Forking and coordination in multi-platform development: a case study

Anh Nguyen Duc[1,2], Audris Mockus[2], Randy Hackbarth[2], John Palframan[2]
[1]Norwegian University of Science and Technology, Sem Saelands vei 7-9, Trondheim, Norway
[2]Avaya Labs Research, 211 Mt Airy Rd, Basking Ridge, NJ, USA

anhn@idi.ntnu.no,{audris,randyh,palframan}@avaya.com

ABSTRACT

With the proliferation of desktop and mobile platforms the development and maintenance of identical or similar applications on multiple platforms is urgently needed. We study a software product deployed to more than 25 software/hardware combinations over 10 years to understand multi-platform development practices. We use semi structured interviews, project wikis, VCSs and issue tracking systems to understand and quantify these practices. We find the projects using MR cloning, MR review meeting, cross platform coordinator's role as three primary means of coordination. We find that forking code temporarily relieves the coordination needs and is driven by divergent schedule, market needs, and organizational policy. Based on our qualitative findings we propose quantitative measures of coordination, redundant work, and parallel development. A model of coordination intensity suggests that it is related to the amount of paralel and redundant work. We hope that this work will provide a basis for quantitative understanding of issues faced in multi-platform software development.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management

General Terms

Management, Measurement, Human Factors

Keywords

Multiple platform development, fork, coordination, empirical study

1. INTRODUCTION

A rapid growth in the popularity of cross-platform software systems has been brought on by the recent proliferation of mobile applications. Software such as web browsers, instant messaging clients, voice communication, e.g., Skype and other mobile applications, needs to be delivered for an ever increasing range of hardware and software platforms. Skype, with 299 million users, is available on Windows, Linux, Mac, iOS, Window phone, Android, Kindle, TV and Xbox [26]. The market for cross-platform mobile development tools is forecast to reach US$8.2 billion by 2016 [2].

The existing literature on multi-platform development is highly pragmatic, with guidelines and technical solutions [10, 1, 27, 21], but with few empirical studies of software development practices and coordination needs that would support multi-platform development. The traditional software engineering approaches may not directly apply in a multi-platform application context that has many unique challenges. On one hand, development teams have to conform to different requirements and constraints for each platform (e.g., iOS, Android, Windows 7, etc.), hardware (e.g., HTC, Google, Samsung), and, potentially, communication protocol (e.g. AIM, XMPP, SIP, H323). On the other hand, developers need to maintain inter-operability, consistent performance and, potentially, look-and-feel of the product across different platforms [8].

The maintenance and upgrades, which account for two-thirds of a typical software project's lifetime cost [22], might take even more effort in a cross-platform system development. Not many developers are intimately familiar with several platforms, thus it is often necessary to have different individuals and teams implement the same product for each platform. As a result, it may be difficult to propagate bug fixes across multiple version of a software application. For example, a recent critical security flaw was fixed for iOS, but remained unpatched for Mac OS [9]. Some code, often the user interface, has to be implemented in a platform-specific framework and ensured that the application features work in a similar fashion. This requires the teams implementing on different platforms to be able to effectively coordinate their work.

To understand how software intensive organizations resolve development and coordination problems in multi platform development we set out to study a set of voice, video, and instant messaging clients at Avaya. The qualitative part of the study investigates the state-of-the-practice in multi platform development and associated coordination mechanisms. In the quantitative part we constructed a network of related file versions across 43 instances of version control systems used by 25 different software and hardware platforms and use it to quantify the extent of forking and coordination. Our primary contributions involve:

1. A description of industrial multi-platform development effort: the practices and challenges; coordination mechanisms; and reasons for codebase forks.

2. Measures of the extent of parallel development, co-

ESEM '14 Torino, Italy

Table 1: Summary of the Projects

Items	Values
Number of platforms	25
Number of codebases	43
Number of files	232065
Number of branches	571
Number of commits	4055968
The timeframe	2010 - 2013

ordination, and redundant work in multiple platform development

3. A model of relationship between the intensity of coordination and the amount of parallel development and redundant work.

The paper is organized as follows: Section 2 introduces related terminologies used in this paper. Section 3 presents research approaches and study context. Section 4 presents findings from qualitative phase. Section 5 describes measures of forking and coordination, Section 6 presents quantitative findings. Section 7 reviews related work, Section 7.3 points out limitations, and Section 8 presents the conclusions.

2. TERMINOLOGY

We start from the terminology used in this paper. **Multi-platform software** refers to the ability of software to operate on more than one platform with identical (or similar) functionality [5]. Platform can refer to (1) different types of operating systems (e.g., Windows, FreeBSD, Linux, Mac OS X, and Solaris systems), (2) types of processors (e.g., x86, PowerPC, SPARC or Alpha), (3) communication protocol (e.g. SIP, RTM, AIM and H323) and (4) types of hardware systems (e.g., iphone, ipad, android phone and windows phone). **Project** has a its own code repositories with active code commits and MRs and results in one or more releases of a client product. In this study, project, platform and fork are used as inter-changeable terms. **Fork** is a clone of an entire VCS repository that has subsequent development independent of the original project [19][1]. **Modification request (MR)** is a piece of work (a story, an epic, a task or a bug) that is tracked in issue tracking system, such as JIRA. **Commit** is a set of changes to one or more file traced by Version Control Systems (VCSs). Commits to the VCS are associated with a MR identifier in the considered projects. **Software dependencies** refer to the relationship between software artifacts, such as source files, components and modules. There are many type of dependencies, such as data-related dependencies (i.e. a data structure modified by a function and used in another function) [11, 3] or functional dependencies (e.g. method A call method B) [11] and logical dependencies (e.g. files being changed together) [7]. **Related files** are files that had identical content in the past (based on their version history) [16]. **MR-clone** is an MR which was created for another project and is associated with an MR in the original projects. MR clones are used as a coordination mechanisms to track tasks among different projects. **Coordination mechanisms** are approaches to manage dependencies among software artifacts and project stakeholders, and is an important research topic [14].

[1]Forking is also used as a way to contribute to the project where the contributor creates a mirror (fork) of the repository and then sends pull requests to update the central repository. Such practice is not used in Avaya.

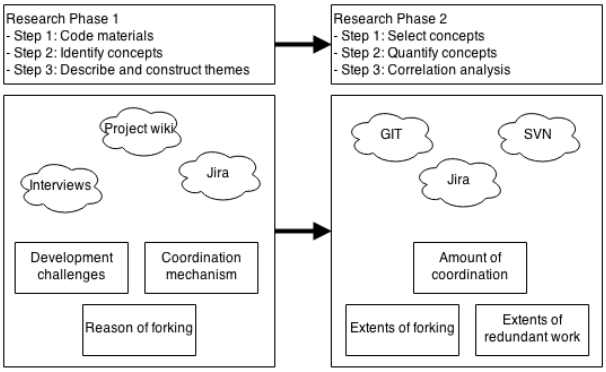

Figure 1: Research phases

3. RESEARCH METHOD

We describe the context of the study in Section 3.1 and design in Section 3.2.

3.1 Context

Avaya is developing large, complex, real-time software systems that are embedded and standalone products. Development and testing are spread through 10 to 13 time zones in the North America, USA, Europe and Asia. R&D department employed many virtual collaboration tools such as JIRA, Git, WIKIs and Crucible. Development teams use Scrum-like development methodologies with a typical 4-week sprint. Several mechanistic coordination approaches have been used, such as (1) daily scrum meeting, (2) sprint planning meeting, (3) sprint review, (4) backlog review meeting, and (5) Scrum of scrums meeting [23].

We focused on a 10+ year old software component, the so-called Spark engine. As a software platform, Spark provides a consistent set of signaling platform functionalities to a variety of Avaya telephone product applications, including those of third parties. The current evolution of Spark is a client platform that provides signaling manager, session manager, media manager, audio manager and video manager. Spark engine started in 2005 and has forked into many different codebases to support different combinations of hardware (Flare, ipad, iphone, Mac, Samsung devices, 1XC, VDI, ACA, AAC, IPO and LYNC), software (iOS, android, windows) and communication protocol (SIP, H323, etc), as shown in Table 1.

3.2 Research design

We conducted a single case study in a large telecommunication organization, following a guideline suggested by Yin [28]. The case represents a typical multi-platform software development with involvement of teams across geographical and organizational boundary. Exploratory case studies are used to investigate little known phenomenon or one without an established theoretical basis [20]. We use this approach to discover the code divergence, forking activities and coordination mechanisms in multi-platform software systems.

Table 2: Interviewees

ID	Positions	Locations	Projects
P1	Scrum Mastser	Tel Aviv Isarel	PrA, PrB
P2	Developer	Tel Aviv Isarel	PrA, PrB
P3	Developer	Ottawa Cananda	PrC
P4	Development manager	Ottawa Cananda	PrD, PrE
P5	Developer	New Jersey, USA	PrF
P6	Development manager	Ottawa Cananda	PrK, PrB
P7	Scrum Master	Ottawa Cananda	PrI, PrJ
P8	Product owner	Ottawa Cananda	PrG
P9	Developer	Ottawa Cananda	PrF
P10	Development lead	New Jersey, USA	PrH, PRK, PrA, PrB

The research consists of two phases and was conducted between December 2012 and March 2014. In Phase 1, we aimed to understand the context of multi-platform development, including technical and management challenges, and current development and coordination practices. A range of data collection methods were used to achieve familiarity with the organizational context and participants, including conversations and interviews, and review of relevant documentation and project artifacts. Semi structured interviews is the primary data source in this phase. We conducted 10 semi-structured interviews with different project roles, such as product manager, technical leader, developer and researcher, as shown in Table 2. The interview guide consists of four main parts:

1. Challenges of working on inter-platform teams,
2. State-of-the-art of codebase divergence,
3. Coordination mechanisms, and
4. Codebase merging activities

The interview guide was designed using the checklist from the literature review and revised multiple times by the authors. Each interview lasted from 30 to 60 minutes, depending on time availability of interviewee. The interviews were flexibly conducted to allow for improvisation and exploration of emerging issues [20]. The interviews were recorded, transcribed and reviewed by interview participants. Table 2 describes the list of interviewees in this study.

Relevant segments of text that expressed either challenges and practices in multiple development platforms were labeled with an appropriate code by open coding. After that, we grouped the relevant codes as a higher-order code using axial coding [28]. Finally, we performed selective coding in secondary data materials, such as project wiki and MR descriptions.

In Phase 2, we aimed at quantifying important concepts found during Phase 1, as shown in Figure 1. In particular, we quantified the extent of forking activities, amount of coordination via cloning MRs and the extent of potentially redundant development. The primary data sources in this phase were Version control systems (Git and SVN) and Issue tracking system (JIRA). We started from data integrated over all Avaya projects as described in previous studies [15, 16]. The data unifies information from most of version control and issue tracking systems in Avaya, but we still need to select the projects related to the client's communication. Three authors of the papers had insights on the products, which helped to identify 67 relevant codebases. We interviewed key stakeholders to confirm and complement the codebase list. To select and link relevant data, we wrote shell, Perl, and R programs to extract and calculate the necessary measures.

After all data was extracted, we performed data cleaning to remove inactive repositories (repositories without changes in the last 12 months) and moved repositories (repositories moved from one VCS to another VCS without subsequent development activities in the original VCS). 20 inactive repositories and 4 moved repositories were removed. Table 1 provides a summary of the final dataset of 43 project repositories for the 25 platforms.

Based on findings in Phase 1, we proposed and described several measures of parallel development and coordination. We performed logistic regression analysis to investigate the relationships among some of the concepts.

4. QUALITATIVE FINDINGS

In this section we summarize our qualitative findings organized into three themes as shown in Figure 2.

- Reasons for diverged code bases (Section 4.1),
- Challenges managing diverged code bases (Section 4.2), and
- Coordination mechanisms deployed to manage diverged code bases (Section 4.3).

4.1 Reasons for forking a codebase

Avaya projects related to Spark recognize the benefits of preserving a common code base across platforms - i.e. not forking the code repository. Benefits include efficiency in development and maintenance because changes are only made once, and improved quality because validation and verification activities can focus on one version of the change instead of being distributed across many versions. Nevertheless, we identified four reasons that Spark related projects created forks leading to diverged code:

- Schedule constraints (Section 4.1.1)
- Technical variation in the platform (Section 4.1.2),
- Coordination overhead in maintaining a single platform (Section 4.1.3), and
- Organizational differences (Section 4.1.4)

4.1.1 Schedule-driven forks

We found that a common reason for code divergence is different project release schedules. In particular, one project may require a stable code base with minimal, tightly controlled changes while another project may be in active development with rapid code changes. For example, Project A shared some common features with Project B, and originally they were in the same codebase. However, Project A team created a separate code to assure a stable code base: "*the next delivery date of PrA is [Date X] while the coming software delivery date of PrB is later at [Date Y]. Therefore, while PrB is still in development phases with testing and debugging, PrA needs to be stable for delivery*". (P1).

Projects with the same release schedule, but different development schedules have also chosen to diverge their code base. For example, we observed that code divergence occurred among projects with different numbers of sprints or with different sprint durations.

4.1.2 Feature-variety-driven forks

We have observed code divergence when one project introduces significant new features compared to the project(s) using the shared code base. For example, two projects PrE and PrF implemented a set of signaling engine functionalities in iOS and other mobile platforms accordingly. Early in its project stage, PrF stopped sharing the signaling engine engine codebase: "*after the first six months, the signaling engine stopped synchronization with project PrE. From that time, it was branched off from PrE and developed independently, due to technical differences. The current situation is that it has features that other iOS clients do not use*" (P5)

Another typical example of forking due to feature variety is project PrG and project PrD implementing client in Windows and iOS platforms. In the begining, there was a close collaboration between the two teams as user interface code was shared for the first two releases: "*... the reason is that the UI team is the same for both Windows and iOS versions. The design and logic behind them is pretty similar...*" (P8). The code base for subsequent releases diverged, however, because of project differences in user interface requirements. Developers in PrG stated that this issue made it difficult to collaborate among teams.

The assesment of feature variety becomes a part of the forking decision-making process in PrA: "*... The decision is usually made by trying to understand what are the main changes coming in from other releases. For example, if we are working for a H323 release and we know that a release from another project introduces a major feature then probably we want to branch out before that. This is a purely risk management activity.*" (P9)

4.1.3 Coordination overload

When two projects share the same code base, cross-project coordination is required to manage the impact of changes to the code base on each project. We have observed that projects have chosen to fork their code base when too much coordination is required. P6 stated: "*... in general, there is no good reason to have separate platforms of the same hardware. Historically there were separate teams working on codebase PrF. Later in the project, they had been communicating too much and (hence) the codebases got separated*". Here is another example: "*... the reason for stopping synchronization (between PrC and PrD) is too much overhead, no dedicated people to maintain the [shared] source code*" (P8) Note that according to our definition, we would refer to feature forks as branches.

4.1.4 Forking due to organizational differences

We have observed code divergence between two projects with common features and common schedule because the projects are in different organizations. The reason is that management needs (or wants) total control of resources if they are responsible for product delivery in their organization: "*... it (coordination efforts) has been mostly on ipad and windows team, purely because they came from the same team right, because we were working very closely together on the same area ... after that, we have been divided into different teams, as the managers were divided into the pure desktop and windows team and the purely mobile teams ... the organization split in the different ways from ...*"

In Spark, codebases are forked for outsource development teams. It is a common case that the outsourced team created its own version of the code base to establish code ownership of features and bug fixes: "*... I find the best thing working with external teams, especially from different timezone and locations is that if you have a really good clear process, especially with feature forking model, it is very easy for people to work together...they work on features and wrap it up, deliver back to mainstream in the end of release ...*" (P7). Strictly speaking, we consider feature work to be an instance of branching, not of forking. However, in this case the outsourced team uses their own version control repository and work on an entire sequence of releases, not on just on individual features.

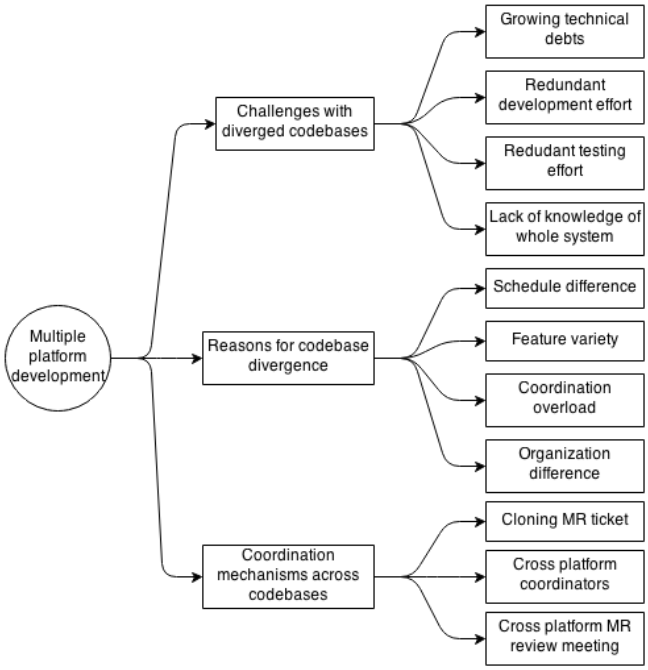

Figure 2: Thematic scheme map

4.2 Challenges managing diverged codebases

We found four primary challenges in managing diverged code bases.

- Technical debt is the extra development that arises when code is implemented in the short run (e.g. to deliver a product release) in a manner that will require additional maintenance and development in the long run (e.g. to deliver subsequent releases or to share code among similar products). Technical debt is often necessary, but we found that a diverged codebase encouraged the growth of technical debt as described in Section 4.2.1.
- A diverged code base encourages redundant development of features as described in Section 4.2.2.
- In addition to redundant development a diverged code base encourages redundant test effort as described in Section 4.2.3.
- Developers and testers who work on a diverged code base tend to become experts on their fork of the code base, but not on the family of products from which the code base originated as described in Section 4.2.4.

4.2.1 Growing technical debt

We found that the muti-platform development increases the technical debt for the whole product family. P10 mentioned: "*We migrate software from platform to platform, or team to team, and this builds up technical debt. We need to continuously invest in software — it is getting old even if you don't touch it*".

Spark diverged codebase has many individual product teams. Because of overall staff constraints, each product team is thinly staffed and yet is responsible for the entire product. As a result, schedules are tight with little time to reduce the MR backlog: "*Chasing a deadline created technical debt. Technical debt needs to be caught and dealt with earlier.*" (P8).

In several projects, multi-platform development comes with rapidly increasing MR backlog, creating pressure on developers, which leads to "*work around*" MR resolutions. Here is an example:"*The project has an ever growing backlog. It has grown to [figure] Severity level 1/2/3 issues for all the releases. The development team is stretched thin. We are focusing on getting [release] 6.2 out, so there is no bandwidth to tackle the backlog. Many issues are open though internal Interop, Product Verification (PV), or System Verification (SV) testing. Many of these are duplicates. If it were not for the urgency of shipping 6.2 the backlog would be addressed. We have stability issues we do not know the cause of*". (P5)

A short-term resolution for a bug or a feature request that is applied to only a subset of platforms would need more work later to be applied to the remaining platforms.

4.2.2 Redundant development effort

We found that product team staff responsible for one Spark platform were often unaware of many of the changes in other Spark platforms. As a result, features and bug fixes are implemented independently for each platform resulting in development that is unintentionally duplicated. Here is an example: "*"Some files included in the root are present multiple times in different codebases. For example some files are generated by multiple packages. This is redundant but also makes it difficult to know for sure which version of the file ends up on which phone release. ... it is important to make sure that you identify the correct source for the file as found on the latest phone release today and remove the other ones. It is more difficult to check across different platforms.*" (P3)

Occurence of multiple identical slightly dirveged files complicates merging features across platforms. "*Team PrA will have to do much of the dual delivery. Many features delivered to PrA need a port to PrB too. But the Team PrB will be involved in the code reviews. They will need to get involved in case the merge does not go smoothly.*" (P2)

In particular, the replication of code related to third parties components are more critical and harder to solve. P1 talked about collaboration between PrA and PrB: "*We found a bug in PrB codebase around the Broadcom hardware. Broadcom provided a fix ... and also introduced patches for that. We were working with Broadcom on PrA and would propagate the fix to PrB. ... We were not in the position to take the patch in the current release of PrB. This happened a couple of times and then we had different patches from Broadcom on PrA and PrB. The main challenge in consolidating two branches was matching the Broadcom patches*".

To mitigate redundant development, some individual product teams on different Spark platforms have made an effort to track changes in the other team's product as illustrated in this example: "*... Product team PrA team needs to watch for every submission to Product team PrB. Product team PrA assigned a developer dedicated to the product of PrB, and have an established line of communication with them via phone and conference call throughout the week ...*" (P2).

4.2.3 Redundant testing effort

Diverged code bases induce redundant test effort because each platform's test team typically independently tests similar features or functionalities from different platforms. Unless they know other teams' test cases of similar features, testers independently create their own test cases as shown in this example: "*the PrA team now only tests for PrA code and PrB team only tests PrB code. If there is a cross delivery, both teams have to test other product as well. Hence, both test teams get hit by the same amount.*" (P2)

We found that duplicated testing effort is more critical in high-level testing teams, such as at PV team and SV team. At this level, testers did not have insight about releases and how they are related across platforms. P8: "*This is something outside of our hand. What we can do is to tell the SV team that these products should be tested, there is a new release with these patches, we need you to pick it up and run with it... We do that by writing in the release note. ... But usually the SV team organizes themselves to secure resources for program vs. testing these patches just like we do in development.*"

A particular and often very important case of redundant testing is testing third party components. "*when a bug involves both [third party vendor] hardware and application products, both sides need to be able to reproduce the bug to find the main root-cause.*" (P6)

4.2.4 Lack of knowledge of the whole system

With diverged code bases, each project has its own priority of features and bug fixes. As a result of continuous evolution, the whole product family becomes extremely complex. Consider, for example, the likely complexity of the diverged code base leads to few, if any, engineers who understand in detail this entire code base.

In addition, individual development teams typically do not have "*who does what*" and "*who knows what*" knowledge about other teams as shown with this example: "*Once you move outside the core, you have different platforms, schedules, and product managers, etc. In many cases working with another project is like working with a different company. There are different things to manage, so the orchestration of it all becomes problematic.*" (P5).

4.3 Coordination mechanisms across platforms

We observed three primary techniques for coordinating development among platforms with diverged code. Many projects deployed at least two of these approaches:

- Clone MRs across platforms (Section 4.3.1) where a cloned MR is a replica of the original MR, containing the same information stored in the original issue — e.g. Summary, Affects Versions, Components, etc. The cloned MR is typically linked to the original MR.

- Establish a cross platform coordinator role (Section 4.3.2)

- Conduct cross-project MR review meetings (Section 4.3.3).

4.3.1 Cloning MR tickets across platforms

Formal defect tracking systems are used as a standard practice in all Spark-related development teams. To coordinate a common bug fix among projects, many projects create a cloned MR as described by P7: "*... when there is a platform MR ticket linked to an application MR ticket that requires resolution, the ticket in the application would be assigned to a developer who is responsible to track the changes. He would do so through the relevant focal point in our team to the platform team. Once the platform ticket is resolved, the person assigned should verify this, add the info to the ticket and sync up with the assigned build person on the revision that needs to be committed. The assignee would update the ticket with the changes done (move it to resolved with relevant info and create clones if required).*" (Wiki page PrH, P7).

Cloned MRs have been used in many projects: "*We take a ticket from one platform branch, cloning them to another platform branch and linking them to the application feature. Basically it helps us to track back the issues.*" (P2)

4.3.2 Cross-platform coordinators

Technical staff members who address cross-platform issues are critical to maintaining knowledge flow among projects. For example, a designated team consisting of developers from different Spark projects, such as SIP, H323 and 1XC coordinate issues of common concern, such as changes in tools or process, buyback planning, and dependencies between projects. Other projects appointed a developer responsible for examining all components being developed for some other platforms. As an example, an appointed developer made the following observation: "*... there is a Spark engine team member who looks on both sides, Windows and iOS to check for a bug across platforms ...*" (P8).

4.3.3 Cross-platform MR review meeting

The MR review meeting is an important standard practice in all Spark related projects as shown by this typical example. "*Every week the scrum team should meet for about an hour to review the product backlog. This is the process of estimating the existing backlog using effort/points, refining the acceptance criteria for individual stories, and breaking larger stories into smaller stories.*" (Wiki page – Project PrK).

Cross-project MR review meetings include project managers, technical managers and lead developers from each participating project. At each meeting, participants review requests from each participating team, review and create tickets, and make sure team members are aware of what is coming to them. "*... everytime there is an open code inspection everyone in the team can participate ... we have a mandatory of minimum inspection from two reviewers when submiting our code ....*" (P10)

Cross-project MR meetings help creating a culture of regular informal communication to clarify conflict issues, cloning issues and other issues as described in this example: "*Synchronization is done by talking to the other team, such as the platform team. In the development level we are discussing everything, such as testing environment. For the Tel Aviv site, every time they put a fix we need to test whether it introduces a bug to the application. Overall, there is a lot of coordination that happens on a daily basic...*" (P4)

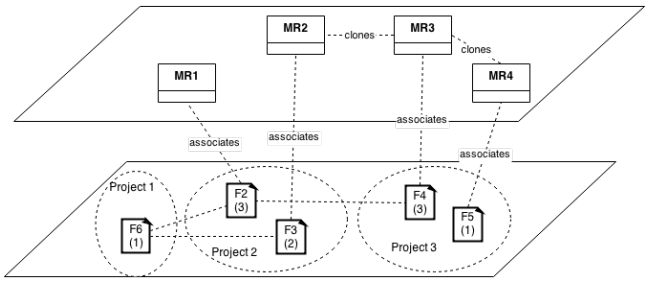

Figure 3: Illustration of proposed measures

5. METRICS

Table 3 provides the definitions for the measures we used to quantify the concepts found in Phase 1 of the study and Figure 3 illustrates them. A change in a file, for instance, F2 in Project 2, can be associated to a specific MR, for example MR1. There is a cloning relationship among MRs, for instance, between MR2 and MR3. In Figure 3, F2 has 3 commits and is cloned by file F6 in Project 1.

5.1 Amount of coordination via cloning MRs

To quantify the amount of cross-platform coordination via cloned MRs we used the number of cloned MRs (NclonedMR). As noted above, cloned MRs are an important bug-fix coordination vehicle with other projects. In the example in Figure 3, NclonedMR(Project 2) = 0 (none for F2) + 1 (MR2 for F3) + 0 (none for F4) = 1 The number of such MRs represents the amount of coordination and the fraction of MRs that is cloned for a project represents the intensity of coordination. We model the intensity of coordination in Section 6.2.

5.2 Extent of cross-platform development

We used two metrics to quantify the extent of cross platform development: average number of forks per file (AvgForkFile) and fraction of files being changed in parallel (FracPara). AvgForkFile quantifies the extent of forking (that indicates potential need for parallel development) for the project code. In the example, AvgForkFile(Project 2) = (2 (Project 1 and Project 3 for F2) + 1 (Project 3 for F3) + 0 (for F4))/ 3 = 1.

FracPara quantifies the realized amount of parallel development and measured by a ratio between the number of files being changed in parallel and total number of file in Project X.We use the ratio because projects with many files might also have many files being changed in parallel.

5.3 Amount of redundant work

We used two measures to quantify the amount of redundant work across platforms: the number of redundant commits (NRCom) and the number of redundant MRs (NRMr). NRCom is defined as a ratio of the redundant to total numbers of commits associated with all related files of a project. A redundant commit is a commit that is repeated among related files. In the example, NRCom(Project 2) = 3 x (3-1)/ 3 (3 commits of F2 being shared with 2 other related files) + 2 x (2-1)/ 2 (2 commits of F3 being shared with another

Table 3: Variable definitions

Varible	Description	Formula
Duration(X)	Duration between the first and last file change in Project X	$max_{f \in X} t_f - min_{f \in X} t_f$ (1)
Nfile(X)	The number of files in Project X	$\mid X \mid$ (2)
Nparalfile(X)	Number of files being changed in parallel. d_f is the period of time over which f is modified	$\mid \{f : \forall f \in X, \exists f' \notin X \wedge d_f \cap d'_f \neq \emptyset\} \mid$ (3)
NclonedMR(X)	Number of cloned MRs in Project X. f is a file and mr_f modifies f	$\sum_{f \in X} \mid \{mr_f : \exists mr : mr \text{ is a clone of } mr_f\} \mid$ (4)
AvgForkFile(X)	Average number of forks related to a file in Project X. $Fork_f$ is a set of forks for file f	$\frac{\sum_{f \in X} \mid \{Fork_f\} \mid}{\mid X \mid}$ (5)
FracPara(X)	Fraction of parallel development in Project X	$\frac{\mid \{f : \exists f' \notin X \wedge d_f \cap d'_f \neq \emptyset\} \mid}{\mid X \mid}$ (6)
NRMR(X)	Number of redundant MRs in Project X	$\sum_{f \in X} \mid \{\cup_{f' \in F_f} mr_{f'}\} \mid * \frac{\mid F_f \mid -1}{\mid F_f \mid}$, where $F_f = \{f' : f' \notin X \wedge f' \text{related to} f\}$ (7)
NrCom(X)	Number of redundant Commits in Project X	$\sum_{f \in X} \mid \cup_{f' \in F_f} Comm_{f'} \mid * \frac{\mid F_f \mid -1}{\mid F_f \mid}$ (8)

related file) + 3 x (1-1)/1 (3 commits of F4 being shared with 0 other file) = 3

For example, a bug-fix commit in one project may need and an independent implementation in another project where developers may not be aware of the original fix. Such an implementation would be clearly redundant because, if aware of the original fix, the second team could reuse it in their project if the code has not diverged too much. Therefore, NRCom may reflect the potential amount of redundant development work .

NRMr is the number of redundant MRs modifying files related to files of a project. A redundant MR is defined as an MR that is repeated among related files. In the example, NRMr(Project 2) = 1 x (3-1)/ 3 (1 MR of F2 being shared with 2 other related files) + 1 x (2-1)/2 (1 MR of F3 being shared with another related files) + 0 (0 MR for F4) = 1.167

Note that redundant MRs do not represent MR clone relationships. Redundant MRs are defined as MRs changing a related file. A high number of redundant MRs implies a large number of related tasks that are implemented independently in several projects.

5.4 Hypotheses about multiple platform coordination

Using the above measures we pose the following hypotheses regarding factors that may affect coordination measured via MR cloning. In particular we expect more coordination measured via MR cloning with the following changes in multi-platform development. We also expect that amount of coordination will increase to solve redundant tasks. These leads to three Hypotheses as below:

H1 The number of cloned MR is possitively associated with the number of files changed in parallel.

H2 The number of cloned MR is possitively associated with the number of redundant commits.

H3 The number of cloned MR is possitively associated with the number of redundant MRs.

6. QUANTITATIVE FINDINGS

We first provide summaries of various measures and then test our hypotheses.

6.1 Descriptive analysis

Table 5 describes some statistics of measures shown in Table 3. Median active duration of a project is 1099 days and maximum value of 1858 days. In average, each project has 2202 files. The smallest project has 81 files and the largest project has 132050 files. This is because some projects involve files from multiple code repositorires and serve as mainline development. Some other more recent projects are branched out to develop a feature or to integrate other functionalities across platforms.

Number of cloned MR per project varies from 0 to 1017 MRs. Average number of related forks per file varies from 0 to 63 forks. 34 projects (79%) have code files spread over more than 10 forks. There are three projects with 70 or more forks, all of which are common engine components that deliver code to many different applications across many platforms. We found a weak correlation (Spearman correlation coefficient = 0.286) between the number of files and

Table 4: Logistic regression model

	Dependent variable:	
	Likelihood of cloning a MR	
	M1	M2
log(NredundantMR/Nfile)	0.055^{***}	
	(0.008)	
log(NrComm/Nfile)		0.017^{**}
		(0.007)
log(FracPara + 1)	-0.204^{***}	-0.182^{***}
	(0.014)	(0.014)
Duration	-0.0005^{***}	-0.001^{***}
	(0.00003)	(0.00003)
Constant	-0.100^{***}	-0.091^{**}
	(0.039)	(0.040)
Observations	32008	32008
Log Likelihood	−1,298.821	−1,318.752
Akaike Inf. Crit.	2,605.642	2,645.503
Note:		**p<0.05; ***p<0.01

the number of forks, suggesting that a large project is not neccesary a project with many related forks.

Fraction of parallel development (FracPara) in a Spark project ranges from 0% to 72%. Six projects had all related files moved without subsequent parallel changes, implying they did not buy back changes from other projects. The median percent of project files changed is 4%. There are two projects with more than 50% of their codebase being changed in parallel. They are relative young projects (less than a median project age) with less than 1000 files.

The number of redundant MRs and commits varies from 6.33 to 34708 (MRs) and 438 to 686792 (commits). This is not unexpected given the large variation of the numbers of forks per project. Altogether there were 280115 redundant MRs across the 43 projects affecting 500783 files.

6.2 What are the probability that an MR would be cloned?

In this section, the results of our logistic regression analysis modeling the intensity of coordination are presented. The response variable is the likelihood that an MR will have a clone (MR clone-proneness). It implies the project's propensity to coordinate work via MRs and reflects the fraction of work that is coordinated in this fashion.

To obtain the response variable, we assigned each MR affecting a project a value of one if it was cloned and zero otherwise. We used a log transformation for all predictors shown in Table 4, except for Duration. This data processing helps to reduce the impact of the highly skewed distribution of collected metrics. The 32008 MRs from 43 projects represent our observations. Because of the high correlation among predictors NRCom and NRMR (Spearman coefficient value of 0.783), we investigated them in separate models M1 and M2.

The logistic regression results are presented in Table 4. M1 uses Fraction of redundant MRs, Fraction of parallel development (FracPara) and Duration as independent factors. M2 uses Fraction of redundnat Commits, FracPara and Duration as independent factors. Residual deviance is 2605 and 2645 for M1 and M2, correspondingly. Each model explains

Table 5: Descriptive statistics of variables

Variable	Min	1/4Q	Median	3/4Q	Max
Duration	428	834	1099	1395	1858
Nfile	81	614	2202	5180	132050
NclonedMR	0	33	97	261	1017
AvgForkFile	0	0	1	3	63
FracPara	0	1	4	16	72
NRMR	6	668	1942	5692	34708
NRComm	438	3990	18243	91980	686792

for c.a 21% of variation. All independent variables in both models are statistically significant at $p < 0.01$.

FracPara is the predictor that explain most variance. Negative coefficient of FracPara shows negative relationships between fraction of files changed in parallel and MR-clone proneness. The result differed from our initial expectation, hence, rejecting H1 (which proposed that an increased number of files changed in parallel increases coordination). This implies that the extent of parallel development decreases the probability of having a cloned MR. Perhaps, forking and more parallel development between independent projects leads to less coordination needs via cloned MRs.

Positive coefficients of NRCom/Nfile and NRMR/Nfile shows a positive relationship between these variables and MR-clone proneness, confirming our hypotheses H2 and H3. Odd ratio value of NRMR/Nfile is 1.505. This observation implies that the amount of redundant work increase the probability of having a cloned MR.

The model indicates that the project duration (reflecting project age) is one of the most important predictors of probability leading to a cloned MR. Interestingly, the model shows a negative relationship between project duration and MR clone-proneness. This may be because more recent projects have been able to take advantage of tools such as JIRA, that has made coordination via MRs easier. Tools in use when older projects started did not facilitate coordination by MRs. Another possibility is that a recent organization policy of refactoring and merging forks lead to a proliferation of young projects with many cloned MRs.

7. DISCUSSION

7.1 Forking and coordination in multi platform development

Joorabchi et al. explored challenges in developing mobile apps and revealed several issues about multiple platform development, such as platform technical variety, maintenance of common core codebases, different scripting language among platforms [10], growing testing space [1, 12], and cross-platform security flaws [1]. In this study we confirmed that challenges occurs between tester-developer and manager-manager. In 2001, Perry et al. conducted an observational case study of parallel development in a legacy multiplatform software system and found that current tools, processes and project management support for this level of parallelism is inadequate [18]. Our study shows that even with support of collaboration tools and modern VCS, identifying technical dependencies and maintaining effective coordination across multiple platforms is still a challenge.

Previous studies suggested inner-project branching strategis to deal with technical issues, such as new branch for new

release, service patch and testing [6]. In this study, we found that inter-project forking is driven by organization strategy and policy. Previous studies found that the main reasons open source projects fork are technical variation, discontinuation of original project, community-driven, sustainability and legal issues [17, 19]. Similar to open source projects, forks are used to divide work into different organizational teams and to deal with technical variations. Unlike open source forks, commercial forks are driven by different schedule release, balance between coordination and development effort and general product development strategies.

Kotlarsky et al. defined an integrative framework with four type of coordination mechanisms: coordination by organization design, work-based coordination, representations of work-in-progress, inter-personal coordination and technology-based coordination [13]. Our study reveals three types of mechanisms: consistent usage of collboration tools across platforms (Cloning MR in Jira), organization design (cross-platform coordinators) and work-based coordination (review meetings). While cloning MR is the main approach to identify and trace duplicated bugs, it is conducted in a rather ad hoc manner. Organization design and work based mechanisms play a role in shaping process and practices in adopting technology-based coordination mechanisms.

7.2 Coordination needs in parallel development

Similar to findings in open source projects [19], most forks evolve in parallel to the original projects. In our study, 65% of forks are active, compared to 73.4% of continued forks in open source projects [19]. In our case, there is a large fraction of MR cloning in Spark (83% of its peak). Schwarz et al. investigated code clone across repositories in an open source ecosystem and found that 14% of all methods are copied from another package in another project [24]. While we did not directly measure the amount of clone across forks in code level, the median value of fraction of cloned MRs is 29.16%. Perry et al. showed that a legacy system might have 3-4 releases undergoing development at any given time [19]. Our studies reveals some period of time Spark projects has all 43 forks changed.

Bird et al. proposed a relationship between goals and virtual teams on different branches [4]. The authors suggested that if two branches have similar goals, they would also have similar virtual teams or be at risk for communication and coordination breakdowns with the accompanying negative effects. Shihab et al. found that branching does have an effect on software quality and that misalignment of branching structure and organizational structure is associated with higher post-release failure rates [25]. Cataldo et al. measured Coordination conguence and found the mismatch in logical coordination requirements and actual coordination via MRs leads to lower software quality. In this study, we found that amount of actual coordination via cloning MR is reduced by defering coordination requirements using independent forks. However, the actual coordination is positively associated with amount of redundant works.

7.3 Threats to validity

While designing and conducting this study, various conscious decisions were taken to strengthen the validity of results. We used a suggested checklist [20] to ensure that we had addressed all the critical requirements of case study design including aims of the study, defining the case, unit of analysis and data collection methods among others.

Table 6: Hypothesis testing

Concepts	Hypotheses	Result
Cross platform dev.	H1	Reverse direction
Redundant work	H2	Expected direction
Redundant work	H3	Expected direction

Construct validity refers to the relations between theory and observation, ensuring that the treatment reflects the construct of the cause and that the outcome reflects the construct of the effect. In this study, perspectives of multiple platform development is quantified based on grounded qualitative findings, which reduce this validity threat.

Internal validity refers to the existance of causal relationships between treatment and outcomes. One way to reduce this threat is to include potential confounding factors that may affect the relationship into the model. We include project duration and several other predictors that may affect the investigated relationship into the model.

External validity concerns about the generalization of observed results to a larger population. The case in our study could represent a large and complex product family in software and telecommunication industry

Credibility concerns the confidence in the result. We adopted several threat mitigation strategies, such as sending interview transcripts to each interviewee for correction; involving three researchers on study design and review, methodological triangulation and detailed documented steps of data collection, processing and analysis. Thus we believe that the resulting detailed protocol serves as a good means to replicate this study.

8. CONCLUSIONS

Our study provides a comprehensive understanding on the state-of-the-practice of multiple platform development. The Spark family has a large amount of parallel development and potentially a lot of redundant work. While intra-project branching is driven by technology, we found that inter-project forking is driven by market and management strategies. Our study also reveals cloning MRs is an important coordination mechanism in the Spark family. Logistic regression analysis suggested that coordination needs decrease in parallel development among independent projects but increase in projects with a large amount of redundant work.

We reveal some opportunities for future resesarch in this area. In this study, we only investigated the extent of forking activities and redundant work via commits and MRs. In future work, we would like to address code divergence and assess the number of identical files between them in a more detail manner. In particular, we would like to investigate if there is any difference between a file that diverges much compared to a file that does not in term of software quality and coordination efforts. This study revealed important coordination mechanisms for multiple-platform development. Future work can focus on evaluating the effectiveness of these mechanisms. Last but not least, our study revealed the main reasons and consequenes of code divergence. Further studies can investigate criteria used to decide whether or not to fork a code repository. The current status of the project, coordination cost and amount of redundant work

would be important inputs for a decision making support tool on whether and when to fork a code repository.

We also derive some implications that practitioners can consider when taking action in multiple development platform. The primary value of VCS is to allow parallel development of branches to delay the merge time. In order to reduce technical debt at system level, periodical synchronization of forks is necessary. Testers need to understand that there might be a lot of redundant functionalities among different platforms. A look beyond scope of a single platform may help to reduce amount of duplicated test cases. Product line managers need to enforce formal processes for cloning MRs across codebases, such as periodical review of cross platform issues. Last but not least, it is helpful to automatically detect code divergence and propagate changes in related files to relevant developers.

9. REFERENCES

[1] S. Amatya and A. Kurti. Cross-platform mobile development: Challenges and opportunities. In *ICT Innovations 2013*, volume 231 of *Advances in Intelligent Systems and Computing*, pages 219–229. Springer International Pub., 2014.

[2] S. P. Analytics. Cross platform mobile development tools market analysis and forecast, 2013.

[3] E. Arisholm, L. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *Software Engineering, IEEE Transactions on*, 30(8):491–506, Aug 2004.

[4] C. Bird, T. Zimmermann, and A. Teterev. A theory of branches as goals and virtual teams. In *Proc. of 4th CHASE*, CHASE '11, pages 53–56, New York, NY, USA, 2011. ACM.

[5] J. Bishop and N. Horspool. Cross-platform development: Software that lasts. *Computer*, 39(10):26–35, Oct 2006.

[6] J. Buffenbarger and K. Gruell. A branching/merging strtegy for parallel software development. In *System Configuration Management*, volume 1675 of *Lecture Notes in Computer Science*, pages 86–99. Springer Berlin Heidelberg, 1999.

[7] M. Cataldo, A. Mockus, J. Roberts, and J. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *Software Engineering, IEEE Transactions on*, 35(6):864–878, Nov 2009.

[8] A. Charland and B. Leroux. Mobile application development: Web vs. native. *Commun. ACM*, 54(5):49–53, May 2011.

[9] S. C. D. Moren, D. Miller. What you need to know about apple's ssl bug, 2014.

[10] R. Fahy and L. Krewer. Using open source libraries in cross platform games development. In *Games Innovation Conference (IGIC), 2012 IEEE International*, pages 1–5, Sept 2012.

[11] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. Softw. Eng.*, 11(8):749–757, Aug. 1985.

[12] M. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 15–24, Oct 2013.

[13] J. Kotlarsky, P. C. van Fenema, and L. P. Willcocks. Developing a knowledge-based perspective on coordination: The case of global software projects. *Inf. Manage.*, 45(2):96–108, Mar. 2008.

[14] R. E. Kraut and L. A. Streeter. Coordination in software development. *Commun. ACM*, 38(3):69–81, Mar. 1995.

[15] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Proc. of 6th MSR*, MSR '09, pages 11–20, Washington, DC, USA, 2009. IEEE Computer Society.

[16] A. Mockus, R. Hackbarth, and J. Palframan. Risky files: An approach to focus quality improvement effort. In *Proc. of 9th ESEC/FSE*, ESEC/FSE 2013, pages 691–694, New York, NY, USA, 2013. ACM.

[17] L. Nyman, T. Mikkonen, J. Lindman, and M. FougÃĺre. Perspectives on code forking and sustainability in open source software. In *Open Source Systems: Long-Term Sustainability*, volume 378 of *IFIP Advances in Information and Communication Technology*, pages 274–279. Springer Berlin Heidelberg, 2012.

[18] D. Perry, H. Siy, and L. Votta. Parallel changes in large scale software development: an observational case study. In *Proc. of ICSE*, pages 251–260, Apr 1998.

[19] G. Robles and J. M. GonzÃąlez-Barahona. A comprehensive study of software forks: Dates, reasons and outcomes. In *Open Source Systems: Long-Term Sustainability*, volume 378 of *IFIP Advances in Information and Communication Technology*, pages 1–14. Springer Berlin Heidelberg, 2012.

[20] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, 2009.

[21] J. Savolainen, M. Mannion, and J. Kuusela. Developing platforms for multiple software product lines. In *Proc of 16th SPLC*, SPLC '12, pages 220–228, New York, NY, USA, 2012. ACM.

[22] S. R. Schach. *Practical Software Engineering*. Richard D. Irwin/Aksen Associates, Homewood, IL, USA, Feb, 1992.

[23] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[24] N. Schwarz, M. Lungu, and R. Robbes. On how often code is cloned across repositories. In *Proc. of 34th ICSE*, pages 1289–1292, June 2012.

[25] E. Shihab, C. Bird, and T. Zimmermann. The effect of branching strategies on software quality. In *Proc. of the ACM-IEEE ESEM*, ESEM '12, pages 301–310, New York, NY, USA, 2012. ACM.

[26] M. Swider. Microsoft highlights 299m skype users, 1.5b halo games played, 2013.

[27] A. I. Wasserman. Software engineering issues for mobile application development. In *Proc. of the FSE/SDP FoSER*, FoSER '10, pages 397–400, New York, NY, USA, 2010. ACM.

[28] R. K. Yin. *Case study research: Design and methods*. Sage Pub., Thousand Oaks, CA, 2003.