Information Needs for Integration Decisions in the Release Process of Large-Scale Parallel Development

Shaun Phillips
University of Calgary
Department of Computer Science
phillist@ucalgary.ca

Guenther Ruhe
University of Calgary
Department of Computer Science
ruhe@ucalgary.ca

Jonathan Sillito
University of Calgary
Department of Computer Science
sillito@ucalgary.ca

ABSTRACT

Version control branching allows an organization to parallelize its development efforts. Releasing a software system developed in this manner requires release managers, and other project stakeholders, to make decisions about how to integrate the branched work. This group decision-making process becomes very complex in the case of large-scale parallel development. To better understand the information needs of release managers in this context, we conducted an interview study at a large software company. Our analysis of the interviews provides a view into how release managers make integration decisions, organized around ten key factors. Based on these factors, we discuss specific information needs for release managers and how the needs can be met in future work.

Author Keywords

Parallel development; release management; version control; integration; decision support; team meetings

ACM Classification Keywords

D.2.9 Software Engineering: Management

General Terms

Management

INTRODUCTION

Software engineering decision support qualifies the process of decision-making through the different stages of the software life-cycle [14], but decisions can only be as good as the information they are based upon. Version control systems are generally considered to be a vital information source in development projects [1, 17]. Version control *branching* refers to the parallelization of development in a version control system. With branching, different software development teams or individual developers can work in separate branches or "copies" of a code base. The decision to branch is typically motivated by the need to prevent workflow disruption, and reduce overall cycle-time, by having teams stabilize their development in isolation before integrating with their organization. *Integrations* involve merging code, resolving merge conflicts, and compiling and testing the merged code.

Different organizations use different numbers and arrangements of branches. As an example, Figure 1 depicts the branching model used by our interview participants' organization. In this model, developers are assigned to teams that make code changes primarily in the feature branches. Staging branches are used for integrating work from several related feature branches in preparation for further integration in the mainline branch. In addition to integrating code up the tree, code also flows downward to ensure that development teams do not operate on out-of-date code bases.

As a release approaches, decisions need to be made around what integration work will be required. Though previous research has looked at various aspects of planning and managing releases (e.g., [15]), the focus of our ongoing research is on understanding information needs for managing releases in a branched context. The interview study reported in this paper aims to identify current approaches and challenges around release management in branched projects, including identifying the information needs of release managers making integration decisions.

From our analysis of the interview data we have identified ten decision factors that are important considerations when making integration decisions and derived several key information needs. We discuss the extent to which these information needs are met by the participants' organization and how they can be implemented in future work.

Several aspects of this study are relevant to the CSCW community. First, the use of version control requires coordination between potentially 1000s of individuals and this complexity demands considerable computer support. Second, while version control is generally considered in the scope of software engineering, it is also required by many organizations that require co-development of other intellectual goods. Finally, this work considers version control from the perspective of key individuals in an organization who are supported by technology and captures practices and problems that have evolved over time.

RELATED WORK

Release management and version control integration are topics well-represented in academic literature. Exploring where

CSCW12, February 11–15, 2012, Seattle, Washington, USA.

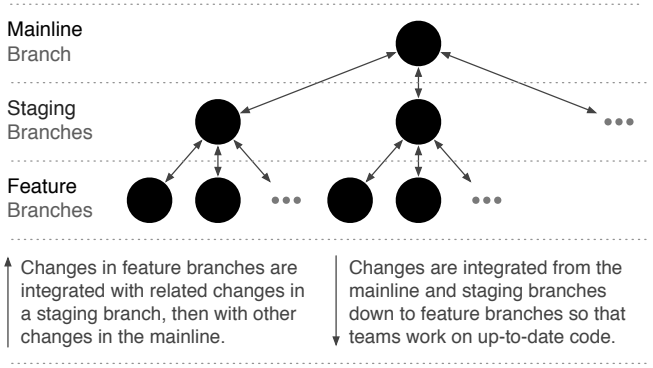

Figure 1. Branching model used by the software company in our study.

these areas intersect is a key contribution of this study. Findings from our interviews show that there are opportunities to improve integration decision-making with software support. Previous work has shown these findings to be applicable to other organizations; for example, Wright and Perry's case study on the Subversion project [21] discussed how certain integration decisions delayed the 1.5 release of the product.

Several studies have explored merging at the code level, for example, the paper from Mens [9]. Current high-level merging practices (such as merge frequency in an organization) were explored in our broad 2010 user survey [12]. The impact of some merging practices was explored by Perry and Votta in an observational case study of large-scale parallel development [11]. A key finding from the study was that development parallelism was higher in the organization than anticipated by tool builders. While merging practices are an important consideration in this study, the focus is on the higher-level decision making process for releasing branched products, of which the practices are one of many factors.

Our study builds on existing release management literature by addressing concerns and information needs for parallel environments. Previously, Sassenburg has briefly discussed how collaborative efforts can impact release decisions [18]. Ruhe has produced extensive work on release processes [15], though the book does not explicitly address issues that are unique to parallel environments. Walrad et. al. have considered how release maintenance can benefit from branching [20] (e.g., creating branches specifically for release purposes), but this differs from our study in that we are focused on product development in branches and the associated release concerns.

Many studies have examined planning in software development projects, such as that from Rönkkö et. al. [13]. Here, we are concerned with integration decisions—an aspect of planning in branched projects. These decisions are usually reached collaboratively between branch representatives and other release stakeholders. Saarelainen et. al. note that group decisions such as these are widely used both in private and public organizations and attitudes towards them are generally positive or neutral [16]. One widely recognized method to improve the effectiveness of group decisions is the use of Group Decision Support Systems. The systems combine computer, communication, and decision technologies to support group activities [8]. Similarly, this paper attempts to better understand the information needs for computer support required for group decision-making towards integration decisions in large-scale parallel software development.

While previous research has studied information seeking for decision support in various contexts including the daily information needs of software developers [7], to the best of our knowledge, no work has looked at the information seeking behaviour of release managers as they make plans for integrating source code from different teams. In this context, release managers need timely information from individuals working on different teams, possibly with similarities to command and control situations where the concept of individual, intragroup, and intergroup "situational awareness" is central [19]. The information seeking process has been conceptualized to include activities identifying, seeking, and retrieving needed information [6] as well as sense-making [10]. The goal of our study is to build on this previous work by studying these information seeking activities in the release planning context.

METHODOLOGY

Organization Information

Our study involved semi-structured interviews with an unaffiliated software organization of roughly 1000 developers. Development teams are organized in the hierarchy depicted in Figure 1: developers are divided amongst 15 product units, or major components, and each product unit is composed of feature teams. Developers on the same team tend to be co-located, but the teams themselves may not be in close proximity (some are in different geographic regions).

The flagship product, approximately 100 million lines of code, has been in development for more than ten years and has undergone several major releases, generally in a two-year cycle. Each product unit has a release team which is part of a larger release organization. Development teams will collaborate with their release team to establish schedules, verify release quality criteria, and assess integration risk. This information is then communicated in release organization meetings to determine overall project strategy, status, and risk. Many different project stakeholders will attend the meetings throughout the release cycle and their input is one of many information sources that must be considered when making complex integration decisions.

Participants

Seven practitioners (P1...P7) in the target organization, with industry experience ranging from 6 to 17 years, participated in our study. P1 and P2 are both release managers who develop release schedules, coordinate development teams, and support the version control and test systems; P3, P4, and P5 manage and lead teams of developers and testers, perform cross-team coordination for end-to-end product scenarios, and are responsible for delivering and integrating their features with the rest of the product group; P6 and P7 develop the product software and perform the actual integrations and testing.

Table 1. Factors identified in our analysis.

Categories	**Factors** (Subcategories)
Branch Evolution	Code Churn, Conflicting Changes
Branch Health	Bugs & Bug Counts, Feature Completeness, Test Runs & Results
Project Awareness	Dependencies, System Architecture, Infrastructure
Project Traffic Control	Branch Organization, Code Flow

Study Design

Each interview was conducted one-on-one by the first author of this paper. The interviews were one hour in length and the audio was recorded and transcribed verbatim. The interviews were semi-structured in that they were initiated with broad questions stemming from our branching and merging survey [12], but allowed to flow as they delved into the participants' integration experiences, and to what extent these experiences have affected their product releases. We specifically targeted participants with different responsibilities in order to capture the diverse perspectives and information needs of those involved in integration and release decisions.

Analysis

To analyze the transcribed interview data we first performed open-coding to identify concepts and passages in our data of particular analytic interest [4]. We next grouped these codes into categories and subcategories with the goal of understanding our participants' integration decision making and information needs. These categories and subcategories are discussed in detail in the next section.

FINDINGS

Given the extensive use of branches in the participants' organization, integration work is an important part of the development process. If an integration results in code with serious problems (e.g., does not compile or run) the product is said to be "on the floor" (P1). The time taken to perform an integration, called its latency, has proven difficult for our participants to predict. Latency ranged from a day to many weeks. Even though our participants' organization incorporates integration costs into its release plans, integration issues continue to result in significant release delays. One particular incident, where over a dozen branches were integrated into the mainline branch in a two-week period, resulted in the project "being on the floor for 45 days" and "threw our schedule totally out of whack" (P1). After that incident, "everyone realized that obviously code flow and scheduling have to be best friends" (P1).

Our analysis has identified ten factors that capture what our participants have learned about making integration decisions. These factors are organized into four categories: branch evolution, branch health, project awareness, and project traffic control. The categories and factors are listed in Table 1 and described in the remainder of this section.

Branch Evolution

In a branched project, branches are changed in parallel and paying attention to these changes is important for planning how and when to integrate. If the amount of change being introduced is large or the changes are in some way incompatible, the integrations may be particularly problematic.

Code Churn

When planning integration work, our participants consider the amount of code churn, which is the total number of lines of code changed and not yet integrated, as a gross measure of how difficult an integration will be. Broadly speaking, the more different two branches are the more likely that there will be integration problems that are difficult to diagnose.

When dealing with thousands of changes across dozens of branches on a daily basis, it is not feasible to throughly examine the impact of individual changes, but code churn is used to make a "high level assessment of risk" (P6) for integration decisions. Indeed, "the biggest [integration] cost is the payload that each integration is bringing into a common branch. The larger the payload...the more chances that something will not work well" (P5) and it was "always the case that a lot of code changes meant a lot of latency" (P1). That is, larger amounts of churn have resulted in longer integrations and increased pressure on the release schedule.

The volume of changes that have to be integrated is also a function of the number of branches being integrated over some period of time, and this number tends to increase towards the end of project milestones—the last chance for significant changes before a release date. When planning integrations, P2 considers how many branches have "significant payload" (or code churn) to integrate at the end of a milestone, because he has found that even integrating four branches with significant changes in one day is "almost impossible to do" without breaking the mainline.

If at any time there is an integration problem that delays subsequent integrations, the volume of changes that need to be integrated in a potentially short amount of time grows as the delay continues. This "mounting pressure" (P2) increases the likelihood of additional integration problems. For example, if an integration results in "a major performance regression" our participants do not allow integrations from other branches until "a root cause analysis on the regression" is completed. A decision then needs to be made about whether to continue delaying integrations to avoid the problem getting "compounded by more code getting piled on top of it" (P2), backing out the integration that resulted in the regression, or allowing the known regression to remain in the mainline. In any given situation, it is difficult to know which option introduces more risk to the schedule.

Being able to view where pressure is building in the project can help influence better integration decisions; in particular, with integration effort estimation and when to sacrifice quality to relieve the code churn pressure. If detected with enough warning, steps can be taken to mitigate risk through stricter integration regulation, particularly when close to a release date.

Conflicting Changes

Conflicting changes are incompatible code modifications—they can be literal, such as changes to the same line of code, or they can be subtle, for example, when "two functions got added with the exact same name in different branches, except one was 'virtual public' and the other 'public virtual'" (P7). These more subtle conflicts are not detected by the version control system during a merge and "are the harder ones that cost more" (P7).

In branched development environments, conflicting changes typically are not discovered until branches integrate. This latency can cause conflicts to build up and be compounded, increasing the risk of delays. The integration policy of the organization attempts to mitigate these risks by first requiring branches to integrate downstream (for example, from the mainline into a staging branch), resolve any conflicts and test, before integrating upstream. This process ensures that any conflicts are fixed locally in the branch before being shared with the rest of the organization.

This process breaks down, however, when several branches attempt to integrate within a short time period. These situations can arise when, due to scheduling or business demands, branches must integrate by a certain date. A branch's initial downstream integration can take several days to complete, and in that time other branches will integrate upstream, forcing the branch to take another downstream integration to absorb the latest changes. As described by P3:

> "Then I submit, and I come up with a bunch of merge conflicts, because of all of a sudden other people have checked in [to the mainline]. Well, at that point, what do I do? The best that I can hope for is resolve the merge conflicts, try and do a build, and then just hope for the best. Or I start the whole integration process over again, in which case I never finish."

Individual branches being integrated may be in good states, but when conflicts are resolved on-the-fly, there is a greater risk of defects. The threat to the release schedule in this case is the "hoping for the best." The code base is far too complex to accurately predict whether conflicts were resolved correctly without running the test suite. But if strict adherence to the downstream before upstream integration policy is required, there can be even greater delays. A key integration decision in this context is choosing an ordering for integrations that can reduce the amount or severity of conflicts. To gather information on conflicts to help with integration planning, P7 often performs a mock integration several days prior to the merge date in order to see "how many conflicts" there are and whether "anything looks bad."

For higher-level integration ordering decisions, a broader view of the project is required. Within the organization, the code base is structured in such a way that directories roughly correspond to development teams or branches. As a result, different parts of the code are expected to be modified in different branches. Our participants consider a large volume of change outside of these directories to indicate the potential for conflicting changes; a situation P2 described as "a storm waiting to happen, there is a convergence zone." To minimize risk, this information can be used to create a more effective integration ordering.

Branch Health

Some considerations around code churn and conflicts, as discussed above, may encourage frequent integrations, however our participants tended to delay integrations based on issues related to *branch health*. Problems in a branch, if integrated up, result in those problems being spread to other branches, impacting the work of other teams and the stability of the mainline. Our participants are concerned with three aspects of branch health in particular: (1) bugs and bug counts, (2) feature completeness, and (3) test runs and results.

Bugs and Bug Counts

The participants frequently identified bugs as a key input to integration decisions; in particular, estimating when a branch will be ready to integrate for scheduling purposes. In this case, the bugs are those that are open in a branch before it integrates. Bug information was used differently by different participants.

Participant P3, a development lead, can not assess which individual bugs in the different branches are "more risky." Across the project there can be hundreds or thousands of open bugs depending on the point in the development cycle. A deep consideration of what the bugs mean is infeasible, so instead he is simply concerned with the number of open bugs for each branch as a predictor of which "branch is at a greater risk of not making it because there is more [expected] churn, so there is more things that could go wrong." P2, a release manager, also uses the number of open bugs as a predictor of future code churn:

> "You take any 30 bugs...some will be small, some will be big, and they will average out in code churn dispersion. So [near a release date], we are pretty good just by looking at bug count and directly correlating that to risk, in terms of how much more stabilization there is going to be, and do we need to move our milestone out. Bug count is really what drives a lot of those decisions."

It is important to note that P2 distinguishes between the time periods close to release dates and during development milestones. Development milestones, also known as coding milestones, are the periods of work where new features tend to be introduced (roughly corresponding the alpha stages of the release). At development milestones, the bug counts tend to be very large and the association with churn is not very accurate—likely because they are accompanied by millions of lines of code in new development. Near release dates, however, typically only bug fixes are integrated (and are heavily regulated) and the association with churn is more accurate. But in general, predicting the impact of bugs on the release schedule remains a "gut feel" (P2).

It may seem unwise to have active bugs during integrations as the defects will be propagated to other branches, "but in reality [there are] always bugs when integrations happen" (P7). There are many reasons to integrate features that have bugs;

for example, if a branch with a dependent feature needs the code (even if incomplete) or would be at risk for delaying the release schedule. The justification is summarized by P4:

> "The style we choose is to let people integrate with exceptions as long as we discuss it and we understand the risk on the schedule, and we approve it. The other side would be, 'no, you cannot integrate unless you fix everything', we do not do that. This latter style makes the predictions you are asking about easier, but at the same time we can miss opportunities. You can end up cutting a feature prematurely from an integration."

Within the organization, bugs are an important input to integration decisions as they can be associated with future code churn. This risk assessment supports the difficult integration decision to either delay an integration because of open bugs, or permit the integration with defects in order to not delay the product or fail to deliver certain features to customers.

Feature Completeness

Feature development in our participants' organization is parallelized among many branches (see Figure 1). Work on these features is broken down into "work items" (P5) and a branch schedule details when those work items should be completed. Tracking progress on work items "provides a sense of completeness" or a measure of the "maturity" (P5) of the code in a given branch; a branch with many outstanding work items, "might still be in a state where nothing is really working" (P5).

In addition to preferring to not integrate code until there are no open bugs, our participants prefer to integrate code once "the work items are complete and no bugs are pending" (P5). However scheduling issues, such as plans around end-to-end testing of product scenarios that require the integration of code across multiple branches, may force features to be integrated before completion to work out any broad integration problems as development progresses.

Knowledge about the "schedule and completeness" (P7) of the features in a branch being integrated is used to estimate "how many integrations you need after your [first] integration" (P4) in order to completely integrate the features. Determining the number of subsequent integrations needed is mostly based on the velocity of work being completed in the branch. Because there are significant costs associated with integrations themselves—in this project, an integration takes 24-48 hours if there are no problems—P4 has found that estimating the number of future integrations is a useful input to scheduling decisions.

While tracking work items is a typical way to track progress in development projects, the distinction to make in branched development environments is that this data needs to be considered, and presented effectively, at the branch level.

Test Runs and Results

Testing was often raised as an important consideration by the interview participants. Of course, there are many testing considerations in the software release process. And like feature completeness, tracking test results is a typical way to measure progress and quality. But here, our primary consideration is the relationship between testing and integration decisions in a parallel development environment. The testing that occurs during an integration "is how I am going to judge the success or failure of the integration" (P3).

However, the time to stabilize during an integration is unpredictable, which complicates scheduling decisions. For an integration to be complete, the branch must be healthy and stable, but "given that the feature [branch] is good, we do not actually know the mainline will be good once it gets in" (P6). We have previously discussed a severe 45 day delay that occurred when many branches integrated in a short time period. In that situation, the majority of the 45 days was spent stabilizing the mainline branch. A contributing factor to this stabilization unpredictability is that "if you are under resource constraints, it is possible your branch will only run 5% of the actual tests versus the [mainline] running 100% of the tests" (P7). In other words, because the full test suite contains hundreds of thousands of tests that would take over a week to execute, teams must settle on running a subset of the suite in their branches. So one of the causes of the delay was that different branches run different sets of tests, and when a greater proportion of the tests are run after an integration, the outcome is not predictable.

The strategy taken by the organization to reduce integration stabilization time and force more predictability into the release process has been to redefine the integration tests. The original integration test suite consisted of about 500 tests that had been slowly accumulating over the course of 8 years. Recently, the 500 tests have been replaced with 24 new integration tests that represent important end-to-end customer scenarios, which all feature branches have some stake in. Great importance is placed on the tests not being "flaky" (P1), in that there is confidence that a failure during an integration represents a true product bug. The new approach has so far led to fewer major integration problems than the previous approach, but it has not yet been used through a full release cycle.

Integration decisions involving testing are typically about how much time to plan for stabilization, and the actions that can be taken to reduce the stabilization time. Our participants would appreciate more information about the testing being done on each branch to help estimate the stabilization costs for the integrations. While they have found that these costs can be minimized by optimizing the tests that are run during integrations, this remains a difficult problem for their organization.

Project Awareness

Project awareness covers knowledge of the system under development and its supporting infrastructure. Because the participants' system is extremely large and complex, it is infeasible for those making integration decisions to possess system-wide, in-depth technical knowledge. Instead, the interviews identified three important high-level considerations: the dependencies between branches, where the development in each

branches is situated in the system architecture, and the reliability of the project's infrastructure systems.

Dependencies

Awareness of dependencies, and especially *new* dependencies, was frequently cited as a factor in integration decisions; in particular, in how much time to plan for an integration. To estimate the time required to integrate a feature branch through a staging branch and into the mainline, an important consideration is how that code will affect the branches at each step, and how other branches will affect it, so a branch dependency becomes a scheduling dependency:

> "Parallel development exercises your ability to think about the bigger picture. Now, you cannot just sit in your office and say, 'The feature is going to be done and integrate on that day.' No, you have to go and look around to figure out what are other teams integrating that depends on you, or what you depend on them, so that [the mainline] is not on floor once everybody meets there. So you need to do dependency analysis and be very familiar with each others schedules." (P5)

Part of the problem is that defects caused by dependency changes may require significant effort to fix and may force design changes, which can cause stabilization times to balloon. In fact, changes in APIs that are used by several features have been a "good portion" (P1) of the most disruptive release delays in the project. P6 describes the implications of dependency failures encountered during merges:

> "Functional dependencies can cause a long tail of bugs to be fixed, and can have bugs that require the involvement of many teams. So when a behaviour is changed, there is no way to change it back. But if you broke somebody, then you start having to negotiate between the people who needed the change and the people who needed it to not have happened."

A "long tail of bugs" refers to quality issues that arise long after the integration is complete. Such issues are more likely when dependency information is unavailable or not used during integration planning. Dependency analysis is challenging for our participants because the project has thousands of components in the feature branches and the dependency graph "is like a hairball" (P2). However, our participants have found that performing frequent (about once every two weeks) and early integrations can help identify dependencies between code being changed in different branches. Frequent integrations have helped add greater predictability to integration planning as the risk of long stabilization times is reduced:

> "Part of the problem that we were having when doing integrations every milestone or so, is that you do not figure out your dependencies or your dependents until really late in the product cycle, when many things are coming together and everyone is trying to determine what were the resulting dependencies. So by doing a lot more integrations often, we weed out some of these dependencies as we go, instead of creating a large amount of debt that we end up having to pay all together." (P5)

System Architecture

From the participants' experience, where a branch's source code changes are in the system architecture can strongly influence the outcome of an integration. P2 describes the system architecture as a "layer cake," with the most fundamental features comprising the bottom layer. In terms of risk during integrations, "the lower the payload in the stack, the more chances that something will not work well" (P5). In this project, the lower layers have had a "humongous impact" on the release schedule by "blasting [out] code that was putting people completely on the floor" (P1).

The key integration decision in this case is to determine an effective ordering for branch integrations. The organization attempts to mitigate the risk of destabilization by identifying the features in the lower layers and encouraging those branches to integrate first. This approach, in theory, allows the fundamental features to stabilize in the mainline branch and be absorbed into the features branches before those feature branches integrate, giving them time to adjust to any breaking dependency changes. The approach is further described by P2:

> "We also recognize that for teams higher up in the cake to be productive, it is best that the [lower layers] get a head start, and it is best that their milestones are offset from the others. So in the broader milestone, we take half of the teams and say, 'you guys are going to try and hit halfway in the milestone' and then the rest of the layer cake, 'you are going to hit at the end.'"

The architecture information is most useful during development milestones, because "towards the end of a product cycle, that assessment is less important because each branch ought to be working on bug fixes" (P3). Nevertheless, a broad understanding of system architecture supports integration decisions by providing guidance on how to better regulate the integrations—particularly, to avoid a situation where a low-level branch integrates after several high-level branches and breaks their integrations after-the-fact.

Infrastructure

Infrastructure systems—version control, build, and test—are the backbone of any integration, and the participants stressed the importance of keeping them "rock solid" (P5). The problem is that infrastructure adds another dimension to integrations that can have a significant impact on their outcome. These additional concerns are described by P6:

> "In the case of integrations, there is a much broader class of things that will go wrong, and I think we have much more trouble predicting the probability that those things will happen. We have build tools issues, we have compiler issues... [There are] more different sources of breakage."

While the infrastructure is there to support integration activities, it can also derail an integration if there is any instability. This problem is exacerbated when developers have little knowledge of the systems involved in the integration and how to troubleshoot them. For instance, P7 discussed how a com-

plex problem with the build system caused a great amount of frustration:

> "There were race conditions in the build system. So, it is possible you do an integration, you build fine and it looks good. Then you build on better hardware, and because stuff gets parallelized more, you end up in the race conditions in the build system. The [merge] was easy, but you're getting regressions from the build system and tracking these down and reproducing them is nearly impossible."

Awareness of the infrastructure supporting integrations is an important factor in costing integrations. Having some measure of confidence in the reliability of the infrastructure systems, and the capabilities of those using them, allows one to adjust the planned length of integrations accordingly.

Project Traffic Control

Integration is fundamentally about moving code changes between branches with the goal of taking work done in parallel and combining it into one code base that can be released. The way branches are organized defines paths for the flow of changes between branches and integration policies determine how frequently code flows along those paths.

Branch Organization

Branch organization is a fundamental component of integration decisions, as branch structure in the version control system defines how code will flow between the branches. For example, knowing how many integrations are required to get a feature into a dependent branch across the organization is a valuable planning aid, because "there is a cost, a fixed cost, from going from branch-to-branch" (P1). That is, for every integration, there is a minimum amount of time and effort that must be spent merging, building, and testing.

> "Even if a feature branch wanted to integrate, it would take them a few weeks before that code actually got up to [the mainline]. And of course, it would have to go down the other side so it could be a month or more before other teams saw the code." (P1)

The organization has used several different branch topologies over 10 years. In 2005, there were 3 branches with 150 developers working in each. In this model, the branches were constantly broken because of the large amount of churn happening on a daily basis. By 2010 the pendulum had swung the other direction, and there were 25 branches underneath the mainline, with even more feature branches underneath those. However, our participants found that with this model the cost to propagate code became overwhelming because of the fixed integration costs multiplied by the large number of branches. The model in use today is depicted in Figure 1, which has 15 branches under the mainline with feature branches underneath those—and it has so far "had the best of both worlds...[and] worked really well" (P1).

By optimizing the branch structure, the goal is to reduce the overall amount of integrations while not sacrificing the benefits of branching. The decision to create a particular structure with features being developed in particular branches is typically done at the beginning of a release cycle. However, decision makers must also be prepared to adjust the structure on-the-fly:

> "When features are in different branches, validating a lot of [customer scenarios] is harder because you can not get features to interact with each other as they are in different branches. So sometimes we end up delaying this type of work until a time when they come together....which introduces uncertainty into the product because you are getting this data a lot later than you would have liked to." (P5)

A key observation from P3's experience is that important end-to-end customer scenarios can be difficult to verify in a branched development environment, which creates uncertainty about the product quality from a release perspective. P3 went on to discuss a potential solution: "We tend to have branches across organizational boundaries, but we are trying to have some of those branches be across end-to-end scenarios...so that we facilitate this type of testing earlier on in the product cycle." P6 echoed support for this new branching model, believing that it is "a dramatically lower cost than any other option."

So, in order to make effective integration decisions, particularly around release scheduling, a clear understanding of the branch organization is required. Similarly, decision makers must be prepared to alter the branch organization on-the-fly to mitigate risk to the release schedule.

Code Flow

The flow of code between branches proved to be significant concern for the interview participants. The problem is that infrequent integrations lead to unpredictability, particularly in stabilization costs. This finding was hard-earned by the organization: "One of our learnings from the last release, was that teams that had the most trouble, the least predictability of getting their code into [the mainline], were the ones furthest separated from [the mainline] in terms of integrations" (P2). The integration decision in this case is determining how frequently branches should perform upstream and downstream integrations. P5 discusses the historical implications of this decision for the organization:

> "We had the pendulum at the two extremes: one where we were not [validating the results of our integrations] and it was cheaper, but we would break things in [the mainline], and then we swung it too much the other way where our integrations would be very expensive and teams would not want to pay for them, so they would delay doing integrations until as late as possible, and there would be a huge amount of integrations in a short amount of time. And everything would be on the floor because of all of these things coming together."

The consequences of having a large amount of code integrated together at once have been detailed in the discussion on code churn, and they can be severe—a 45 day delay was the worst reported case. The difficult integration decision in this

context is to determine the ideal integration frequency given the practical constraints in the development environment. P5 continues on to describe the outcome of this decision for the organization:

> "We have tried to put the pendulum somewhere in the middle where the size of the integrations and validations are smaller, and we are encouraging teams to do integrations more often to avoid the big integrations at the end. We are not fully done because not all the systems are in place, but I would say it has gotten a bit better than it was in the past."

Regardless of the strategy being used, in order to make integration decisions, it is important to know how frequently integrations are happening to and from various branches. If any branch is "stale" (P4), not having integrated up or down for a long time, then there is greater chance for destabilization and a delay in the schedule because "that branch is going to have a much harder time integrating up" (P5). In these cases it can take a branch "two weeks to absorb whatever has been changing around them" (P3).

It is clear that code flow is a critical input to making effective integration decisions. It is important for decision makers to be aware of how frequently integrations are occurring between branches, and if any branches are introducing risk in the project by going stale. Plans should be adjusted based on this feedback to make the release schedule more predictable by encouraging integrations to and from branches that are trending towards staleness.

INFORMATION NEEDS

Despite many years of experience, integration decisions continue to be challenging for our participants. In the previous section we discussed 10 factors that are important in the participants' integration decision making process. In this section we discuss specific information needs implied by those factors and the extent to which those needs are met in the target organization.

Buse and Zimmerman have noted that there is significant research into information needs of developers, but a deficit in understanding the needs of managers [3]. Hence, the novelty of these information needs does not stem from the information itself, as many tools readily provide the data. Rather, the novelty is uncovering how the data can be processed, abstracted, and displayed to promote effective decision making for release management. For example, in a release meeting in our participants' organization where dozens of team representatives are present, we need to determine what information should be presented, and in what manner, to quickly facilitate good integration decisions.

Predicting Storms

It is valuable to identify conflicting changes as early as possible as they are a major source of integration risk. Early warnings of impending conflicts can be achieved by executing mock, or temporary, merges and builds between branches, which would identify the location and scale of conflicting code. This approach is similar to the speculative analysis proposed by Brun et. al. [2]. As each pair of branches needs to be continuously mock-integrated, the process should be automated and executed in manner that would not seriously degrade the performance of the version control system. Currently, such a system does not exist in the organization; P7 does perform mock-integrations on an ad-hoc basis, but the resulting conflict information is used for managing his personal workload. Ideally the data should be project-wide, generated close to real-time, and presented in a single view that is appropriate for large collaborative meetings.

The next concern is where and when the conflicts will meet. The conflicts represent impending storms (destabilization) and the challenge is to predict the point where they will materialize. Two pieces of information are required: the integration paths between branches and the branch integration schedules. The paths describe how code is intended to flow through the branches, such as the depiction in Figure 1. The schedules are the dates and times of each branch's intended upstream and downstream integrations. This information is typically communicated in meetings and added to internal wikis; however, the data is not accessible to other tools.

To more accurately predict collision points, teams should enter their (often changing) integration plans into a tool that can be leveraged by the mock-integration system. Alerts could then be provided about when, in which branch, and with what severity integration problems may occur. Such a system could promote better cooperation as teams can work together to alleviate the integration problems before they arise, or adjust their schedules to accommodate increased integration effort at future collision points.

Detecting Pressure

Pressure reflects the amount of code churn that has built up in a branch. The greater the volume of code, the more likely a branch will encounter integration problems. That is, as the amount of change grows between any two branches, even if separated by intermediate branches, then there is an increasing risk to the release schedule. Like a dam ready to burst, the large amount of code in the originating branch threatens to wash out the target.

Pressure can be prevented from rising to unacceptable levels by making good decisions about when to integrate and also when to back-out integrations that are causing delays. Code churn data is available from the version control system, but there is no system in place that effectively visualizes churn across all branches so that potential problems can be easily recognized in release meetings. Currently, last integration dates for each team are tracked on a wiki, but this data is input manually and can only provide a rough estimate of how much change has occurred. Precise churn data may be a better predictor of risk and can be frequently and automatically extracted.

Monitoring Feature and Dependency Flow

To make informed integration decisions, two key considerations are which features are currently in each branch and

where are their dependencies. However, the version control system used by the organization monitors code flow only at the change-level, and because features can consist of thousands of individual changes, it is tedious to determine if a branch contains a particular feature in its entirety. There is no mechanism to associate multiple changes to the high-level concept of a *feature*.

Late in the release cycle, tracking changes individually is usually not a problem. Questions in release meetings tend to be around whether certain branches contain certain bug fixes, which are usually single changes. But in development milestones, there is a large amount of code churn and changes flow rapidly between branches. During this time, questions in release meetings are more likely to be about which branches contain a particular feature X, as well as which branches contain the feature dependencies of X. Answering these questions can quickly put branch test results into context (e.g., failures might be expected if certain features are not present), and can help facilitate efficient integration planning by knowing when and which branch to pull code from to have features meet and be tested.

Because these questions involve high-level abstractions of code flow, the solution must enable the association of a set of code changes to a revision of a feature. The idea is to streamline code flow analysis in release meetings by moving away from developer-centric per-change analysis to release-centric per-feature analysis. Such a tool would remove tedious version control log parsing, or having to build and run the product to see which features are present.

Tracking Health

Tracking test results, bugs, and task completion is the usual way to measure project health. But in branched software development, these metrics must be considered at the branch level. For example, due to resource constraints, individual branches in the organization do not run the entire test suite. Rather, branches run a subset of tests that are (believed to be) the most relevant to the features being developed in them. However, this practice creates blind spots for the release organization. The pass rate may be 100%, but if the branch is only running 1% of the test suite there is a large degree of uncertainty. When that branch integrates and a greater proportion of the test suite is run against its features, bugs can be discovered and significantly lengthen stabilization times.

The organization generally tracks metrics at the branch level. However, these metrics tend to be spread across many different systems—each with its own reporting mechanism. In a release meeting where status must be communicated quickly, this limitation slows information finding and integration decision making.

The ideal solution is to combine all branch metrics into a single view that can be expanded to reveal specific details. For example, in Figure 1, each branch could be coloured in a manner to suggest their overall health based on a weighting of the tracked metrics. Then, in a release meeting, attention can be drawn to the branches that are creating risk in the project.

THREATS TO VALIDITY

While the interview participants in our study act in a diverse set of roles, they nevertheless work in the same organization using many of the same tools and processes. The benefit of having participants from the same organization is that we are able to deeply explore experiences from several different perspectives, but it may be the case that the experiences are shaped by the particular organizational culture.

Efforts were made to increase validity by initiating the interviews with questions resulting from prior work [12]. Nevertheless, care should be taken when generalizing our findings. The organization is very large, as is the product under development. The problems encountered in this environment may not occur in smaller companies, or may not occur to the same degree.

FUTURE WORK

From our experience, the lack of information support experienced by our participants is not unique. In many software development organizations, the necessary data may be available; however, the information is dispersed and is not processed and visualized in a way that is useful for release management. Nevertheless, a systemic review of existing tools is needed to broadly understand the extent that tools support release management in branched development projects. Such an endeavor is part of a larger effort to implement our ideas in an industry setting. We have begun work with another large organization, one that is encountering similar integration challenges to those described in this paper, by developing a tool that incorporates the described information needs and is measured statistically for effectiveness.

Future contributions to CSCW research will consider how organizations can manage branching of any intellectual product. We expect branch evolution and branch health to be concerns in any product that requires people to work in parallel. Awareness is already well studied in CSCW [5], but here we introduce the notion of project awareness as essential to managing branches in version control. Project traffic control considers how work is organized to facilitate awareness and collaboration within large teams, and why it is important to constantly monitor it for ineSfficiency. As with our findings, the information needs of those managing software projects also applies to those managing other products. Predicting storms, detecting pressure, monitoring flow, and tracking health are all general concepts. While our emphasis on software engineering may be domain-specific, these concepts outline a framework for considering and researching version control branching as a general CSCW activity.

SUMMARY

We have conducted an interview study focused on how our participants make branching and integration decisions while managing releases. We found that making these decisions is complex and requires considering many factors such as code churn, potential conflicts, bug counts, and dependencies between branches. We also identified several information needs that can support integration decision-making in

a branched context by helping release decision makers predict storms of conflicts, detect pressure building up from non-integrated changes, monitor code flow between branches, and track branch health.

More generally, we found that parallel development with version control branching can have a significant impact on the ability to release software on time. The interview participants in this study experienced many such delays, some several weeks in duration. For businesses, tool makers, and researchers, a key message to take away from this paper is that branching, while beneficial for some aspects of collaborative software development, leads to complex integration decisions with significant costs. These considerations should be incorporated into their work to ensure the results are useful and relevant in industry settings.

ACKNOWLEDGMENTS

We would like to express our appreciation to our seven interview participants for their generous sharing of knowledge and experience, and to Dr. Saul Greenberg for his valuable feedback. This study has been funded through the Natural Sciences and Engineering Research Council of Canada (NSERC) discovery grants 250343-07 and 341367-07.

REFERENCES

1. Barcellini, F., Détienne, F., Burkhardt, J.-M., and Sack, W. A socio-cognitive analysis of online design discussions in an open source software community. *Interact. Comput. 20* (January 2008), 141–165.
2. Brun, Y., Holmes, R., Ernst, M., and Notkin, D. Speculative analysis: exploring future development states of software. In *FSE/SDP workshop on Future of software engineering research*, FoSER '10, ACM (2010), 59–64.
3. Buse, R., and Zimmermann, T. Information needs for software development analytics. Tech. rep., Microsoft Research, 2011.
4. Corbin, J., and Strauss, A. *Basics of Qualitative Research*, 3rd ed. SAGE Publications, 2008.
5. Gutwin, C., and Greenberg, S. The importance of awareness for team cognition in distributed collaboration. In *Team Cognition: Understanding the Factors that Drive Process and Performance*, APA Press (2004), 177–201.
6. Hansen, P., and Jrvelin, K. Collaborative information retrieval in an information-intensive domain. *Information Processing and Management 41*, 5 (2005), 1101 – 1119.
7. Ko, A., DeLine, R., and Venolia, G. Information needs in co-located software development teams. In *International conference on Software Engineering*, ICSE '07, IEEE Computer Society (Washington, DC, USA, 2007), 344–353.
8. Limayem, M., Banerjee, P., and Ma, L. Impact of gdss: opening the black box. *Decis. Support Syst. 42* (November 2006), 945–957.
9. Mens, T. A state-of-the-art survey on software merging. *Software Engineering, IEEE Transactions on 28*, 5 (may 2002), 449–462.
10. Paul, S., and Reddy, M. Understanding together: sensemaking in collaborative information seeking. In *Computer supported cooperative work*, CSCW '10, ACM (New York, NY, USA, 2010), 321–330.
11. Perry, D., Siy, H., and Votta, L. Parallel changes in large scale software development: an observational case study. In *International Conference on Software Engineering* (apr 1998), 251–260.
12. Phillips, S., Sillito, J., and Walker, R. Branching and merging: an investigation into current version control practices. In *International workshop on Cooperative and human aspects of software engineering*, CHASE '11, ACM (2011), 9–15.
13. Rönkkö, K., Dittrich, Y., and Randall, D. When plans do not work out: How plans are used in software development projects. *Comput. Supported Coop. Work 14* (October 2005), 433–468.
14. Ruhe, G. Software engineering decision support—a new paradigm for learning software organizations. In *Advances in Learning Software Organizations*, vol. 2640 of *Lecture Notes in Computer Science*. Springer, 2003, 104–113.
15. Ruhe, G. *Product Release Planning: Methods, Tools and Applications*. CRC Press, 2010.
16. Saarelainen, M.-M., Koskinen, J., Ahonen, J., Kankaanpaa, I., Sivula, H., Lintinen, H., Juutilainen, P., and Tilus, T. Group decision-making processes in industrial software evolution. In *International Conference on Software Engineering Advances*, IEEE Computer Society (2007), 78–.
17. Sack, W., Détienne, F., Ducheneaut, N., Burkhardt, J.-M., Mahendran, D., and Barcellini, F. A methodological framework for socio-cognitive analyses of collaborative design of open source software. *Comput. Supported Coop. Work 15* (June 2006), 229–250.
18. Sassenburg, H. A multi-disciplinary view on software release decisions. In *International workshop on Workshop on interdisciplinary software engineering research*, WISER '06, ACM (2006), 45–52.
19. Sonnenwald, D., and Pierce, L. Information behavior in dynamic group work contexts: interwoven situational awareness, dense social networks and contested collaboration in command and control. *Information Processing and Management 36*, 3 (2000), 461 – 479.
20. Walrad, C., and Strom, D. The importance of branching models in scm. *Computer 35* (September 2002), 31–38.
21. Wright, H., and Perry, D. Subversion 1.5: A case study in open source release mismanagement. In *ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, FLOSS '09, IEEE Computer Society (2009), 13–18.