Analysis of a Clone-and-Own Industrial Automation System: An Exploratory Study

Nick Lodewijks
University of Amsterdam,
The Netherlands
nicklodewijks@gmail.com

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. Unfortunately, current literature lacks quantitative information about the positive and negative effects of clone-and-own. In this paper, we present the results of our exploratory analysis of an industry system developed using the clone-and-own approach. We found that products from the same product family can vary significantly in change activity over time, divergence from their origin and synchronization activity. We will further investigate these factors to develop quantitative measures for the assessment of clone-and-own benefits and drawbacks.

1 Introduction

Cloning is often considered to be a practice harmful to the quality of source code, and potentially a cause of maintainability problems (Kapser and Godfrey, 2006; Thummalapenta et al., 2010). 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 (Dubinsky et al., 2013).

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 ($\pm$1 million lines of Java code) proprietary factory automation system. Over the past 17 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 worldwide, the company now seeks to reduce maintenance overhead. 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 the elimination of all clones by merging the variants into a single platform, and others propose to maintain multiple variants as-is (Rubin, Czarnecki, and Chechik, 2013). 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 (Antkiewicz et al., 2014). Eliminating clones will increase coupling, and changing shared code may require re-testing of all systems that use it (Dubinsky et al., 2013). If the success of the product highly depends on the benefits of clone-and-own, then its merits should be considered before moving away to a different approach.

The main objective of our study is to explore the evolution of MES-Toolbox systems and to gain insight into how clone-and-own may have affected ongoing project development and maintenance. In this paper, we show how version control system metadata, source-code differencing, and visualization techniques can be used to identify clone-and-own related points of interest in the evolution of a product family.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org).
07-09 June 2017, Madrid, Spain.

2 Subject System

The system studied in this work is the MES-Toolbox; a 17-year-old proprietary Java-based factory automation system developed by ENGIE. The main purpose of the systems is for 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 the 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. 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.

Over the past 17 years, the system has grown to contain more than 6500 Java files, with a total of approximately 1 million lines of Java code. While the design of the system has a modular structure and aims to separate *common* code from *customer* implementation code as much as possible, it's a monolithic application. Nearly all source-code is contained in a single *project*, which is developed, built and versioned as a whole. Internally, this project is called the *Standard* project, as it is used as a basis for all new projects. This project, which can be considered as the main platform of the product family, contains a constantly growing set of reusable core components and ready-to-use standard solutions.

Within the organization there is a clear distinction between platform development and application development, this distinction is often found in a Software Ecosystem (SECO) (Lettner, Angerer, Grünbacher, et al., 2014). A small team of five developers is responsible for the overall design, development, and maintenance of the 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.

Even though the system is highly configurable, cloning is used to address the specificity and high degree of variation of customer requirements often found in the domain of industrial automation (Schrock, Fay, and Jager, 2015). For every new factory, a clone of the codebase 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. Each clone corresponds to the automation system of a factory somewhere around the world for some specific customer. Each clone is a variant of the base platform. We refer to the collection of all MES-Toolbox variants as the MES-Toolbox product family.

Between clones there exists a varying degree of commonality, and there is often no clear relation between the clones. Clones developed for the same customer might have more in common than clones developed for different customers. For example, if a company requires all their production facilities to have identical branding and communication interfaces with third-party systems. However, even clones that appear to be unrelated in terms of end-user requirements may still have some forms of commonality, such as the graphical user interface components or the configuration framework that is used.

3 Research Questions

Dubinsky et al. (2013) 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. But how much independence is needed, and how can it be measured? In this section, we describe three research questions that we will use to explore independence-related characteristics of the MES-Toolbox product family.

RQ1: *Do MES-Toolbox systems change in parallel?* When cloning is used to develop systems independent of each other, developers can decide *when* to change the codebase of each individual system. The development of each system can follow its own release and development schedule that is based on available resources and requirements for the system. The development of complex systems with many customer-specific modifications may require and allow for months of continuous, frequent change, while relatively straightforward and simple systems might have strict deadlines and require only a few changes within the first weeks.

To explore whether MES-Toolbox systems may have benefited from this *time* aspect of independence,

we want to gain a rough understanding of the degree of parallel change in the MES-Toolbox product family. We hypothesize that a schedule-driven need for independence may lead to a lack of parallel development, whereas some relation between systems (e.g.: systems developed the same customer) or a collaboration-overhead driven need for independence may lead to parallel change. For the purpose of this exploratory study, we do not yet use a strict definition of *parallel change*. Instead, we are interested in any form of seemingly parallel change. Do systems change in parallel every week, month or year? Do many systems change at roughly the same time, or is this only the case for specific systems?

RQ2: *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. As the product family grows it often becomes increasingly hard to keep an overview of the available functionality (Stanciulescu, Schulze, and Wąsowski, 2015; Berger et al., 2014; Duc et al., 2014). We hypothesize that the degree of divergence can be used to quantify the complexity caused by cloning. Therefore, we are interested to see how this property of the MES-Toolbox product family has evolved over time.

A developer of the MES-Toolbox platform 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.

RQ3: *Have all MES-Toolbox systems been synchronized with their origin?* Cloning is said to increase maintenance overhead because changes to one clone may have to be propagated to all clones. Studies have shown however that change propagation is not always performed (Stanciulescu, Schulze, and Wąsowski, 2015), 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 changes from the platform release to customer 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 arguably not causing much maintenance overhead. Consequently, techniques that purely reduce repetitive task would have limited effect on maintenance overhead caused synchronization for these systems. In the MES-Toolbox product family, synchronization with products and their origin can occur in both ways. Bugs are often found and fixed on a product, after which the change is propagated to the platform project. From there, the change can be propagated to all the other products derived from that platform version.

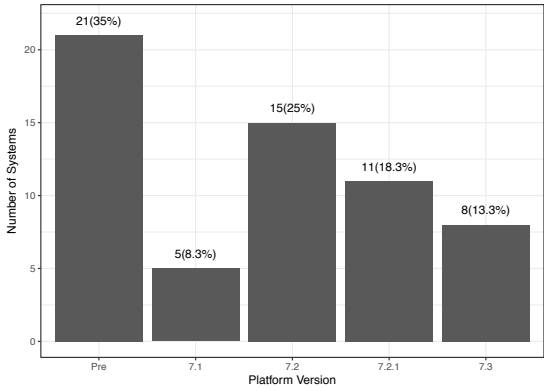

Figure 1: Distribution of System Versions

4 Research Methodology

To explore the evolution of MES-Toolbox systems, 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. Our tool is embedded in a modified version of JMeld[1], an open source differencing tool written in Java.

First, we make a local copy of the SVN repository with the command `svnadmin hotcopy`, and verify its integrity with `svnadmin verify` in the analysis environment. This local repository is used for all data collection to ensure that the data source does not change during subsequent analysis.

4.1 Selecting MES-Toolbox Systems

We extract all systems present in the local copy of the repository by scanning the output of `svn ls`[2] 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 the systems were never put into production. We excluded these systems from the analysis. Finally, we noted

[1] `https://sourceforge.net/projects/jmeld/`

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

whether the system was directly branched from the platform, or from another branch (its *nesting depth*).

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. Figure 1 shows the distribution of versions among systems in the version repository. 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 these have all been put into production, and are derived from a comparable base platform within the last five years. The main difference between these versions is the internationalization of all text visible to the end-user. There are no significant differences in terms of architecture or functionality.

We refer to the codebase of a specific platform version in the form of `PL-VERSION`, for example, we use $PL_{7.2}$ to refer to version 7.2 of the platform. The internal name of the system 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 in the form of `P-NUMBER`. In this paper, we often refer to this name as P_n, which can be read as *project n* or *product n*.

4.2 Mining Commit Metadata

For each system we selected, we extract the version history using a bash script. This bash script uses the `svn log` [3] command to export the version history in xml format. For each system we collect all revisions from its change history, and extract the relevant change metrics. We used the definition and format of the change metrics dataset published by Yamashita et al. (2017) as a basis for our data set.

We extract the *revision number*, *author* and *date* of each revision. Next, from the output of `svn diff`[4], 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. From the full path of the files, we extract the file name and file extension. 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.

[3] svn log --xml --stop-on-copy -v <variant.repositoryPath> > <variant>-log.xml

[4] svn diff -x -U0 -c {revisionNumber} {repositoryPath}

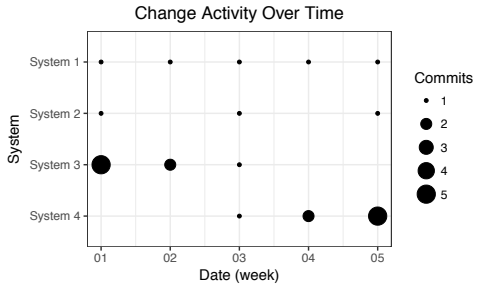

Figure 2: Example visualization for change activity. All systems exhibit different change activities, with a varying degree of parallel change.

4.3 Detecting Parallel Change

To determine whether systems change in parallel, 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 the radius of the dot). This leads us to the view shown in Figure 2.

The vertical axis represents the systems, and the horizontal axis the time of the changes. Each dot represents a point in time when 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 activity 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. In Figure 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.

4.4 Measuring Divergence

To measure how much systems have diverged from their origin, we developed a tool that calculates how much the difference between each system and its origin has changed over time. We do so by calculating the differences for each system, for each file, at every revision that changed either the system or its origin. We perform these measurements on a local copy of the actual codebase of the systems. For the platforms and each system, we locally replay their change history by sequentially updating the local working copy with `svn update`. After each update of a platform codebase, we re-calculate the differences with code-differencing on all systems that have been derived from the platform. Similarly, after each update of the codebase of a system, we re-calculate the differences between the system and its origin. This technique is computationally intensive but does allow us to explore how much each revision has affected divergence.

We measure differences at line-level granularity (number of lines different) with the Java implementation of GNU diff [5]. Using the file-level granularity measurement, we aggregate to file-level granularity. By using a line-level granularity instead of a file-level granularity (number of files different), we will be able to aggregate to file-level granularity and report on both levels. We define the difference in number of lines as `diff`. During analysis, we keep track of how much the difference has increased or decreased compare to the previous revision, the `diffDelta`.

We illustrate the divergence calculation on an artificial example in Table 1. In this example, $PL_{7.2.1}$ is the origin of system P_{17}. First, we update the local copy of the codebase of $PL_{7.2.1}$ to revision 1 and calculate the differences between $PL_{7.2.1}$ and P_{17}. We see that in revision 1, `Main.java` was modified on $PL_{7.2.1}$, causing a difference of five lines. Next, we update P_{17} to revision 2 and re-calculate the differences. We see that `Main.java` was changed, reducing the difference by five lines. This pattern of increasing and decreasing divergence is typically caused by change propagation when revision 1 is merged to system P_{17} in revision 2. As the measurements continue, we see that `Main.java` was modified two more times on P_{17}, 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 $PL_{7.2.1}$.

4.5 Detecting Synchronization

Systems retrieving changes from their origin, or contributing changes to their origin is often done by *merging* the revision from the system to its origin or vice versa. 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[6]. We classify each revision commit as `MERGE` or `NON_MERGE` by scanning the output of `svn diff` for an occurrence of `svn:mergeinfo`.

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

Table 1: Example data of divergence over time calculation.

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.

5 Results and Analysis

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

5.1 Parallel Change

RQ1. Do MES-Toolbox systems change in parallel?
Figure 3 shows the change activity of the $PL_{7.2}$ and $PL_{7.2.1}$ platforms, and all systems derived from these platforms that were included in our study. 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[7] for system P_4.

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

[6] `http://svnbook.red-bean.com/en/1.7/svn.branchmerge.basicmerging.html`

[7] 124 days, 14 March 2014 to 16 July 2014

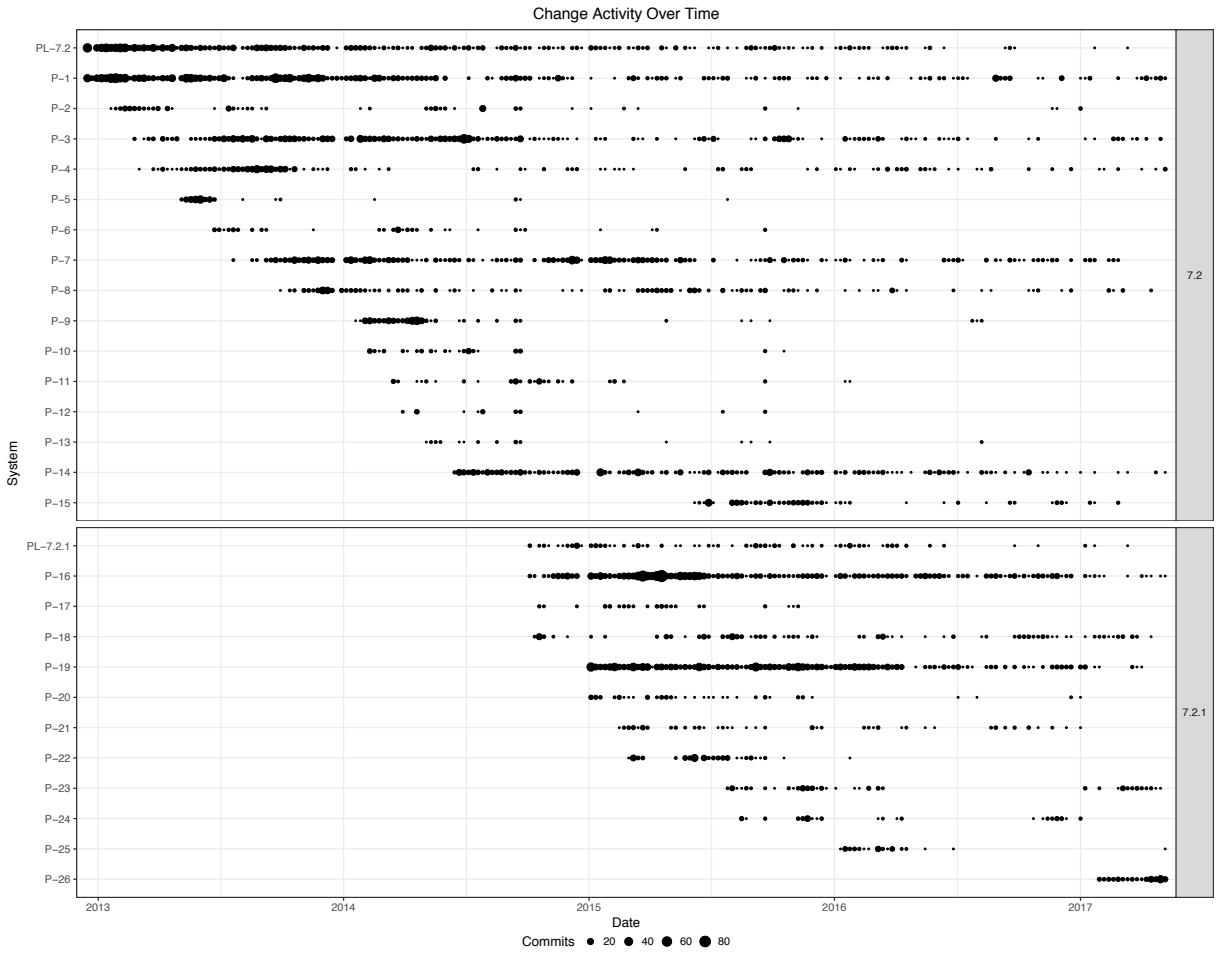

Figure 3: The change activity of $PL_{7.2}$ and $PL_{7.2.1}$ systems.

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. This seems similar to the change frequency of Keba, an industrial automation ecosystem studied by Lettner, Angerer, Grünbacher, et al. (2014). In the Keba ecosystem, the change frequency reportedly largely depends on customer requirements, and most changes happen within the first three to four weeks in a customer project. In our case, for many systems, most change activity does appear to occur in the first period of the project, but this period is much longer (2-4 months). Manual examination of some of the changes that occurred after this initial period, suggests that they are often (critical) bug-fixes or minor changes requested by the customer. For example, P_5 was changed on 31 July 2015 after being inactive for almost a year (311 days). Manual analysis of this change shows that this change was triggered by a customer request after the physical production line was modified.

The change activity of MES-Toolbox systems seems consistent with observations by Lettner, Angerer, Grünbacher, et al. (2014), who stressed the importance of platform quality characteristics like stability and backward compatibility, and long-term platform evolution in the domain of industrial automation. The oldest system we analyzed was 11 years old, and still continuously changed. Some systems were inactive for years before becoming active again due to new customer demands. This is not necessarily the case for other systems developed with clone-and-own. Stanciulescu, Schulze, and Wąsowski (2015) found forks in the Marlin ecosystem, an open source firmware for 3D printers, to be characterized by a short maintenance lifetime (101 days on average).

With regard to whether and to what extent MES-Toolbox systems have been changed in parallel, we clearly see that multiple MES-Toolbox systems are changed roughly at the same time. However, the degree of parallel change is not the same for all systems, nor is it constant over time. Many systems appear to

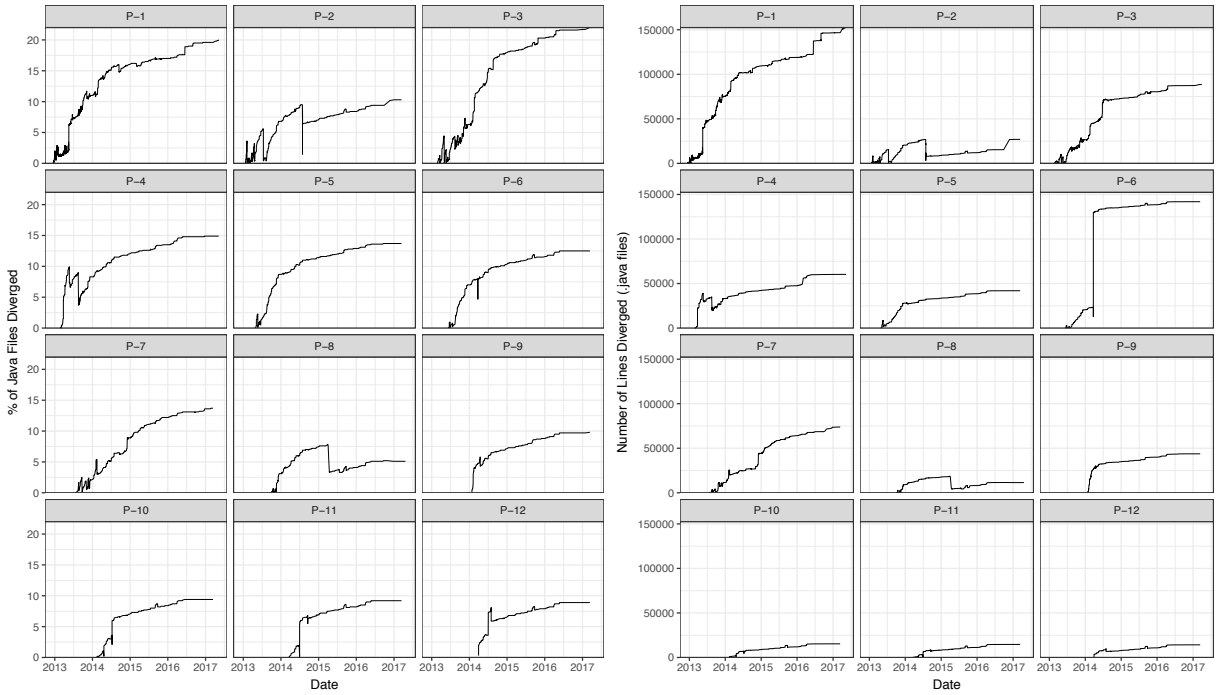

Figure 4: Divergence over time for a subset of $PL_{7.2}$ and $PL_{7.2.1}$ systems in percentage of files and number of lines.

be changed in parallel initially until the development of one system is done and they no longer change in parallel. For example, if we look at systems P_4 and P_5, we see a major reduction in change activity of system P_5 after June 2013, but the development of system P_4 continues. This type of pattern is what we would expect to see due to a schedule-driven need for independence.

Furthermore, we observe at least two vertically aligned dot patterns. These patterns occur if multiple systems are changed at roughly the same time, while many of those systems did not change before or after that time. Manual inspection of these patterns shows that both instances were critical bug fixes, manually merged to most systems on the same day, regardless of the development schedule of the systems. 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 in the MES-Toolbox product family.

5.2 Divergence

RQ2. How much do MES-Toolbox systems diverge from their origin? In this research question, we calculate how much MES-Toolbox systems are different from their origin, and explore how this property has changed over time. Figure 4 shows the divergence measurements over time, in terms of percentage of files and number of lines.

It may be seen clearly that while divergence tends to increase over time, there is a 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 compared to the other systems. 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.

Overall, we see that divergence measured in percentage of Java files can be significantly different from divergence measured in terms of number of lines. In 2014 the diverged number of lines for system P_6 rapidly increased from less than 25.000 lines to more than 140.000 lines. We do not see this growth in the file-based measurement. Manual analysis of this anomaly shows that a developer deleted a module from the codebase which was not required for the project but was causing merge-conflicts.

Even though the codebase of many systems reportedly hardly required any customer-specific modifications, they still diverged significantly. For these systems, this divergence was not caused by changes to the systems, but by the lack of synchronization of changes

Figure 5: Synchronizing Changes of $PL_{7.2}$ and $PL_{7.2.1}$ systems.

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 (Schmorleiz and Lammel, 2016).

Thummalapenta et al. (2010) 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 may be much more susceptible to misclassification, as in our case the systems are often synchronized at arbitrary points in time. System P_8 did not retrieve any new changes from its origin 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 are not necessarily relevant.

5.3 Synchronization

RQ3. Have all MES-Toolbox systems been synchronized with their origin?

To detect whether a change to either $PL_{7.2.1}$ or $PL_{7.2}$ was contributed by one of the systems, we identify revisions that caused at least one system to converge one line. In the combined history of $PL_{7.2.1}$ and $PL_{7.2}$, there were 501 revisions for which this was the case. We manually inspected these revisions and found that 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. A detection strategy solely based on the presence of merge-info would have missed these revisions.

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. The change history of system P_4 contained 18 revisions with merge-info, of which 14 caused convergence. Out of these 14 revisions, 12 (85%) were correctly classified as changes retrieved from its origin.

Figure 5 shows the synchronizing changes over time. We see 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 (2015) found that 15% of all forks, and 34% of all active forks synchronized at least once with the main Marlin repository.

While all systems retrieve changes from their origin, some do so significantly more frequent than others. Systems P_1 and P_3 retrieved changes from their origin respectively 202 and 89 times. Furthermore, we see that the period of time between subsequent synchronizations can be relatively long. For example, system P_6 retrieved changes from its origin on 22 July 2013, and 8 months later on 24 March 2014. This is consistent with the results in the previous section, where we identified long time-interval late propagation in the visualization of divergence over time.

Finally, 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 in the MES-Toolbox product family.

6 Threats to Validity

Internal Validity During our study, the MES-Toolbox product family continued to change. To prevent this change from affecting our results, we obtained a local copy of the repository. This local copy of the repository was used throughout the study.

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.

While the experience of the author as a developer of the system may provide a detailed interpretation of fine-grained changes, this can cause some bias. We aimed to reduce this threat as much as possible by providing quantitative data to support our findings and collaboration with an external supervisor.

External Validity Development practices in other organizations that use clone-and-own might have different effects on the evolution of the system, which may lead to different observations. However, some of our findings are consistent with those of other, independent studies.

In our analysis of synchronizing changes, we looked at 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.

7 Related Work

Clone Evolution Patterns

Thummalapenta et al. (2010) 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 used these patterns to characterize some of the change patterns we observed in the evolution of the product family. For example, Delayed Propagation was used as a strategy to validate the correctness of changes on some variants, before propagating them to all variants. Independent Evolution was used to keep the variant as-is after the project had been commissioned and the testing phase had already finished.

Similar characteristics were found by Stanciulescu, Schulze, and Wąsowski (2015) 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 (Cordy, 2003). 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 (2006). 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, Angerer, Grünbacher, et al. (2014) 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, Angerer, Prähofer, et al. (2014), the *change characteristics* and *software evolution challenges* of the same ecosystem were investigated. The software change taxonomy of Buckley et al. (2005) 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 is different in a sense that we support our findings with visual representations of the evolution of the system. For example, we know that in this case changes are also propagated by hand, so we developed a technique to show how frequent this is actually done in the MES-Toolbox product family.

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 modules (Marin, Deursen, and Moonen, 2007). 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 (2005) 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. 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. (2009) 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 (Kellens, Mens, and Tonella, 2007; Tourwé and Mens, 2004; Ceccato et al., 2006). For example, fan-in analysis looks for crosscutting functionality by detecting methods that are explicitly invoked from many methods scattered throughout the code (Marin, Deursen, and Moonen, 2007). History-based concern mining techniques analyze change-history to detect which program entities change together frequently (Breu and Zimmermann, 2006; Adams, Jiang, and Hassan, 2010). Hashimoto and Mori (2012) developed a tool that improves history-based concern mining by combining it with fine-grained change analysis based on abstract syntax tree differencing.

In future work, we intend to use these tools and techniques to gain a deeper understanding of the change and divergence patterns we found.

Clone-and-Own in Product Line Engineering

Dubinsky et al. (2013) 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 (2013) 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.

Hetrick, Krueger, and Moore (2006) 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. (2014) 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.

8 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 identify clone-and-own related points of interest. First, we explored whether MES-Toolbox systems have changed in parallel. Next, we investigated how much the codebase of the systems diverged from their origin, and to what extent this changed over time. Finally, we studied the synchronization activity between systems and their origins.

We observed that many MES-Toolbox systems are changed roughly at the same time, but that the degree of parallel change is not the same for all systems, nor is it constant over time. Many systems appear to be changed in parallel initially until the development of one system is done and they no longer change in parallel. This is consistent with a schedule-driven need for independence. We further observed a schedule-independent cause for parallel change, which was the need to propagate critical bug fixes to many systems on the same day. This form of mass-synchronization appeared to have occurred only twice in the history of the systems we analyzed.

With regard to divergence, we found that all MES-Toolbox systems we analyzed, including those which reportedly hardly required any customer-specific modifications, diverged significantly from their origin. In terms of the proportion of Java files, all systems diverged between 7% and 22.5% from their origin. 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. We identified one case where the divergence measured in percentage of Java files was significantly different from divergence measured in terms of number of lines.

During our analysis of divergence over time, we were able to identify points in time when systems were synchronized with their origin. Our analysis of synchronizing changes confirms these findings, and we found that all systems we analyzed retrieved changes from their origin at least once, but not all systems contributed changes back to their origin.

Overall, these results show that products from the same product family can vary significantly in terms of change activity over time, divergence from their origin and synchronization activity. It is important to keep this in mind when studying product families realized via clone-and-own, as these variations may play an important role in reducing maintenance overhead. In future work, we will further investigate these factors to develop quantitative measures for the assessment of clone-and-own benefits and drawbacks.

Acknowledgements

We thank prof. dr. J.J. Vinju, the reviewers and other participants of the SATToSE 2017 seminar for their helpful input on related literature and the direction of this study.

References

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

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

Berger, T. et al. (2014). "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, pp. 302–319.

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

Buckley, J. et al. (2005). "Towards a Taxonomy of Software Change". In: *Journal of Software Maintenance and Evolution: Research and Practice* 17.5, pp. 309–332.

Ceccato, M. et al. (2006). "Applying and Combining Three Different Aspect Mining Techniques". In: *Software Quality Journal* 14.3, pp. 209–231.

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

Dubinsky, Y. et al. (2013). "An Exploratory Study of Cloning in Industrial Software Product Lines". In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pp. 25–34.

Duc, A. N. et al. (2014). "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, pp. 1–10.

Figueiredo, E. et al. (2009). "Crosscutting Patterns and Design Stability: An Exploratory Analysis". In: *IEEE International Conference on Program Comprehension*, pp. 138–147.

Hashimoto, M. and A. Mori (2012). "Enhancing History-Based Concern Mining with Fine-Grained Change Analysis". In: *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, pp. 75–84.

Hetrick, W. A., C. W. Krueger, and J. G. Moore (2006). "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, pp. 798–804.

Kapser, C. and M. Godfrey (2006). "'"Cloning Considered Harmful" Considered Harmful". In: *2006 13th Working Conference on Reverse Engineering*. IEEE, pp. 19–28.

Kellens, A., K. Mens, and P. Tonella (2007). "A Survey of Automated Code-Level Aspect Mining Techniques". In: *Transactions on Aspect-Oriented Software Development IV*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 143–162.

Lettner, D., F. Angerer, P. Grünbacher, et al. (2014). "Software Evolution in an Industrial Automation Ecosystem: An Exploratory Study". In: *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*. IEEE, pp. 336–343.

Lettner, D., F. Angerer, H. Prähofer, et al. (2014). "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, pp. 40–49.

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

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

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

Schmorleiz, T. and R. Lammel (2016). "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, pp. 1466–1471.

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

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

Thummalapenta, S. et al. (2010). "An Empirical Study on the Maintenance of Source Code Clones". In: *Empirical Software Engineering* 15.1, pp. 1–34.

Tourwé, T. and K. Mens (2004). "Mining Aspectual Views using Formal Concept Analysis". In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. IEEE Comput. Soc, pp. 97–106.

Yamashita, A. et al. (2017). "Software Evolution and Quality Data from Controlled, Multiple, Industrial Case Studies". In: *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE, pp. 507–510.