soft. Eng.*, pp. 81–90, 2001.

[27] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter, "Distance, Dependencies, and Delay in a Global Collaboration," *2000 ACM Conference on Computer Supported Cooperative Work*, pp. 319–328, 2000.

[28] GitHub, "Social Coding: Building Software Together." [Online]. Available: https://github.com/.

[29] "Java based tools for extracting information from GitHub." [Online]. Available: https://github.com/pridkett/gitminer.

[30] "The World's Leading Graph Database." [Online]. Available: http://neo4j.org/.

[31] "A graph traversal language." [Online]. Available: https://github.com/tinkerpop/gremlin/.

[32] A. Sarma, D. Redmiles, and A. van der Hoek, "Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 889–908, Aug. 2011.

[33] M. Kersten and G. C. Murphy, "Mylar: A Degree-of-Interest Model for IDEs," *4th International Conference on Aspect-Oriented Software Development*, pp. 159–168, 2005.

[34] SourceForge.net, "Dependency Finder." [Online]. Available: http://depfind.sourceforge.net/.

[35] J. Anvik and G. C. Murphy, "Reducing the Effort of Bug Report Triage: Recommenders for Development-Oriented Decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, article 10, pp. 1–35, 2011.

[36] S. C. Brailsford, C. N. Potts, and B. M. Smith, "Constraint Satisfaction Problems: Algorithms and Applications," *European Journal of Operational Research*, vol. 119, no. 3, pp. 557–581, Dec. 1999.

[37] V. Kumar, "Algorithms for Constraint Satisfaction Problems : A Survey," *AI Magazine*, vol. 13, no. 1, pp. 32–44, 1992.

[38] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," *Theory and Practice of Software*, pp. 337–340, 2008.

[39] R. Dechter and D. Frost, "Backtracking Algorithms for Constraint Satisfaction Problems," UCI Technical Report, 1999.

[40] R. Barták, "Theory and Practice of Constraint Propagation," *3rd Workshop Constraint Prog. Deci. Cntrl*, pp. 7–14, 2001.

[41] N. Jussien and O. Lhomme, "Local Search with Constraint Propagation and Conflict-Based Heuristics," *Artificial Intelligence*, vol. 139, no. 1, pp. 21–45, 2002.

[42] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "The Barcelogic SMT Solver," *20th International Conf. on Comp. Aided Verif.*, pp. 294–298, 2008.

[43] B. Dutertre and L. De Moura, "The Yices SMT Solver," *Tool paper at http://yices.csl.sri. com/tool-paper.pdf*, 2006.

[44] R. Nieuwenhuis and A. Oliveras, "On SAT Modulo Theories and Optimization Problems," *Theory and Applications of Satisfiability Testing*, vol. 4121, pp. 156–169, 2006.