Defining a Strategy to Introduce a Software Product Line Using Existing Embedded Systems

Kentaro Yoshimura
Hitachi Europe
Automotive R&D Lab
18 rue Grange Dame Rose
78140 Velizy, France
+33 1 34 63 05 00
kentaro.yoshimura@hitachi-eu.com

Dharmalingam Ganesan
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
+49 (0) 631 / 68 00 - 2232
ganesan@iese.fraunhofer.de

Dirk Muthig
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
+ 49 (0) 631 / 68 00 – 1302
muthig@iese.fraunhofer.de

ABSTRACT

Engine Control Systems (ECS) for automobiles have numerous variants for many manufactures and different markets. To improve development efficiency, exploiting ECS commonalities and predicting their variability are mandatory. The concept of software product line engineering meets the business background of ECS. However, we should carefully investigate the expected technical, economical, and organizational effects of introducing this strategy into existing products.

This paper explains an approach for assessing the potential of merging existing embedded software into a product line approach. The definition of an economically useful product line approach requires two things: analyzing return on investment (ROI) expectations of a product line and understanding the effort required for building reusable assets. We did a clone analysis to provide the basis for effort estimation for merge potential assessment of existing variants. We also report on a case study with ECS. We package the lessons learned and open issues that arose during the case study.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Domain-specific architectures

General Terms

Design, Measurement

Keywords

Software Product Line, Engine Control Systems, Software Economics, Reverse Engineering, Clone Detection and Classification.

1. INTRODUCTION

1.1 Background

Figure 1 shows an overview of an ECS. The ECS is one of the core components for engine management systems. The ECS monitors engine status and driver requests, and controls the engine by regulating the amount of fuel injection, ignition timing, quantities of intake air, and so on. From a domain point of view, ECS share a significant portion of common properties; also, many future variations for different customers and market segments can be predicted in advance. However, embedded software in ECS was optimized to reduce hardware costs (i.e., microprocessor and memory chips). This optimization turned out to be not favor of component reuse in new products. Therefore, new ECS software was developed by "clone-and-own" from similar existing ECS.

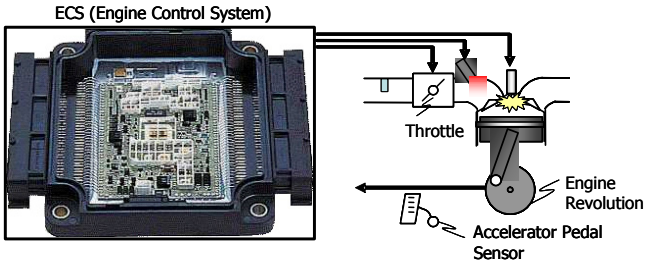

Figure 1: An overview of ECS.

With the increase of ECS business, the number of ECS variations (see Figure 2) has exploded and software development costs are increasing at an alarming rate. This situation and the business goals of ECS match scenarios for introducing product line engineering, which is a reuse- and architecture-centric paradigm that systematically takes advantage of commonalities and predicted variability [2] [4] [5] [8]. The core theme of product line engineering is to achieve systematic reuse by developing assets (e.g., common architecture, software components) that can be reused for a family of similar products.

Customer		A				B			
Engine Type		Direct Injection		Multi Port Injection		Direct Injection		Multi Port Injection	
Engine Size		1.6	2.0	1.0	1.6	2.0	3.0	1.0	2.0
Market	Japan	X	X	X	X	X		X	
	U.S.		X		X	X	X		X
	Europe	X	X			X		X	X

Figure 2: ECS variants - An example.

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

To that end, research in product line engineering has mostly focused on the construction of new product line infrastructures and activities: scoping, domain analysis, architecture creation, and variability management. On the other hand, existing products contain a lot of domain expertise and actual reliability. From an industry point of view, one of the most important issues is how to define future product lines from existing variants. This implies that introducing product line engineering often means merging the existing software from several similar systems into a common product line infrastructure. Unfortunately, there is limited or no support in the existing literature for assessing the merge potential of large industrial systems. The other important point issue is how to analyze the risk of introducing product line engineering into existing products. Identifying ROI in advance is mandatory from a management point of view.

We have developed the starting method [17], software clone analysis [18] and ROI estimation [8], for introducing software product line engineering. In this paper, we combine the results from the technical clone analysis and the ROI estimations to outline an economically useful product line approach for an existing domain of embedded software. Moreover, we apply the proposed method to the existing ECS embedded software.

1.2 Outline of This Paper

This paper is structured as follows: Section 2 describes an overview of the migration to a product line. Section 3 describes the ROI calculation for the investment in a product line. Section 4 describes our method for the merge potential assessment. The application of the proposed method to existing ECS is the topic of Section 5. Section 6 packages some important lessons learned and open issues we experienced. Section 7 describes our work in progress, which covers variability management at the implementation level. Section 8 provides related work followed by a conclusion.

2. Migration to a Software Product Line

In many organizations, similar systems are developed by separate groups. Usually, these systems have the same origin, but in order to satisfy the different requirements and schedules of individual customers, different projects develop the systems in parallel. As a consequence, reuse across systems is ad-hoc. In other words, the same copy of code is maintained by different groups, resulting in increased development costs. To solve this problem, we introduce a migration strategy.

Basically, the current development style will be improved by becoming a reuse-oriented style. In the future style, there are two activities, namely family or domain engineering and application engineering. In domain engineering, the requirements of the current and future products are analyzed and the common and variable parts are identified. Then a product line infrastructure containing reusable components is constructed. In application engineering, the individual products are constructed from the reusable components together with product-specific requirements.

Although the future style of software development looks simple and logical, a migration from the current process to the future process is very challenging. We consider the following to be the important challenges, faced by many organizations, in migrating from single system development to product lines: (1) Estimating economic benefits resulting from migration is difficult, and (2) addressing the immediate needs of customers is the main priority for managers, architects, and developers. Reasoning about long-term benefits from reuse and developing software with reuse are not given a priority. (3) It is not obvious in advance which components should be developed for reuse, and which ones should not. (4) Adapting existing systems for future reuse is yet another challenge. (5) How the quality of the resulting end products will be affected by reuse-oriented software development is an important question. (6) Organizational issues like funding and management structure of domain and application engineering groups should be solved as well. (7) How to shorten the overhead of the migration step is another issue.

Despite the above practical challenges, organizations are willing to migrate to product line engineering in order to reduce the development costs and time-to-market. Hence, a systematic migration strategy, which takes into account the above mentioned difficulties, should be developed first to ensure smooth and successful migration.

Figure 3 shows a merge strategy. The strategy includes answers to the following questions: From an organizational point of view: (a) what is the economic benefit for target products, (b) how to re-define the development process, and (c) how to restructure the organization for successfully merging the existing implementations into a product line. From a technical point of view: (d) how to assess the merge potential of the existing software variants, and (e) how to perform merging the existing implementation. Questions (a) and (d) are the topics addressed in this paper.

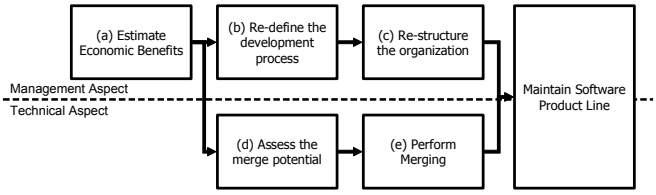

Figure 3: An overview of a migration process.

3. Predicting ROI for Introducing Software Product Line Engineering

First of all, managers have to decide whether they should introduce software product line engineering into their development process. Software product line engineering is often an economical development method for software series. However, the method is not "the silver bullet". For example, when only one software product is developed, nothing can be gained from software product line engineering. It is worth estimating ROI using a software economics model before introducing the product line development method.

We have developed a method for predicting ROI with uncertainty of software development [8] based on Monte-Carlo simulation. The simulation model was carried out using the economic model introduced in [4]. To practically perform the Monte-Carlo simulation, we used Crystal Ball, a commercial tool. The following steps apply Monte-Carlo simulation with the product line economic model with uncertainty:

Step 1 – All input variables for which experts cannot provide sufficiently accurate predictions are identified as uncertain variables.

Step 2 – Each of the uncertain variables identified is then mapped to a suitable probability distribution, which defines the range of accepted values for each variable and a function specifying how likely a particular value is to occur.

Step 3 – Random input numbers for uncertain variables based on the selected probability distribution are generated. The ROI is calculated for each set of random input values. That is, many ROI estimations may be computed.

Step 4 – The computed ROI predictions are put together and a frequency distribution is constructed, which tells how likely it is to achieve certain ROI values or ranges through product line engineering.

Regarding step 2, the commonality level among the products, for example, is mapped to a Normal distribution with a mean of 60% (i.e., on average, products have 60% in common). The probability distribution thus defines that it is equally likely to experience a commonality level below as above the average within defined lower and upper boundaries. Normal distribution is, however, only one of many possible probability distributions, which thus does not hold in general for all uncertain variables. The number of products in a product line, for example, is mapped to a Uniform distribution. Note that the most suitable mappings may be different in different contexts. We refer the readers to [8] for more detailed discussions on the economic model and the simulation of ROI.

4. Merge Potential Assessment Method

We propose a process for merging existing products as shown in Figure 4. The first step in that process is to analyze ROI of a software product line for an existing product family. We can get benefits from product line engineering if the software systems have certain kinds of characteristics, such as commonality, number of products, size of software, and so on. In this step, we analyze whether we should introduce a software product line or not. The next step is to assess the merge potential of existing products. Since all products are assumed to have the same conceptual software architecture, this is used as the reference point for comparison. Hence, we assess the merge potential of every component in each existing product by using software architecture decomposition. Once the component is assessed for its merge potential, the next step in our process model is to perform the actual merging of the component in different products. That is, the component is transformed into a generic reusable component together with current and predicted future variants. After transforming a component into a reusable entity, existing individual products should be adapted to use it. Adapting existing products involves many technical activities like restructuring the build process, directory structure, configuration management, testing.

The main target of our research is on proposing an automatic assessment method for existing systems. Since today's embedded control software encompasses huge systems, automatic analysis of commonalities and variabilities is mandatory. To assess the commonality between the systems, we can check 3 layers, namely the Requirement Level, the Executable Model Level, and the Source Code Level. We focus into source code level, because the existing systems were not developed by Model-Driven Development. (Some part of system is developed by MDD.) Requirement level commonality analysis might be good idea, but the current requirements of an embedded control system are not formally specified. This means we are not able to compare the requirements automatically.

In the next section, we explain how the merging process shown in Figure 4 is applied to target existing software products.

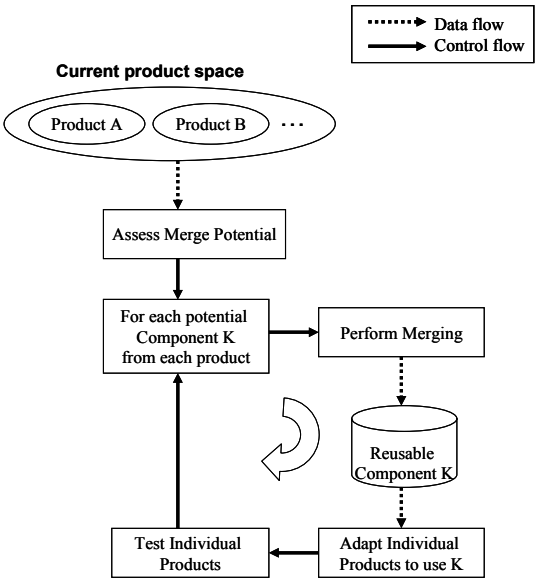

Figure 4: Process for iterative technical merging.

4.1 Implementation level Commonality Analysis

The main topic of this section is the analysis of clone data from product A to product B. In order to interpret the collected clone data for assessing the merge potential, we propose a hierarchical clone analysis approach. Figure 5 shows an overview of our approach.

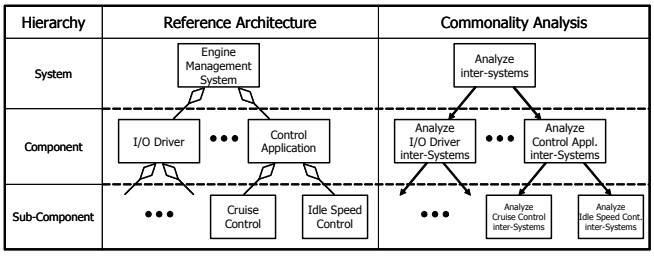

Figure 5: Clone analysis using decomposition hierarchy.

First, we assume that both products A and B have one monolithic component, and analyze the clone classification. Next, we analyze the clone classification of each component from product A to product B, based on a reference. Then, we continue to analyze the sub-components within a component. In short, our clone analysis for merge potential assessment is carried out at different levels of abstraction using the decomposition hierarchy shown in Figure 5.

4.2 Commonality Analysis Method

4.2.1 Definition

First, we define clone and clone coverage before explaining the details behind clone analysis to assess the merge potential of software product variants.

Clone: Figure 6 shows an inter-system clone pair. Two code fragments form a clone pair if their program text is similar. In our approach, we restrict this to clones between functions, that is, clones from a function in product A to a function in product B. The main reason for this restriction is that in the latter phase, in

order to resolve clones, we can replace existing cloned functions with generic functions that can be instantiated for each product. We used a commercial tool called *CloneFinder* for finding cloned code functions. Clonefinder can find clones that are either exact copy from one product to another or a copy with some modifications (e.g., renamed function). However, this tool does not classify clones into the different types defined below. We wrote some wrappers around the tool and extracted the different types of clones. It is worth clarifying that the level of granularity for reuse is not at the function level, but at the component level. Using the *clone coverage* metric, we measured the commonality level among components of existing products.

Clone Coverage: Let J and K be two components. Then the clone coverage in K from J is defined as follows:

$$CloneCoverage(K) = \frac{\#of\ Lines\ Cloned\ \ from\ J}{\#\ of\ \ Lines\ in\ K} * 100$$

Interpretation of Clone Coverage: If *CloneCoverage(K)* is near 100%, it means that nearly all the lines in K are cloned from J, and if it is near 0%, that means there is hardly any text similarity with J. This clone coverage metric can be applied at any level of abstraction. That is, we can compute clone coverage from one product to another product, and then to the next level of the product decomposition hierarchy. From now onwards, the number of lines in a component refers to the sum of the numbers of non-commented lines in each function within the component.

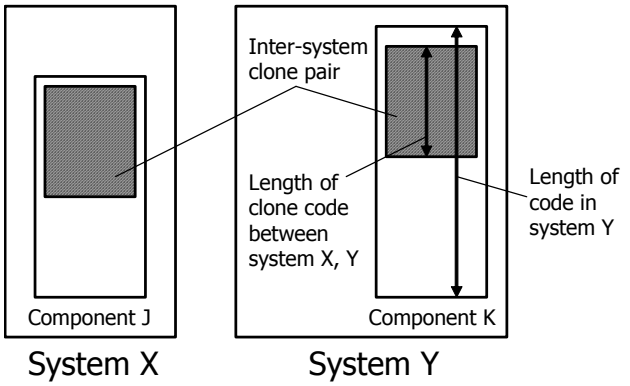

Figure 6: Inter-system clone pair.

4.2.2 Clone classification

To facilitate the merge potential assessment, we propose classifying clones from product A to B into different types as follows. Please note that we will not discuss clones within product A or B; all discussions about clone analysis are from product A to B.

Type 1: Exact interface and implementation copy from product A to product B. Figure 7 is an example of a type 1 clone.

```
int foo(int j) {
  if (j < 0)
    return j;
  else
    return j++;
}
```

Product A

```
int foo(int j) {
  if (j < 0)
    return j;
  else
    return j++;
}
```

Product B

Figure 7: An example of a clone of type 1.

Type 2: Interface copy, but the implementation is modified to satisfy product-specific requirements. Figure 8 is an example of a type 2 clone.

```
int foo(int j) {
  if (j < 0)
    return j;
  else
    return j++;
}
```

Product A

```
int foo(int j) {
  if ( j <= 0 && j >= -5 )
    return j;
  else
    return j++;
}
```

Product B

Figure 8: An example of a clone of type 2.

Type 3: Only the interface is copied, but implementation differs too much, so that our common sense will consider it as different code (see Figure 9).

```
int foo(int j) {
  if (j < 0)
    return j;
  else
    return j++;
}
```

Product A

```
int foo(int j) {
  return j--;
}
```

Product B

Figure 9: An example of a clone of type 3.

The difference between type 2 and type 3 clones lies in the choice of the threshold for the clone coverage rate. Type 3 clone is introduced especially to identify variable parts in the implementations.

Type 4: Interface is renamed, but the implementation is cloned (see Figure 10).

```
int foo(int j) {
  if (j < 0)
    return j;
  else
    return j++;
}
```

Product A

```
int goo(int j) {
  if ( j < 0 )
    return j;
  else
    return j+2;
}
```

Product B

Figure 10: An example of a clone of type 4.

Note that with the above four types, we have considered all possible function clones, and not ignored any other type of function clones. The motivation for classifying clones into Type 1, Type 2, and Type 3 was to understand and identify the common and variable parts in the implementations of products A and B quickly. Type 4 was defined in case programmers renamed the interfaces but cloned the implementation from one product to another.

To merge the existing systems, we need to increase type 1 clones, reduce type 2 clones, and keep type 3 clones only if the product needs the same interface but a different implementation, and move type 4 clones into type 1. Since existing clone detection tools can not provide us with clone classification into the above four types, we developed our own tools for classifying clones. Due to space limitations, we skip our algorithm for classifying clones. In short, given two systems, our algorithm can classify function clones into the above four types.

5. Case Study: Engine Control Systems

5.1 Overview of Case Study

In this section, we apply the proposed process with a case study to assess the merging potential of two ECS products for customers A and B. The current products were taken from an initial version, and different groups were formed to address the needs of the global market. Although these products share a common conceptual architecture, their implementation and maintenance are controlled by different groups. Hence, deriving a merging strategy was a wise decision before introducing a product line.

To assess the merge potential of ECS products, we used the software architecture as a reference point. We assumed that target ECS products share the architecture shown in Figure 13. We compare and assess the merge potential of a component in product A with the same component in product B. To support this assessment, we analyze the product level. Next, we analyze the component level and the sub-component level commonality. After that, we plan a merge strategy for each sub-component. Finally, we discuss the result of the clone analysis from the domain point of view, using the proposed method.

5.2 Predicting ROI

As the first step of introducing a software product line, we estimate the ROI of ECS case. We selected uncertain input variables as follows:

- Number of products
- Commonality level of core asset base
- Fraction of core asset base difficulty
- Fraction of core asset base that changes with each new version of the product line
- Rate of building each product's unique part

For example, the commonality level of core asset base is mapped to normal distribution with a mean of 70%. The type of distribution is selected by domain experts using metric data of existing products. The specific number of input variables can not be disclosed due to company confidentiality reasons.
Figure 11 and We can observe that it is 70% certain that we will take more than 64.0% ROI. This result means that characteristic of ECS meets software product line engineering and the organization will take benefit by introducing product line engineering.

In the following subsections, we analyze the merge potential of ECS product variants.

Figure 12 show the result of Monte-Carlo simulation after 3 ECS products in software product line strategy. To estimate the distribution of ROI, we have simulated 20,000 times. In Figure 11, the horizontal axis shows ROI and the left vertical axis means probability.

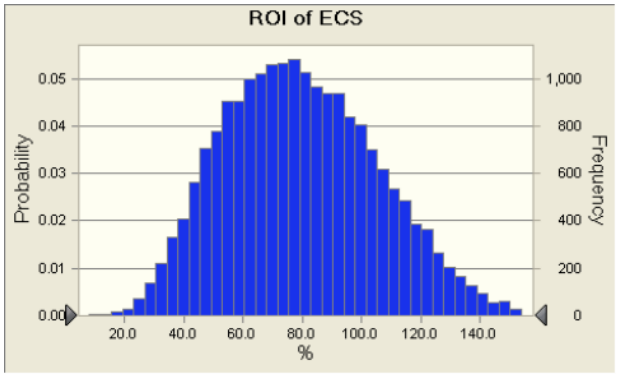

Figure 11: Distribution of estimated ROI of ECS (after 3 generations).

We can observe that it is 70% certain that we will take more than 64.0% ROI. This result means that characteristic of ECS meets software product line engineering and the organization will take benefit by introducing product line engineering.

In the following subsections, we analyze the merge potential of ECS product variants.

Figure 12 shows the result of the clone coverage analysis of the product view. In this case study, if the clone coverage rate of function f of product B from product A is less than 20%, we consider function f to be type 3.

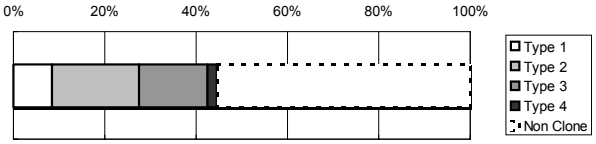

Figure 12: Clone coverage from product A to B.

In the case of the analyzed ECS products, lines of code of type 1 clones in product B from A cover around 9% of all function code in B. We noticed that type 2 clones in product B from A cover around 19% of all function code in B. Ultimately, we would like to reduce type 2 clones by separating common and variable parts, thereby reducing code duplication and introducing systematic reuse. Type 3 clones also exist in our current products. The existence of type 3 clones, in our case, has two reasons: a) some portions of ECS are implemented by different groups, but the interface was reused from the initial root version, and b) product-specific functionality implementation was needed, but with the same interface for both products. For product line migration, in order to avoid code duplication, type 3 clones should be kept only if products require different implementations but with the same interface. We had very little type 4 clones ,which means that programmers have not changed function names from product A to B. 55% of function code in product B is not a clone at all. That is, 55% of function code in product B has a different implementation than in product A.

We can observe from Figure 12 that type 1 and type 2 clone coverage from product A to B is around 28%. This result shows that a part of ECS can be merged and another part cannot be merged. To understand this issue more clearly, we used the hierarchical clone coverage view introduced earlier. In the next subsection, we analyze which components of the architecture are

implemented in a different style, and which components have high clone coverage from product A to B.

5.3 Clone Coverage: Component View

In the previous subsection, we have shown the clone coverage view from product A to product B. This view is at a high level of abstraction, and is only useful for understanding the merge potential from the system level. That is, Figure 12 does not contain any information about the architectural components of ECS. Ideally, we would like to know the clone coverage per component so that the component merging potential can be assessed. But the difficulty lies in the abstraction level: Architectural components are not directly visible in the source code, but the clone detection results are always at the code level and not at the component level.

To solve this problem, we employ mappings as done in the reflexion model [12]. That is, we map the abstract components to source code for both products from the domain point of view. Figure 13 shows the reference architecture of this case study. For example, every file under the IO_Driver directory belongs to the IO_Driver component. Using this mapping, we lifted the collected clone data to the component level. This reference architecture is based on the AUTOSAR (AUTomotive Open System ARchitecture) software architecture [13]

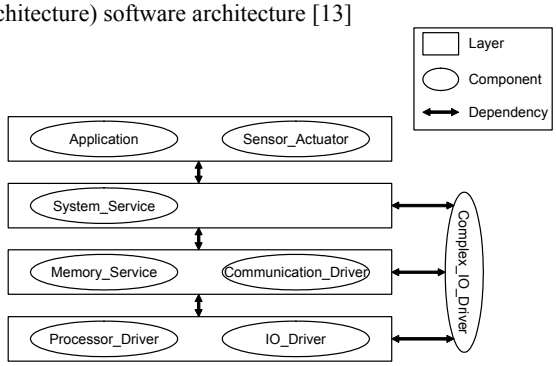

Figure 13: Software architecture of ECS products.

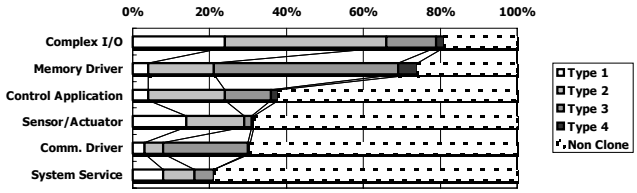

Figure 14: Component level clone coverage.

Figure 14 shows the clone coverage per component from product A to product B. Using this view and the domain knowledge of the architect, we reasoned about the clone coverage for each ECS component. In this subsection, we present the analysis of clone coverage at the component level for the components of ECS.

The Memory_Service, Sensor_Actuator, and Communication_Driver components implement product-specific functionalities, and hence low type 1 clone coverage (below 15%) reflected this scenario.

The Memory_Service component has around 5% type 1 clone coverage, because it implements a functionality related to flash memory operations, which is mainly supplier dependent. As a result, the implementation of Memory_Service in product A is significantly different from product B. Also, around 50% of the Memory_Service component code is type 3 clone. This is because for both products, the external interfaces of Memory_Service are the same, and hence interfaces are reused from the initial root version of ECS.

For the Complex_IO_Driver component, type 1 clone coverage is around 25%. This matches our estimation because this component is "complex" and the developers tried to maintain commonality. However, we can notice that the type 2 clone coverage is around 35% for this component. We plan to resolve type 2 clones in future.

The System_Service component implements system level service routines, and hence it is mostly product-specific. We can see from the clone coverage view that around 80% of System_Service code is not a clone.

There were also some unexpected surprises in the clone coverage results. For example, the Application component of ECS has only 5% type 1 clone, but our expectation was around 30% to 40%. From the domain point of view, the Application component in both products contains common domain concepts, but the clone coverage metric does not show a high commonality. To understand the reason for the differences, we analyzed the clone coverage per sub-component within the Application component.

5.4 Clone Coverage: Sub-Component View

Figure 15 shows the clone coverage of sub-components in the Application component. The Application component consists of 9 sub-components. From Figure 15, we can tell the clone distribution for the sub-components in the application component.

We discuss a merge strategy of software components based on this assessment result in the next subsection.

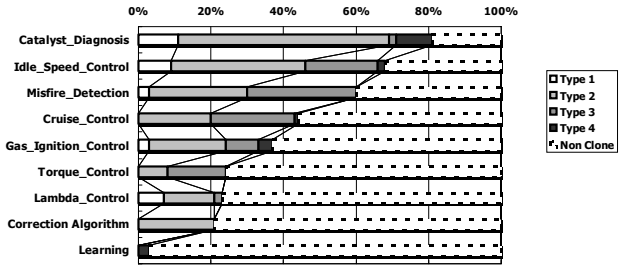

Figure 15: Clone coverage for Application sub-components.

5.5 A Merge Strategy of Sub-Components

Engine_Gas_Injection_Control is a traditional component with stable requirements for the engine control systems. But there are also some differences or variations from one car model to another model. Nevertheless, this component should be merged and transformed into a generic component with variation points. In Figure 15 we can notice that type1 and type 2 clone coverage for the Engine_Gas_Injection_Control component is low: our expectation was at least 50% from the domain point of view. In this case, our merge plan is to transform the Engine_Gas_Injection_Control component from the latest version, which is product A, into a generic component with variation points, which can be instantiated for product B and other future products.

Similarly, the requirements of the Idle_Speed_Control component are stable for engine control systems. The type 1 and type 2 clone coverage for this component from product A to product B is around 50%, which already gives us an indication that this component can be transformed into a generic component. In this case, our merge strategy is to merge this component from product A and product B by first separating common and variable parts from both implementations.

The functionality of Torque_Base_Control shares significant commonalities among products A and B. However, clone coverage was low (around 80% are non-cloned code) because the root version of ECS did not contain this component, and in the latter stage it was implemented in different styles by developers belonging to different groups. To merge this component, it is not rational to compare its code because there are much more code differences than functionality differences. Therefore, we will follow the same merge strategy as for the Engine_Gas_Injection_Control component.

The Cruise_Control component has 0% type 1 clone, and around 60% are non-cloned lines. Cruise_Control is an unstable component and not traditional with respect to engine control software; rather, it belongs to the vehicle control domain. Therefore, we will not give priority to merging the implementations of this component into a generic component.

For the component Misfire_Detection, type 3 clone coverage is around 35%. This means that the same application framework is used in both product A and B; however, the implementations are different for specific customers. In this case, we will integrate only the application framework. We will not try to merge the implementations of these components into generic reusable components.

The Learning component does not have any clones from product A to B, because the learning behavior is different from one car model to another. Hence, these components are also no candidates for merging into generic reusable components. In this case, we will keep variability at the component level (i.e., we will select different learning components for different car models).

Our merge strategy is to transform the components, namely, Idle_Speed_Control, Torque_Base_Control, and Engine_Gas Injection_Control into generic reusable components for the ECS products.

5.6 Discussion of Case Study Result

We have shown that for two ECS products, type 1 and type 2 clone coverage from product A to product B was only 28%. Although these products have a significant amount of commonalities, the clone coverage does not reflect the domain view. As mentioned earlier, products A and B have a common origin, but started evolving separately to address different market segments (see Figure 16). In addition, these products are controlled by developers who belong to different groups.

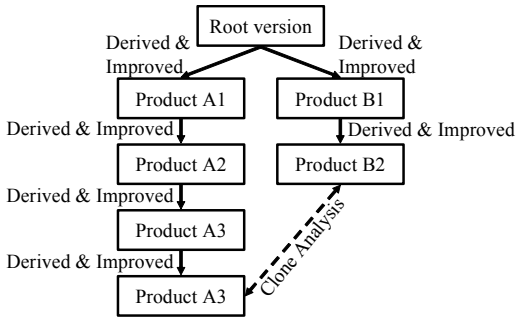

Figure 16: Evolution tree of ECS products A and B.

The clone analysis was performed on the two latest versions of product A and product B, and analyzing the evolution history was not in the scope of the project due to organizational issues. We did some additional analysis to understand the reasons for low clone coverage, and found two activities with respect to product A: a) around 30% of product A's code was generated automatically using model-driven development, b) some portion of the existing assembly code in product A was migrated to the C language. These two activities were not performed in product B. As a result, the code in product A and product B is textually different and hence, low clone coverage occurred.

Another reason for low clone coverage from product A to B is due to the ECS domain itself. ECS is a mixture of multiple hardware parts, mechanics, and software. Also, there are market-specific regulations, too, for example, emission rules are different in Japan, Europe, and the United States. To handle all these issues, developers in different groups tend to change existing code in various ways, and when more and more requirements have to be handled in a sequence of releases, the code commonality among similar products of the same origin tends to shrink.

6. Lessons Learned and Open Issues

6.1 Lessons

In this subsection, we share a few lessons, that we believe will be of interest to other practitioners and researchers.

Software cloning may not be a good way to realize product line engineering: Software clones might be good to quickly realize a first few variants. Later, due to organization and technical reasons, clones will disappear, and organizations will have more or less independent products. So, it is our position that software cloning is not an economically sound solution from a long-term point of view.

ROI predictions can strongly motivate the management to invest in product line engineering: In practice, without strong support of the management, it may not be possible to introduce product line engineering to an organization. Hence, it is wise to first motivate the management by showing the economic benefits. ROI figures support the management in the decision-making process.

Architecture-centric clone analysis is a useful and practical approach to assess the merge potential of the existing systems: Software cloning occurs at the implementation level. Hence, the measurement of cloning also occurs at the implementation level. However, industrial software contains thousands of files, making it almost impossible to reason about software clone distribution. Therefore, clone measurement and analysis should be raised to the architecture level. Moreover, the architecture-centric clone

analysis supports the understanding of clones from a semantic point of view, because architecture is nothing but a domain abstraction.

Clone coverage is a sound toolkit to motivate the technical audience to do product line engineering: Before introducing a product line, it is often necessary to convince the developers why traditional development is not good. And moreover, the reasons why software cloning is not a sound way to implement product line engineering should be clarified. In our case, we have shown the clone distribution of the existing variants to the developers and conveyed the key message that the products, which were basically clone-and-own from a root version, tend to lose the clones quickly. As a consequence, the clones disappear and the products become more or less completely stand-alone, although the products have so much in common with respect to the domain. To avoid this problem, a more disciplined approach to reuse has to be introduced.

Domain experts and reverse engineering experts need to work together: We have shown that, by employing reverse engineering techniques, industrial-strength software variants can be analyzed for commonalities and variabilities. However, on the one hand, reverse engineers alone cannot solve the challenge of merging the existing software variants. They lack the domain knowledge. On the other hand, domain experts may not be aware of reverse engineering. So it is important that both the domain experts and the reverse engineers work together to successfully merge the existing variants into a product line.

6.2 Open Issues:

In this subsection, we share a few important open issues, which need further research to support the merging of existing variants into a product line.

Clone visualization: It is commonly accepted that visualization supports humans in understanding large data sets. However, most visualization research is concerned with visualizing software clones that are present in a single system. How to visualize clones across systems is not well-addressed. It would be useful for practitioners to visually assess the merge potential of the existing variants, based on the clone distribution.

Clone refactoring: How to remove clones is not an easy question. Some of the existing clone detection tools replace the clones with macros or preprocessing statements. However, the scope of such refactoring is restricted to a single system. Also, macros are not always the best implementation technology for implementing the variants, given that it is not type checked. If more than one variant exists, how to refactor the detected clones is not trivial.

Clone error reduction: Clone detecting tools usually compare the source code for syntactically similar patterns. However, the problem is that syntactically similar patterns are not always the same semantically. As a result, not all clones are really clones. That means, there might be false positives in the detected clones. How to reduce false positives in clone detection is an open problem, whose solution is of interest to practitioners.

Refactoring effort estimation: This is rather a business issue. Once the clones are detected, we should remove them. This task requires effort. How much effort is needed to remove clones is of interest to managers. Currently, there is no support to answer such questions.

7. WORK IN PROGRESS

7.1 Classification of Variability

Once the merge potential is assessed using the clones and the domain concepts, we need to plan for resolving the clones so that code duplication is reduced and systematic reuse is in place.

Type 1 clones need not be reviewed because the code is textually the same in both products. But to resolve type 2 clones, we first need to understand the nature of the difference between clone pairs. The difficulty is that from the implementation-level differences, we can not conclude that the component contains some variability. To solve this problem, at least partially, we used the knowledge of our architect to reason about differences. But the challenge is the effort required to analyze each clone pair for the component. Therefore, for now, we focused on clone pairs with high clone coverage (more than 75%).

Figure 17 shows the distribution of clone review results for the sub-components of the Application component of ECS. Around 21% of the reviewed clone pairs contain variants. That is, some portion of code was modified to support product-specific requirements. An example of a variant: In the case of product A, the number of cylinders is fixed, but in product B, there can be a variable number of cylinders. To resolve such kinds of variants, we may use configuration files that specify the number of cylinders.

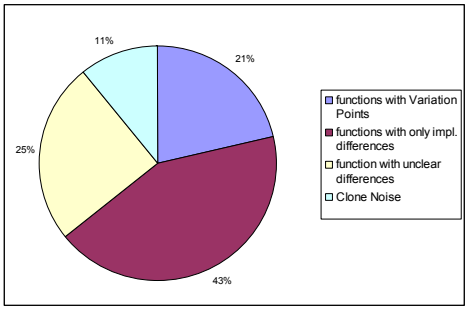

Figure 17: Classification of reviewed type 2 clones.

We can notice that around 40% of the reviewed type 2 clones contain implementation-level differences that are not related to variants. The differences fall into different categories: a) Change of data types, for example, *int* type to *short int* type; these kinds of changes were performed because one programmer thinks *int* is enough, but another programmer in a different group later realizes *short int* is better; b) Change of variable or array names or library routines, and c) Change of programming style; for example, some programmers like to have "{" in the same line with the *if* statement, others like to put "{" in a new line. Currently, we are developing approaches for classifying the implementation-level differences into different categories to support architects and to reduce the effort for clone pair review.

We also noticed during the clone review that the architect cannot find the reason for differences in the clone pair from the domain point of view. We mark this situation as unclear, and plan for discussion with the developers in future. The noises produced by clone detection tools are marked as "Clone Noise" in Figure 17. Noises refer to those pairs of clone that are reported as clones by tools, but where the architect disagrees with this detection because it is not actually code duplication.

7.2 Selection of Variability Implementation Techniques

Once the clone pairs are identified and reviewed, we should resolve them systematically by employing appropriate variability implementation techniques. In our case, ECS is implemented mainly in the C language, and there are many ways one can realize variants, for example, using macros, conditional compilations, dynamic linking, etc. The obvious question is which variability implementation technique one should choose from the collection of existing ones. For example, in some cases, we can use macros, and in other cases we can use conditional compilations.

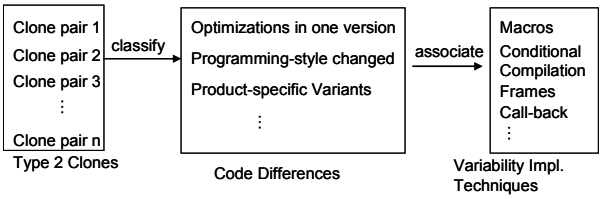

Figure 18: Process for resolving clone of type 2.

In [3], the authors replace clones with macros to reduce the quantity of source code and facilitate maintenance. We consider macros as being one of the variability implementation techniques because they are type-checked and hence might behave unexpectedly, which is not desirable for a safety-critical system like ECS. The choice of variability implementation technique is not just a technical strategy. That is, we have to choose a technique that is suitable for the application domain, as well as being known or familiar to developers. Therefore, we believe it is an interesting problem to resolve clones of type 2 into appropriate variability implementation techniques. Figure 18 depicts this problem.

8. RELATED WORK

There are many works about product line engineering for automotive software (e.g., [14] [15]). Researchers have focused mainly on requirements, variability management, and architecture design for product lines. In contrast, our approach is a mixture of bottom-up reverse engineering with top-down software architecture to migrate to a product line.

SIMPLE (The Structured Intuitive Model for Product Line Economics) [6] is an economics model for software product line engineering. However, its focus was mainly on discussing several scenarios of introducing software product line into a software development organization. We estimated the economic benefit of a software product line with uncertainty of input valuables. Simulating ROI predictions for different scenarios is not in the scope of SIMPLE, but is already addressed by our approach.

AUTOSAR [13] is a consortium to establish open standard software architecture for vehicles. The AUTOSAR goals include reusability of software modules in vehicles. However, the standard is a kind of application framework. In the future, we will explore the connection between AUTOSAR and a product line for our ECS.

In [1], an assessment of reengineering opportunities (e.g., parameterization, delegation) based on clone information has been investigated by classifying clones into different types. However, their focus was on resolving clones within a version to facilitate software maintenance. We also classified clones into different types to assess the merge potential of existing products.

In [7], the authors located the common and variable parts within a product using clones. We used clone detection tools to derive the merge strategy by analyzing clones across products using software architecture decomposition. Another important difference is that in our case, we also captured interface cloning, which is particularly effective in the context of product line migration to identify the variable parts in the implementation.

In [10], the authors described the refactoring activities performed to migrate the Image Memory Handler (IMH) of current products of office appliances into a reusable product line component. Their major focus was on improving reusability and handling the variants by introducing modularity by resolving clones. Our objective is also to improve reusability. Their clone analysis is done only on the exact implementation copy within a version, but our clone analysis includes copy-and-paste-modified from one variant to another variant.

In [9][16], an approach for comparing programs was proposed. These research results are, in fact, complementary to our approach. In principal, a program P from product A can be compared to a program Q in product B by using program comparison approaches to identify common and variable parts. However, the problem of merging two programs is comparatively easier than merging two systems, which are made up of many programs.

9. CONCLUSION

In this paper, we proposed an approach to assess the potential to merge existing systems into a product line. First, we proposed an approach to estimate the economical benefit of software product line engineering with uncertainty of software development in the future. Next, we proposed a method to identify the commonality of current implementations across the "future" software product line variants.

In the case study of Engine Control Systems (ECS), we observed an alarming lesson from the investigation reported above: products derived from the same origin by different teams lose identical parts much quicker than necessary and thus also than expected. That is, many conceptually identical requirements are implemented in different, often inconsistent ways, which practically prevents merging them later to share and save maintenance effort in the future. In our ECS case, the portion of functional commonality among two products is about 60-75%; their implementations, however, share as little as around 30% of code. From our point of view, the following requests in the engine control domain are responsible for the low implementation level commonality: continuous demand for new features, integration of diverse configurations of varying hardware, software and mechanical parts, and uncoordinated concurrent development of similar features by different teams.

We will extend our work by addressing organizational-aspects to better support the practical execution of migration strategies based on identified technical merge strategies. In any case, before "blindly" applying product line engineering in real teams and projects, organizations must carefully investigate the expected technical, economical, and organizational effects.

Acknowledgement

It is our pleasure to thank the Hitachi Research Laboratory and Engine Management Systems business unit for fruitful

discussions on product line engineering. Thanks to Sonnhild Namingha of Fraunhofer IESE for editing the final version of this paper. Last, but not least, the insightful comments from the anonymous reviewers helped us to improve the paper.

10. REFERENCES

[1] M. Balazinska et al. Measuring clone based reengineering opportunities. *Proc. of Sixth Intl. Soft. Metrics Symposium*, 292 – 303, 1999.

[2] J.Bayer et.al. PuLSE: A Methodology to Develop Software Product Lines. *Proc. of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99)*, 122–131, Los Angeles, CA, May 1999.

[3] I.D. Baxter et al. Clone detection using abstract syntax trees. *Proc. of the 2nd Working Conf. on Reverse Eng (WCRE)*. IEEE Computer Society Press, July 1995.

[4] G. Böckle, P. Clements, J.D. McGregor, D. Muthig, and K. Schmid. Calculating ROI for Software Product Lines. *IEEE Software*, 21(3), 23-31, June 2004.

[5] A. Childs, J. Greenwald, G. Jung, M. Hoosier and J. Hatcliff. CALM and Cadena: Metamodeling for Component-Based Product-Line Development. *IEEE Computer*, 39(2), 42-50, Feb 2006.

[6] P. C. Clements, J. D. McGregor, S. G. Cohen. The Structured Intuitive Model for Product Line Economics (SIMPLE). *Technical Report CMU/SEI-2005-TR-003*, 2005.

[7] P. C. Clements and L. M. Northrop. Salion, Inc. A Software Product Line Case Study. *Technical Report CMU/SEI-2002-TR-038*, 2002.

[8] D. Ganesan, D. Muthig and K. Yoshimura. Predicting Return-on-Investment for Product Line Generations. *In Proc. of the Int. Conf. Soft. Prod. Lines (SPLC),* 2006.

[9] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *ACM SIGPLAN Notices*, 25(6), 234-245, 1990.

[10] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A Case Study in Refactoring a Legacy Component for Reuse in a Product Line, *In Proc. of Int. Conf. on Soft. Main. (ICSM*), 2005.

[11] C. W. Krueger and D. Churchett. Eliciting Abstractions from a Software Product Line. *In Int. Work. on Product Line Engineering The Early Steps (PLEES): Planning, Modeling, and Managing*, 2002.

[12] G. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. *In Proc. of Third ACM SIGSOFT Symp. on Foun. of Soft. Eng*., 18-28, 1995.

[13] Th. Scharnhorst et al. AUTOSAR – Challenges and Achievements 2005. *Proc. of the Int. Conf. Electronics Systems for Vehicles*, 395-408, Baden-Baden, Germany, Oct. 2005.

[14] M. Steger et al. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. *In Proc. of the Int. Conf. Soft. Prod. Lines (SPLC)*, 2004.

[15] S. Thiel et al. A Case Study in Applying a Product Line Approach for Car Periphery Supervision Systems. *SAE World Congress 2001: In-Vehicle Software, SAE Technical Paper* 2001-01-0025, March 2001.

[16] W. Yang. Identifying Syntactic Differences Between Two Programs. *In Soft. Practice and Exp.,* 21(7), 739-755, 1991.

[17] K. Yoshimura, J. Bayer, D. Ganesan, D. Muthig. Starting a Software Product Line by Reengineering a Set of Existing Product Variants. *Proceedings of Society of Automobile Engineers World Congress (SAE 2006). In In-Vehicle Software Session*, 2006.

[18] K. Yoshimura, D. Ganesan and D. Muthig. Assessing Merge Potential of Existing Engine Control Systems into a Product Line. *ICSE Workshop on Software Engineering for Automotive Systems (SEAS)*. Shanghai, China, May 2006.