Branching Strategies Based on Social Networks

Noureddine Kerzazi
Department of Research & Development, Payza
Montreal, Canada
noureddine@Payza.com

***Abstract*—Effective code branching strategy must be adapted to the unique needs of each organization. Teams and workflows organization as well as software architecture should be reflected in the branching strategies to maximize productivity and to minimize development risks. When conceptualized carefully, proper branching structure produces superior results. This paper proposes an analytic approach for adapting structure of branches based on Social Network Analysis to find out Branch-dependencies. The article provides context-based scenarios of successful application of such branching strategies in different situations.**

***Index Terms*—Release Engineering, Release Cycles, Branching Structure, Network Matrix, Branch-Dependency.**

I. Introduction

All software project of certain team and system sizes will inevitably require branching structures to be conducted in parallel [1]. To mitigate risks, organizations try to structure the parallel works within a breakdown structure of branches, in the version control system, aiming to improve teams' productivity or to decrease risks of freezing the release flow. The question is not "should we adopt structure of branches to conduct a parallel development effort?", but "how should be aligned the branching structures with the technical and organizational aspects to support effective parallel development and continuous integration?"

Adopting an inadequate branching strategy can result in process overheads [2, 3], tedious integration and a discontinuity of the release flow. Source code integration from a given branch into the mainline code stream could ends up being a frustrating experience for the entire teams [4]. The source code transit from branches to the production environment introduces new kind of defects related to merge issues, latent defects that do not surface until changes from different branches reach each other in the releasable branch.

Finding an adequate branching structure that fit the organization context and their specific needs can become a tedious task. Especially when companies had little information available to assess how their branching structures are effective [2, 5]. The effectiveness of branching structures can be assessed aiming different goals (e.g., Software quality in terms of Post-release failures, Productivity in terms of parallel development and integration workload, Risks in terms of release flow discontinuity, etc.).

Thus, in order to study these perspectives, we need to carry out a further analysis of branching structures according to at least two dimensions: Technical and Organizational. First, number of studies has analyzed entire teams [4]. The source code transit from branches to the production environment introduces new kind of defects related to merge issues, latent defects that do not surface until changes from the technical aspects aiming to understand the relationship between the code characteristics and the post-release failures [6, 7], yet we still need to figure out the crosscutting relationship between the code scattered within branches. Second, only limited work has focused on the relationship between branch dependencies and certain attributes at project level such as collaboration and congruence [8]. Our work aims to examine the organizational dependencies derived from collaboration structure on the branching structures. On the other words, how to predict extra effort of integration and post-release failures at branch level based on branch-dependency, team-dependency, and resource-dependency.

This paper presents an analytic approach based on Social Network Analysis (SNA) [9] to support the organization of branches. SNA can be used to identify dependencies between branches, teams, and resources. Our research goal is to examine to what extent teams' interactions, communication, and resources sharing (e.g., files, APIs, etc.) support code churn information in designing a context-based structure of branches.

Section 2 presents common branching strategies. Section 3 describes the approach to represent branch-dependencies and discusses four scenarios of branching. Section 4 concludes.

II. Methodology

We based our exploratory study on classical Glaserian grounded theory [10]. Grounded theory is a qualitative research methodology that starts from general research questions. Both research questions and data collection instruments are refined throughout the study progress. A growing demand on coordination in software release process suggests that the identification and management of dependencies is a fundamental challenge in structuring branches. We aim to understand the central concern of those involved in branching strategies and how they resolved their central concern.

The study takes place in a large industrial organization that develops a complex financial system. We had the opportunity to be on site for an extended period of time (more than 14 months). We collected data using collaborative framework, Team Foundation System (TFS), Source Code Repository, IRC, Release Calendar, Test Reports, participant observation, and semi-structured interviews. Also, we were able to observe social phenomena that are uncovered in interviews such as within the daily release meetings.

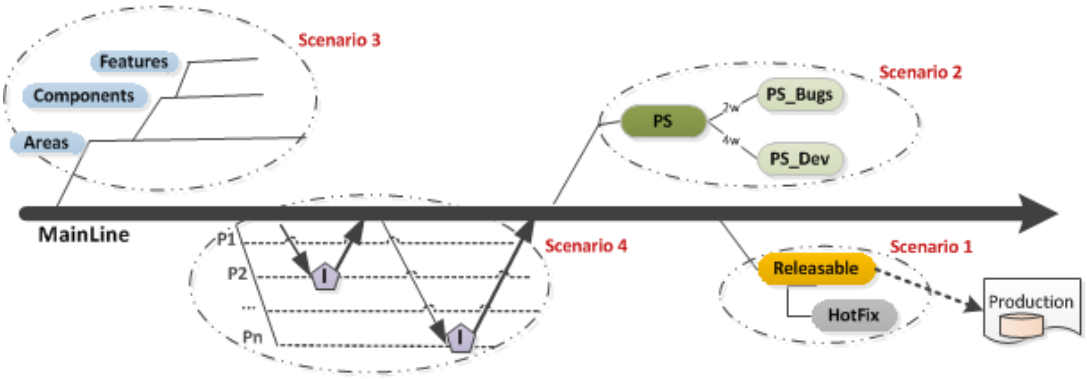

Fig. 1. Scenarios of Branching Strategies

- RQ1: How do Release Engineers Structure and Manage Parallel Development?
- RQ2: What factors that drive designing the Branch Breakdown Structures (BBS)?
- RQ3: What is the relative impact of a misalignment between Branching Structures and teams' organization?
- RQ4: Do poor Branching Structures affect the integration effort and the software quality?

III. BRANCHING REQUIREMENTS

Branches do not exist in closed environments, neither with the purpose of isolating contributors. Continuous integration and releases remind us that branches are open workspaces. In this sense, the context-based requirements of collaboration come before any best way of organizing.

Figure 1 highlights four strategies of branching. A consistent source code is maintained in the "*Mainline*" branch, while releases are carried on "*Releasable*" branch. Critical Bug fixes might be merged from "*HotFix*" branch to the "*Releasable*", and then merged back to the mainline to avoid regression. Below four scenarios related to different situations that we faced within the same organization and under the same complex project.

A. Scenario 1

Organization needs a branch for resolving quick hot fixes as depicted in figure 1. Hotfixes could be related to post-release bugs discovered in production or related to a late merge issue that happens when code from different branches reach each other in the releasable branch. The requirements expressed by the organization are:

- S_1C_1: The content of the Hotfix branch must be tested and integrated to the releasable branch with minimum of integration effort and minimum of injected defects.
- S_1C_2: All developers should be able to work in this branch.
- S_1C_3: One expects that little code is added to fix bugs.

B. Scenario 2

Organization takes the decision to create two new professional services (PS) teams, working within scrum iterations. Teams are dedicated to bugs resolution and system maintenance. Both teams are composed of five members (2 dev, 1 Team Lead, 1 BA, and 1 Tester). The first Team, time-boxed under two weeks, works on branch called PS-Bugs. The second team, time-boxed under four weeks, works on branch called PS-Dev. Two constraints have been imposed:

- S_2C_1: PS-Bugs and PS-Dev teams can share their workspace at any time without necessarily waiting for the release. In other words, the branches can merge their content without having to cross the main line, which means integrating and releasing the content of the branch.
- S_2C_2: due to the limit of testing resources, PS-Bugs and PS-Dev must share a common test space.

C. Scenario 3

The organization has created a specific team dedicated to maintain the architecture and the core framework. Those architectural components as well as the framework, which are crosscutting, will be used by all developers at different levels.

- S_3C_1: Development starts in an experimental branch and follows a set of promotion from ideas to delivery.

D. Scenario 4

The organization initiates projects that could not fit time-boxing (i.e., iterations) due to external factors such as coordination with other organizations.

- S_4C_1: A branch is created for parallel development and destroyed after the release.
- S_4C_2: Integration must be done in the branch to avoid breaking the mainline branch.

To learn more about the impact of the branching structure described in figure 1, let consider this situation:

>Dev1: I'm working on PS-Bugs branch. In order to resolve the bug 725 assigned to me, I need that code produced by PS-Dev team. Also, I need the latest code related to the UI control that is in development by FED-UI team.

>RM: No problem for the code coming from PS-Dev, I need just the change set number. However, I cannot merge the code from the FED-UI branch because it's not releasable yet. If the code passes through the mainline branch, this means that we are releasing FED-UI project. Also, as you know the baseless merge between branches will result in many more conflicts than a normal merge.

This situation, related to a poor branching structure, freezes the development and causes teams' frustrations with feeling of isolation. Next section presents the common branching strategies and their purposes.

E. Common Branching Strategies

This section outlines the common branching strategies used in the industry. Every branch exists to satisfy some concerns. Concerns may be addressed according to many perspectives.

TABLE I. COMMON BRANCHING STRATEGIES ADAPTED FROM [1]

Branching by	Purpose
Features	Common approach used for parallel development within organizations that develop mainly one product. With shortest life of branches, no need for extra effort to maintain those branches.
Components	Aligned with the system architecture. Each branch embodies a streamline of one component. Main branch serves for the integration purpose.
Teams	Branches aligned with the organizational structure. Branches match team boundaries making communication and coordination much simpler.
Releases	Product vision that define the content and date of the next release. Parallel development is done mainly on one branch with continuous integration.
Code Promotion	Common approach used especially by the open source community. Branches are used to promote the code through multiple stabilization phases.
Technologies	Branches are aligned with technology platforms. For example, adapted product for different OS. Branches can share the same framework.

Rather than seeking the compliance with one model of branching, organizations use a mix of those models to response to their specific needs.

IV. NETWORKS AND SOCIAL NETWORK ANALYSIS

A. Approach

Based on structural social organization network proposed by Ashworth and Carley [11], we analyzed the interaction between the sources code provided by teams working on separate branches. To construct social networks, we start by examining collaboration structure based on developers networks derived from code churn information. Code Churn information has been used to predict failures at files level [12]. This information captured in the source code repository is directly linked to the collaborative work items such as description of bugs or tasks [13, 14]. Thanks to check-in policies each *Changeset* must be linked to a work item. Thus, we construct a map as shown is figure 2.

Then we represent the flow of relationship using matrix, as shown in figure 3. When direct relationship is not visible, matrices are powered to two in order to extract indirect relationships (e.g., if R (B_1, B_2) and R (B_2, B_3) then R (B_1, B_3). Remarks:

- While a work item belong to a single branch, resources can be scattered in different branches.
- Components or files could be modified in different branches.
- Developers are not isolated. We observe that developers often have more than one workspace. Each workspace related to one branch.

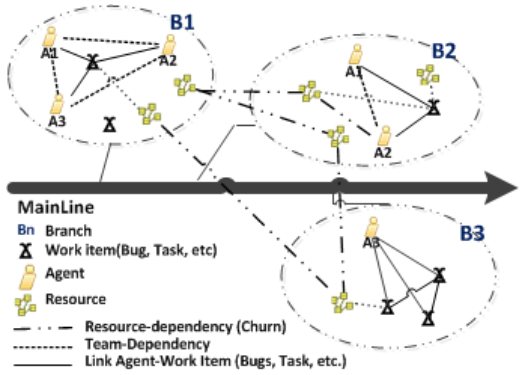

Fig. 2. Social Network Construction Example

The relationships between branches and socio-technical concerns are represented as a set of networks. As suggested by [9] , a network *N* represented as a matrix, symbolizes a set of edges E between two sets of nodes U and V, E ϵ U * V. Each element $e_{ij} \epsilon$ E represents a relationship between the node $U_i \epsilon$ U and $V_j \epsilon$ V.

Figure 3 represents an example of such representation of networks. The matrix (AB) shows the relationship of the agent A_i assigned to work on branch B_j. The matrix (BB) shows the relationship between branches of scenarios 1 and 2 (see figure 1). B_1 to B_6 represent respectively the branches {*Mainline*; *PS*; *PS-Bugs*; *PS-Dev*; *Releasable*; *HotFix*}. Hence, we can read the first line of the matrix as the {*Mainline* branch is connected to *PS* and *Releasable* branches (means e_{ij} = 1), while second line says that *PS* is connected to the *Mainline* but not to the *Releasable* branch. In other words, we can not release *PS* without crossing the *Mainline* branch. Thus, the matrix (BB) represents direct path from each branch to the Releasable branch.

$$(AB) = \begin{array}{c} \\ A1 \\ A2 \\ A3 \end{array} \begin{array}{c} \begin{array}{cccccc} B1 & B2 & B3 & B4 & B5 & B6 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

$$(BB) = \begin{array}{c} \\ B1 \\ B2 \\ B3 \\ B4 \\ B5 \\ B6 \end{array} \begin{array}{c} \begin{array}{cccccc} B1 & B2 & B3 & B4 & B5 & B6 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

$$(BA) or (AB)^T = \begin{array}{c} \\ B1 \\ B2 \\ B3 \\ B4 \\ B5 \\ B6 \end{array} \begin{array}{c} \begin{array}{ccc} A1 & A2 & A3 \end{array} \\ \begin{pmatrix} 2 & 0 & 1 \\ 0 & 2 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 2 \\ 1 & 0 & 0 \end{pmatrix} \end{array}$$

Fig. 3. Examples of Network Matrices Representation

Representing the branches network provides a concrete and real structure of collaboration in a software team project. A direct path between two branches means that the teams working on these branches can share resources without delay. A further analysis based on churn metrics can provides insights

on how both teams can collaborate effectively and how to work on separate concerns to avoid workload of integration. In contrast, when there is no direct link between two branches, the team working on branch A has to wait for the release of the content of the branch B to get the latest version of some resources. Facing this kind of branch-dependencies, an adjustment of the branching structure is required.

B. Practical Implication

In the interest of creating a context-based branching structure that promotes parallel development and strives to maximize teams' productivity, our approach has several implications. SNA could make it possible to manage code integration with very little effort.

Team awareness - Teams working on independent branches can benefit from visualizing their social network. This information is particularly relevant for the release manger who is involved in the merge of branches and integration of source code. Further analysis could identify the mismatch between team organization and branches they use.

Post-Release defects prediction - Although code Churn metric can support the prediction of post-release failure, it is too fine-grained. Social Networks can be used as an abstraction of this metric at work item level.

Assessing integration overhead - Release engineers can use Social Network Analysis to make their integration effort more focused. In addition, SNA can helps localizing where and when forward merges from the mainline branch to other branches should be done.

V. CONCLUDING REMARKS

Effective branching strategy should takes into account organizational and technical dimensions. From technical dimension, code churn information can tell us how developers collaborated through resource sharing. We know who worked on what and when [11]. From organizational dimension, we can examine a social network of developers who have collaborated on the same work items during the same period of time. Consequently, branching structure can be adapted based on collaboration structure, which underlies identifying the interactions between developers and where each developer' work lies in the overall branching structure.

As discussed in the four scenarios above, the organizational dependency can affect decision making regarding the branching structures. Socio-Technical dependency analysis would bring more insights on the factors affecting the integration effort as well as the quality of the outcome software. An overall view of social networks helps in predicting the integration effort for the upcoming release, and at this point, the release manager could make proactive adjustments within the branching structure to attempt to prevent an overload effort of software integration.

VI. ACKNOWLEDGMENT

This work has been supported by NSERC Canada. We are grateful to anonymous readers for their valuable discussions.

REFERENCES

[1] B. Appleton,S. Berczuk,R. Cabrera*, et al.* Streamed Lines: Branching Patterns for Parallel Software development *Pattern Languages of Programs Conference*, 1998.

[2] E. Shihab,C. Bird and T. Zimmermann, The effect of branching strategies on software quality. in *Int'l Symposium on Empirical soft. Eng.and Measurement*, (Lund, Sweden), 301-310, 2012.

[3] H.K. Wright and E.P. Dewayne. Release Engineering Practices and Pitfalls *ICSE*, Zurich, Switzerland, 1281-1284, 2012.

[4] J. Humble and D. Farley *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[5] C. Bird and T. Zimmermann, Assessing the value of branches with what-if analysis. in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, (Cary, North Carolina), 2012.

[6] M. Eaddy,T. Zimmermann,K.D. Sherwood*, et al.* Do Crosscutting Concerns Cause Defects? *IEEE Trans. Soft. Eng.*, *34* (4): 497-515, 2008.

[7] A. Meneely,L. Williams,W. Snipes*, et al.*, Predicting Failures with Developer Networks and Social Network Analysis. in *FSE*, (Atlanta, Georgia, USA), 13-23, 2008.

[8] M. Cataldo,A. Mockus,J.A. Roberts*, et al.* Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Trans on Softw. Eng.*, *35* (6): 864-878, 2009.

[9] S. Wasserman and K. Faust *Social-Network Analysis: Methods and Applications*. Camb. Univ. Press, 1994.

[10] S. Adolph,W. Hall and P. Kruchten Using grounded theory to study the experience of software development. *Empirical Software Engineering*, *16* (4): 487-513, 2011.

[11] M.J. Ashworth and K.M. Carley Who You Know vs. What You Know: The Impact of Social Position and Knowledge on Team Performance. *The Journal of Mathematical Sociology*, *30* (1)2006.

[12] N. Nagappan and T. Ball, Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. in *First International Symposium on Empirical Software Engineering and Measurement*, 363-373, 2007.

[13] M. Cataldo,P.A. Wagstrom,J.D. Herbsleb*, et al.* Identification of Coordination Requirements Implication for the design of collaboration and Awareness Tools *Computer Supported Cooperative*, Alberta, Canada, 353-362, 2006.

[14] T. Wolf,A. Schroter,D. Damian*, et al.* Mining Task-Based Social Networks to Explore Collaboration in Software Teams. *Software, IEEE*, *26* (1): 58-66, 2009.