Assessing Merge Potential of Existing Engine Control Systems into a Product Line

Kentaro Yoshimura
Hitachi Europe
Automotive Research and Development Laboratory
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 many variants for many manufactures and several markets. To improve their development efficiency, exploiting ECS commonalities and predicting their variability are mandatory. The concept of software product line engineering meets this ECS business background. However, we should carefully investigate the expected technical, economical, and organizational effects of introducing the strategy into existing products. Thereafter, a strategy will be derived systematically and realize the desired benefits.

This paper reports an experience with the up-front investigation performed for Hitachi's ECS. We focus on the approach to plan the migration of the existing family of individual systems into a future product line. The approach assesses potential ways of merging software from existing variants and eventually defines a procedure for performing the migration. To get a high quality strategy, we integrate the approach of software measurement, the expertise of software architects, and reverse engineering techniques.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Domain-specific architectures

General Terms
Measurement, Reuse

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SEAS'06, May 23, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005...$5.00.

1. INTRODUCTION

1.1 Background

Hitachi releases – besides many other products – many variants of engine control systems (ECS) to many car manufactures. Figure 1 shows an overview of an ECS (Engine Control System). An ECS is one of the core components for engine management systems. An ECS monitors the engine status and the driver request, 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, the 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 made it too strong to reuse as generic components in new products. Therefore, new ECS software was developed by "clone-and-own" from similar existing ECS.

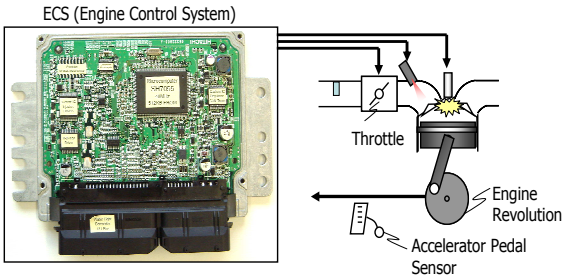

Figure 1: An Overview of ECS.

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 [1] [3]. The core theme of product line engineering is to achieving systematic reuse by developing assets (e.g., common architecture, software components) which can be reusable 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.

To that end, the 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, how to define the future product lines from existing variants is one of the most important issues. 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. In this paper, we discuss on the merge potential assessment to support planning the technical transformation of the existing system family of individual systems into a future product line.

To asses merge potential and hence derive the merge strategy of existing systems we make a few assumptions which we believe are common in industry. First, we assume that existing systems have conceptually the same architecture. Second, existing systems have been developed using "clone-and-own" or copy-and-paste. We discuss on clone classification and a migration strategy to product line based on the clone analysis results.

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 our method for the merge potential assessment; the application of proposed method to Hitachi's ECS is topic of section 4. Section 5 provides related work. We conclude and discuss the future work in section 6.

2. Migration to 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 schedule 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 into the 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 due to reuse and developing software with reuse is not given a priority. (3) It is not obvious to know in advance which components should be developed for reuse, and which ones should not be. (4) Adapting existing systems to reuse generic components is yet another challenge. (5) How the quality of the resulting end products will be affected due to the 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.

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 for smooth and successful migration.

A merging strategy includes answers to the following questions: (a) What is the merge potential of existing implementations, (b) how to perform the actual technical merging in an efficient way, and (c) how to restructure the organization for successfully merging existing implementations into a product line. It is worth clarifying that only the first question is the topic addressed in this paper. That is, We propose an assessment method to know the merge potential of existing implementations.

3. Merge Potential Assessment Method

3.1 Overview

We propose an iterative process for merging of existing products as shown in Figure 3. The first step in that process 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 the 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, etc. and Testing.

The main target of our research is to propose an automatically assessment method for existing systems. Since engine control systems are huge systems, the automatic analysis is mandatory. To assess the commonality between the systems, we can check 3 layers namely The Requirement Level, Executable Model Level and Source Code Level. We focus into source code analysis 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 requirement of ECS is not formally specified. This means we are not able to compare the requirement automatically.

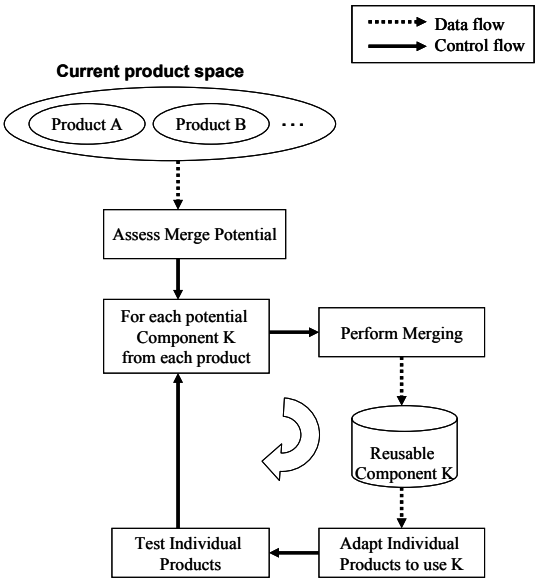

Figure 3: Process for iterative technical merging.

In the next section, we explain how the merging process shown in Figure 3 is applied to Hitachi ECS.

3.2 Hierarchical Clone 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 4 shows an overview of our approach.

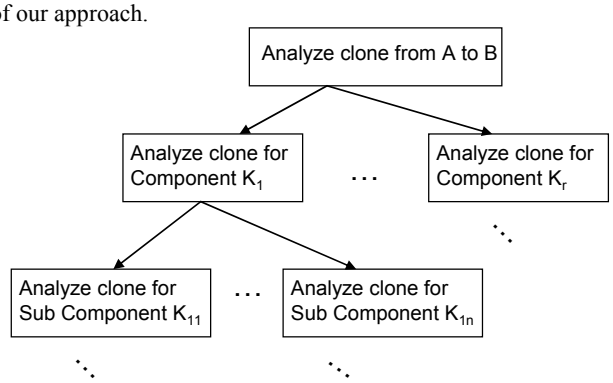

Figure 4: Clone analysis using decomposition hierarchy.

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 continued to analyze the sub-components within a component. In short, our clone analysis for merge potential assessment was carried out at different levels of abstraction using the decomposition hierarchy shown in Figure 4.

3.3 Commonality Analysis Method

3.3.1 Definition

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

Clone: 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.

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.

3.3.2 Clone classification

To facilitate the merge potential assessment, we propose to classify 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 5 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 5: An example of a clone of type 1.

Type 2: Interface copy but the implementation is modified to satisfy product specific requirements. Figure 6 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 6: 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 7).

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

Product A

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

Product B

Figure 7: An example of a clone of type 3.

The difference between type 2 and type 3 clones lies in the choice of 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 8).

```
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 8: 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 into type 1. Since existing clone detection tools can not provide us with the 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.

4. Case Study: Engine Control Systems

4.1 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 need 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 10. 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 product level. Next, we analyze component level and 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 methods.

4.2 Clone Analysis: Product View

Using our clone coverage metric, we assessed at a high level, the merge potential of product A and B. The result tells us the abstracted merge potential, for example, if type 1 clones from A to B have high clone coverage, we can merge products A and B relatively easily. If type 1 clones from A to B have high clone coverage, then this already gives us an indication that the products can be merged relatively easily by moving all the type 1 clones into a reusable component repository.

Figure 9 shows the result of the clone coverage analysis of 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.

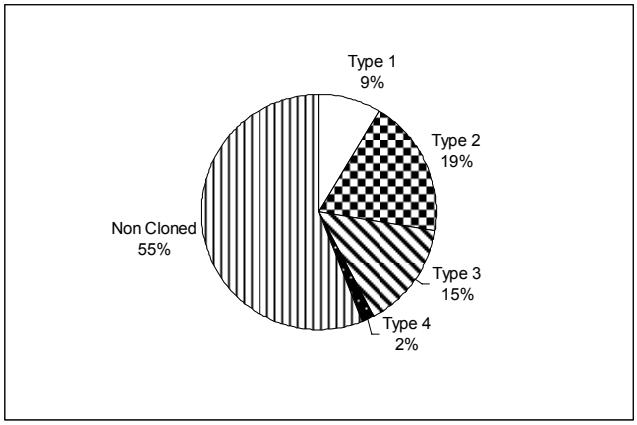

Figure 9: Clone Coverage from Product A to B.

In the case of the analyzed ECS products, 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 portion 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 different implementation than in product A.

We can observe from Figure 9 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.

4.3 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 9 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 [7]. That is, we map the abstract components to source code for both products from domain point of view. Figure 10 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 ECU architecture [5].

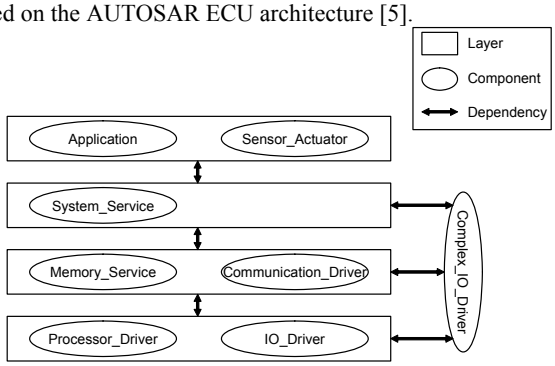

Figure 10: Software architecture of ECS products.

Figure 11 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 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 in a clone.

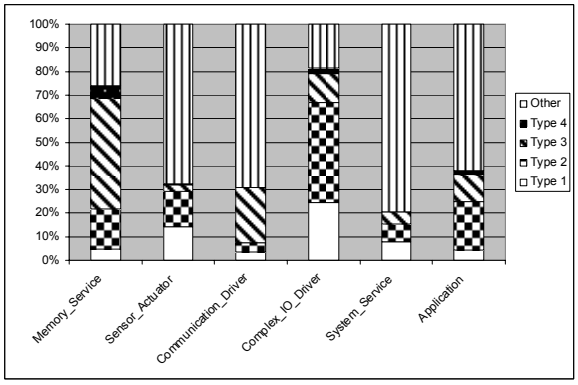

Figure 11: Component level clone coverage.

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 Application component.

4.4 Clone Coverage: Sub-Component View

Figure 12 shows the clone coverage of sub-components in the Application component. The Application component consists of 9 sub-components. From Figure 12, we can know the clone distribution for the sub-components in the application component.

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

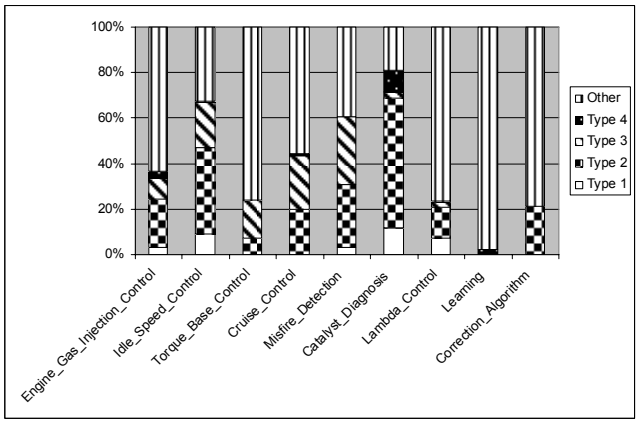

Figure 12: Clone coverage for Application sub-components.

4.5 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 12 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, the 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 the functionality differences. Therefore, we will follow the same merge strategy as in 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 the engine control software, rather it belongs to the vehicle control domain. Therefore, we will not give priority to merge 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: Idle_Speed_Control,Torque_Base_Control,Engine_Gas_Injection_Control into generic reusable components which can be instantiated for the current and future ECS products.

4.6 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 13). In addition, these products are controlled by developers who belong to different groups.

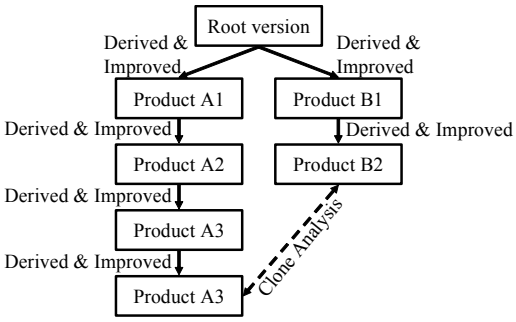

Figure 13: Evolution Tree of ECS products A and B.

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.

5. RELATED WORK

There are many works about product line engineering for automotive software (e.g. [8][9]). 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 the top-down software architecture to migrate to a product line.

AUTOSAR (AUTomotive Open Software ARchitecture) [5] is a consortium to establish open standard software architecture for vehicles. Hitachi is a premium member of AUTOSAR. AUTOSAR goals include reusability of software modules in vehicle. However, the standard is a kind of application framework. In the future, we will explore the connection between AUTOSAR and a product line for Hitachi ECS

In [1] the 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 [4][6] 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..

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed an approach to assess potential to merge existing systems into a product line. In the case of Hitachi Engine Control Systems (ECS), we observed an alarming lesson from the investigation reported above: products derived from the same origin by different teams loose much quicker identical parts 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 the Hitachi's 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. For the low implementation level commonality, from our point of view, the following requests in the engine control domain are responsible for: continuous demand for new features, integration of diverse configurations of varying hardware, software and mechanical parts, and uncoordinated development of similar features concurrently by different teams.

We will extend our work by addressing more extensively organizational-aspects to better support the practical execution of migration strategies based on identified technical merge strategies. Therefore, we will quantify benefits and return-on-investment (ROI), for example, by applying existing models of product line economics [3]. 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.

7. 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] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.