Clone-and-Own
Analysis of an Industrial Automation System

Nick Lodewijks
nicklodewijks@gmail.com

October 31, 2017, 48 pages

Supervisor: Prof. Dr. Jurgen Vinju
Host organisation: ENGIE Industrial Automation

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

In industry, the development of similar products is often addressed by cloning and modifying existing artifacts. This so-called "clone-and-own" approach is often considered to be a bad practice but is perceived as a favorable and natural software reuse approach by many practitioners. Current literature lacks deep qualitative information about the positive and negative effects of clone-and-own, and in particular, it lacks insight on the contextual factors which influence it. In this thesis, we show how version control system metadata, source-code differencing, and a variety of visualization techniques can be used to explore and quantify the benefits and drawbacks of clone-and-own. We apply the techniques we developed on a large ($\pm$1 million lines of Java code) proprietary factory automation system. Our results show that all MES-Toolbox systems we analyzed benefited from clone-and-own to some extent, but also caused maintenance overhead. However, some systems caused significantly more maintenance overhead than others.

Chapter 1

Introduction

Cloning is often considered to be a practice harmful to the quality of source code, and potentially a cause of maintainability problems [16, 31]. Yet, in industry the development of similar products is often addressed by cloning and modifying existing artifacts. This so-called *clone-and-own* approach is perceived as a favorable and natural software reuse approach by many practitioners, mainly because of its simplicity and availability [8].

While the general belief is that clone-and-own is a bad and unsustainable development technique, it has been used successfully for the development of the MES-Toolbox; a large (±1 million lines of Java code) proprietary factory automation system. Over the past 16 years, for each new customer an existing system was cloned and modified in any possible way to add, modify or remove functionality. With over 70 implementations of the systems running world-wide, the company now seeks to reduce maintenance overhead and to cope with the complexity caused by clone-and-own. Unfortunately, the decision on how to move forward from a successful clone-and-own approach is not straightforward.

Over the past decade, several tools and techniques for dealing with cloned product variants have been proposed. Some of them advocate elimination of all clones by merging the variants into a single platform, and others propose to maintain multiple variants as-is [27]. What approach works best for a given situation depends on the domain and context of that situation. In some cases eliminating all clones and adopting an integrated platform is neither possible nor beneficial [2]. Eliminating clones will increase coupling, and changing shared code may require re-testing of all systems that use it [8]. If the success of the product highly depends on the benefits of clone-and-own, then completely moving away to a different approach without considering its merits can be a reckless decision.

Current literature lacks deep qualitative information about the positive and negative effects of clone-and-own, and in particular it lacks insight on the contextual factors which influence it. Therefore, the main objective of this study is to explore the evolution of MES-Toolbox systems, and to gain insight into how clone-and-own has affected ongoing project development and maintenance. In this thesis, we show how version control system metadata, source-code differencing, and a variety of visualization techniques can be used to explore and quantify the benefits and drawbacks of clone-and-own.

1.1 Research Questions

To gain insight into how clone-and-own has affected the development and maintenance of MES-Toolbox systems, we define two main research questions; one for the benefits of clone-and-own and one for the drawbacks.

RQ1: *Have any MES-Toolbox systems benefited from the independence provided by clone-and-own?* Dubinsky et al. [8] observed that independence provided by clone-and-own is one of the major reasons for considering cloning as an efficient reuse mechanism. Developers can make any change required to satisfy customer requirements, without affecting other clones. They do not have to collaborate with teams working on other systems, that may have different priorities or scheduling constraints. These characteristics of clone-and-own have to be considered when new change mechanisms are introduced, since different techniques may not provide the same degree

of independence.

RQ2: *To what extent has clone-and-own caused maintenance overhead for MES-Toolbox systems?*

1.2 Research Method

To gain a quantitative understanding of the benefits and drawbacks of clone-and-own, we take a system-centric view by focusing on the source code of cloned systems and their evolution. To answer the aforementioned research questions, we developed a tool to extract data from the version control system, which we further analyze in R[1]. We discuss the details of our analyses infrastructure and data collection process in chapter 3. This includes the selection process of systems we included in our analyses. Next, we address the benefits (**RQ1**) and drawbacks (**RQ2**) in chapter 4 and chapter 5, respectively. For each questions we define two sub-questions that focus on a particular aspect of the main question, and define hypotheses that lead our study.

1.3 Contributions

Contribution #1: Quantitative approach to Clone-and-Own Benefits and Drawbacks
The benefits and drawbacks of cloning have been studied before, but mostly qualitatively. Qualitative studies are important to gain an in-depth understanding of the problem in general or in specific context, but they only provide a textual description of characteristics of the problem. We translate qualitative findings of previous studies to quantitative measures, and use these to study the evolution of the product family.

Contribution #2: Visualization Techniques
We developed visualization techniques that can be used to gain insight into the evolution of a product family, and to identify clone-and-own related points of interest. The characteristics we focused on were the rate in which cloned systems changed, differences in change distribution, and how much clones diverged from their origin.

Contribution #3: Industry Case Study
We provide a significant amount of data on the evolution of an industry system that has never been studied before. We support our interpretation of the data with unfiltered numerical and graphical representations of the data. For example, we not only state that systems change continuously, we *show* they change continuously. Very few studies publish this kind of data, as this is often subject to confidentiality.

Contribution #4: Clone-and-Own Analysis Tool
To effectively study the evolution of the MES-Toolbox product family, we developed an analysis tool with the following functionality:

- Replay the change history of cloned product variants.
- Perform code differencing at each step in the history.
- Combine change history and code differencing results of multiple variants in a single view.

[1] www.r-project.org

Chapter 2

Background

In this chapter we discuss the background information and context of this study. In the next section (2.1) we explain what clone-and-own is from the perspective of *software reuse*, and discuss how it relates to *Product Line Engineering*. Next, in section 2.2 we describe the history and organization development practices of the systems we study.

2.1 Software Reuse

In 1968 the term *software reuse* was coined to overcome the *software crisis* – the problem of building large, reliable software systems in a controlled, cost-effective way [19]. Computing power and the complexity of the problems rapidly increased, and could no longer be addressed efficiently with existing software development techniques. Krueger [19] described software reuse as *the process of creating software systems from existing software rather than building software systems from scratch.*

Back then software engineers already recognized that software systems built for different purposes can share functionality. Instead of developing the same functionality twice, it would be more efficient to develop it once in the form of a *library* – a collection of software entities that can be reused for the development of multiple systems. Figure 2.1 illustrates the difference between systems developed with and without libraries. In this example, the two systems are highly similar. Instead of duplicating components *a* and *b* in each system, these components are contained in a shared library.

Systems that have similar but not identical functionality are called *product families* or *Software Product Lines* (SPL). In this thesis, we use the words *system* and *product* interchangeably. Each product in a product family is a *variant*. Maintenance of products families adds an extra dimension to the already challenging task of software maintenance and development. Each individual variant can have its own peculiarities which have to be considered during maintenance of the system. To deal with this additional complexity, the field of Software Product Line Engineering (SPLE) emerged.

2.1.1 Software Product Line Engineering

Software product Line Engineering is a development paradigm that promotes the combination of a common platform, and mass customization to satisfy customer needs, for the development of software-intensive systems [25]. Its main focus is the identification, tracing, and manipulation of common and variable artifacts, where common artifacts are part of all products in the product family, and variable artifacts are those that are specific to some individual products [26]. The common and variable artifacts are mapped to *features*; high-level descriptions of functionality from the customer's point of view. The product line is described by a *feature model*, a model containing the features and relations among features. Individual products are *derived* from the product line based on a selection of features from the feature model. Figure 2.2 illustrates this concept. The platform contains both common and variable artifacts, and the products are derived using their feature model.

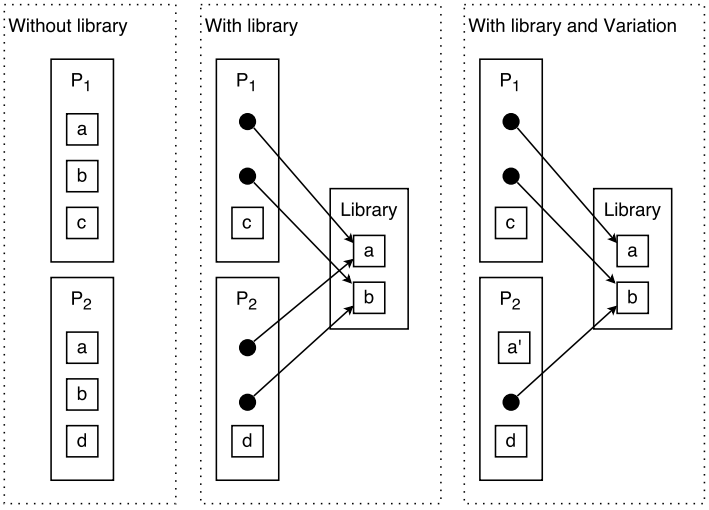


Figure 2.1: Software reuse by sharing functionality contained in a library.

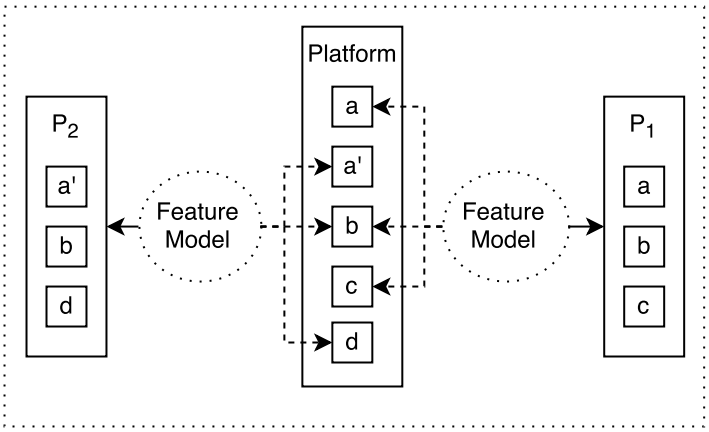


Figure 2.2: Individual products are derived from a common platform using a *feature model*.

2.1.2 Clone-and-Own: Cloned Product Variants

Clone-and-own is a technique where whole systems or sub-systems are copied and modified to satisfy requirements that are similar, but not identical to the original project. *Clone* refers to *copying* of artifacts and *own* to taking ownership of the copy by making modifications. Figure 2.3 illustrates this technique. When cloning is used to address new or changing customer requirements, after a while there will be a significant number of variants of the system. This uncontrolled growth of systems is called *Software Mitosis*[10], which inevitably leads to a loss of overview. If new techniques have been developed to deal with this complexity, then why is cloning still used?

Dubinsky et al. [8] studied the processes and perceived advantages and disadvantages of the clone-and-own approach of six industrial software product lines. They show that cloning is perceived as a favorable and natural reuse approach by the majority of practitioners in the studied companies, mainly because of its simplicity and availability. They found that practitioners lack the awareness and knowledge about forms of reuse, and many alternative approaches fail to convince them that they yield better results.

Several tools and techniques for dealing with cloned product variants have been proposed. Some of them advocate elimination of all clones by merging the variants into a single platform, and others propose to maintain multiple variants as-is [27]. To deal with this software mitosis, Faust and Verhoef [10] propose to use a *grow-and-prune* technique. This approach allows phases of uncontrolled growth to address customer needs, followed by pruning phases where commonalities between products are

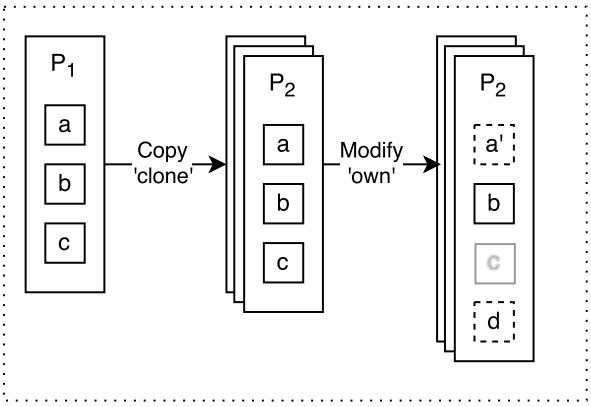


Figure 2.3: Individual products are developed by modifying a copy of an existing products. Files can be modified (*a'*), deleted (*c*) or added (*d*).

identified, and generic solutions can be created.

2.2 Subject System: MES-Toolbox

The system studied in this work is the MES-Toolbox; a proprietary Java-based factory automation system developed by ENGIE. Development of this system started 17 years ago, and it has grown to contain more than 6500 Java files, with a total of approximately 1 Million Lines Of Code (MLOC).

The MES-Toolbox is designed for industrial automation of batch and continuous production processes. It can visualize, control and register every step of an entire production process. From the intake of raw material (unloading from trucks, ships, bags, pallets, containers), preparation (dosing, weighing, heating), processing (pressing, grinding, mixing), storage, to distribution of end products to customers. Depending on what customers require for their production process, the system performs article and recipe management, quality registration, production planning, tracking and tracing of materials used in production, stock control, shift registration, production performance analysis and communicates with ERP systems.

2.2.1 Architecture

This functionality is spread out over some services that run in parallel on multiple JVM's on a dedicated server in the factory. Most of these services can safely be rebooted or reconfigured without disturbing the production process. To monitor and control physical production equipment (e.g., conveyors, mixers, weigher, buttons, lights), the MES-Toolbox communicates with Programmable Logic Controller's (PLC's) that perform the actual low-level control of these physical devices. Much of this PLC code is generated from the configurations contained the MES-Toolbox, as they often include detailed information about the physical equipment and production processes.

The system has a modular structure, and its design aims to separate *common* code from *customer* implementation code as much as possible. In practice, however, this has proven to be very challenging due to the specificity and high degree of variation of customer requirements in the domain of industrial automation [29]. While these ever-changing requirements have led to a variant rich and highly configurable platform, it has also caused many clones to *diverge* from their origin.

2.2.2 Clone-and-Own Organization

Within the organization there is a clear distinction between platform development and application development, this distinction is often found in a Software Ecosystem (SECO) [21]. A small team of five developers is responsible for the overall design, development, and maintenance of this system. The founder and writer of the first line of code of this system is also still part of this team. Work of this team is focused on maintenance of the core platform, development of complex customer specific

features, standardization of functionality, development of product configuration tools, and provide support to application engineers.

While the platform contains a constantly growing set of reusable core components and ready-to-use standard solutions, for every new factory, a clone of the latest platform release is realized by creating a branch with the *Subversion* version control system. The clone is then configured and changed in any possible way by Application Engineers to add, modify or remove functionality. Most of the application engineers co-located with the platform engineers.

2.2.3 Version History

There are currently four platform versions: 7.1, 7.2, 7.2.1 and 7.3. The first version of the platform (7.1) was released on 7 March 2012 and was followed relatively fast by the next release (7.2) on 18 December 2012. Version 7.2.1 of the platform was released on 7 October 2014, and version 7.3 on 14 December 2016. There are currently over 70 clones of the platform, as can be seen in Figure 2.4 which shows the revision graph of the repository generated by TortoiseSVN[1].

[1] https://tortoisesvn.net/

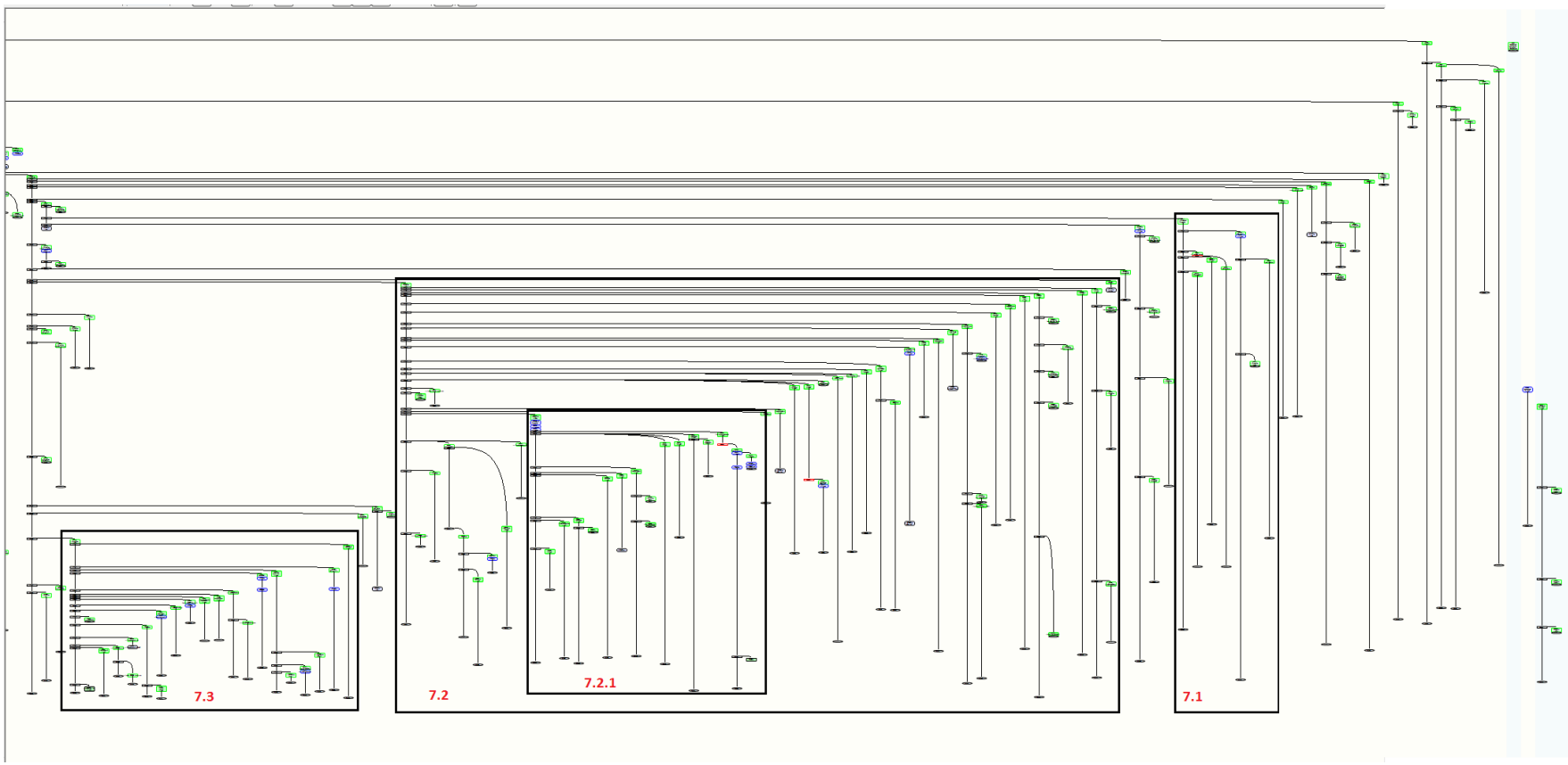


Figure 2.4: Revision Graph of the repository generated by TortoiseSVN. Each green box represents a branch in the repository.

Chapter 3

Repository Mining

To answer the research questions defined in section 1.1, we built a tool that retrieves changes to each system from the subversion (SVN) repository, performs source-code differencing and exports the relevant information to a csv file for further analysis in R[1]. Our tool is embedded in a modified version of JMeld[2], an open source differencing tool written in Java. The main purpose of the data we collect is to explore how the systems have evolved and to identify possible benefits and drawbacks of clone-and-own in this particular industry case. In this section we discuss the infrastructure we used to perform our analysis. This infrastructure is illustrated in figure 3.1.

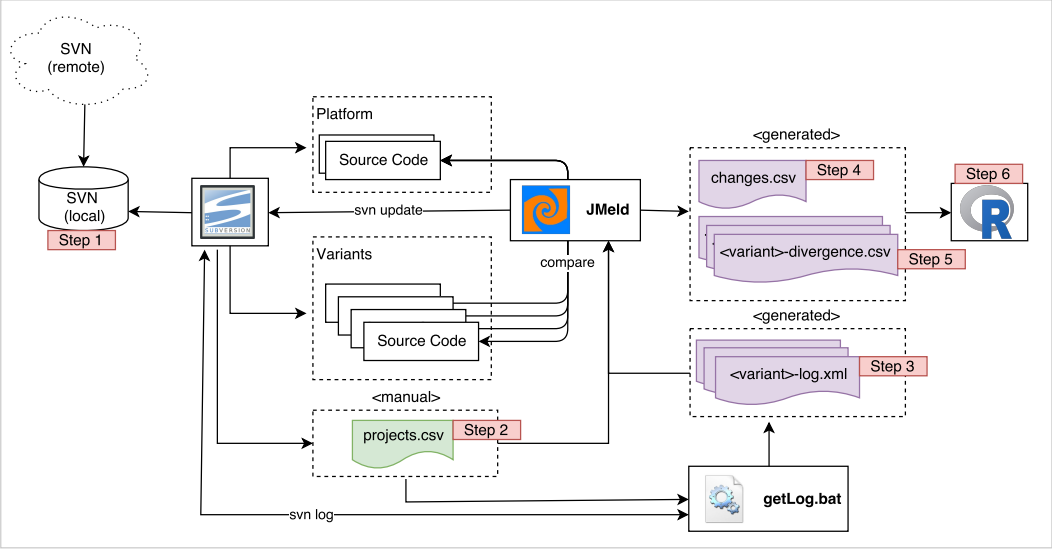


Figure 3.1: Schematic representation of the repository mining process

3.1 Data Collection

In this section we explain how we collected the data that was used for this study.

Step 1: Local Copy We make a local copy of the SVN repository with the command `svnadmin hotcopy`, and verify its integrity with `svnadmin verify` on the analysis environment. This local repository is used for all data collection to ensure that the data source does not change during subsequent analysis.

[1] www.r-project.org
[2] https://sourceforge.net/projects/jmeld/

Step 2: projects.csv We extract all systems present in the repository by scanning the output of `svn ls`[3] for paths in the form of `projecten/.*/trunk/$`. We then manually validate these paths, and documented for each system the platform version it was branched from, the name of the project, an anonymised name, the repository path, and any unusual properties of the system that we have to consider during analysis. For example, development of some systems was discontinued and were never put into production. We excluded these systems from the analysis. Finally, we noted whether the system was directly branched from the platform, or from another branch (its *nesting depth*). We use the platform version *pre* for projects that predate the first platform release, and were directly branched from the development branch of the platform.

The name of the project can contain the name of the customer, and the location of the production facility. Since this information is subject to confidentiality, we manually defined an anonymised name for each system. The main focus of this study is on systems derived from platform version 7.2 or later. These systems are coded with an anonymous name in the form of `P-NUMBER`. In this thesis, we often refer to this name as P_n. Systems which pre-date the 7.2 platform release are anonymised in the form of `PR-N`.

Figure 3.2 shows the distribution of versions among the systems we selected for analysis. In total, we identified 60 systems for this study. Twenty-one systems pre-date the first platform release. There are five 7.1 systems, fifteen 7.2 systems, eleven 7.2.1 systems and eight 7.3 systems. For this study, we mainly focus on 7.2 and 7.2.1 systems, as all of them have been put into production, and are derived from a comparable base platform within the last five years.

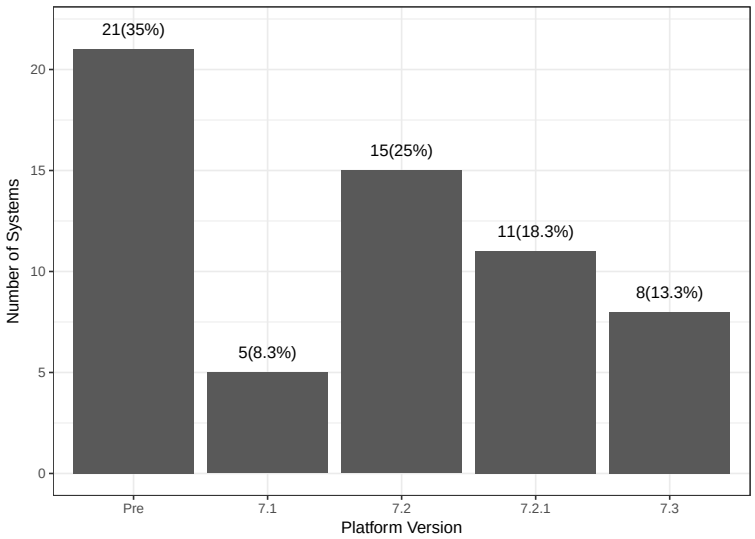


Figure 3.2: Distribution of System Versions

Step 3: <variant>-log.xml For each system we defined in `projects.csv`, we extract the version history using a bash script. This bash script uses the `svn log` command to export the version history in xml format.

```
svn log --xml --stop-on-copy -v <variant.repositoryPath> > <variant>-log.xml
```

Step 4: changes.csv This file contains the change history of all systems combined. For this file we used the definition of the change metrics dataset published by Yamashita et al. [34].

First, for each system we collect all revisions from its change history (`<variant>-log.xml`), and extract the *revision number*, *author* and *date* of each revision. Next, for use `svn diff` to determine what was changed in the revision.

[3] svn ls -R {svnRepo} | egrep "projecten/.*/trunk/$"

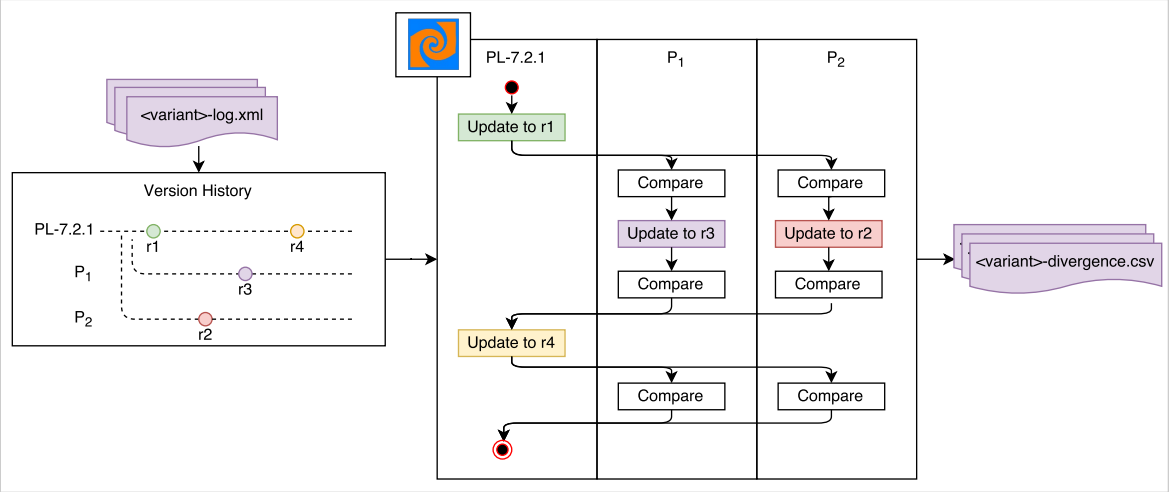


Figure 3.3: Schematic representation of divergence measurement technique. Systems are compared to their origin in parallel.

```
svn diff -x -U0 -c <revisionNumber> <variant.repositoryPath>
```

From the output of `svn diff`, we determine the *full path* of the files that were changed, the *type of change* (added, deleted or modified), and calculate for each file how many lines were changed, added or deleted. Note that we use `svn diff` to determine which files were changed, and not `svn log`. The reason for this is that when a directory is deleted, the output of `svn log` only contains the name of the directory, and does not contain the names of the files contained in the directory.

From the full path of the files we extract the *file name*, *file extension*, *package name*, and determine whether the file is in a *common* or *customer* part of the codebase.

Step 5: <variant>-divergence.csv For each system we measure divergence over time by calculating the differences between the system and its origin, for each file, at every revision that changed either the system or its origin. We measure differences at line-level granularity (number of lines different) with the Java implementation of GNU diff [4]. We define the difference in number of lines as `diff`. During analysis we keep track of how much the difference has increased or decreased, the `diffDelta`. To speed up the process, we parallelized the comparison of systems with their origin. Figure 3.3 illustrates the technique we used to calculate the differences.

Step 6: Analysis in R We perform the numerical and graphical analysis of the data we collect in R.

[4] http://www.bmsi.com/java/#diff

Chapter 4

Analysis of Clone-and-Own Benefits

The main question we answer in this chapter is: *Have any MES-Toolbox systems benefited from the independence provided by clone-and-own?*. In the following section, section 4.1, we discuss how independence can be considered beneficial by comparing clone-and-own development to development using a shared codebase. Based on theory, we identify two hypotheses. In section 4.2 we propose metrics to quantify independence, and explain the techniques we use to explore independence related characteristics. In section 4.3 we report our results, which we interpret in section 4.4. Finally, we conclude in section 4.6.

4.1 Research Questions

To determine whether systems benefited from independence, we consider two types of independence: independence in space and independence in time. To understand how MES-Toolbox systems may have benefited from independence in time and in space, we compare clone-and-own development to development using a shared codebase.

Independence in Time

If the codebase of systems is shared, then any change will affect all systems at the same time. This is often unwanted, as systems for different customers may be developed by different teams, following different release or development schedules. When cloning is used, developers can decide *when* to change the system. Each system can have its own release and development schedule.

Changing a system after it has been tested and released can be costly, as it requires re-testing of the system. Ideally the system should not change after it has been tested and released, we refer to this as being *stable in time*. According to the software change taxonomy of Buckley et al. [5], changes can be performed continuously, periodically or at arbitrary intervals. Complex projects may require and allow for many months of continuous, frequent change, while relatively straightforward projects might require only a few changes within the first weeks. If these systems were to be developed on a shared codebase, than the activity of the complex project would negatively affect the stability in time of the more simple project. We define benefit from independence in time as follows:

Axiom 4.1: Benefit: Independence in Time

Systems benefit from independence in time if their stability in time is not negatively affected by the change activity of other systems.

Do MES-Toolbox systems change in parallel? To determine whether MES-Toolbox systems benefited from independence in time, we are interested in the degree to which systems change in parallel. If systems have always changed at the same time, then their stability in time could have been the same if clone-and-own was not used. This leads us to the first hypothesis:

H_1: *Some MES-Toolbox systems always changed at the same time, thus would have been equally stable in time if a shared codebase was used.*

Independence in Space

If the codebase of systems is shared, then any change will affect all systems. When cloning is used, this is not the case and developers can decide *what* to change. Due to differences in requirements, changes to one system may be different from changes to other systems. For example, if system A does not use functionality B, it does not require changes that solely affect functionality B. Independence in space can be considered at different levels of granularity. On a fine-grained level we can consider the space of change as *lines*, and on a course-grained level as *files*. For this study, we consider the space of change as *files*. If a file is not changed, we define this as *stable in space*. We define benefit from independence in space as follows:

Axiom 4.2: Benefit: Independence in space

Systems benefit from independence in space if their stability in space is not negatively affected by the change distribution of other systems.

Do MES-Toolbox systems have a similar file change distribution? In the MES-Toolbox product family, some systems are regarded as complex and require a significant amount of modification to satisfy customer requirements. This is not the case for all systems, as some systems supposedly only use functionality already available in the system. If this is true, than we expect these systems to have a similar file change distribution, thus not benefiting from independence in space. This leads us to the second hypothesis:

H_2: *Some MES-Toolbox systems have an identical file change distribution, thus would have been equally stable in space if a shared codebase was used.*

4.2 Research Method

In this section we discuss how the study on independence provided by clone-and-own has been set up. To study whether systems change independently in time, we explore how frequent systems have changed and quantify how frequent systems changed at the same time (section 4.2.1). Similarly, to study whether systems change independently in space, we examine for each system what files have been changed and quantify the degree of similarity between two systems (section 4.2.2). For both independence in space and independence in time, we first develop a technique to visually identify points of interest, and propose quantitative metrics to test our hypothesis.

Figure 4.1 illustrates our interpretation of independence in time and independence in space.

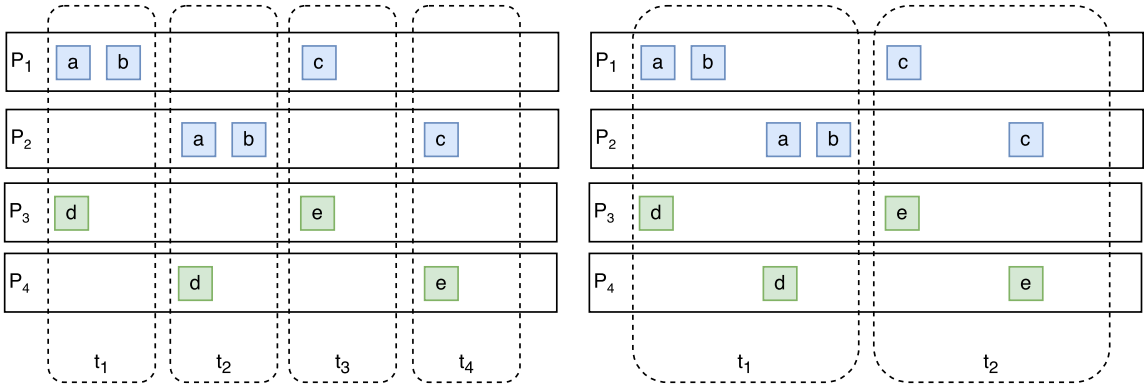


Figure 4.1: Systems developed using clone-and-own can change independently in time and independently in space. In the example on the left, systems P_1 and P_2 changed independently in time, but not in space. Systems P_1 and P_3 changed independently in space, but not in time. Systems P_1 and P_4 changed independently both in time and in space. However, the example on the right shows that courser-grained time periods may result in the observation that all systems change at the same time.

4.2.1 Independence in Time

For hypothesis **H_1** we are interested in the time aspect of change at system-level granularity. We decided to use a visualization which allows us to gain insight into whether (a) systems change in parallel, (b) systems change continuously, periodically or at arbitrary moments in time, and (c) to identify variance between systems.

For this visualization we chose *systems* as the first dimension and *time* as the second dimension. To prevent overplotting, we group data-points by week or month. By grouping data we will not be able to distinguish between systems that changed many times a week, or only once a month. To mitigate this effect we introduce an additional dimension which is *number of commits* (proportional to radius of dot). This leads us to the view shown in figure 4.2.

The vertical axis represents the systems, and the horizontal axis the time of the changes. Each dot represents a point in time where a system was changed. The radius of the dot is proportional to the number of commits that occurred. In this example, we group the data-points by week. Continuous change will give rise to a sequence of horizontally aligned dots. Changing a system twenty times a week will result in a thicker horizontal dot pattern compared to changing a system only once a week.

Interpretation In figure 4.2 we observe that system 1 was under continuous maintenance, as it was changed every week. System 2 was changed every other week, which appears to be more periodical but due to the week-based granularity may still be considered as continuous to some extent. The change activity for systems 3 and 4 is continuous for the first three weeks, but declining for system 3 and increasing for system 4. Finally, we see that systems 1, 2 and 3 all changed in the first week, but system 3 has been modified more frequent.

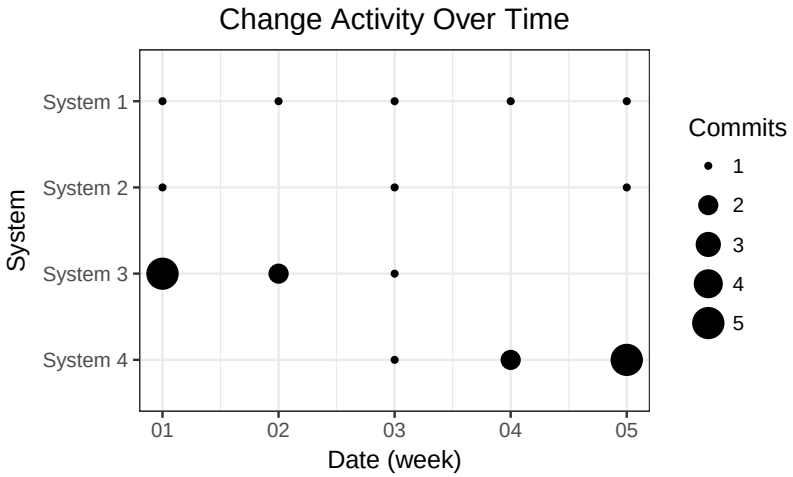


Figure 4.2: Example visualization for change frequency

Quantifying Independence in Time

While the change frequency view should give is some insight into whether systems change in parallel, it will not provide us with a quantitative answer. To determine whether systems have benefited from independence in time at the system-level, we calculate the degree to which systems have changed at the same time (in parallel). We mathematically define this as follows.

Let $T_M(s)$ be the points in time in which system s was modified. We define $T_{MP}(s_x, s_y)$ as the points in time in which system s_x and s_y were both modified:

$$T_{MP}(s_x, s_y) = T_M(s_x) \cap T_M(s_y) \tag{4.1}$$

To determine the fraction of parallel development, we only consider the time in which *both* systems existed. For the purpose of our study, it would be illogical for a system to affect the stability in time of a system system that does not yet exist. We define $T(s)$ as all points in time in which system s existed. We define $T_{MC}(s_x, s_y)$ as the points in time in which system s_x was modified contemporary with the age of s_y:

$$T_{MC}(s_x, s_y) = T_M(s_x) \cap T(s_y) \tag{4.2}$$

To calculate the proportion of parallel change, we divide the number of points in time in which both systems were changed by the number of contemporary change periods of both systems combined:

$$fracPara(s_x, s_y) = \frac{|T_{MP}(s_x, s_y)|}{|T_{MC}(s_x, s_y) \cup T_{MC}(s_y, s_x)|} \tag{4.3}$$

It is possible that systems have not always changed in parallel, but that the activity of one system is fully contained in the other. To quantify this property, we define $fracParaC(s_x, s_y)$ as the number of parallel change periods of systems s_x and s_y, divided by the number of contemporary change periods of system s_x. Note that this function is asymmetric:

$$fracParaC(s_x, s_y) = \frac{|T_{MP}(s_x, s_y)|}{|T_{MC}(s_x, s_y)|} \tag{4.4}$$

We report the results of this calculation in the format shown in table 4.1. Each cell contains the value of $fracParaC(s_x, s_y) \mid fracPara(s_x, s_y)$, where rows represent s_x and columns s_y.

Interpretation We see in table 4.1 that the change activity of system 2 contemporary with the age of system 4 ($T_{MC}(s_2, s_4)$) is fully contained in the activity of system 4 ($T_M(s_4)$). However, only 66% of the change activity of system 4 is contained in the activity of system 2. This suggests that the systems have not always changed in parallel. Hypothetically, if both systems were developed using a

	s_y	System 2	System 4
s_x			
System 2		-	1.00 | 0.66
System 4		0.66 | 0.66	-

Table 4.1: Fraction of overlapping change activity for system 1 and system 4 from example figure 4.2.

shared codebase, then the changes required for system 2 would not have affected the stability in time of system 4. Thus, system 4 has not benefited from independence in time from system 2. However, system 2 *has* benefited from independence from system 4, as 33% of the activity for system 4 would have negatively affected the stability in time for system 2.

4.2.2 Independence in Space

For hypothesis **H_2** we are interested in the file change distribution of MES-Toolbox systems. To visualize the change distribution of files we adopted a 2-dimensional view proposed by Van Rysselberghe and Demeyer [33]. In this view, as shown in figure 4.3, each change to a file is represented as a dot. The horizontal axis represents the file that is changed, and the vertical axis indicates the time of the change. Files have been sorted in alphabetical order, which causes files in the same directory to be close to each other on the horizontal axis. Each file has been given a unique identifier (fileID). We split `MERGE` and `NON_MERGE` changes such that we will be able to identify files that are frequently merged.

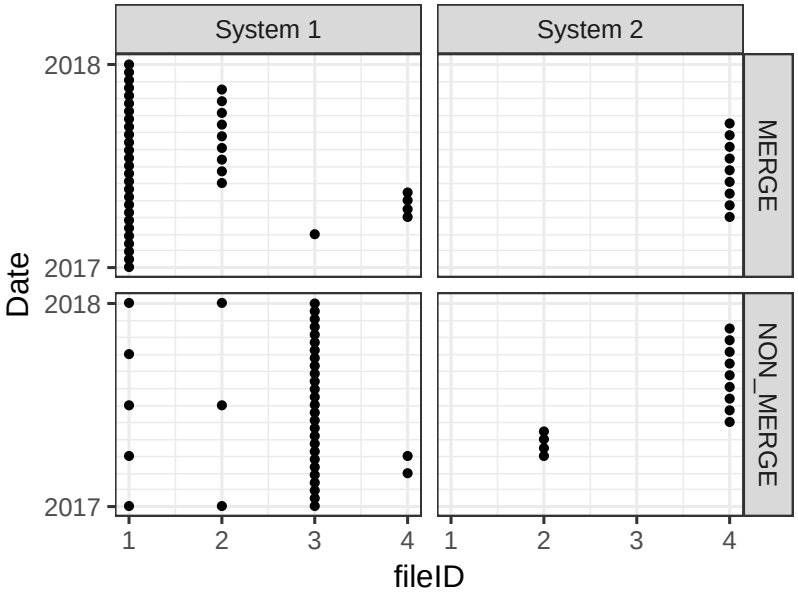


Figure 4.3: Example visualization for change distribution

Interpretation Figure 4.3 shows that in the history of system 1, files 1, 2, 3 and 4 were modified. In the history of System 2 only file 2 and 4 were modified. The change distribution of system 1 appears to contain much more dots than the distribution of system 2, suggesting that the distribution of system 1 may be different from the distribution of system 2. We numerically calculate the similarity to confirm whether this observation is valid.

Quantifying Independence in Space

To determine whether systems have a similar file change distribution, we need a measure for change distribution. We define $F_M(s)$ as the set of all files that were modified in system s. We calculate the degree of overlapping change distribution as the number of files changed by both systems, divided by the number of files changed by the systems combined:

$$fracDist(s_x, s_y) = \frac{|F_M(s_x) \cap F_M(s_y)|}{|F_M(s_x) \cup F_M(s_y)|} \tag{4.5}$$

It is possible that the change distribution does not completely overlap, but that the distribution of one system is contained in the distribution of the other system. We define $fracDistC(s_x, s_y)$ as the number of files modified by both systems, divided by the number of files modified by s_x.

$$fracDistC(s_x, s_y) = \frac{|F_M(s_x) \cap F_M(s_y)|}{|F_M(s_x)|} \tag{4.6}$$

We report the results of this calculation in the format shown in table 4.2. Each cell contains the value of $fracDistC(s_x, s_y)$ | $fracDist(s_x, s_y)$, where rows represent s_x and columns s_y.

	s_y	System 1	System 2
s_x	n	2	4
System 1	2	-	1.0 | 0.5
System 2	4	0.5 | 0.5	-

Table 4.2: Fraction of overlapping change distributions for systems in example figure 4.3. The change distribution of system 1 and system 2 overlaps for 50%. However, the change distribution of system 1 is fully contained in the distribution of system 2.

Interpretation In table 4.2 we see that the file distributions of System 1 (s_1) and System 2 (s_2) consisted of respectively 2 files and 4 files. We see that the distribution of s_1 is fully contained in the distribution of s_2. The stability in space of System 2 would not be negatively affected if these systems were to be combined. However, only 50% of the files changed in s_2 were changed in s_1, meaning that the stability of s_1 would be negatively affected if the systems were combined. We conclude that s_1 benefited from independence in space from system s_2.

4.3 Results

In this section, we present the results of our exploratory analysis.

4.3.1 Independence in Time

The change frequency view of the PL-7.2 and PL-7.2.1 platforms and all systems derived from these platforms can be seen in figure 4.4. For completeness, figure 4.6 illustrates the change frequency view all systems we analyzed.

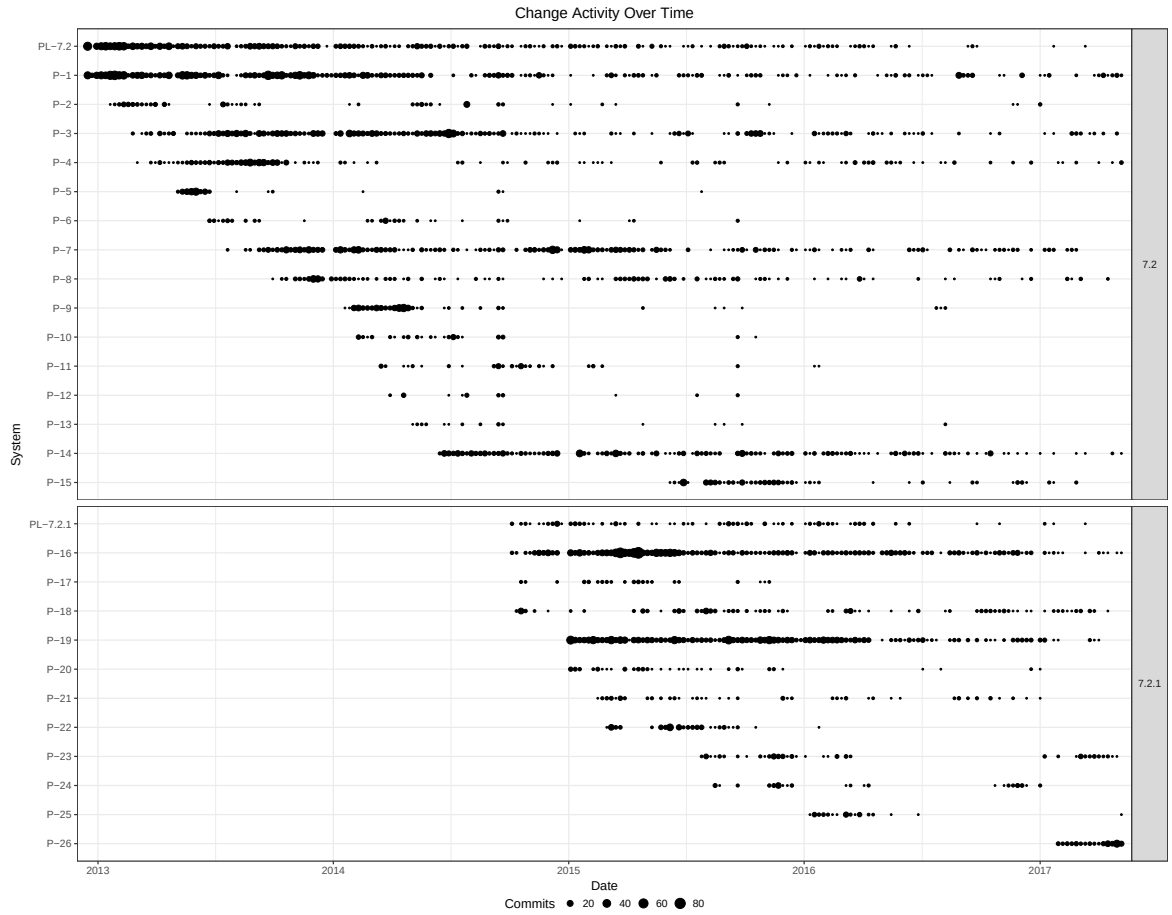


Figure 4.4: Visualization of system change frequency of PL 7.2 and PL 7.2.1 systems

We see that many systems appear to be modified almost continuously, even years after the first change was made. For example, systems P_1 and P_3. Systems P_4, P_8 and P_{18} also appear to be changed continuously, but to a lesser extent than the first group. The change activity for these systems appears less dense and contains more periods of inactivity. The longest period of inactivity for these systems is approximately four months[1] for system P_4.

Furthermore, we observe that active periods can be separated by relatively long periods of inactivity. For example, P_{23} has two periods of activity separated by an inactive period of almost ten months[2].

The majority of the systems show an initial burst of activity at the beginning of the project, followed by a varying amount of activity afterward. Manual examination of changes suggests that they are often (critical) bug-fixes or minor changes requested by the customer. For example, P_5 was changed on 31/07/2015 after being inactive for almost a year (311 days). The commit message says 'Added

[1] 124 days, 14/03/2014 to 07/16/2014
[2] 299 days, 16/03/2016 to 09/01/2017

alcohol flow meter failure contact'. Twenty lines were added to `PhysicalModelConfiguration.xml`, and a failure indicator was added to the factory visualization. This change was triggered by a customer request, after the failure indicator was physically added to the production line.

Parallel Change

The change activity view in figure 4.6 suggests that a significant number of systems change at the same time. Figure 4.5 illustrates the number of systems changed within the same month between 2005 and 2016. Here we observe that this number has increased over time. The number of systems changed within the same month has grown from only 5 systems in 2011, to 24 systems in 2015. Note that this number is a lower-bound, as this only includes all systems shown in figure 4.6. These are only the platform releases, the platform development branch, and most but not all system development branches. Thus, any change occurring on feature branches or release branches for each system is not included.

The highest number of systems changed in one month is 30 in 2016. This peak corresponds with the vertically aligned dot pattern we observed in figure 4.6.

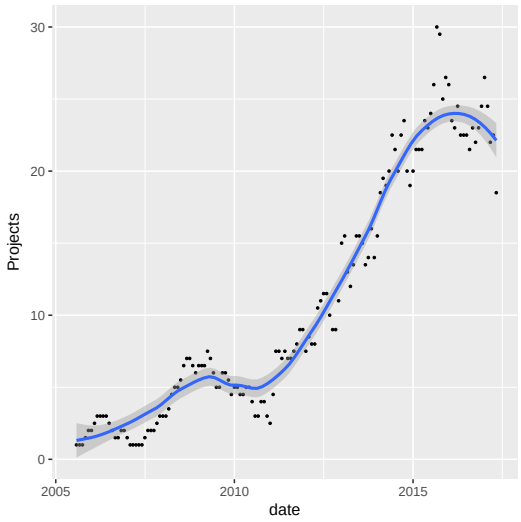


Figure 4.5: Average number of active projects increases over time

We hypothesized that some systems had always changed at the same time, thus would have been equally stable in time if a shared codebase was used instead of clone-and-own ($\mathbf{H_1}$). To test this hypothesis we use the technique we discussed in section 4.2.1. Based on the historical data of each system we determine the date periods in which they changed, and calculate for each pair of systems the fraction in which these periods overlap. The results of these measurements with week-granularity can be seen in table 4.3 and with month-granularity in table 4.4. Values equal or greater than 0.75 are highlighted with a star (*).

The change activity of all systems overlap to some extent. The highest degree of overlap is for systems P_9 and P_{13}, which is 83% by week and 91% by month. However, the degree of overlap for the other systems is much lower. Only the change activity of systems P_1, P_3, P_4, P_7 and P_8 overlaps for more than >74% by month. By week, this number ranges from 28% (P_4, P_8) to 58% (P_1, P_3), which is significantly less.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}
P_1	-	0.22 | 0.21	0.72 | 0.58	0.45 | 0.37	0.11 | 0.11	0.19 | 0.19	0.69 | 0.51	0.47 | 0.37	0.22 | 0.20	0.16 | 0.16	0.19 | 0.18	0.08 | 0.07	0.08 | 0.07	0.78*| 0.50	0.42 | 0.30
P_2	0.81*| 0.21	-	0.78*| 0.19	0.50 | 0.16	0.14 | 0.10	0.41 | 0.26	0.78*| 0.15	0.55 | 0.11	0.35 | 0.16	0.37 | 0.21	0.39 | 0.19	0.28 | 0.22	0.33 | 0.22	0.77*| 0.09	0.80*| 0.09
P_3	0.74 | 0.58	0.20 | 0.19	-	0.42 | 0.34	0.09 | 0.09	0.21 | 0.20	0.77*| 0.59	0.49 | 0.39	0.24 | 0.22	0.21 | 0.21	0.22 | 0.21	0.10 | 0.10	0.12 | 0.12	0.78*| 0.50	0.47 | 0.33
P_4	0.70 | 0.37	0.19 | 0.16	0.65 | 0.34	-	0.14 | 0.13	0.19 | 0.16	0.64 | 0.30	0.51 | 0.28	0.20 | 0.15	0.08 | 0.06	0.14 | 0.10	0.06 | 0.05	0.12 | 0.10	0.80*| 0.34	0.34 | 0.18
P_5	1.00*| 0.11	0.27 | 0.10	0.80*| 0.09	0.80*| 0.13	-	0.38 | 0.09	0.71 | 0.04	0.80*| 0.04	0.75*| 0.09	0.75*| 0.14	0.67 | 0.08	0.67 | 0.18	0.67 | 0.12	1.00*| 0.03	0.00 | 0.00
P_6	0.86*| 0.19	0.41 | 0.26	0.93*| 0.20	0.52 | 0.16	0.10 | 0.09	-	0.80*| 0.14	0.60 | 0.12	0.63 | 0.32	0.63 | 0.43	0.41 | 0.20	0.40 | 0.32	0.36 | 0.18	0.88*| 0.06	1.00*| 0.02
P_7	0.65 | 0.51	0.16 | 0.15	0.72 | 0.59	0.36 | 0.30	0.04 | 0.04	0.15 | 0.14	-	0.54 | 0.48	0.21 | 0.20	0.16 | 0.16	0.20 | 0.19	0.07 | 0.06	0.10 | 0.10	0.73 | 0.51	0.62 | 0.51
P_8	0.64 | 0.37	0.12 | 0.11	0.65 | 0.39	0.38 | 0.28	0.05 | 0.04	0.14 | 0.12	0.80*| 0.48	-	0.27 | 0.23	0.21 | 0.19	0.18 | 0.15	0.10 | 0.10	0.13 | 0.11	0.75*| 0.37	0.53 | 0.32
P_9	0.71 | 0.20	0.23 | 0.16	0.77*| 0.22	0.35 | 0.15	0.10 | 0.09	0.39 | 0.32	0.77*| 0.20	0.65 | 0.23	-	0.54 | 0.44	0.43 | 0.26	0.29 | 0.24	0.88*| 0.82*	0.85*| 0.10	0.50 | 0.07
P_{10}	0.76*| 0.16	0.33 | 0.21	1.00*| 0.21	0.19 | 0.06	0.14 | 0.14	0.57 | 0.43	0.86*| 0.16	0.71 | 0.19	0.71 | 0.44	-	0.47 | 0.24	0.41 | 0.35	0.54 | 0.33	1.00*| 0.08	1.00*| 0.05
P_{11}	0.72 | 0.18	0.28 | 0.19	0.84*| 0.21	0.28 | 0.10	0.08 | 0.08	0.28 | 0.20	0.84*| 0.19	0.48 | 0.15	0.40 | 0.26	0.32 | 0.24	-	0.26 | 0.22	0.29 | 0.20	0.89*| 0.15	1.00*| 0.07
P_{12}	0.70 | 0.07	0.50 | 0.22	0.90*| 0.10	0.30 | 0.05	0.20 | 0.18	0.60 | 0.32	0.70 | 0.06	0.70 | 0.10	0.60 | 0.24	0.70 | 0.35	0.60 | 0.22	-	0.50 | 0.21	1.00*| 0.07	0.50 | 0.02
P_{13}	0.47 | 0.07	0.40 | 0.22	0.73 | 0.12	0.40 | 0.10	0.13 | 0.12	0.27 | 0.18	0.67 | 0.10	0.53 | 0.11	0.93*| 0.82*	0.47 | 0.33	0.40 | 0.20	0.27 | 0.21	-	0.91*| 0.09	0.75*| 0.07
P_{14}	0.58 | 0.50	0.09 | 0.09	0.58 | 0.50	0.37 | 0.34	0.03 | 0.03	0.06 | 0.06	0.63 | 0.51	0.42 | 0.37	0.10 | 0.10	0.08 | 0.08	0.15 | 0.15	0.07 | 0.07	0.09 | 0.09	-	0.46 | 0.39
P_{15}	0.51 | 0.30	0.09 | 0.09	0.53 | 0.33	0.28 | 0.18	0.00 | 0.00	0.02 | 0.02	0.74 | 0.51	0.44 | 0.32	0.07 | 0.07	0.05 | 0.05	0.07 | 0.07	0.02 | 0.02	0.07 | 0.07	0.72 | 0.39	-

Table 4.3: Fraction of overlapping change activity (by Week). Fractions equal or greater than 0.75 indicated with a star (*).

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}
P_1	-	0.45 | 0.45	0.92*| 0.88*	0.82*| 0.78*	0.17 | 0.17	0.35 | 0.35	0.93*| 0.89*	0.86*| 0.82*	0.38 | 0.38	0.24 | 0.24	0.30 | 0.30	0.19 | 0.19	0.29 | 0.29	1.00*| 0.97*	0.77*| 0.77*
P_2	1.00*| 0.45	-	0.95*| 0.43	0.90*| 0.43	0.26 | 0.23	0.58 | 0.46	1.00*| 0.41	0.93*| 0.36	0.53 | 0.36	0.43 | 0.35	0.46 | 0.33	0.31 | 0.25	0.46 | 0.35	1.00*| 0.34	1.00*| 0.24
P_3	0.96*| 0.88*	0.44 | 0.43	-	0.83*| 0.78*	0.18 | 0.18	0.36 | 0.36	0.93*| 0.85*	0.88*| 0.81*	0.38 | 0.37	0.25 | 0.25	0.29 | 0.28	0.20 | 0.20	0.27 | 0.26	0.97*| 0.86*	0.67 | 0.58
P_4	0.95*| 0.78*	0.45 | 0.43	0.93*| 0.78*	-	0.20 | 0.20	0.31 | 0.28	0.92*| 0.74	0.86*| 0.70	0.31 | 0.27	0.16 | 0.14	0.27 | 0.24	0.20 | 0.19	0.21 | 0.18	0.97*| 0.78*	0.65 | 0.54
P_5	1.00*| 0.17	0.62 | 0.23	1.00*| 0.18	1.00*| 0.20	-	0.71 | 0.28	1.00*| 0.14	1.00*| 0.11	0.67 | 0.12	0.67 | 0.20	0.50 | 0.08	1.00*| 0.29	0.50 | 0.09	1.00*| 0.06	1.00*| 0.06
P_6	1.00*| 0.35	0.69 | 0.46	1.00*| 0.36	0.75*| 0.28	0.31 | 0.28	-	1.00*| 0.34	0.92*| 0.28	0.82*| 0.53	0.73 | 0.67	0.70 | 0.50	0.50 | 0.42	0.75*| 0.50	1.00*| 0.20	1.00*| 0.06
P_7	0.95*| 0.89*	0.41 | 0.41	0.91*| 0.85*	0.80*| 0.74	0.14 | 0.14	0.34 | 0.34	-	0.90*| 0.88*	0.39 | 0.39	0.24 | 0.24	0.31 | 0.31	0.19 | 0.19	0.29 | 0.29	0.97*| 0.89*	0.81*| 0.81*
P_8	0.95*| 0.82*	0.37 | 0.36	0.92*| 0.81*	0.79*| 0.70	0.11 | 0.11	0.29 | 0.28	0.97*| 0.88*	-	0.40 | 0.39	0.26 | 0.26	0.30 | 0.29	0.21 | 0.21	0.32 | 0.32	0.97*| 0.81*	0.74 | 0.64
P_9	1.00*| 0.38	0.53 | 0.36	0.93*| 0.37	0.67 | 0.27	0.13 | 0.12	0.60 | 0.53	1.00*| 0.39	0.93*| 0.39	-	0.64 | 0.64	0.46 | 0.33	0.38 | 0.33	0.91*| 0.91*	1.00*| 0.29	1.00*| 0.29
P_{10}	1.00*| 0.24	0.67 | 0.35	1.00*| 0.25	0.56 | 0.14	0.22 | 0.20	0.89*| 0.67	1.00*| 0.24	1.00*| 0.26	1.00*| 0.64	-	0.75*| 0.46	0.62 | 0.50	1.00*| 0.60	1.00*| 0.14	1.00*| 0.12
P_{11}	1.00*| 0.30	0.55 | 0.33	0.91*| 0.28	0.73 | 0.24	0.09 | 0.08	0.64 | 0.50	1.00*| 0.31	0.91*| 0.29	0.55 | 0.33	0.55 | 0.46	-	0.45 | 0.38	0.44 | 0.27	1.00*| 0.23	1.00*| 0.12
P_{12}	1.00*| 0.19	0.57 | 0.25	1.00*| 0.20	0.86*| 0.19	0.29 | 0.29	0.71 | 0.42	1.00*| 0.19	1.00*| 0.21	0.71 | 0.33	0.71 | 0.50	0.71 | 0.38	-	0.60 | 0.25	1.00*| 0.14	1.00*| 0.12
P_{13}	1.00*| 0.29	0.60 | 0.35	0.90*| 0.26	0.60 | 0.18	0.10 | 0.09	0.60 | 0.50	1.00*| 0.29	1.00*| 0.32	1.00*| 0.91*	0.60 | 0.60	0.40 | 0.27	0.30 | 0.25	-	1.00*| 0.26	1.00*| 0.24
P_{14}	0.97*| 0.97*	0.34 | 0.34	0.89*| 0.86*	0.80*| 0.78*	0.06 | 0.06	0.20 | 0.20	0.91*| 0.89*	0.83*| 0.81*	0.29 | 0.29	0.14 | 0.14	0.23 | 0.23	0.14 | 0.14	0.26 | 0.26	-	0.74 | 0.74
P_{15}	1.00*| 0.77*	0.24 | 0.24	0.82*| 0.58	0.76*| 0.54	0.06 | 0.06	0.06 | 0.06	1.00*| 0.81*	0.82*| 0.64	0.29 | 0.29	0.12 | 0.12	0.12 | 0.12	0.12 | 0.12	0.24 | 0.24	1.00*| 0.74	-

Table 4.4: Fraction of overlapping change activity (by Month). Fractions equal or greater than 0.75 indicated with a star (*).

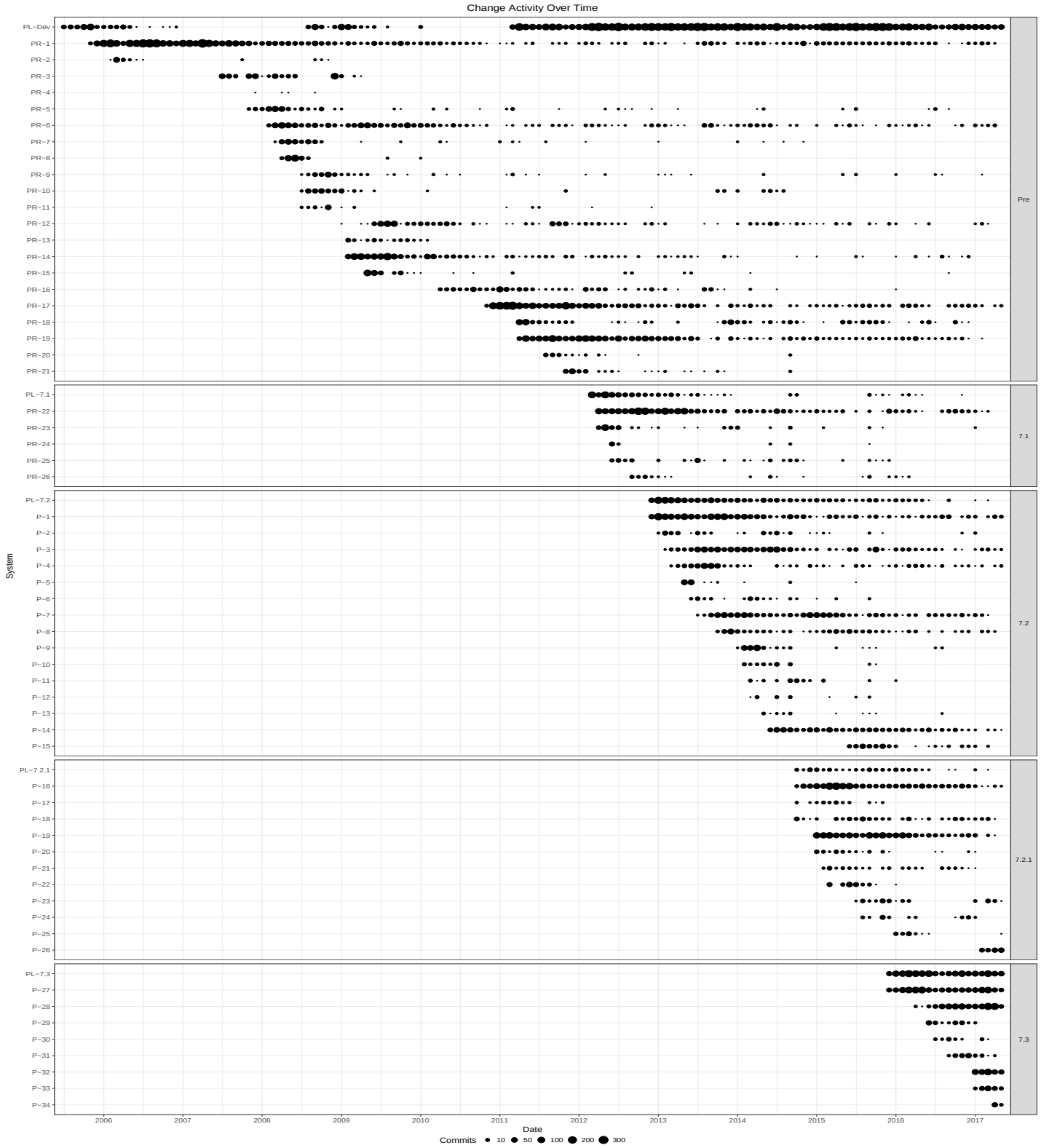


Figure 4.6: Visualization of system change frequency

4.3.2 Independence in Space

The change distribution for the 7.2 platform and all systems derived from this version can be seen in figure 4.7. Overall, we see that the changes are relatively spread out over the codebase. We do however see some differences in the density of the changes, and observe some patterns.

Visually, the change distribution of P_6, P_9, P_{10} and P_{11} appear to be relatively similar for both merge and non-merge commits. However, table 4.5 shows that the similarity between P_6, P_9 and P_{10} is approximately 25%. The similarity between P_{10} and P_{11} is 77%, which is the highest of all systems. Surprisingly, the similarity between P_2 and P_3 is 63%, which is the second-highest, even though the visual change distributions of these systems does not appear to be similar.

	n	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{13}	P_{15}
n	-	2400	1537	2036	937	243	920	1012	653	377	538	609	26	214
P_1	2400	-	0.54 | 0.49	0.57 | 0.44	0.32 | 0.30	0.10 | 0.10	0.29 | 0.27	0.33 | 0.31	0.25 | 0.25	0.10 | 0.10	0.14 | 0.13	0.17 | 0.15	0.01 | 0.01	0.07 | 0.06
P_2	1537	0.85*| 0.49	-	0.91*| 0.65	0.49 | 0.44	0.15 | 0.15	0.47 | 0.41	0.41 | 0.32	0.36 | 0.34	0.18 | 0.17	0.35 | 0.35	0.37 | 0.36	0.01 | 0.01	0.07 | 0.07
P_3	2036	0.67 | 0.44	0.69 | 0.65	-	0.38 | 0.36	0.11 | 0.11	0.36 | 0.33	0.34 | 0.29	0.29 | 0.28	0.15 | 0.14	0.26 | 0.26	0.30 | 0.29	0.01 | 0.01	0.06 | 0.06
P_4	937	0.83*| 0.30	0.80*| 0.44	0.84*| 0.36	-	0.24 | 0.24	0.40 | 0.25	0.38 | 0.22	0.29 | 0.20	0.14 | 0.11	0.15 | 0.11	0.18 | 0.12	0.01 | 0.01	0.09 | 0.08
P_5	243	0.97*| 0.10	0.94*| 0.15	0.96*| 0.11	0.94*| 0.24	-	0.44 | 0.10	0.48 | 0.10	0.52 | 0.17	0.27 | 0.12	0.25 | 0.08	0.28 | 0.09	0.03 | 0.03	0.18 | 0.10
P_6	920	0.77*| 0.27	0.78*| 0.41	0.80*| 0.33	0.41 | 0.25	0.12 | 0.10	-	0.60 | 0.40	0.47 | 0.37	0.29 | 0.26	0.31 | 0.25	0.33 | 0.25	0.02 | 0.02	0.08 | 0.07
P_7	1012	0.79*| 0.31	0.62 | 0.32	0.67 | 0.29	0.35 | 0.22	0.12 | 0.10	0.55 | 0.40	-	0.42 | 0.34	0.18 | 0.15	0.18 | 0.14	0.21 | 0.15	0.02 | 0.02	0.20 | 0.19
P_8	653	0.93*| 0.25	0.85*| 0.34	0.90*| 0.28	0.41 | 0.20	0.19 | 0.17	0.66 | 0.37	0.65 | 0.34	-	0.29 | 0.22	0.41 | 0.29	0.48 | 0.33	0.02 | 0.02	0.13 | 0.11
P_9	377	0.67 | 0.10	0.73 | 0.17	0.79*| 0.14	0.35 | 0.11	0.17 | 0.12	0.71 | 0.26	0.48 | 0.15	0.50 | 0.22	-	0.50 | 0.26	0.51 | 0.24	0.07 | 0.07	0.10 | 0.07
P_{10}	538	0.65 | 0.13	0.99*| 0.35	0.99*| 0.26	0.26 | 0.11	0.11 | 0.08	0.53 | 0.25	0.35 | 0.14	0.50 | 0.29	0.35 | 0.26	-	0.97*| 0.83*	0.01 | 0.01	0.06 | 0.05
P_{11}	609	0.65 | 0.15	0.93*| 0.36	0.99*| 0.29	0.27 | 0.12	0.11 | 0.09	0.50 | 0.25	0.34 | 0.15	0.51 | 0.33	0.31 | 0.24	0.86*| 0.83*	-	0.02 | 0.02	0.06 | 0.05
P_{13}	26	0.62 | 0.01	0.58 | 0.01	0.62 | 0.01	0.50 | 0.01	0.31 | 0.03	0.62 | 0.02	0.62 | 0.02	0.62 | 0.02	1.00*| 0.07	0.31 | 0.01	0.46 | 0.02	-	0.27 | 0.03
P_{15}	214	0.73 | 0.06	0.50 | 0.07	0.61 | 0.06	0.39 | 0.08	0.20 | 0.10	0.33 | 0.07	0.93*| 0.19	0.41 | 0.11	0.18 | 0.07	0.15 | 0.05	0.18 | 0.05	0.03 | 0.03	-

Table 4.5: % Overlapping Change Distribution (.java)

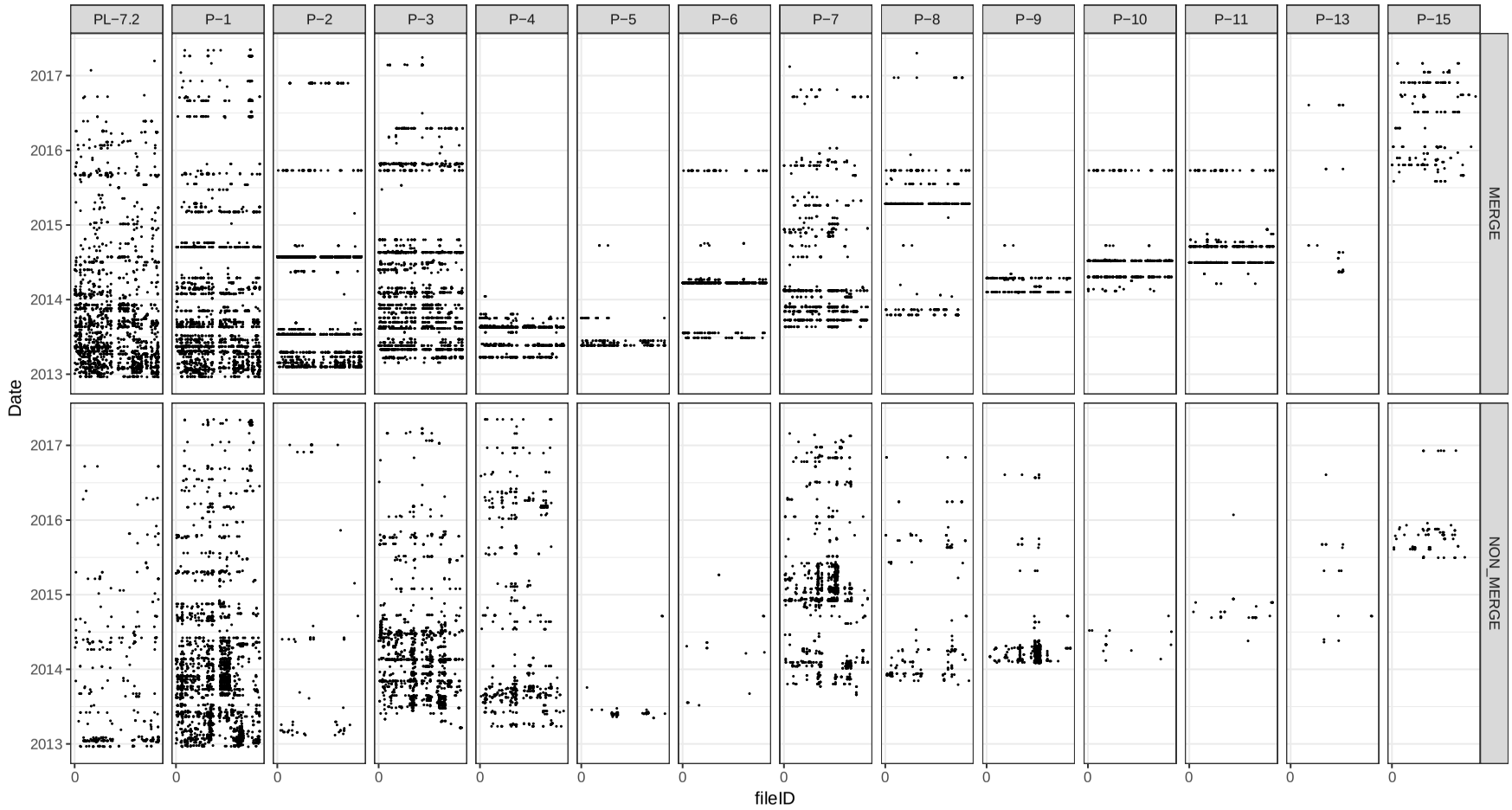


Figure 4.7: Change distribution of the 7.2 platform and 7.2 systems.

4.4 Discussion

Here we interpret the results presented in Section 4.3. We first discuss our overall findings and then answer the sub-questions that led our exploration.

Independence in Time

Overall, we observed in figure 4.6 that changes to MES-Toolbox systems are performed continuously, periodically, and at seemingly arbitrary intervals. Some systems are changed continuously for years showing almost no periods inactivity, while others are only changed frequently during the first few months.

The change frequency of MES-Toolbox systems appears to be mostly determined by customer requirements. This is consistent with observations by Lettner et al. [21], who reported the same for the Keba SECO, an industrial automation software ecosystem. Most changes to the Keba system reportedly happen within the first three to four weeks in a customer project. In our case, many systems were most active in the first period of the project, but this period was much longer (2-4 months).

Furthermore, Lettner et al. [21] stressed the importance of platform quality characteristics like stability and backwards compatibility, and long-term platform evolution in the domain of industrial automation. They stated that *"application engineer B reported that he had to update a ten-year-old version of the platform software, because an important customer had decided to leave out several platform releases and then requested a new feature. This led to significant difficulties in merging the old software version with the new functionality."*

Our results confirm the importance of these characteristics in the domain of industrial automation. The oldest system we analyzed was 11 years old, and still continuously changed (see fig 4.6, system P_1). Some systems were inactive for years before becoming active again due to new customer demands. This is different from other types of systems, like Marlin, an open source firmware for 3D printers. Stanciulescu, Schulze, and Wąsowski [30] found forks in the Marlin ecosystem to be characterized by a short maintenance lifetime (101 days on average).

Do MES-Toolbox systems change in parallel? In figure 4.5 we observed that many MES-Toolbox systems change in parallel, and that that the number of systems changed in the same month has significantly increased over time. This number has grown from only five systems in 2011, to at most 30 systems in 2016.

H_1: *Some MES-Toolbox systems always changed at the same time, and would have been equally stable in time if a shared codebase was used.*

We calculated the degree of parallel change using two time intervals; by week and by month. The change activity of all systems overlapped to some extent, but did not overlap completely. The change activity by week overlapped at most 82% for system P_9 and P_{13}, and by month 97% for systems P_1 and P_{14} Therefore, we reject hypothesis $\mathbf{H_1}$.

> **H_1**: **Rejected.** There were no MES-Toolbox systems that would have been equally stable in time if they were to be developed using a shared codebase.

However, this does not prove that the systems could not have been developed using a shared codebase. We assume that change activity of systems would be exactly the same if this would have been the case, which is not necessarily true in reality. For example, systems P_9 and P_{13} were built for the same customer. All changes for P_9 were propagated to P_{13} and vice versa. This resulted in a highly similar change activity (82%), but not identical. Using our technique we conclude that a shared codebase would affect stability in time, while in practice these systems *could* have been combined. This suggests that the degree of parallel change can be considered as merely an indicator, as the threshold does not necessarily have to be 100%.

Independence in Space

To explore the change distribution of files in MES-Toolbox systems, we adapted a visualization originally proposed by Van Rysselberghe and Demeyer [33] to discover changes that are relevant for software re-engineering activities. We did not use the visualization to discover relevant changes, but as a way to discover variation between different systems and over time. The scale in which we used it (5500 files, 4 years, faceted by system and merge/non-merge changes) caused too much overplotting to be able to accurately identify specific changes.

Furthermore, the authors noted that committing a large number of files around the same time, will cause a horizontal line pattern. In the context of clone-and-own, we observed that this pattern is mainly caused by periodical propagation of changes from a release to systems. When systems are periodically synchronized, a large number of revisions, thus large number of files, are often merged to the system as a single change-set.

Do MES-Toolbox systems have a similar file change distribution? To determine whether Mes-Toolbox systems benefited from independence in space, we explored the similarity in file change distribution of MES-Toolbox systems. We hypothesized that some systems would have an identical file change distribution, and would have been equally stable in space if they were developed using a shared codebase.

H_2: *Some MES-Toolbox systems have an identical file change distribution, thus would have been equally stable in space if a shared codebase was used.*

We found that the highest degree of similarity was 77% for system P_{10} and P_{11}. All other systems scored significantly lower, Therefore, we reject hypothesis $\mathbf{H_2}$.

> **H_2**: **Rejected.** There were no systems with an identical file change distributions

Krinke [18] and Hotta et al. [15] analyzed the evolution of open source software systems, and found that cloned code is generally more stable than non-cloned code. This was later confirmed by Harder and Göde [12]. Therefore, they concluded that maintenance of cloned code is not necessarily more expensive than the maintenance of non-cloned code. Our findings suggest that this can also be the case for cloning in-the-large. We observed that individually cloned systems were more stable in time and in space, compared to if they were to be combined into a single system.

Furthermore, a reason why systems have not changed in parallel is because application engineers only work on a limited number of systems at the same time. Working on multiple systems requires the application engineer to switch between development environments. A (partially) shared codebase reduces the effort required to make changes for different systems, thus potentially resulting in more parallel development.

4.5 Threats to Validity

We performed a literature review to understand what aspects of clone-and-own are considered beneficial. Based on what we found, we defined the research questions in section 4.1. We limited the scope of our analysis to what we defined as independence in time, and independence in space. While these two aspects of clone-and-own might not cover the whole spectrum of possible benefits, we consider them to be fundamental to clone-and-own.

In this study we use historical data from the version control system to determine the change activity of systems. Changes present in the version control system are triggered by the user, therefore it is possible that systems change more frequent or at different times than what the history suggests. However, based on the frequency at which systems change, we conclude that the history accurately represents the real change activity of the systems. For calculating the overlapping proportion of change activity, we further mitigate this issue by using relatively course grained periods of time (week and month).

4.6 Conclusion

The main question for this chapter is: *Have any MES-Toolbox systems benefited from the independence provided by clone-and-own?*. We studied the independence provided by clone-and-own from the perspective of independence in space, and independence in time. To get insight into these two dimensions, we explored the change frequency of systems and change distribution of files across systems. In summary, we conclude that all MES-Toolbox systems have benefited from independence provided by clone-and-own to some extent.

Overall, we found that MES-Toolbox systems can change continuously, periodically and at seemingly arbitrary moments in time. We hypothesized that some systems would have an identical change frequency, and similar change distribution; and therefore could have been developed using a shared codebase without negatively affecting stability in time or in space. We did not find such a system. The highest similarity in change activity was 82% by week, and 97% by month. The highest similarity for change distribution was 77%.

Chapter 5

Analysis of Clone-and-Own Drawbacks

The main question we answer in this chapter is: *To what extent has clone-and-own caused maintenance overhead for MES-Toolbox systems?*. In the following section, section 5.1, we identify for each sub-question two hypotheses. In section 5.2 we explain the techniques we use to explore the drawback related characteristics. In section 5.3 we report our results, which we interpret in section 5.4. Finally, we conclude in section 5.6.

5.1 Research Questions

To determine to what extent clone-and-own has caused maintenance overhead, we focus on two possible causes of maintenance overhead: Decentralization of Information, and Repetitive Tasks.

Decentralization of Information

When systems are developed using a shared codebase, all modifications and extensions are contained a single codebase. A clone-and-own development approach causes these modifications and extensions to be spread out over multiple places in the repository. As the product family grows it becomes increasingly hard to keep an overview of the available functionality [30, 3, 9]. We define disadvantage due to decentralization of information as follows:

Axiom 5.1: Drawback: Decentralization of Information

Cloned systems suffer from decentralization of information if systems diverge from their origin.

How much do MES-Toolbox systems diverge from their origin? Clone-and-own allows developers to add, remove or modify files without affecting their origin. These changes will inherently cause systems to diverge from their origins; they are no longer identical. We consider this to be a form of decentralization of information.

If systems do not diverge much from their origin, then decentralization of information likely not causing significant maintenance overhead. Some developers stated that diverged Java files often make it difficult to propagate changes, but expected that the Java codebase would not significantly diverge for most of the 7.2 and 7.2.1 systems. Many of these systems are considered relatively simple, and hardly require any customer-specific modification of the codebase.

H_3: *The Java codebase for most version 7.2 and 7.2.1 systems does not diverge significantly (>10%) from their origin.*

A metric commonly used to quantify the complexity caused by cloning is the *number of variants* of each file. For example, Hetrick, Krueger, and Moore [14] illustrate the complexity caused by

branching by using a *file branch factor* metric, which is defined as the average number of branched files per product, normalized by the number of products. Based on initial experiments, we hypothesize that divergence measured on the file-level granularity might not accurately represent the complexity caused by cloning. For example, this metric will not be able to distinguish divergence due to trailing whitespace at the end of a file, and divergence due to rewriting the file.

H_4: *Divergence measured at file-level granularity can lead to different conclusions compared to divergence measured at line-level granularity.*

Repetitive Tasks

When systems are developed using a shared codebase, any change will affect all systems. When cloning is used, the codebases are separated and changes to one system may have to be *repeated* on the codebase of other systems. This can be considered as a form of redundant work, and we refer to this as *repetitive tasks*. We define this disadvantage due to repetitive tasks as follows:

Axiom 5.2: Drawback: Repetitive Tasks

Cloned systems suffer from repetitive tasks if changes are frequently propagated to systems.

Have all MES-Toolbox systems caused repetitive task maintenance overhead? Cloning is said to increase maintenance overhead because changes have to be propagated to all clones. Studies have shown however that change propagation is not always performed [30], which suggests that cloning does not necessarily increase maintenance overhead due to change propagation.

In the organization we study, changes are manually propagated at the discretion of teams developing the systems. Application engineers stated that they periodically merge new changes to systems, but only while they are still under active development. Because some systems are developed relatively fast, we expect that some systems retrieve only very few changes from their origin, thus not causing much maintenance overhead. Consequently, techniques that purely reduce repetitive task would have limited effect on maintenance overhead caused by these systems.

H_5: *Some MES-Toolbox systems never synchronized with their origin, causing no repetitive task maintenance overhead.*

A technique to reduce repetitive task maintenance overhead, is to extract artifacts that are common to many systems into a shared library. This reduces the effort required to propagate changes, as changes do not have to be merged to each individual system. For this technique to reduce maintenance overhead, common artifacts that cause repetitive tasks have to be identified.

The initial architect of the MES-Toolbox system recognized that separating commonality and customer-specific artifacts can potentially reduce maintenance overhead. He stated that "Ideally, we distribute the common part of each platform release in the form of binaries, and the customer-specific part as source files.". Therefore, the design of the system aims to separate *common* and *customer* functionality, where *common* is supposed to represent commonality between all systems derived from the same platform release.

If common artifacts truly are common to many systems, then we expect these artifacts to be merged significantly more frequent than customer-specific artifacts. However, as the product family grows, artifacts once developed as customer-specific are re-used and become a commonality between systems. Subsequent changes to these customer artifacts will then cause repetitive task maintenance overhead, even though the artifacts might not be common to all systems. Thus, frequent merging of customer-specific artifacts could indicate that the artifacts are either incorrectly classified as customer, or (parts of) these artifacts have become common to some extent.

To study whether this is the case for the MES-Toolbox product family, we hypothesize that customer artifacts cause a significant portion of the repetitive task maintenance overhead. If we find this to be true, the implications can be significant, as it implies that a file-level common – customer separation may be insufficient to fully address repetitive task maintenance overhead.

H_6: *A significant portion of repetitive task maintenance overhead is caused by customer artifacts.*

5.2 Research Method

In this section we discuss the techniques we use to explore the hypothesis defined in the previous section (5.1). First, we explain how we measure, visualize, quantify and interpret *divergence*, and then *repetitive tasks*.

5.2.1 Decentralization of Information

In this study we consider *Decentralization of Information* in the form of *divergence*.

Measuring Divergence

For hypothesis $\mathbf{H_3}$ we measure how much systems have diverged from their origin. We measure divergence over time by calculating the differences between systems and their origins, for each file, at every revision that changed either the system or its origin.

We hypothesize ($\mathbf{H_4}$) that divergence measured at file-level granularity can be very different from divergence measured at line-level granularity. Therefore, we measure differences at line-level granularity (number of lines different) with the Java implementation of GNU diff [1]. We define the difference in number of lines as `diff`. During analysis we keep track of how much the difference has increased or decreased, the `diffDelta`. Based on the differences in number of lines, we determine whether the difference in number of files has increased or decreased. The difference in number of files increases ($+\mathbf{1}$) if files go from equal to not-equal (`diffDelta > 0 && diff == diffDelta`), decreases ($-\mathbf{1}$) if files go from not-equal to equal (`diffDelta < 0 && diff == 0`), and otherwise stays the same ($\mathbf{0}$).

We illustrate the divergence calculation on an artificial example in table 5.1. In this example, `PL-7.2.1` is the origin of system P_{17}.

revision	system	file	diffDelta	diff	fileDiffDelta
1	PL-7.2.1	Main.java	5	5	1
2	P_{17}	Main.java	-5	0	-1
3	P_{17}	Main.java	10	10	1
4	P_{17}	Main.java	5	15	0
5	PL-7.2.1	Main.java	-15	0	1

Table 5.1: Example data of divergence over time calculation.

Interpretation In table 5.1 we see that in revision 1, `Main.java` was modified on `PL-7.2.1`, causing a difference of five lines. This file was then modified three times on P_{17}, first reducing the difference by five lines in revision 2, increasing the difference by ten lines in revision 3, and five lines in revision 4. Finally, in revision 5 the difference was reduced by fifteen lines by a change on `PL-7.2.1`.

5.2.2 Repetitive Tasks

For hypothesis $\mathbf{H_5}$ we are interested in the extent to which systems synchronize with their origin. Systems retrieving changes from their origin, or contributing changes to their origin can be seen as a form of *synchronization*.

Detecting Synchronization

Subversion automatically registers the merged revision(s), and the origin of the merge in a so-called `svn:mergeinfo` property attached to files and directories [2]. Changes to this property can be detected by scanning the output of `svn diff` for an occurrence of `svn:mergeinfo`. If this is the case, we classify the commit as `MERGE`, and `NON_MERGE` otherwise.

[1] http://www.bmsi.com/java/#diff
[2] http://svnbook.red-bean.com/en/1.7/svn.branchmerge.basicmerging.html

Unfortunately, we cannot blindly trust the validity of Subversion properties. Subversion properties can be changed by hand, developers might forget to commit the changes to properties, or they could manually copy changes between systems without using the merging system. We aim to mitigate these issues by taking into account whether revisions have caused convergence or divergence. We expect that most changes to systems will cause them to diverge from their origin and that merging these changes to their origin will cause them to converge. Similarly, we expect that changes to the origin of systems will cause them to diverge, and merging the change to the systems will cause them to converge. We manually validate a large sample of data to ensure this is a reliable technique to detect synchronization.

In the combined history of PL-7.2.1 and PL-7.2, 501 revisions caused at least one system to converge one line. Out of 501 revisions that caused convergence, 372 revisions (74%) were correctly classified as changes contributed by the converging system(s). Out of these 372 revisions, 17 revisions did not have merge-info.

To detect whether systems retrieved changes from their origin, we identify for each system, all revisions that have merge-info, and caused at least one Java file to converge with the origin of the system. We manually validated this for system P_4. The change history of system P_4 contained 18 revisions with merge-info, of which 14 caused convergence. Out of these 14 revisions, 12 were merged from the origin of system P_4, one revision was accidentally merged from another system, and in one revision this revision was reverted.

These results confirm that merely looking at merge-info can be unreliable, and suggest that our technique is suitable to detect synchronizing changes.

Change Distribution View

To investigate hypothesis **H_6**, we are interested in the similarity of files changed in each system. To gain insight in this property, we again use a variant of the 2-dimensional view proposed by Van Rysselberghe and Demeyer [33]. In this view, as shown in figure 5.1, each change to a file is represented as a dot. The horizontal axis represents the file that is changed, and the vertical axis indicates the time of the change. Files have been sorted in alphabetical order, which causes files in the same directory to be close to each other on the horizontal axis. Each file has been given a unique identifier (fileID). We color the dots based on whether they are in a customer or common part of the codebase.

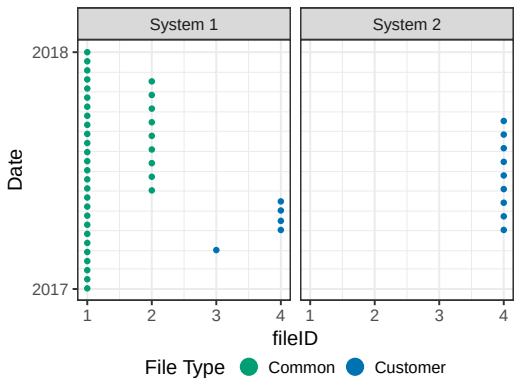


Figure 5.1: Example visualization for change distribution

Interpretation Figure 5.1 shows that files 1 and 2 are customer files, and files 3 and 4 are common files. In this case, files 1 and 2 appear to be merged more frequent than files 3 and 4. This would be consistent with our hypothesis, as we expect that a significant portion of repetitive tasks maintenance overhead is caused by customer files.

5.3 Results

In this section we present the results of the methods discussed in section 5.2.

5.3.1 Decentralization of Information

The visualization of divergence over time can be seen in figure 5.2. It may be seen clearly that while divergence tends to increase over time, there is a significant variance both in the degree of divergence and rate of divergence. In the first year of the history of systems P_1, P_2, P_3, and P_7, the proportion of diverged Java files appears to be highly volatile. This can also be seen in divergence in number of lines, but is less clear.

In terms of percentage of Java files, all systems at some point in time diverged between 7% and 22.5% from their origin. This suggests that all systems, even those that do not frequently change, can diverge significantly. In terms of diverged number lines, most systems did not exceed 50.000 lines (<5%), and only two systems diverged more than 75.000 lines.

The spike in lines diverged for system P_6 clearly shows that divergence measured in percentage of Java files can be significantly different from divergence measured in terms of number of lines. Divergence at file-granularity of systems P_5 and P_6 appear to evolve almost identically, while divergence in number of lines does not. The activities that led to this anomaly occurred as follows:

- On 2014/03/24 a feature branch had to be merged to system P_6. However, merging of the feature branch caused a significant number of conflicts, as the system had not retrieved any changes from its origin since 2013/07/22 (see figure 5.4). The system first had to retrieve all changes from its origin.
- Some changes could not be merged due to conflicting changes in the *bulkloading* module.
- First, non-conflicting changes were merged to system P_6, reducing file divergence from 487 to 288 (-199) and line divergence from 23.305 to 12.776 (-10.529).
- Next, the remaining (conflicting) changes were merged, and the decision was made to delete the *bulkloading* module from system P_6. The system did not use this module, thus no effort was spent in resolving the conflicts. This change caused file divergence to increase from 288 to 397 (+109), and line divergence from 12.776 to 126.753 (+113.977)
- Finally, the feature branch was merged to the system. Increasing file divergence from 397 to 480 (+83), and line divergence from 126.753 to 129.854 (+3.101)

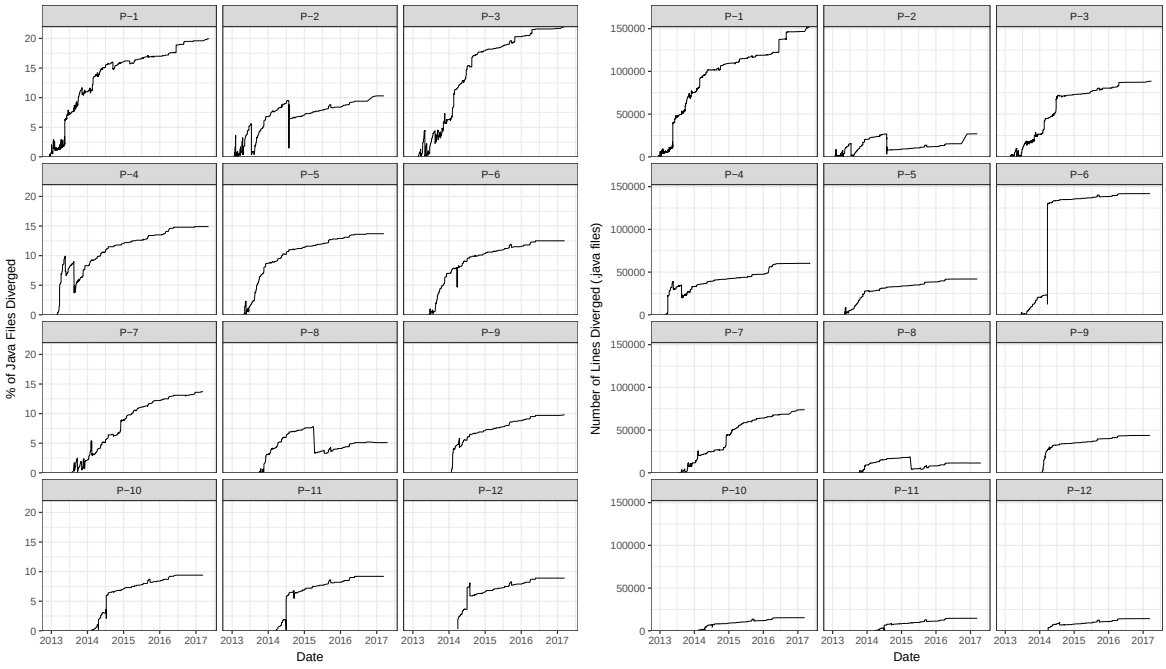
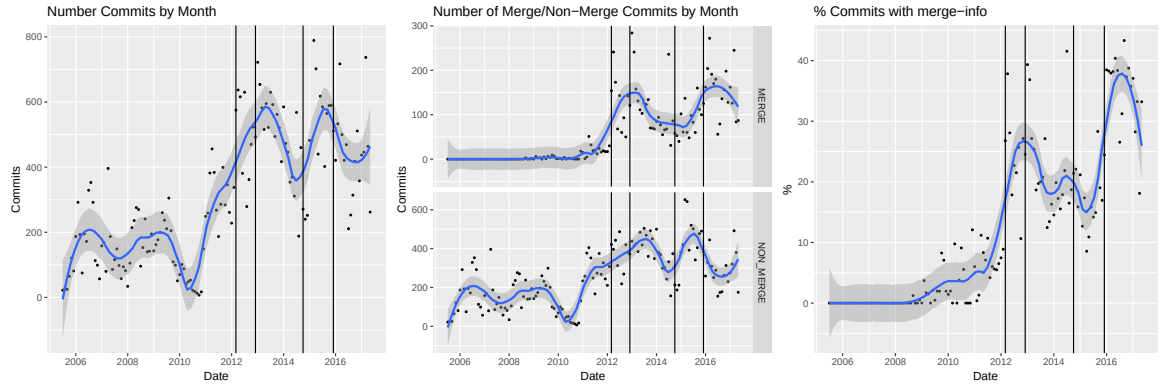


Figure 5.2: Rate of divergence

5.3.2 Repetitive Tasks



(a) All commits (b) Merge and non-merge commits (c) % merge commits

Figure 5.3: Total number of merge and non-merge commits over time.

In figure 5.3 we see the distribution of merge and non-merge commits over time. We have highlighted platform releases with vertical black lines. The lines represent releases 7.1, 7.2, 7.2.1 and 7.3. We observe in figure 5.3c that the proportion of changes with merge-info has increased from 0% in 2008 to at most 43% in 2016. The lack of changes with merge-info before 2009 can be explained by the fact that Subversion introduced merge tracking in version 1.5, released 19 June 2008[3]. Before this time, no merge-info was recorded.

The increase from 20% in 2015 to 40% in 2016 appears to be caused by the release of platform version 7.3. This release was actively changed after it was released. All changes to this branch were propagated to the platform development branch, meaning that for every non-merge commit on the 7.3 platform release, there was a merge commit on the platform development branch.

[3] https://subversion.apache.org/docs/release-notes/release-history.html

Synchronization with Origin

Figure 5.4: Synchronizing Changes

From Figure 5.4 we observe that all systems retrieve changes from their origin, but some do so significantly more frequent than others. The activity pattern for systems P_1 and P_3 is more constant compared to other systems. This is confirmed by the data in table 5.2. These systems retrieved changes from their origin respectively 202 and 89 times.

Furthermore, we observe at least two instances of vertically aligned dots. These patterns can be caused by multiple systems retrieving changes from their origin roughly at the same time. Manual inspection of these patterns shows that both instances were critical bug fixes, manually merged to most systems on the same day. The fact that we do not see many of these vertical line patterns suggests that mass-synchronization of many systems at once does not happen often.

We see that the period between subsequent synchronizations can be significant. For example, system P_6 retrieved changes from its origin on 22 July 2013, and 8 months later on 24 March 2014.

Synchronization	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{13}	P_{15}	P_{16}	P_{17}	P_{18}	P_{20}	P_{21}	P_{23}	P_{25}	P_{26}
from Origin	202	55	89	15	5	19	47	20	6	11	12	4	14	52	10	14	11	9	12	1	1
to Origin	193	8	32	26	0	4	12	5	0	3	3	0	0	1	0	4	1	3	0	1	0

Table 5.2: Number of Synchronizing Commits

Merge Change Distribution

The change distribution for the 7.2 platform and all systems derived from this version can be seen in figure 5.5. Every green dot represents a change to a *common* file, and every blue to a change to a *customer* file. Visually, the density of merge changes to *common* files appears to be higher compared to the density of merge changes to *customer* files. Table 5.3 list for each system the number of merge changes directed at common and customer artifacts. This data confirms that the majority of merge changes are directed at common artifacts.

Nonetheless, we see that a significant portion of merge changes is directed at customer artifacts. The proportion of merge changes directed at customer Java artifacts is 21%. However, we see that PL-7.2, P_1, P_2 and P_3 are responsible for 65% (10774) of all merge changes. If we exclude these outliers from the dataset, the proportion of merge changes directed at customer Java artifacts is slightly higher (24%).

	PL-7.2	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{13}	P_{15}	Total
COMMON	2051	2589	2025	1960	758	193	818	714	482	181	498	575	15	107	12966
CUSTOMER	556	548	514	531	234	50	277	243	130	56	111	127	12	110	3499
Total	2607	3137	2539	2491	992	243	1095	957	612	237	609	702	27	217	16465

Table 5.3: 21% of all merge changes are directed at Java customer artifacts

Table 5.4 lists the 10 most merged files. Six files are considered as *customer* artifacts. The top 3 most merged files are `IntakeModel.java`, `TripModel.java` and `IngredientResolver.java`, which have been merged respectively 101, 86 and 66 times. Interestingly, out of ten files end with `*Model.java`. The *model* classes implement the behavior of similarly named *view* classes. For each user interface element defined in the *view* there is a corresponding `ModelAdapter` that implements its behavior in the *model*. In the latest version of the 7.2 platform release, the file `IntakeModel.java` contains at least 161[4] ModelAdapters.

relativePath	customerCommon	Commits
modules/intake/klant/src/gti/ui/main/IntakeModel.java	CUSTOMER	101
modules/bulkloading/klant/src/gti/ui/main/TripModel.java	CUSTOMER	86
common/src/gti/batch/s88/recipe/resolve/IngredientResolver.java	COMMON	66
klant/src/gti/ui/main/ProductionOrderModel.java	CUSTOMER	64
klant/src/gti/domain/util/CRecipeUtil.java	CUSTOMER	63
common/src/gti/batch/s88/recipe/RecipeExecutor.java	COMMON	62
klant/src/gti/ui/main/RecipeModel.java	CUSTOMER	56
modules/visualization/common/src/gti/ui/main/VisualizationEditorView.java	COMMON	56
common/src/gti/batch/pfc/wizard/BatchProcedureWizardModel.java	COMMON	53
common/src/gti/batch/s88/phase/DefaultPhaseAction.java	COMMON	53

Table 5.4: Top #10 most merged Java files.

[4] At revision 74632, counted the number of occurrences of `'extends .*ModelAdapter'`

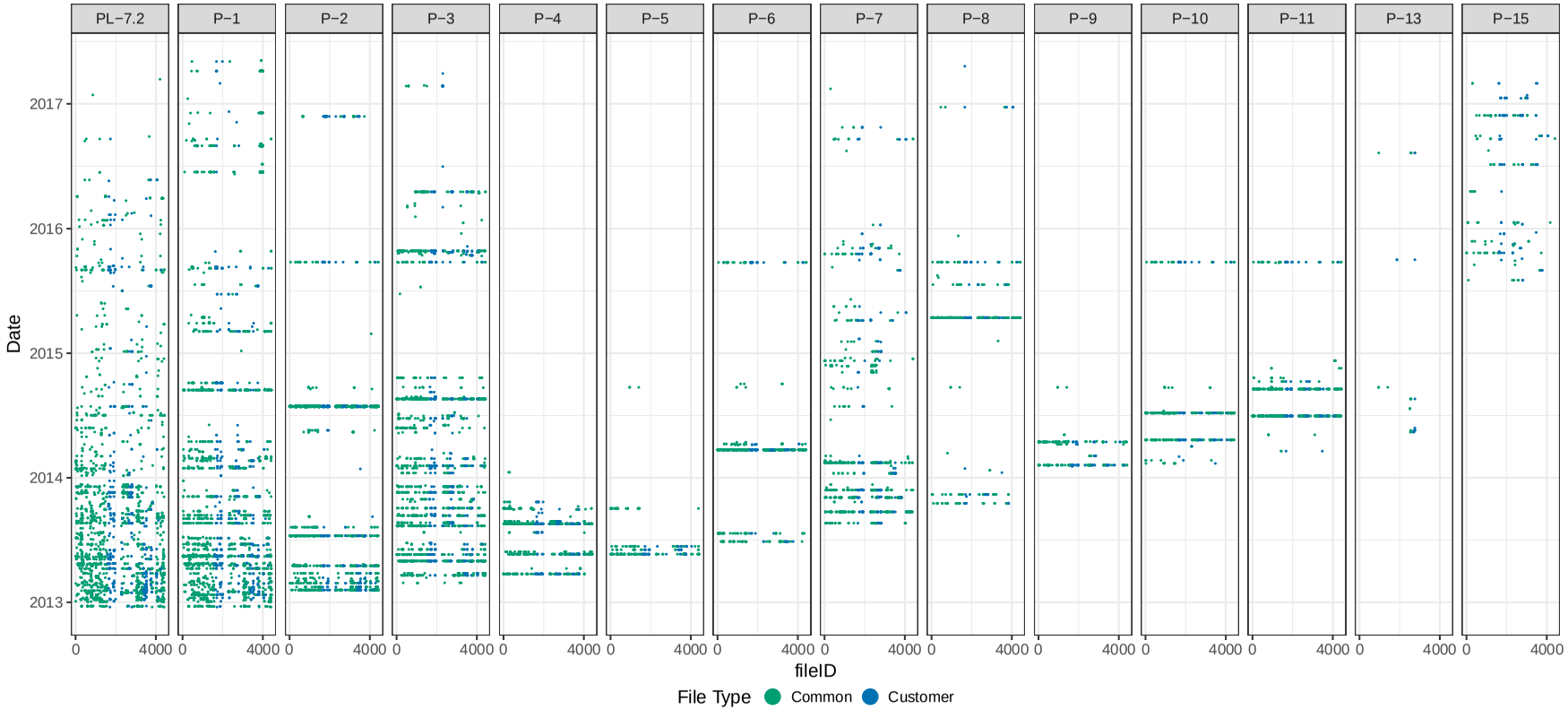


Figure 5.5: Merge Change distribution of Java files, 7.2 platform and 7.2 systems.

Detailed Change Analysis

We manually selected a merge change to `IntakeModel.java` and used the multi-system differencing functionality (see Chapter 3) of our modified version of JMeld to analyze this change in more detail. Figure 5.6 illustrates the merge change we focused on. We see that the method `setValueAt` of `Intake_Article_Code_ModelAdapter` is changed to correct its behavior. Table 5.5 lists the activities related to this bug fix in chronological order.

Revision	Date	Action
	18 February 2013	The incorrect behavior of the article code field was reported for system PR_{22} (issue #6950).
r44727	19 February 2013	The issue was resolved by changing the `ModelAdapter` of the article code field in system PR_{22}.
r52497	3 December 2013	The change (r44727) was merged from system PR_{22} to system P_1.
r52500	3 December 2013	The change (r52497) was merged from system P_1 to platform PL-7.2. At this time, there were eight version 7.2 systems, of which one already had the change (P_1).
r52641	6 December 2013	The change was merged to system P_3.
r55398	24 March 2014	The change was merged to system P_6.
r59534	29 July 2014	The change was merged to system P_2.
r59711	13 August 2014	The change was merged to system P_7.
r65045	15 April 2015	The change was merged to system P_8.
		The change was never merged to systems P_4 and P_5.

Table 5.5: Detailed analysis repetitive tasks for a bug-fix to *customer* artifact `IntakeModel.java`.

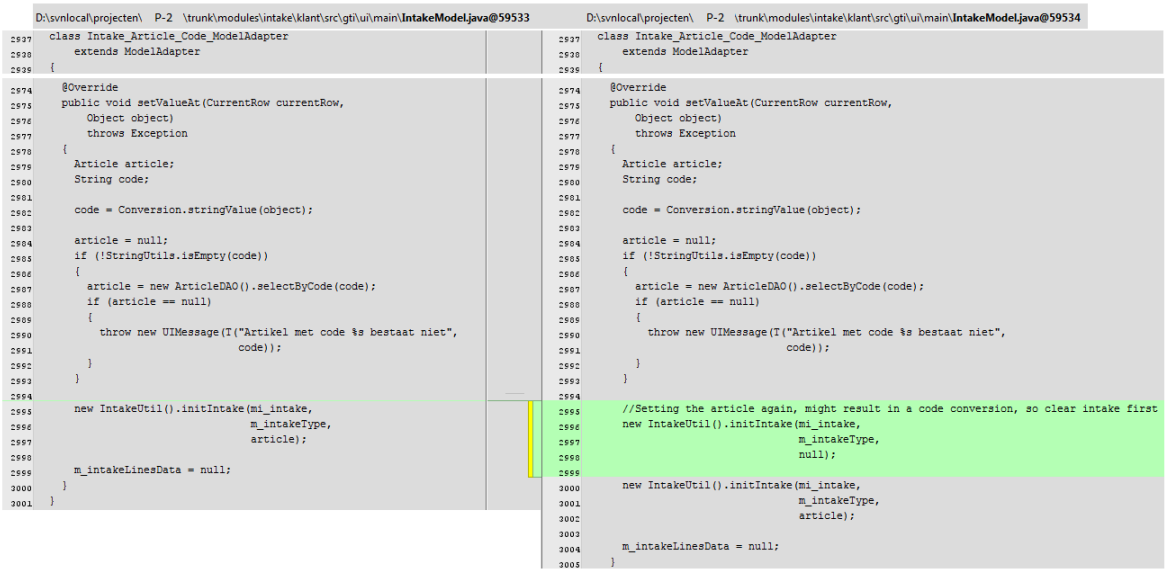


Figure 5.6: Merged bug-fix in customer artifact `IntakeModel.java` for system P_2.

5.4 Discussion

In this section we interpret the results from the previous section (5.3).

Decentralization of Information

Our results show that divergence can vary significantly between different products and over time.

How much do MES-Toolbox systems diverge from their origin?

H$_3$: *The Java codebase for most version 7.2 systems does not diverge significantly (>10%) from their origin.*

Even though the codebase of many systems hardly required any customer-specific modifications, they still diverged significantly. This divergence was not caused by changes to the systems, but by the lack of synchronization of changes from their origin to the system. This is a form of *independent evolution*, a pattern of commits where clones diverge throughout the studied time-interval. However, some clones were eventually synchronized which is a form of *late propagation*, a pattern of commits where clones diverge, and later in time converge again after changes are propagated [28].

Thummalapenta et al. [31] studied clone evolution patterns for cloning in-the-small, and confirmed the possibility of *late propagation* being misclassified as *independent evolution*. However, they found that *late propagation* patterns always took place in much less time than their total time interval of observations, thus concluded that such misclassification would occur only rarely. Our data suggest that cloning in-the-large is much more susceptible to misclassification, as the systems are often synchronized at arbitrary points in time. One system did not retrieve any new changes for almost a year, after which a bulk of changes were propagated at once, reducing the proportion of diverged Java files from 7.5% to less than 4%.

The maintenance overhead caused by divergence due to late propagation is arguably different from divergence due to customer-specific modifications. This raises the question; how do we distinguish between these types of divergence, and how do they affect analysis tools and techniques? Analyzing differences between variants is the primary activity performed when migrated to a more structured software product line approach. Based on these differences, variants can be merged into a single variant or points where variation is needed can be identified. In the context of variation analysis, differences caused by late propagation can be considered as noise, as they are not necessarily relevant.

> **H$_3$**: **Neutral.** In terms of percentage of Java files, all version 7.2 systems eventually diverged between 10% and 22.5% from their origin. However, in terms of diverged number lines, most systems did not exceed 50.000 (<5%)

H$_4$: *Divergence measured at file-level granularity can lead to different conclusions compared to divergence measured at line-level granularity.*

Hetrick, Krueger, and Moore [14] use a *file branch factor* metric to illustrate complexity caused by branching. This metric is defined as the average number of branched files per product, normalized by the number of products. Our study shows that the number of branched files per product can vary significantly between systems and over time, hence care has to be taken when using the average.

However, both techniques can be useful for different purposes. We observed that synchronization of systems with their origin was much clearer on the file-based divergence. Adding or removing files from clones has a significant effect on line-based divergence. If systems frequently add or remove files, line-based divergence may contain too much noise to be useful.

> **H$_4$**: **Accepted.** Systems with a similar percentage of files diverged can vary significantly in terms of total number of lines diverged. Therefore, using file-level divergence to quantify the complexity caused by cloning should be done with care.

Repetitive Tasks

Have all MES-Toolbox systems caused repetitive task maintenance overhead?

H$_5$: *Some MES-Toolbox systems never synchronized with their origin, causing no repetitive task maintenance overhead.*

We found that all systems retrieved changes from their origin at least once, and most but not all systems contributed changes to their origin. This is different from Marlin forks, as Stanciulescu, Schulze, and Wąsowski [30] found that 15% of all forks, and 34% of all active forks synchronized at least once from the main Marlin repository. This suggests that tool support for automated change propagation (e.g.: VariantSync [24]) could reduce maintenance overhead for MES-Toolbox systems. Therefore, we reject hypothesis **H$_5$**.

> **H$_5$**: **Rejected.** While some systems synchronized significantly more frequent than others, all systems retrieved changes from their origin at least once and most systems contributed changes to their origin.

The design of the systems aims to separate *common* from *customer* artifacts. The goal of this separation is to eventually extract the common artifacts from the cloned codebase, and distribute them in the form of shared libraries. This could significantly reduce repetitive task overhead for the common part of the codebase. To evaluate the potential of this approach, we hypothesized that a significant portion of the repetitive task maintenance overhead is caused by customer artifacts.

H$_6$: *A significant portion of repetitive task maintenance overhead is caused by customer artifacts.*

We found that approximately 21% of the merge changes in the 7.2 platform and 7.2 systems were directed at Java *customer* artifacts. This is consistent with our hypothesis. We manually inspected some of these changes, and conclude that even though customer artifacts are not common to all systems, they are re-used for potential future systems.

To better understand why this is happening, we introduce the notion of a *feature*. A feature can be described as functionality from the customer's point of view. Features can be implemented in a single asset (modular), distributed across multiple assets (cross-cutting), or can be part of assets that implement other features (tangled) [2]. We found that many of the *customer* files that were frequently merged, contain *tangled* features. Because the current separation is on file-level granularity, it is difficult to propagate changes only to the systems that require them.

> **H$_6$**: **Accepted.** A significant portion of merge commits is directed at customer artifacts. We suspect that this is caused by the separation of customer functionality at file-level granularity. Many customer files contain *tangled features*. This causes the files to be partially common for multiple systems.

5.5 Threats to Validity

To study the maintenance overhead caused by repetitive tasks we focused on *merge* changes. While merging of changes between systems is a form of task repetition, other forms of repetitive task may cause significantly more maintenance overhead. For example, if a bug has to be uniquely resolved for each system, because the systems are too different to re-use previous changes.

To quantify repetitive tasks we used the number of synchronizing commits. The number of commits can be affected by the behavior of individual developers. Developers can choose to merge each individual revision, or merge a large number revisions at once. The first style clearly results in a higher number of commits compared to the latter, but arguably requires more effort too.

We used the merge-info property to determine whether a commit was a merge. Since this property can be incorrect, we additionally checked whether commits caused systems to converge. We cross-checked the precision of this technique by manually inspecting revisions, and achieved a good precision.

5.6 Conclusion

The main question for this chapter is: *To what extent has clone-and-own caused maintenance overhead for MES-Toolbox systems?*. We studied maintenance overhead caused by clone-and-own from the perspective of Decentralization of Information, and repetitive tasks. In summary, we conclude that clone-and-own has caused maintenance overhead for all MES-Toolbox systems, but for some systems more than others.

Chapter 6

Related Work

Clone Evolution Patterns Thummalapenta et al. [31] proposed an approach for the identification of the evolution of cloned code fragments over time and categorized the evolution patterns as (a) Consistent Evolution, (b) Late Propagation, (c) Delayed Propagation, and (d) Independent Evolution. In our study, we may be able to use these patterns to characterize some of the change patterns in the evolution of the product family. For example, Delayed Propagation may be used as a strategy to validate the correctness of changes on some variants, before propagating them to all variants. Or, Independent Evolution may be used to keep the variant as-is after the project has been commissioned and the testing phase has already finished.

Similar characteristics were found by Stanciulescu, Schulze, and Wąsowski [30] in a study on the advantages and disadvantages of forking using the case of Marlin, an open source firmware for 3D printers. They found that important bug-fixes were not propagated and functionality was sometimes developed more than once. Intuitively you may consider these findings to be bad practices and drawbacks of clone-and-own. However, there are situations where this may be desirable, as the authors found that "Once the firmware is configured and running on the printer, new changes are not desired".

In an environment where the potential cost of an error can be significant, systems are changed as little as possible when maintained [7]. In a clone-an-own based system, this characteristic can be detected by looking for patterns like Independent Evolution, the lack of synchronization with the origin, or redundant code. This is in line with some of the cloning patterns described by Kapser and Godfrey [16]. They argued that code duplication can also have benefits, and described the pros and cons in a catalog of cloning patterns used in real-world systems.

Software Ecosystem Characteristics Lettner et al. [21] studied the relevance of characteristics of Software Ecosystems in the domain of industrial automation and found some additional characteristics that according to them are of particular importance in the industrial automation domain. For example, platform quality characteristics like stability and backward compatibility, and long-term platform evolution seemed to be essential to the success of the studied system. One of the reasons for this conclusion was that *"application engineer B reported that he had to update a ten-year-old version of the platform software because an important customer had decided to leave out several platform releases and then requested a new feature. This led to significant difficulties in merging the old software version with the new functionality."*. Developers of the system we study have reported similar issues with upgrading customer systems to a new release.

In a later study by Lettner et al. [20], the *change characteristics* and *software evolution challenges* of the same ecosystem were investigated. The software change taxonomy of Buckley et al. [5] was used to describe qualitatively when, where, and how changes were made in different parts of the system and what was affected by changes. The authors found that the ecosystem is subject to both continuous and periodic evolution. The core platform is continuously changed to include new features and bug-fixes, while those changes are only periodically released to platform users. The granularity of these changes is reportedly primarily coarse for customer requirements, and fine for bug fixes. Propagation of changes is done by hand, and change impact analysis is performed manually, based on

expert knowledge.

The system we study is in the same domain and seems to be developed similarly. Our study will be different in a sense that we aim to support qualitative findings with quantitative data by analyzing the evolution of the system. For example, we know that in this case changes are also propagated by hand, so we intend to study when this is done, how frequently, and what factors influence the decision on whether or not to propagate changes from project to a release, between releases or between projects.

Crosscutting Concerns A possible area of interest in the analysis of clone-and-own evolution is the presence and development of *crosscutting concerns* in the system. A crosscutting concern is a feature whose implementation is spread across many different modules [22]. If product variants, or clones, exhibit a high degree of variation in the implementation of crosscutting concerns, we expect that this may also affect the extent to which changes are propagated, and how the code-bases diverge.

Marin, Moonen, and Deursen [23] propose a classification system for crosscutting concerns in terms of *sorts*, where a *sort* is a description based on a number of distinctive properties. A *sort* we expect to find often in this case study is *Entangled Roles*. In Object Oriented terminology this sort is defined as *Implement a method with (entangled) functionality that belongs to a different concern than the main concern of that method*. A characteristic of clone-and-own is that it allows application engineers to make these kinds of fine-grained changes quickly. For example, a customer wants to be notified when stock levels exceed a certain value for a specific product. If there is no such monitoring system in place, then the fastest solution can be to add this functionality to a method that deals in some way with stock-control. Implementation of a generic solution may exceed the level of expertise of the application engineer, and waiting for a platform engineer to develop the solution may take too much time.

Figueiredo et al. [11] describe 13 patterns of crosscutting concerns identified in three case studies, one of which was a software product line. The authors found that some patterns consistently emerged in situations with the frequent use of inheritance. They found that this was often the case in product lines because "*Program families rely extensively on the use of abstract classes and interfaces in order to implement variabilities. The inappropriate modularization of such crosscutting concerns might lead to future instabilities in the design of the varying modules*"

Detection of crosscutting concerns is called *aspect mining*. Various aspect mining techniques have been proposed [17, 32, 6]. For example, fan-in analysis looks for crosscutting functionality by detecting methods that are explicitly invoked from many different methods scattered throughout the code [22]. History-based concern mining techniques analyze change-history to detect which program entities change together frequently [4, 1]. Hashimoto and Mori [13] developed a tool that improves history-based concern mining by combining it with fine-grained change analysis based on abstract syntax tree differencing.

The goal of our study is not to investigate the effectiveness of these tools and techniques, but they may provide insight into the characteristics of change patterns we will find.

Clone-and-Own in Product Line Engineering Dubinsky et al. [8] studied the processes and perceived advantages and disadvantages of the clone-and-own approach of six industrial software product lines. They show that cloning is perceived as a favorable and natural reuse approach by the majority of practitioners in the studied companies, mainly because of its simplicity and availability. They found that practitioners lack the awareness and knowledge about forms of reuse, and many alternative approaches fail to convince them that they yield better results.

Rubin, Czarnecki, and Chechik [27] proposed a framework to organize knowledge related to the development, maintenance and merge-refactoring of product lines realized via cloning. This framework is a step towards a recommender system that can assist users in selecting tools and techniques that are useful in their situation. The characteristics we have found in our exploratory analysis may aid the qualification of situations where different tools and techniques can be useful.

Hetrick, Krueger, and Moore [14] report on the experience of a structured, incremental transition from a clone-and-own approach to software product line practices. They show that it is possible to make this transition without a significant upfront investment and disruption of the ongoing production schedules. The authors indicate that the *file branch factor* gradually reduced during the transition,

to a point where all branches from product line core assets were completely eliminated. This metric is defined as the average number of branched files per product, normalized by the number of products. Our study shows that the number of branched files per product can vary significantly between systems and over time. Hence care has to be taken when using the average. Furthermore, we found that products with a similar percentage of files diverged can vary significantly in terms of total number of lines diverged.

Antkiewicz et al. [2] propose an incremental and minimally invasive strategy for adoption of product-line engineering. The strategy is called *virtual platform*, and should allow organizations to obtain incremental benefits from incremental changes to the development approach.

By studying the development practices of our industry case, we gain insight into an industry context and the needs of practitioners. This may serve as input for recommender systems, requirements for the *virtual platform*, and can be helpful to practitioners, researchers and tool developers.

Chapter 7

Conclusion

In this work, we presented the results of our exploratory analysis of an industry product family developed using a clone-and-own approach. The goal of this analysis was to gain insight into how the product family has evolved, and to validate the correctness of our data, tools, and techniques. We show how relatively simple visualizations can be used to explore the evolution of a product family and to identify variance in change characteristics.

The results of our study suggest that products from the same product family can vary significantly in terms of change frequency and change distribution. This suggests that not all MES-Toolbox systems may benefit from the independence provided by clone-and-own to the same extent. While all systems showed signs of continuous, periodical and arbitrary change intervals, some systems changed significantly more frequent than others. It is important to keep this in mind when studying product families realized via clone-and-own, as different change characteristics of systems within the same product family may require different techniques to cope with maintenance overhead.

Bibliography

[1] B. Adams, Z. M. Jiang, and A. E. Hassan. "Identifying Crosscutting Concerns Using Historical Code Changes". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. Vol. 1. ACM, 2010, pp. 305–314. DOI: 10.1145/1806799.1806846.

[2] M. Antkiewicz et al. "Flexible Product Line Engineering with a Virtual Platform". In: *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*. ACM, 2014, pp. 532–535. DOI: 10.1145/2591062.2591126.

[3] T. Berger et al. "Three Cases of Feature-Based Variability Modeling in Industry". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8767. Springer, 2014, pp. 302–319. DOI: 10.1007/978-3-319-11653-2{_}19.

[4] S. Breu and T. Zimmermann. "Mining Aspects from Version History". In: *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 221–230.

[5] J. Buckley et al. "Towards a Taxonomy of Software Change". In: *Journal of Software Maintenance and Evolution: Research and Practice* 17.5 (2005), pp. 309–332. DOI: 10.1002/smr.v17:5.

[6] M. Ceccato et al. "Applying and Combining Three Different Aspect Mining Techniques". In: *Software Quality Journal* 14.3 (Sept. 2006), pp. 209–231. DOI: 10.1007/s11219-006-9217-3.

[7] J. R. Cordy. "Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation". In: *Program Comprehension, 2003. 11th IEEE International Workshop on*. IEEE, 2003, pp. 196–205.

[8] Y. Dubinsky et al. "An Exploratory Study of Cloning in Industrial Software Product Lines". In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*. 2013, pp. 25–34. DOI: 10.1109/CSMR.2013.13.

[9] A. N. Duc et al. "Forking and coordination in multi-platform development: a case study". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14*. New York, New York, USA: ACM Press, 2014, pp. 1–10. DOI: 10.1145/2652524.2652546.

[10] D. Faust and C. Verhoef. "Software product line migration and deployment". In: *Software: Practice and Experience* 33.10 (Aug. 2003), pp. 933–955. DOI: 10.1002/spe.530.

[11] E. Figueiredo et al. "Crosscutting Patterns and Design Stability: An Exploratory Analysis". In: *IEEE International Conference on Program Comprehension*. 2009, pp. 138–147. DOI: 10.1109/ICPC.2009.5090037.

[12] J. Harder and N. Göde. "Cloned code: stable code". In: *Journal of Software: Evolution and Process* 25.10 (Oct. 2013), pp. 1063–1088. DOI: 10.1002/smr.1551.

[13] M. Hashimoto and A. Mori. "Enhancing History-Based Concern Mining with Fine-Grained Change Analysis". In: *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 75–84. DOI: 10.1109/CSMR.2012.18.

[14] W. A. Hetrick, C. W. Krueger, and J. G. Moore. "Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice". In: *International Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, 2006, pp. 798–804. DOI: 10.1145/1176617.1176726.

[15] K. Hotta et al. "Is duplicate code more frequently modified than non-duplicate code in software evolution?" In: *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE) on - IWPSE-EVOL '10*. New York, New York, USA: ACM Press, 2010, p. 73. DOI: 10.1145/1862372.1862390.

[16] C. Kapser and M. Godfrey. ""Cloning Considered Harmful" Considered Harmful". In: *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 19–28. DOI: 10.1109/WCRE.2006.1.

[17] A. Kellens, K. Mens, and P. Tonella. "A Survey of Automated Code-Level Aspect Mining Techniques". In: *Transactions on Aspect-Oriented Software Development IV*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 143–162. DOI: 10.1007/978-3-540-77042-8{_}6.

[18] J. Krinke. "Is Cloned Code More Stable than Non-cloned Code?" In: *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, Sept. 2008, pp. 57–66. DOI: 10.1109/SCAM.2008.14.

[19] C. W. Krueger. "Software reuse". In: *ACM Computing Surveys* 24.2 (June 1992), pp. 131–183. DOI: 10.1145/130844.130856.

[20] D. Lettner et al. "A Case Study on Software Ecosystem Characteristics in Industrial Automation Software". In: *Proceedings of the 2014 International Conference on Software and System Process - ICSSP 2014*. ACM, 2014, pp. 40–49. DOI: 10.1145/2600821.2600826.

[21] D. Lettner et al. "Software Evolution in an Industrial Automation Ecosystem: An Exploratory Study". In: *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*. IEEE, 2014, pp. 336–343. DOI: 10.1109/SEAA.2014.43.

[22] M. Marin, A. van Deursen, and L. Moonen. "Identifying Crosscutting Concerns Using Fan-In Analysis". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17.1 (2007), pp. 1–37. DOI: 10.1145/1314493.1314496.

[23] M. Marin, L. Moonen, and A. van Deursen. "A Classification of Crosscutting Concerns". In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 673–676. DOI: 10.1109/ICSM.2005.7.

[24] T. Pfofe et al. "Synchronizing Software Variants with VariantSync". In: *Proceedings of the 20th International Systems and Software Product Line Conference on - SPLC '16*. New York, New York, USA: ACM Press, 2016, pp. 329–332. DOI: 10.1145/2934466.2962726.

[25] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering*. Vol. 49. 12. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, p. 467. DOI: 10.1007/3-540-28901-1.

[26] J. Rubin. "Cloned Product Variants: From Ad-hoc to Well-managed Software Reuse." PhD thesis. University of Toronto, 2014.

[27] J. Rubin, K. Czarnecki, and M. Chechik. "Managing Cloned Variants: A Framework and Experience". In: *Proceedings of the 17th International Software Product Line Conference - SPLC '13*. ACM, 2013, p. 101. DOI: 10.1145/2491627.2491644.

[28] T. Schmorleiz and R. Lammel. "Similarity management of 'cloned and owned' variants". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC '16*. New York, New York, USA: ACM Press, 2016, pp. 1466–1471. DOI: 10.1145/2851613.2851785.

[29] S. Schrock, A. Fay, and T. Jager. "Systematic interdisciplinary reuse within the engineering of automated plants". In: *Systems Conference (SysCon), 2015 9th Annual IEEE International*. 2015, pp. 508–515. DOI: 10.1109/SYSCON.2015.7116802.

[30] S. Stanciulescu, S. Schulze, and A. Wąsowski. "Forked and Integrated Variants in an Open-Source Firmware Project". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2015, pp. 151–160. DOI: 10.1109/ICSM.2015.7332461.

[31] S. Thummalapenta et al. “An Empirical Study on the Maintenance of Source Code Clones”. In: *Empirical Software Engineering* 15.1 (Feb. 2010), pp. 1–34. DOI: 10.1007/s10664-009-9108-x.

[32] T. Tourwé and K. Mens. “Mining Aspectual Views using Formal Concept Analysis”. In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. IEEE Comput. Soc, 2004, pp. 97–106. DOI: 10.1109/SCAM.2004.15.

[33] F. Van Rysselberghe and S. Demeyer. “Studying Software Evolution Information By Visualizing the Change History”. In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 328–337. DOI: 10.1109/ICSM.2004.1357818.

[34] A. Yamashita et al. “Software Evolution and Quality Data from Controlled, Multiple, Industrial Case Studies”. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE, 2017, pp. 507–510. DOI: 10.1109/MSR.2017.44.