Service-oriented Commonality Analysis Across Existing Systems*

Alexander Harhurin and Judith Hartmann
Technische Universität München
Department of Informatics
Chair of Software and Systems Engineering
Boltzmannstr. 3, 85748 Garching, Germany
{harhurin,hartmanj}@in.tum.de

Abstract

This paper introduces an extractive approach to building-up a product line based on existing systems. Thereby, we focus on the analysis of common functionalities across different systems. Our commonality analysis is performed on the functional level which offers the highest reuse potential. We use the service diagram, a formally founded specification technique, for the functional specification. This allows us to perform the commonality analysis based on a formal definition of the system behavior. Concepts for the comparison of service diagrams and a methodology for building-up a service diagram of a product line are described in the paper. Additionally, since legacy systems rarely have an accurate functional specification, we present a methodology for extracting such a specification out of a given logical architecture.

1 Introduction

Increasing complexity due to a multitude of different functions and their extensive interaction as well as a rising number of different product variants are just some of the challenges that must be faced during the development of multi-functional system families.

Addressing this trend, we presented an approach to model-based development of software product lines (PLs) and to supporting the configuration of concrete variants (see [4]). In this paper, we show how to use it for the reverse engineering of a PL based on existing systems. Thereby, we focus on the analysis of common functionalities across different systems. According to Clements and Northrop [2], a PL is "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of reusable core assets in a prescribed way". Thus, the exploitation of commonality across related systems plays an important role in establishing the baseline for PL development. Moreover, a profound commonality analysis is crucial for deciding about the adequacy of setting up a PL. "One must have confidence that there is a family worth building. Performing a commonality analysis is a systematic way of gaining such confidence and of deciding what the scope of the family is. It reduces the risk of building systems that are inappropriate for the market and provides guidance to architects." [17]

According to Krueger [12], approaches to analyzing commonalities can be categorized into three classes, namely *proactive*, *reactive*, and *extractive* ones. While the proactive approach (a PL is analyzed from the very beginning) is well explored (see [2, 15]), it is not obvious how PL techniques can be applied in the presence of legacy software. However, building a PL is rarely a green-fields process: different already implemented systems have to be integrated into a family. Thereby, the PL development may borrow heavily from existing artifacts (specification, architecture) of related legacy systems. Consequently, we present an extractive approach to building a PL based on existing systems.

Our commonality analysis is performed on the functional specification – the most abstract description – of single systems. We are guided by the fact that the component architecture and the source code are usually a very poor expression of the user-visible functionality. They include a lot of details which are not observable at the overall system boundaries. Abstracting from technical realizations and capturing the pure functionality of systems, the functional specification offers the highest potential of reuse. To that end, we introduced a formally founded specification technique, namely *service diagrams*, in [6]. They describe systems as a set of related user-visible functional requirements (*services*). In contrast to prevalent commonality analyses (e. g. FORM [9] or FAST [18]) which primarily compare

*This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the VEIA project under grant 01ISF15A. The responsibility for this article lies with the authors.

feature names, we use a formal definition of the system behavior to identify *functional* commonalities between systems. The result of our analysis is a common service diagram which explicitly specifies common and different functionalities of a PL including the existing systems.

Drawing from our experience, we assume that model-based software development generally starts with the design of a logical architecture. Consequently, legacy systems rarely have an accurate functional specification. Therefore, the first part of our approach deals with extracting functional specifications based on the logical architectures of existing systems.

Contributions With the concepts described in this paper we make the following contributions.

We analyze the internal structure of the existing architecture and the behavior of single components to define black-box interaction patterns of the system. These result in a functional specification (a service diagram) - the formal basis for an accurate commonality analysis.

Based on the formal semantics and not on the names of the services, we compare service diagrams mined from different systems. Thereby, we also consider the possibility that services are similar but not identical or that the same functionality is specified by different sets of services. Thus, our approach allows to identify functional commonalities based on differently decomposed or labeled services.

Running Example The concepts introduced in the remainder of the paper will be illustrated by a simplified example of a cruise control. In this example, the legacy architectures of two different cruise control systems already exist. The basic cruise control (*BCC*) controls the vehicle speed for a constant target speed. The advanced cruise control (*ACC*) additionally offers a distance control to automatically follow a target vehicle and a pre-crash warning which displays a warning as soon as a potential crash is detected. Both controls comprise the acceleration/deceleration of the vehicle triggered by the respective pedals. In the remainder of the paper, based on the given logical architectures of both systems, the respective service diagrams are systematically extracted and compared to each other. Furthermore, a common service diagram of the cruise control PL is set up. All relevant details of the example are described at the appropriate places.

Outline The rest of this paper is organized as follows. In Section 2, the semantics of the service diagram is briefly presented. In Section 3, we introduce our approach to mining a service diagram of a single system based on its legacy architecture. Section 4 describes our commonality analysis across single service diagrams and shows the methodology to obtain a common service diagram of a PL. Finally, we compare our concepts to related approaches in Section 5 before we conclude in Section 6.

2 Service Diagram

This section introduces the *service diagram*, a hierarchical model for the specification of the functionality of a PL. The behavior is specified from a black-box view, i. e. as relation between input and output messages observable at the system boundaries. An implementation satisfies the specification formalized by a service diagram if it shows the same I/O behavior as specified by the diagram.

A service diagram consists of hierarchically decomposed services – modular specifications of single functionalities – and four kinds of relationships between them, namely *aggregation*, *functional dependencies*, *optional* and *alternative relations* (cp. Figure 4). Structurally, a service diagram reminds of a FODA tree [8]. In contrast to features where behavior is only informally associated by choosing suitable names, the behavior of services is formally defined in our approach.

All relevant concepts are briefly sketched in the following paragraphs. A more detailed description of the basic concepts can be found in [6].

Services The service diagram is based on the notion of a *service* [1] as the fundamental concept. Intuitively, a service represents a piece of functionality by specifying requirements on the I/O behavior. Hence, the service diagram is a restrictive specification. Each service imposes a requirement on the system and, thus, further restricts the valid I/O behavior. Formally, a service is a stream-processing function which maps streams of input messages to streams of output messages. Here, a stream s of elements of type $Data$ can be thought of as a function $s\colon \mathbb{N} \rightarrow Data$.

Each service provides a *syntactic interface* $I \blacktriangleright O$, which consists of a set I of typed input ports and a set O of typed output ports. With each port we associate a set of streams representing the syntactically correct communication over this port. Formally, for a given set of ports P, a port history is a mapping which associates a concrete stream to each port: $h\colon P \rightarrow (\mathbb{N} \rightarrow Data)$. $\mathbb{H}(I_S) \times \mathbb{H}(O_S)$ denotes the set of all *syntactically correct* I/O history pairs (x, y) for a service S with interface $(I_S \blacktriangleright O_S)$.

There are different ways to specify the *semantics* – the stream-processing function – of a single service, e.g. I/O automata [13], first-order logic predicates or a tabular notation. By specifying the semantics, the set of all syntactically correct histories is restricted to a subset of *(semantically) valid* histories. We say, the *behavior* of a service is the set of all valid history pairs for this service.

Note, a service restricts the output histories only for a subset of input histories (*service domain*). For the input histories outside its domain, a service specifies no restrictions on the outputs – any arbitrary value is allowed.

Structuring of Services The *aggregation* relation arranges individual services into a service hierarchy. The semantics of a *compound service* (composed of several *sub-services*) is defined as being the union of all concurrently operating sub-services.

The *alternative* relation is defined as a relation between a *variation point* and a set of mutually alternative services (*variants*). An variation point specifies a set of history pairs which are valid for at least one of its variants, respectively.

An *optional* service represents an alternative between the presence and the absence of this service. Consequently, it can be transferred into an alternative variation point.

By *functional dependencies*, we denote relations between services in a way that the behavior of one service influences the behavior of other ones. As our approach aims at the specification of the user-visible behavior, only those dependencies are specified which are observable at the overall system boundaries. There are a lot of methodological significant dependencies. However, our cases studies have shown that all of them can be realized by introducing additional services and priority dependencies. A *priority* dependency "service S_1 takes priority over service S_2" defines that if both services require different messages on their common output ports within a time interval, the message of S_1 takes priority over the message of S_2. Within this time interval, a valid output history has to fulfill the requirements formalized by S_1 but not those of S_2 – only service S_1 is *efficacious*.

3 Mining Service Diagrams

This section describes our approach to extracting a service diagram based on a legacy logical architecture. In the following, we shortly sketch our definition of the logical architecture. Then, we explain our methodology for extracting single services out of a logical architecture. Finally, we concentrate on the hierarchical structuring of services and their dependencies.

3.1 Logical Architecture

The *logical architecture* describes the system as a hierarchical network of communicating components connected by channels (cp. Figure 1). Analogously to services, each component provides a syntactic interface and a behavioral semantics. The syntactic interface of a component is given as a set of I/O ports. For the behavioral specification of a component, several specification techniques can be used (e. g. I/O automata, statecharts). In contrast to the service diagram, the view on the system is changed from a pure black-box to a white-box view. While each service describes an aspect of the black-box behavior, the logical architecture focuses on how the black-box behavior is realized by (internally) communicating components. Thereby, the formation of the logical architecture can be influenced by different criteria, especially by non-functional requirements. Depending on the considered criteria, the resulting logical architecture will turn out differently.

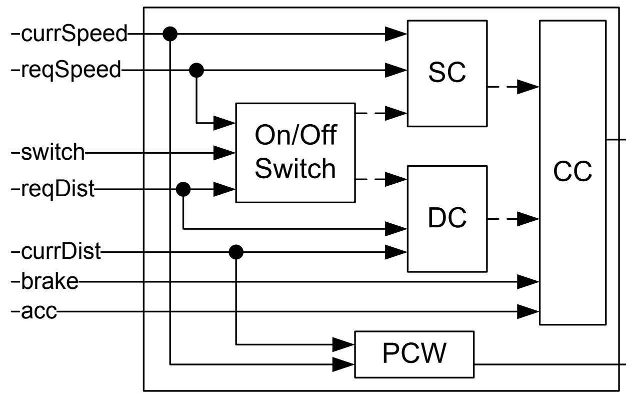
3.2 Mining Services

To extract services out of an existing logical architecture, we analyze the static structure as well as the behavior of the component architecture. We propose the following three steps. First, analyze the static dependencies between input and output ports. Then, identify *input patterns* which classify inputs according to their behavioral effects on the outputs and *output functions* which describe different output reactions. Lastly, combine input patterns which provoke an equivalent output reaction into one service.

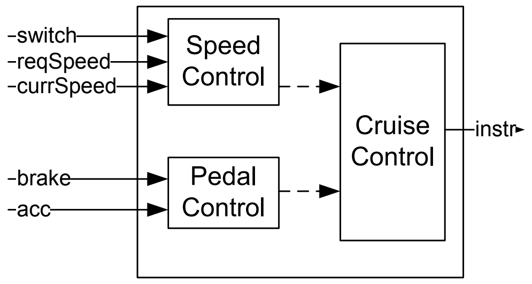
Static Analysis of I/O Dependencies In the first step, the dependencies between input and output ports of the overall system are analyzed. For each output port, we determine the set of input ports which influence the given output port. Therefore, the data flow (a chain of internal components connected by channels) is retraced starting from the output port back to the input ports. The result of this analysis is mirrored by a predicate $d : I \times O \mapsto \mathbb{B}$, which indicates if a port $o \in O$ depends on a port $i \in I$. The set $\mathcal{P} = \{(I', o) \mid o \in O \wedge I' = \{i \in I \mid d(i, o)\}\}$ consists of tuples where o is an output port of the system and I' is the set of all input ports o depends on.

In our example (cp. Figure 1(a)), port `instr` depends on all input ports, while port `warning` only depends on ports `currSpeed` and `currDist`. Thus, $\mathcal{P} = \{(I, \texttt{instr}), (\{\texttt{currSpeed}, \texttt{currDist}\}, \texttt{warning})\}$.

Identification of History Patterns In the next step, for each tuple $(I', o) \in \mathcal{P}$, we analyze the I/O histories on the respective ports. We identify a finite number of history patterns mapping input patterns to corresponding output functions. An *input pattern* is a predicate which characterizes a set of input histories until a time interval t: $IP_t \colon \mathbb{H}(I') \mapsto \mathbb{B}$. An *output function* F_t defines the output on a given port at time interval $(t + 1)$ depending on the inputs until time interval t. Given an input history x, an output function F_t can be seen as a predicate over the output histories: $F_t(x) \colon \mathbb{H}(o) \mapsto \mathbb{B}$. A *history pattern HP* relates an input pattern to an output function such that inputs satisfying the input pattern must be mapped to outputs in compliance with the output function. Intuitively, it defines under which circumstances (which input pattern) which output



(a) Advanced Cruise Control



(b) Basic Cruise Control

Figure 1. Legacy Logical Architectures

function is efficacious. Formally, a history pattern defines a set of history pairs which fulfill the following condition: for all time intervals t in which the input history satisfies the input pattern, the output at $(t+1)$ must be defined by the corresponding output function:

$$HP = (IP, F) \overset{\text{def}}{=} \{(x,y) \in \mathbb{H}(I') \times \mathbb{H}(o) \mid \forall t \in \mathbb{N} : IP_t(x) \Rightarrow F_t(x,y)\}$$

Note, for any time interval, a set of history patterns partitions the I/O histories into different (not necessarily disjunctive) subsets.

While identifying the history patterns, we apply established concepts used in the generation of white-box tests [14]. Based on the internal structure of the logical architecture and the specifications of single components, the domain of each port is partitioned into equivalence classes such that every guard of each branching in the control flow from inputs to outputs is set to both true and false at least once. In the test community this method is called *condition coverage*. For example, component `On/Off Switch` (automaton omitted here) partitions the domains of the ports `switch`, `reqDist`, `reqSpeed` into two equivalence classes, respectively: $dom(switch) = \{0\} \cup \{1\}$, $dom(reqSpeed) = \{x \mid x < 30\} \cup \{x \mid x \geq 30\}$ and $dom(reqDist) = \{x \mid x < 300\} \cup \{x \mid x \geq 300\}$. Following the data flow up to the output port `instr`, subsequent components cause a decomposition of other port domains.

In summary, the result of this step is a table of history patterns for each tuple of the set $\mathcal{P}$. The number of the tables equals the number of the output ports.

Table 1 shows the history patterns for the output port `instr` of the *ACC*. Each column of the table defines a single history pattern. Besides concrete values of the respective type, variables and predicates (first-order logic) are used to describe the inputs. The asterisk * denotes an arbitrary value.

The first column specifies the following pattern: if the value on port `brake` is greater than 0 (brake pedal is pushed) and all other values are arbitrary, then the speed instruction on port `instr` is calculated by means of function k.[1] The second column describes the situation in which the acceleration pedal is pushed (`acc`>0). In this case, the speed instruction is calculated by means of function k, too. The next three history patterns describe the situation in which brake and acceleration are not applied (`brake`=0 and `acc`=0) and either the ACC is off (`switch`=0), the required speed is less than the specification allows (`currSpeed`<30) or the required distance is less than permitted (`reqDist`<300). In this case, the speed instruction equals 0. Otherwise, if the distance control is on and a target vehicle is detected (`currDist`$\neq \varepsilon$), the speed instruction is calculated by the function f, else by the function g (described by the last two history patterns).

Combination of History Patterns into Services Finally, we build a set of initial services based on the tables from the last step. The resulting services describe disjunctive subfunctionalities. Thus, they are neither hierarchically structured nor related by dependencies.

For each table, all history patterns with the same output function are combined into one service. The syntactic interface of this service consists of a set of input ports $I'' \subseteq I'$ and the output port o. Thereby, an input port is included in I'' only if it is further specified (i. e. not identified by *) in at least one of the combined patterns. Otherwise, the port is of no importance in the history patterns and, therefore, excluded from I''. The semantics of the service (i. e. the mapping from inputs to outputs) is defined as conjunction of the concerned history patterns. Whenever the input his-

[1]Functions k, f, g and h are four different methods to calculate the speed instruction. Their details implemented in component `CC` of *ACC* and `Cruise Control` of *BCC* are of no importance for our approach.

switch	*	*	0	*	*	1	1
reqSpeed	*	*	*	<30	*	$s \geq 30$	$s \geq 30$
reqDist	*	*	*	*	<300	$d \geq 300$	≥ 300
currSpeed	*	*	*	*	*	x	x
currDist	*	*	*	*	*	$y \neq \epsilon$	$y = \epsilon$
brake	$x > 0$	x	0	0	0	0	0
acc	y	$y > 0$	0	0	0	0	0
instr	$k(x, y)$	$k(x, y)$	0	0	0	$f(x, y, s, d)$	$g(x, s)$

Table 1. I/O History Patterns of Advanced Cruise Control

	I Ports	O Ports	Semantics
1	acc, brake	instr	$(acc > 0 \vee brake > 0) \Rightarrow instr = k(acc, brake)$
2	acc, brake, reqDist, reqSpeed, switch	instr	$acc = 0 \wedge brake = 0 \wedge (switch = 0 \vee reqSpeed < 30 \vee reqDist < 300) \Rightarrow instr = 0$
3	acc, brake, reqDist, reqSpeed, switch, currSpeed, currDist	instr	$acc = 0 \wedge brake = 0 \wedge switch = 1 \wedge reqSpeed \geq 30 \wedge reqDist \geq 300 \wedge currDist \neq \epsilon \Rightarrow instr = f(currSpeed, currDist, reqSpeed, reqDist)$
4	acc, brake, reqDist, reqSpeed, switch, currSpeed, currDist	instr	$(acc = 0 \wedge brake = 0) \wedge (switch = 1 \wedge reqSpeed \geq 30 \wedge reqDist \geq 300) \wedge currDist = \epsilon \Rightarrow instr = g(currSpeed, reqSpeed)$
5	currSpeed, currDist	warning	$warning = h(currSpeed, currDist)$

Table 2. Services of the Advanced Cruise Control

tory is in accordance with at least one of the input patterns, the output must be defined by the common output function. Since every initial service describes different output functions, the input patterns of the initial services are required to be disjunctive in order to avoid inconsistencies.

In our example, Table 1 yields four services for the output port `instr`. For the output port `warning` there is only one history pattern (table omitted here), thus, only one service. Together, this results in five initial services for the logical architecture from Figure 1(a) (cp. Table 2). Note, we purposely use numbers instead of names to label services in order to emphasize the role of the service semantics but not the terminology in our subsequent analysis.

3.3 Construction of Service Diagrams

The result of the service mining is a set of services that unambiguously specify the black-box behavior of the system. As already mentioned, this set is unstructured and the services are completely independent from each other. As a consequence, the initial services have more input ports and specify more constraints on the inputs than they actually need for the calculation of their outputs. Exemplary, even though the output function f of Service 4 from Table 2 only depends on the current and the requested speed, it also includes other input ports. They are required to define the situation in which the service is efficacious (cp. Section 2).

Drawing from our experience, the commonality analysis (see Section 4) performs better on a service diagram which hierarchically structures modular services and explicitly models dependencies. Therefore, the goal of the second stage of the mining process is to rebuild the set of initial services into a service diagram. Thereby, we aim at separating the pure functionality specified by a service from the preconditions under which the service is efficacious.

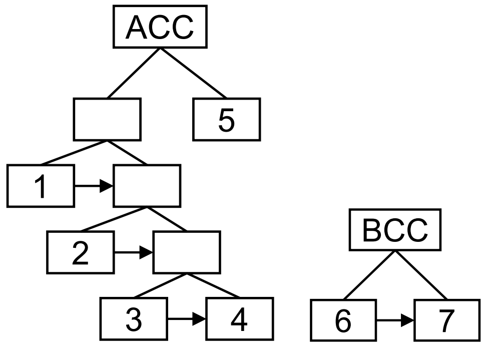
We analyze the syntactical interface of the initial services and order them according to an inclusion relation between the input ports: $S_1 < S_2 \Leftrightarrow I_1 \subset I_2$. The resulting ordering is not total, since two services S_1 and S_2 with $I_1 \not\subseteq I_2 \wedge I_2 \not\subseteq I_1$ are not comparable. Looking at the initial services influencing the output port `instr` (cp. Table 2), we notice the following inclusion relation: $I_1 \subset I_2 \subset I_3 = I_4$. Thus, the services are ordered according to their interfaces: $1 < 2 < 3 = 4$.

Now, we differentiate the input ports in *necessary* and *additional* ports. For each service, we classify an input port as *necessary* if the respective output function depends on it. The port is essential for the functionality of the service. All ports, which have no influence on the output function but only define the precondition of a service to be efficacious are classified as *additional*.

In the following behavioral analysis, we aim at eliminating redundant conditions by introducing priority dependencies between services. Furthermore, we remove additional ports if they are no longer needed in the precondition.

Starting with the "smallest" service, we compare the conditions on the common input ports. If a service S_2 has as input condition a negation of the input condition of a service S_1, there is an implicit priority dependency between them. As introduced in Section 2, a *priority* dependency "service S_1 takes priority over service S_2" defines that S_2 is only efficacious if the input condition of S_1 is not fulfilled. Thus, the introduction of an explicit priory dependency between both services allows us to avoid a repeated treatment of the common condition in service S_2. By this, the services can be stepwise reduced to their pure functionality.

In our example, Service 1 requires $acc > 0 \vee brake > 0$. All other services require $acc = 0 \wedge brake = 0$, the negation of the input pattern of Service 1. Furthermore, `acc` and `brake` are additional input ports of all services except for Service 1. By introducing a priority dependency between Service 1 and the rest, the $acc/brake$-part can be eliminated from all other services. In the service diagram depicted in Figure 2(a), this step results in two new services: Service 1 and the service aggregating the rest. The analysis of Service 2 and the remaining services yields that the negation of the input condition of 2 is reformulated in the input patterns of 3 and 4 – an explicit priority dependency makes sense. However, only port `switch` can be eliminated since all other ports are necessary and can not be eliminated from the compound service of 3 and 4. The resulting service diagrams of the completed analysis of both systems are depicted in Figure 2 and Tables 3 and 4. To improve the comprehensibility of the example, the tables additionally contain an informal description of the resulting services.



(a) ACC (b) BCC

Figure 2. Service Diagrams

4 Commonality Analysis

Having mined service diagrams from logical architectures, we have reached an appropriate abstraction level to start with a formal commonality analysis across legacy system functionalities. The main goal of the analysis is to obtain a *common* service diagram of the (potential) PL. The common diagram explicitly specifies common and different functionalities of the existing systems. Thus, this diagram serves both as basis for the decision to create a PL or not and as starting base in the model-based development of a future PL according to [4]. Thereby, we use the formal definition of the service behavior to identify *functional* commonalities between systems. To that end, we introduce three different matching relations to reason about the similarity of two single services, before we describe our methodology to compare diagrams and to build up the common service diagram.

4.1 Matching Relations

Additionally to the basic relation which classifies two services as identical or not, we introduce two further matching relations. These relations indicate whether one service specifies a sub-functionality of the other one or whether both services have common sub-functionalities.

Note, services describe the black-box behavior of systems embedded in the same environment. Since the set of sensors/actuators is relatively small and well-known to all designers, we assume that the port names are unambiguous to all services. Thus, the same port names refer to the same ports and different port names refer to different ports.

Identical Services We consider two services S_1 and S_2 as *identical* if they exactly match – syntactically and behaviorally. Formally, this is the case if they have the same syntactic interface and describe the same set of valid I/O histories.

In our example, Services 4 of `ACC` and 7 of `BCC` are identical (cp. Tables 3 and 4). They have the same syntactic interface and they show the same behavior. Both services describe a simple speed control which calculates the speed instruction based on the difference between current and requested speed.

Sub-Service Intuitively, a sub-service specifies a sub-functionality of its super-service (cp. compound service in Section 2). More precisely, S_2 is a sub-service of a S_1 if the interface of S_2 is a sub-interface of S_1 and every valid history pair of S_2 is extendable to a valid history pair of S_1. In other words, if a pair is valid for the sub-service S_2, then there is a pair valid for the super-service S_1 such that they are equal on all common ports.

Service 1 of `ACC` is a sub-service of Service 6 of `BCC` (cp. Tables 3 and 4). The interface of 1 is a sub-interface of 6. For each history pair valid for Service 1 there is a history pair valid for Service 6 such that they are equal on the ports `acc`, `brake` and `instr`. This results from the fact that both services describe the same behavior, given `acc`>0,

	I Ports	O Ports	Semantics	Informal Description
1	acc, brake	instr	$acc > 0 \vee brake > 0 \Rightarrow instr = k(acc, brake)$	Reaction to the actuation of the brake or acceleration pedal.
2	reqDist, reqSpeed, switch	instr	$switch = 0 \vee reqSpeed < 30 \vee reqDist < 300 \Rightarrow instr = 0$	ACC is off, the required speed or the required distance is less than permitted.
3	reqDist, reqSpeed, currSpeed, currDist	instr	$currDist \neq \epsilon \Rightarrow instr = f(currSpeed, currDist, reqSpeed, reqDist)$	A target vehicle is detected. The speed depends on the distance to the vehicle.
4	reqSpeed, currSpeed	instr	$instr = g(currSpeed, reqSpeed)$	The speed instruction only depends on the current and required speed values.
5	currSpeed, currDist	warning	$warning = h(currSpeed, currDist)$	Warning signal depends on the current speed and distance values.

Table 3. Leaf Services from Figure 2(a)

	I Ports	O Ports	Semantics	Informal Description
6	acc, brake, switch	instr	$acc > 0 \vee brake > 0 \vee switch = 0 \Rightarrow instr = k(acc, brake)$	If one of the pedals is actuated or the speed control is off, the reaction is the same.
7	reqSpeed, currSpeed	instr	$instr = g(currSpeed, reqSpeed)$	The speed instruction depends on the current and required speed values.

Table 4. Leaf Services from Figure 2(b)

`brake`>0, and `switch`=0. In other words, each history pair valid for Service 1 can be enlarged to an I/O pair valid for 6 by setting `switch`=0.

Common Sub-Service The third matching relation describes situations in which neither S_1 is a sub-service of S_2 nor vice versa, but both services have a common sub-service – they specify the same sub-functionality. Formally, both services have a subset of common I/O ports on which they specify the same behavior.

In our example, Service 2 of `ACC` and Service 6 of `BCC` have a common sub-service. Service 2 requires on port `instr` the output 0 whenever `switch`=0 (independently of the values on the ports `reqDist` and `reqSpeed`). If additionally `acc`=0 and `brake`=0, the output calculated by Service 6 is also 0 since $k(0, 0) = 0$. Thus, if `switch`=0, `acc`=0, and `brake`=0 both services require the output 0. This means, whenever the automatic cruise control is turned off and the pedals are not pushed, the speed instruction is 0. This holds for both systems.

4.2 Comparison of Service Diagrams

Now, we show how the introduced matching relations can be used to compare service diagrams.

Comparison of Leaf Services We start the analysis by syntactically comparing the leaf services of the considered diagrams. This yields three sets of potential candidates for the introduced matches: the set C_{id} of services with identical interfaces, the set C_{sub} of pairs of services (S, T) where S has a sub-interface of T, and the set C_c of services which have at least one input and one output port in common. According to the definitions from the last subsection, these sets are not disjunct, rather $C_{id} \subseteq C_{sub} \subseteq C_c$.

Based on these candidate sets, we start our behavioral analysis. We identify the behaviorally identical services from the candidates in C_{id}. The analysis of the second set C_{sub} yields the services which are in a sub-service relation. Thereby, we do not consider services already identified as identical. Lastly, we eliminate identical services and sub-services from C_c and analyze the remaining services in order to determine the services which have a common sub-functionality.

In our example, the syntactical analysis yields the following sets $C_{id} = \{(4, 7)\}$, $C_{sub} = \{(4, 7), (1, 6), (7, 3)\}$ and $C_c = \{(4, 7), (1, 6), (7, 3), (2, 6), (2, 7)\}$. The behavioral analysis of C_{id} shows that the Services 4 and 7 are identical. Consequently, the pair $(4, 7)$ can be removed from the remaining candidate sets. The behavioral analysis of the remaining set $C'_{sub} = \{(1, 6), (7, 3)\}$ classifies 1 as sub-service of 6. By analyzing the remaining candidate set $C'_c = \{(7, 3), (2, 6), (2, 7)\}$ we see that the Services 2 and 6 have a common sub-functionality. Summarized, we have the pair $(4, 7)$ of identical services, the pair $(1, 6)$ of services in a sub-service relation, and the pair $(2, 6)$ of services with common sub-functionalities.

Comparison of Compound Services Besides leaf services, we also include compound services into our analysis.

Two compound services are identical if they aggregate exactly the same sub-services, or if all sub-services of one compound service are sub-services of the second one and vice versa. In the former case, the same functionality is exactly divided into the same set of sub-functionalities. In the latter case, the same functionality is divided into two different sets of sub-functionalities.

A compound service and a leaf service are identical if each sub-service of the compound service is a sub-service of the leaf service and the leaf service is a sub-service of the compound service. In this case, the same functionality is divided into a set of sub-functionalities within one diagram and remains undivided within the other diagram.

The sub-service and common sub-service relations for compound services are defined analogously.

Note, the sub-service relation allows us to avoid the introduction of a normal form of the service diagram. The same functionality described by two different sets of services will be identified by means of the sub-service relation. In our experience, differing decompositions of the overall functionality are quite usual if functionalities are mined from different logical architectures.

4.3 Common Service Diagram

Having identified related services of both diagrams, we rebuild them into a common service diagram.

Extracting Common Services Before building up the common service hierarchy, we treat situations where a service of one diagram is a sub-service of a service of the other diagram or where two services describe a common sub-functionality. We aim at extracting the maximum identical part of the behavior from the diagrams. For that propose, each of these services is split up into a common sub-service which is identical across the different diagrams and a rest sub-service.

In our example, Service 1 is a sub-service of 6, Services 2 and 6 share a common functionality. We decompose Service 6 into two sub-services, $6A$ and $6B$ (cp. Figure 3 and Table 5). Service $6A$ exactly specifies the sub-functionality specified by Service 1 – the reaction to the actuation of the brake or acceleration pedal. Service $6B$ specifies the speed instruction `instr`=0 if the automatic is off (`switch`=0). A priority dependency between $6A$ and $6B$ determines that $6B$ is only efficacious if neither the acceleration nor the brake pedal is pushed. Furthermore, Service 2 is decomposed in a Service $2A$, which is identical to $6B$, and a rest Service $2B$. This service describes, that the speed instruction is 0 if the requested speed or the requested distance are too low. Note, that the new Services $2A$ and $2B$ are independent. There is no priority relation between them.

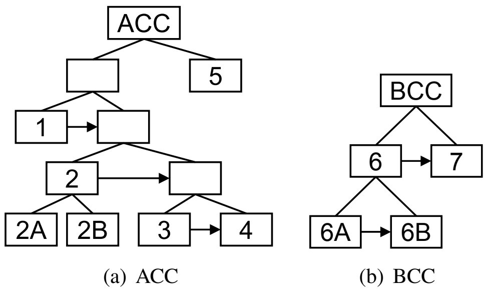


Figure 3. Adapted Service Diagrams

Building Up a Service Hierarchy Now, we have identified the maximum identical part of the behavior from both diagrams. All pairs of identical services – original ones, as well as the ones extracted in the last step – become mandatory services within the common diagram. If services are sub-services or have a common sub-functionality, but it is not possible to extract a common part, these services become alternative variants of a variation point. All other services become optional.

In our example, the commonality analysis of the leaf services of the adapted service diagrams (cp. Figure 3) yields the set of identical services $\{(1, 6A), (2A, 6B), (4, 7)\}$. The identical services result in three mandatory services, namely `Pedal Control` (reaction to the actuation of the pedals), `SC` (speed control), and `On/Off` (functionality to switch the automatic on/off) within the common diagram of Figure 4. All other services of the `ACC` result in optional services. The newly introduced Service `2B` is mirrored by the Service `CorrectValues` (functionality to check the requested speed and distance ranges), the distance control functionality represented by `3` becomes Service `DC`, and the pre-crash warning `7` results in `PCW`. To advance the intuitive comprehensibility of the common service diagram, we assigned meaningful names to the leaf services. However, from a technical point of view, this is not necessary.

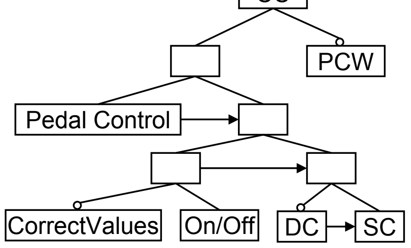


Figure 4. Common Service Diagram

	I Ports	O Ports	Semantics
2A	switch	instr	$switch = 0 \Rightarrow instr = 0$
2B	reqDist, reqSpeed	instr	$reqSpeed < 30 \vee reqDist < 300 \Rightarrow instr = 0$
6A	acc, brake	instr	$acc > 0 \vee brake > 0 \Rightarrow instr = k(acc, brake)$
6B	switch	instr	$switch = 0 \Rightarrow instr = 0$

Table 5. Additional Sub-Services from Figure 3

Analysis of Functional Dependencies The next step is to analyze the dependencies between services. There is a dependency between two mandatory services within the common diagram if both single diagrams contain the same dependency between the original services. Thereby, transitive relations – via ancestors – must be considered adequately.

In our example, the direct relation between Services 6 and 7 causes a transitiv relation between $6A$ and 7 in Figure 3(b). The same relation exists between Services 1 and 4 in Figure 3(a). This results in a transitive relation between Services `Pedal Control` and `SC` in Figure 4. The same goes for the remaining relations.

A common service diagram may contain optional dependencies. It is obvious that dependencies which involve optional or alternative services are efficacious only if the involved services are selected within the configuration of a product. Also, dependencies between mandatory services are optional if they are not present between the original services within all single diagrams.

In our example, if the `BCC` had no dependency between Services 6 and 7, then the common diagram would have an optional instead of a mandatory dependency between the compound services of `On/Off` and `SC`.

Identification of PL Specific Dependencies The last methodological step is to analyze the common service diagram for PL specific dependencies. In the beginning, there are *requires* dependencies between all variable (optional or alternative) services which originate from the same service diagram and *excludes* dependencies between all variable services coming from different diagrams. This results in a PL which exactly consists of the original products. However, we aim at stepwise enlarging the product space by new combinations of variable services even though they originally belong to different products. This requires a further analysis of the involved services.

If two variable services are not functionally related, the options can be selected independently. In this case, the *requires* dependency is eliminated. If two services cause no functional conflict, e. g. they specify no conflicting behavior on common output ports, the *excludes* dependency is eliminated. Otherwise, an excludes dependency may be replaced by a *priority* dependency. For more details about conflict detection and resolution between services within a PL see [6].

In our example, since no optional service functionally depends on any other variable service, as well as no inter-service conflicts exist, the common diagram contains neither *requires* nor *excludes* dependencies. This leads to a set of new systems, e. g. the basic cruise control with a pre-crash warning.

5 Related Work

Related work to our approach can be mainly found in two different areas, namely approaches to mining existing assets and commonality analyses.

Several methods have been recently proposed for locating features in existing systems (see, e. g. an overview by Wilde et al. [19]). For example, a semi-automatic technique introduced by Koschke et al. [11] combines dynamic and static analysis to reconstruct the mapping of features to code. The software reconnaissance method introduced by Wilde et al. [20] uses test cases to locate features. An obvious drawback of these approaches is the same as those of the classical feature-oriented reuse approaches like [9, 5, 10] – they all only focus on the modeling of relationships between features, using uninterpreted features as the corresponding basic concept. In contrast, we use a formal definition of the system behavior to identify functional commonalities between systems.

Another class of researches focuses on discovering Use Cases from source code. For example, El-Ramly et al. [3] analyze traces of the interaction between a system and its users to discover the behavior of the system. However, the absence of a formal semantics of Use Cases prevents a commonality analysis of different systems.

Several commonality analyses are proposed in the context of software reuse, library retrieval and PL design (see, e. g. an overview by Thevenot et al. [16]). For example, so-called *specification matching* approaches are introduced in [21, 7]. These approaches compare software components based on formal descriptions of its behavior. The components are specified using pre- and postconditions written in first-order logic. Even though containing some inspiring ideas, these approaches address a different problem and are not suited to the commonality analysis for PLs. They only aim at comparing components, but not at identifying and extracting common parts of them. Furthermore, the component architecture is usually a very poor expression of the user-visible functionality. It includes a lot of components which functionalities are not observable at the overall system boundaries. In our approach, the commonality analysis takes place at the functional level where a system is seen only according to its user-visible functionality without consideration of implementation details.

In [17], Weiss presents a systematic process for defining families as part of the FAST process. In contrast to our approach, it presents no formally founded concepts to compare functionalities and to extract the commonalities. The identification of commonalities is performed primarily based on terminology. Furthermore, this approach is designed more for the proactive PL development. Existent systems are not adequately taken into consideration.

In summary, to the best of our knowledge, there is no approach to analyze functional commonalities across systems based on the formal definition of the system behavior.

6 Conclusion and Future Work

In this paper, we have introduced an extractive approach to building-up a PL based on existing systems. Thereby, we have focused on one of the most important tasks of PL development, namely the analysis of common functionalities across different systems.

Our commonality analysis is performed on the functional level which offers the highest reuse potential. Moreover, this facilitates the integration of additional requirements for new variants, which are obviously not given in form of implemented code. Although being on the requirements engineering level, our specification technique, the service diagram, is formally founded. Thus, our commonality analysis is based on a formal definition of the system behavior. This functional analysis results in a common service diagram which explicitly specifies common and different functionalities of existing systems.

Moreover, since legacy systems rarely have an accurate functional specification, we have further presented a methodology for extracting single services out of a component architecture and their hierarchically structuring into a service diagram.

We are currently integrating Data Mining approaches to automatically analyzing traces of the interaction between a running legacy system and its environment to allow the service mining from the implemented code. Beyond this, our future work includes traceability improvements aiming at chronologically interrelating services and components to support the reuse of already implemented components.

References

[1] M. Broy. Service-oriented systems engineering: Modeling services and layered architectures. In *FORTE*, 2003.

[2] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2002.

[3] M. El-Ramly, E. Stroulia, and P. Sorenson. Mining system-user interaction traces for use case models. In *Proceedings of IWPC '02*. IEEE Computer Society, 2002.

[4] A. Gruler, A. Harhurin, and J. Hartmann. Development and configuration of service-based product lines. In *Proceedings of Software Product Line Conference*, 2007.

[5] J. V. Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. *WICSA*, 2001.

[6] A. Harhurin and J. Hartmann. Towards consistent specifications of product families. In *FM: 15th International Symposium on Formal Methods*. Springer Verlag, 2008.

[7] D. Hemer. Specification matching of state-based modular components. In *Proceedings of the APSEC'03*. IEEE Computer Society, 2003.

[8] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, SEI, CMU, Pittsburgh, PA, 1990.

[9] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5, 1998.

[10] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented project line engineering. *IEEE Softw.*, 19(4):58–65, 2002.

[11] R. Koschke and J. Quante. On dynamic feature location. In *Proceedings of the 20th international Conference on Automated software engineering*. ACM, 2005.

[12] C. W. Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering*, 2002.

[13] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[14] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[15] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering*. Springer, 2005.

[16] H. Thevenot and T. Simpson. Commonality indices for product family design: a detailed comparison. *Journal of Engineering Design*, 17(2), 2006.

[17] D. M. Weiss. Commonality analysis: A systematic process for defining families. *LNCS*, 1429, 1998.

[18] D. M. Weiss and C. T. R. Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley, 1999.

[19] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds. A comparison of methods for locating features in legacy software. *J. Syst. Softw.*, 65(2), 2003.

[20] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1), 1995.

[21] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6(4), 1997.