T U M

I N S T I T U T F Ü R I N F O R M A T I K

A Formal Approach to Specifying the Functionality of Software System Families

Alexander Harhurin and Judith Hartmann

TUM-I0720
Oktober 07

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

Abstract. Addressing the challenges faced today during the development of multi-functional system families, we suggest a service-oriented approach to formally specifying the functionality and, particularly, the functional variability already in the requirement engineering phase. In this paper, we present and precisely define the underlying concepts, such as the notion of individual services, the combination of services, and variability. Thereby, in contrast to prevalent approaches, we especially focus on *explicitly* modeling *behavioral* (functional) commonalities and differences between alternative variants.

1 Introduction

Today, innovative functions – mainly realized by software – are one of the key potentials for competitive advantage in various application domains, e.g. the automotive domain. 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 are faced during the development of multi-functional system families.

Because of complex interactions between different functionalities there is a need for adequate modeling techniques for functional requirements. The functional specification where a system is seen only according to its user-visible functionality is the appropriate abstraction level to start with a formal description and analysis of a system. The prevalent approaches to modeling functional requirements (e.g. UML Use Cases, FODA [1]) lack a precise semantics in general. However, in order to assure the consistency of a specification, a precise semantics of the modeling techniques is inevitable. Based on such a formal foundation, discrepancies between conflicting functionalities can be detected and resolved already in the early phases of the development process. Consecutively, we focus on the formal definitions of system functionalities and relations between them, and show how the upcoming service-oriented paradigm is used to handle the aforementioned functional intricacy. Our notation technique, the *Service Diagram* introduced in [2] and [3] describes the system as a set of (formally related) functionalities (*services*) and aims at understanding how these services depend on and interfere with each other.

Regarding product families, we enlarge our approach by variability concepts which allow for the formal specification of functional variability in the requirement engineering phase. Approaches like FODA only provide a two-valued variability, i.e. a functionality might be present or not present in a system. "As a consequence, these approaches focus on the analysis of dependencies, however abstracting away from the causes for these dependencies" [4]. We, however, introduce the semantics of a product line as the behavior that can be delivered by at least one of its variants. Aiming at a consistent specification of the product line, we additionally introduce a further variability specification, the *common* specification. This specification allows to identify and explicitly specify behavioral commonalities of variants and greatly helps to reduce the effort to analyze requirements on a product line. Thus, our definitions of variability focus on explicitly modeling behavioral (functional) commonalities and differences between variants, i.e. what behavior different variants have in common, and what behavior is individual for a specific variant.

All in all, given these formal foundations, our approach can be used for an automated analysis of the functional requirements and, particularly, for detecting functional conflicts between different variants in a product line efficiently.

Running Example The concepts introduced in the remainder of the paper will be illustrated by a simplified example of a seat heating system (cp. Figure 1). There exist two variants of the heating system, a manual and an automatic one. The functionalities of the manual heating are as expected, e.g. turning the system on/off and setting the desired seat temperature. The automatic version provides an interface to a further sensor to receive the current seat temperature. Based on these data, the controller calculates the necessary instructions for the heating device. Additionally, there is a child seat controller for detecting a child seat mounted on the front seat. Among others, this controller influences the heating sub-systems. All relevant details will be 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 presented. In particular, we introduce the formal specification of a single system functionality by services in Section 2.1. Subsequently, in Section 2.2 and 2.3 we concentrate on concepts for structuring the services and modeling their dependencies. In Section 2.4 our approach is enlarged by variability concepts in order to specify variation points in a product line. Contributions of our approach are listed in Section 3. Finally, we compare our service model to related approaches in Section 4 before we conclude the paper in Section 5.

2 Service Diagram

In this section, we concentrate on the formal specification of the functionality of a system. First, we focus on specifying individual system functionalities and, subsequently, on structuring the functionalities and modeling their dependencies. This results in an hierarchical structure of the system functionality, the *Service Diagram*, where the overall functionality is subdivided in sub-services with defined relations between them. Thereby, the Service Diagram gives a specification of the system behavior as observable from the environment viewing the system as a black-box, i.e. the behavior is specified as a causal relation between input and output messages. Both, the individual services offered by a system and the user visible relations between them are specified. However, we do not consider the architecture of the system, i.e. the decomposition into components.

Formally, a Service Diagram consists of hierarchically subdivided services and three kinds of relationships between them, namely aggregation, functional dependency, and alternative relation (cp. Figure 1). These concepts will be introduced in the following subsections.

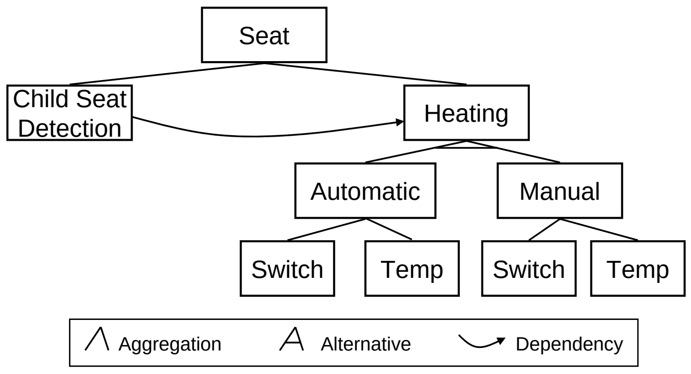

Fig. 1. Service Diagram

2.1 Single Service Specifications

The Service Diagram is based on the notion of a *service* as fundamental concept of the model. Intuitively, a service represents a piece of functionality by specifying a relation between certain inputs and outputs. Because this does not require any implementation details, a service precisely specifies the black-box behavior of a functionality. The definition of a service is based on the idea of timed streams as introduced in [5]. Mathematically, a service is a (partial) stream-processing function which maps streams of input messages to corresponding streams of output messages. Here, a stream s can be thought of as a (possibly infinite) ordered sequence of elements of type *Data*, or more precisely as a function

$$s: \quad \mathbb{N} \rightarrow Data.$$

The ordering corresponds to the progressing time in which certain messages are received or sent by a service.

Syntactic Interface Every service has a *syntactic interface* ($I \blacktriangleright O$), which consists of a set I of typed input ports and a set O of typed output ports.

In our example, the service `Manual` (cp. Figure 1), the manual version of the seat heating controller, has two input ports `switch` and `request` to receive messages from the switch button and from the "desired seat temperature" button as well as an output port `heater` to send control messages to the physical heating unit. The type of all these ports is defined by $type(p) = \mathbb{N} \cup \{\varepsilon\}$ – each message can be a natural number or ε. Note that no interaction is explicitly modeled by the empty message ε.

With each port we associate a stream representing the communication over this port. Formally, we model the mapping from ports to streams by introducing the notion of a *port history.* For a given set of ports P, a port history is a mapping which associates a stream of the appropriate type to each port:

$$h: \; P \to (\mathbb{N} \to \mathit{Data}).$$

$\mathbb{H}(P)$ denotes the set of all histories and $h \in \mathbb{H}(P)$ a possible history.

To be prepared for the definitions in the following sections, we introduce the notion of a *history projection.* Let P and P' be sets of typed ports with $P' \subseteq P$. We define for the history $h \in \mathbb{H}(P)$ its projection

$$h|P' \in \mathbb{H}(P')$$

to be the history containing only streams which are attached to the ports in P'. Furthermore, we use $h[p]$ to denote the stream associated with the port p by the history h, i.e. $h[p] = h(p)$. Then, the term $h[p](t)$ denotes the message contained in the stream $h[p]$ on the port p within the time interval $t \in \mathbb{N}$. In the rest of the paper, we briefly say that a service S receives/sends a message m through port p in time interval t to express that there exists a history $h \in \mathbb{H}(I_S \cup O_S)$ with $h[p](t) = m$.

In our example, one of the possible port histories of the service `Manual` with the ports `switch`, `request`, and `heater` is given by the mapping h_1:

$$h_1 : \begin{pmatrix} \mathit{switch} \\ \mathit{request} \\ \mathit{heater} \end{pmatrix} \mapsto \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & ... \\ 10 & 10 & 10 & 10 & 10 & 10 & ... \\ 20 & 20 & 20 & 20 & 20 & 20 & ... \end{pmatrix}.$$

$h_1[\mathit{switch}] = (111010\ldots)$ results in the corresponding stream of the switch button, and $h_1[\mathit{switch}](0) = 1$ denotes that the switch button is turned on in the first time interval.

Service Behavior There are several equivalent techniques to specify the behavior of a service, i.e. to specify the characteristic mapping from inputs to outputs. For our purpose, we use an assumption/guarantee notation (A/G) which consists of two formulas, namely, an assumption and a guarantee. The assumption (a predicate over inputs) specifies the domain of a service. Thus, we do not require that a service can react to every possible input, i.e. there may be inputs which are not explicitly covered by the service specification. The guarantee (a predicate over inputs and outputs) characterizes the reaction of a service to its inputs if and only if the inputs are in accordance with the assumption. Formally,

$$\begin{aligned} A &: \mathbb{H}(I) \to \mathbb{B}\mathit{ool}, \\ G &: \mathbb{H}(I) \times \mathbb{H}(O) \to \mathbb{B}\mathit{ool}. \end{aligned}$$

A service S with the syntactic interface $(I \blacktriangleright O)$ is defined as a relation from the set of input port histories (according to the assumption) to the powerset of output port histories (according to the guarantee):

$$S : \mathbb{H}(I) \rightarrow \mathcal{P}(\mathbb{H}(O))$$
$$S(x) \equiv \{y | A_S(x) \wedge G_S(x, y)\}.$$

Thus, a service is a *restrictive* specification which restricts the set of all syntactically correct histories to a subset of *valid* histories. Note, if an input history is not in accordance with the assumption, the service behavior is not defined ($S(x) = \emptyset$) – no output history can be classified as correct or not. Furthermore, by mapping to the powerset $\mathcal{P}(\mathbb{H}(O))$ we explicitly allow a service to have non-deterministic behavior.

In our example, the service `Switch` (a sub-service of `Manual`, to turn the heating on/off) has only one input port `switch`. It assumes one of the following messages on this port: 0 for off or 1 for on. The guarantee of this service specifies that there must be an empty message on the output port `heater` if a 0 is received on the port `switch` and any nonempty output if 1 is received. Formally, with $x \in \mathbb{H}(switch)$ and $y \in \mathbb{H}(heater)$:

$$\begin{aligned} A(x) &\equiv \forall t \in \mathbb{N} : x[switch](t) \in \{0, 1\} \\ G(x, y) &\equiv \forall t \in \mathbb{N} : x[switch](t) = 0 \Rightarrow y[heater](t + 1) = \varepsilon \wedge \\ &\qquad x[switch](t) = 1 \Rightarrow y[heater](t + 1) \neq \varepsilon. \end{aligned}$$

Thus, $\big(1\ 1\ 1\ 0\ ...\big)$ is a valid but $\big(3\ 1\ 1\ 0\ ...\big)$ an invalid input history for the service `Switch`. The I/O-pair $\big(1\ 1\ 1\ 0\ ...\big), \big(\varepsilon\ 30\ 30\ 30\ \varepsilon\ ...\big)$ is one possible behavior defined by the service.

The service `Temp` (the second sub-service of `Manual`, to control the seat temperature) can accept a natural number between 10 and 40 as a desired temperature request on the port `request` (`req`) and guarantees that either the same number or ε is sent through the port `heater` within the next time interval[1]:

$$\begin{aligned} A(x) &\equiv \forall t \in \mathbb{N} : x[req](t) \in [10..40] \\ G(x, y) &\equiv \forall t \in \mathbb{N} : y[heater](t + 1) \in \{x[req](t), \varepsilon\}. \end{aligned}$$

2.2 Aggregation

The aggregation relation allows to arrange individual services into a service hierarchy. It directly reflects the idea that the functionality offered by a system can be subdivided into several sub-functionalities. A service composed of several sub-services is called a *compound service*. The semantics of a compound service is defined as being a container of all *concurrently* operating sub-services. We do not specify the compound service explicitly, because its behavior can be derived from its sub-services using the well-defined semantics of the aggregation relation.

[1] We explicitly allow to send ε in order to ensure that the services `Temp` and `Switch` can be combined without conflicts (cp. Section 2.2).

The *interface* $(I_C \blacktriangleright O_C)$ of a compound service C composed of a set of sub-services S aggregates all I/O ports of all its sub-services:

$$I_C \equiv \bigcup_{s \in S} I_s, \quad O_C \equiv \bigcup_{s \in S} O_s.$$

A compound service is defined for all inputs for which *all* its sub-services yield a defined output as well. It is not defined for an input history, if there exists at least one sub-service, which is not defined for the corresponding history projection. Furthermore, the guarantees of all sub-services must hold in the compound service. Thus, the *behavior* of a compound service is defined as:

$$A_C(x) \equiv \bigwedge_{s \in S} A_s(x|I_s),$$
$$G_C(x,y) \equiv \bigwedge_{s \in S} G_s(x|I_s, y|O_s),$$

where $x \in \mathbb{H}(I_C)$, $y \in \mathbb{H}(O_C)$, $x|I_s \in \mathbb{H}(I_s)$, $y|O_s \in \mathbb{H}(O_s)$.

In our example, the service `Manual` can be obtained from its sub-services `Switch` and `Temp`. The compound service receives messages on its input ports `switch` (`sw`) and `request` (`req`) and sends corresponding messages through its output port `heater` (`h`):

$$\begin{aligned} A(x) &\equiv \forall t \in \mathbb{N} : x[sw](t) \in \{0,1\} \wedge x[req](t) \in [10..40] \\ G(x,y) &\equiv \forall t \in \mathbb{N} : x[sw](t) = 0 \Rightarrow y[h](t+1) = \varepsilon \wedge \\ & \qquad x[sw](t) = 1 \Rightarrow y[h](t+1) = x[req](t). \end{aligned}$$

Thus, the pair $\begin{pmatrix} 1 & 1 & 0 & 0 & ... \\ 30 & 30 & 30 & 30 & ... \end{pmatrix}, \begin{pmatrix} \varepsilon \; 30 \; 30 \; \varepsilon \; \varepsilon \; ... \end{pmatrix}$ is a valid behavior specified by the service `Manual`.

Note that the nondeterminism (underspecification) of the sub-services is often eliminated (in parts) in the compound service. Thus, the introduced nondeterminism of the sub-service `Temp` (cp. Section 2.1) is eliminated in combination with the sub-service `Switch`.

In the same way the automatic variant `Automatic` can be specified. It receives identical messages on its input ports `switch` (`sw`) and `request` (`req`) and, additionally, the current temperature of the seat on the port `measure` (`m`). Immediately after turning on, it sends the requested temperature through its port `heater` (`h`) and afterwards it adapts this instruction according to the difference between the actual and the desired seat temperature within each time interval. Exemplarily, the pair $\begin{pmatrix} 0 & 1 & 1 & 1 & ... \\ 23 & 23 & 23 & 23 & ... \\ 21 & 21 & 22 & 23 & ... \end{pmatrix}, \begin{pmatrix} \varepsilon \; \varepsilon \; 23 \; 24 \; 23 \; ... \end{pmatrix}$ is a valid behavior specified by the service `Automatic`.

2.3 Dependencies

By *dependencies*, we mean relations between services in a way that the behavior of one service influences the behavior of another one. Although, there are a lot of methodological significant dependencies like *enables*, *modifies* or *needs*, the scope of this paper is to approach a way of specifying of these relations rather than to enumerate them.

As our approach aims at a specification of the user-visible behavior during the requirements engineering phase, only these dependencies are specified which are observable at the overall system boundaries. Thereby, the interfaces of the affected services are not changed (no additional ports are added) nor is the communication between them characterized (no additional channels are added). Dependencies modify the user observable behavior of the influenced services without explicitly modifying their modular specifications. They specify a mapping from the original output port histories (specified by the modular A/G specification) to new ones. Depending on the original I/O port histories of all affected services the output port history of the influenced service is modified. Formally, a dependency d between an influencing service S_1 and an influenced service S_2 is a function of the form

$$d : \mathbb{H}(I_{S_1}) \times \mathbb{H}(O_{S_1}) \times \mathbb{H}(I_{S_2}) \times \mathbb{H}(O_{S_2}) \rightarrow \mathcal{P}(\mathbb{H}(O_{S_2})).$$

To simplify matters, we limited the definition to dependencies between two services. However, the extension to $n : 1$ dependencies is straightforward.

In our example, among other not further specified functionalities, the service `Child Seat Detection` (`CSD`) modifies the behavior of each variant of the service `Heating`. The specification requires that whenever the `CSD` detects a child seat mounted on the front seat, both heating variants are prevented from heating the front seat according to their modular specifications. The `CSD` permanently receives a message (1 or 0) from the environment through its port `in` whether a child seat is mounted or not. This results in the dependency formally defined by the following function:

$$\begin{aligned}
&d(x_{CSD}, y_{CSD}, x_H, y_H) \equiv y'_H \ \mathit{with} \\
&\quad \forall t \in \mathbb{N} : (x_{CSD}[in](t) = 0 \Rightarrow y'_H[heater](t+1) = y_H[heater](t+1)) \wedge \\
&\qquad\qquad (x_{CSD}[in](t) = 1 \Rightarrow y'_H[heater](t+1) = \varepsilon),
\end{aligned}$$

where y'_H denotes the modified output stream of the service `Heating`.

Aggregation with Dependency In Section 2.2 we introduced the aggregation of sub-services without any dependencies between them. Now, we enlarge this concept by the dependencies introduced so far.

The guarantee of a compound service aggregating two services S_1 and S_2 with a dependency d in-between (S_1 influences S_2) is defined as follows. In the compound service the guarantee of the influencing service S_1 must hold. Additionally, for a valid output history y of the compound service there must exist an output history y' that fulfills the guarantee of S_2, such that y' is transformed to $y|O_{S_2}$ by the dependency d.

$$\begin{aligned}G(x, y) \equiv \exists y' \in \mathbb{H}(O_{S_2}) :&\\ &G_{S_1}(x|I_{S_1}, y|O_{S_1}) \wedge G_{S_2}(x|I_{S_2}, y')\\ &\wedge y|O_{S_2} = d(x|I_{S_1}, y|O_{S_1}, x|I_{S_2}, y').\end{aligned}$$

Since a dependency does not influence input port histories of a service, the assumption of the compound service is not modified.

Consistent Specification The basic idea of our approach is that the overall specification is the combination of modularly specified sub-functionalities. Thereby, the modular specifications can overlap, i.e. different services might be defined over the same in- or output ports. Thanks to the formal definitions of the inter-service relations (aggregation and dependency), we can detect conflicts between functional requirements automatically and, thus, assure the consistency of a specification. A specification of a single product is consistent if there is no conflict between any services of this specification. There is a conflict between two services if their assumptions require different messages on a common input port or if they send different messages through a common output port in the same time interval. Formally, there is an input conflict between two sub-services if there is no valid input history for their common compound service C:

$$\{x \in \mathbb{H}(I_C) \mid A_C(x)\} = \emptyset,$$

and there is an output conflict if there exists a valid input history for which no valid output history is defined by their common compound service C:

$$\exists x \in \mathbb{H}(I_C) : A_C(x) \wedge \{y \in \mathbb{H}(O_C) \mid G_C(x, y)\} = \emptyset$$

2.4 Variability

The Service Diagram as introduced so far can only be used to specify a single software system. Now, we enlarge our approach by the concept of variability aiming at the explicit modeling of commonalities and differences of alternative services. In contrast to other approaches, e.g. FODA, which do not specify the behavior but only the presence or absence of certain features, we focus on explicitly modeling *behavioral* variability.

The basic concept to model variability are *variation points* (VP). Intuitively, a VP is a super-service composed of some alternative and/or optional sub-services (both are also called *variants*). By describing the behavior of a VP, we pay special attention to the following two topics. Firstly, we want to assure the *consistency* of service specifications of product families. The specification of a product family is consistent if each possible configuration[2] is consistent, i.e.

[2] By *configuration* we mean an instance of a product family specification where all VPs are resolved, i.e. certain variants are selected.

there are no conflicts between the services of the respective configuration. Secondly, we want to specify the interplay between the system under consideration and its environment, i.e. what the system demands and guarantees in the interaction with the surrounding systems. We specify how the environment must behave to correctly interact with all or with at least one of the configurations.

VPs can be seen as a special kind of services themselves, so that the same concepts can be used for their definition. In the following, we describe the syntactic interface and, subsequently, the behavior of a VP. Since an optional service can be transfered into an alternative VP, we only show the specification of VPs comprising alternative sub-services.

Syntactic interface An alternative VP comprising a set of mutually alternative services S_V has the *set-valued interface*

$$\mathbb{I}_{VP} \equiv \{(I_s \blacktriangleright O_s) | s \in S_V\}\,.$$

Only the histories which alternatively conform to the interface of *one* of the variants are *syntactically valid* inputs/outputs of the VP.

In our example, the VP `Heating` has the following set-valued interface:

$$\begin{aligned}\mathbb{I}_{Heating} \equiv \{(I_{AS} \blacktriangleright O_{AS}), (I_{MS} \blacktriangleright O_{MS})\} \equiv \{&(\{switch, measure\} \blacktriangleright \{heater\}),\\ &(\{switch, measure, request\} \blacktriangleright \{heater\})\}.\end{aligned}$$

Thus, in combination with the aggregation relation, we are able to specify the interface of a product line. The mandatory and optional ports of a system can be easily identified by means of the set-theoretical operations over the set-valued interfaces.

Furthermore, to provide the syntactical basis for the semantical specifications, we introduce the *maximum interface* $(I_{max} \blacktriangleright O_{max})$ to describe the sum of all possible ports of a VP and the *minimum interface* $(I_{min} \blacktriangleright O_{min})$ to describe the common ports of its variants. For a set S_V of alternative services they are defined as follows:

$$\begin{aligned}I_{max} \equiv \bigcup_{s \in S_V} I_s, \quad O_{max} &\equiv \bigcup_{s \in S_V} O_s\\ I_{min} \equiv \bigcap_{s \in S_V} I_s, \quad O_{min} &\equiv \bigcap_{s \in S_V} O_s.\end{aligned}$$

Behavioral Semantics The behavioral specification of a VP is the disjunction of the specifications of its variants. Before we give a precise definition of the behavior of a VP, we informally describe the set of histories defined by it.

The assumption of a VP describes all input histories which are valid for at least one variant. An input history is in accordance with the assumption of the VP if at least one projection to the interface of a sub-service fulfills the assumption of this sub-service. In terms of sets, this assumption describes the union of the *enlarged sets* of input histories of all variants. Thereby, an enlarged set is defined over the maximum interface and contains histories which projections to the original interface of a variant satisfy the assumption of this variant, while the projections to the other ports of the maximum interface are not subject to any restrictions. Formally, the set specified by the assumption of a VP is defined as $\bigcup_{s \in S_V} \{x \in \mathbb{H}(I_{max}) \mid A_s(x|I_s)\}$.

The guarantee of a VP describes all I/O history pairs which are valid for at least one variant. An I/O history pair is in accordance with the guarantee of a VP if there exists at least one variant which assumption *and* guarantee are fulfilled by the projected I/O history pair. In terms of sets, the guarantee of a VP describes the union of the enlarged sets of I/O history pairs of all variants: $\bigcup_{s \in S_V} \{(x,y) \in \mathbb{H}(I_{max}) \times \mathbb{H}(O_{max}) \mid A_s(x|I_s) \wedge G_s(x|I_s, y|O_s)\}$.

Thus, the specification of a VP combining a set S_V of alternative services is defined over the maximum interface ($x \in \mathbb{H}(I_{max})$ and $y \in \mathbb{H}(O_{max})$) by the following formula:

$$\begin{aligned} A_{VP}(x) &\equiv \bigvee_{s \in S_V} A_s(x|I_s) \\ G_{VP}(x,y) &\equiv \bigvee_{s \in S_V} (A_s(x|I_s) \wedge G_s(x|I_s, y|O_s)). \end{aligned}$$

In our example, the specification of the VP `Heating` specifies I/O histories which can be accepted/produced if either the manual `Manual` or the automatic `Automatic` heating is chosen:

$$\begin{aligned} A_{Heating}(x) &\equiv A_{man}(x|I_{man}) \vee A_{aut}(x|I_{aut}) \\ G_{Heating}(x,y) &\equiv (A_{man}(x|I_{man}) \wedge G_{man}(x|I_{man}, y|O_{man})) \vee \\ &\quad\ (A_{aut}(x|I_{aut}) \wedge G_{aut}(x|I_{aut}, y|O_{aut})). \end{aligned}$$

E.g., the pair $\left(\begin{array}{c} 1\ \ 1\ \ 0\ \ 0\ ... \\ 30\ 30\ 30\ 30\ ... \\ 70\ 70\ 70\ 70\ ... \end{array}\right)$, $\left(\varepsilon\ 30\ 30\ \varepsilon\ \varepsilon\ ...\right)$ fulfills the specification of the VP `Heating` because its projection to the interface of the service `Manual` (first two lines) fulfills the specification of `Manual`. However, it is not in accordance with the specification of `Automatic` since it does not fulfill the corresponding assumption (third line: $x[m](t) = 70 \notin [0..50]$). The pair $\left(\begin{array}{c} 0\ \ 0\ \ 0\ ... \\ 30\ 30\ 30\ ... \\ 70\ 70\ 70\ ... \end{array}\right)$, $\left(\varepsilon\ \varepsilon\ 30\ 30\ ...\right)$ does not fulfill the specification of the VP – no variant of the heating is allowed to send a nonempty output if it is off.

The specification of a VP defines the history set which exactly contains the histories that can be accepted/produced by at least one of the variants. The consistency of this specification assures the consistency of the specification of each single configuration. Also, this specification defines the maximum requirements on the environment to correctly interact with all configurations of the system. These requirements are sufficient to guarantee that the environment can handle the output of any configuration. However, this specification identifies no commonalities between variants. Thus, to focus on commonalities between variants and to perform consistency checks more efficiently, we introduce the *common specification* of a VP.

Common Specification The common specification is designed to derive properties for a VP independently from a concrete configuration. It defines the behavior that all alternative variants have in common, i.e. the behavior that can be definitively expected from a VP. This specification defines input histories that can be accepted by all variants. The defined output histories are only specified within time intervals in which all variants produce the same message. In all other intervals as well as on non-common ports $o' \notin O_{min}$ the VP has a totally non-deterministic behavior. Thus, this specification only defines a part of the behavior – the common behavior.

Note that the common specification is defined over the maximum interface $(I_{max} \blacktriangleright O_{max})$. This is necessary because the histories on different ports are not independent in general. The assumption of a variant $s \in S_V$ with $I_s \supset I_{min}$ might specify dependencies between histories on input ports $i \in I_{min}$ and histories on input ports $i' \notin I_{min}$. The same goes for the guarantees. By regarding only the common ports these dependencies might get lost.

The *common assumption* specifies the set of input histories which can be processed by *all* variants. The set defined by this assumption is the intersection of the enlarged sets of input histories of each alternative sub-service. Thereby, the set of input histories of a variant is enlarged equivalently to the disjunctive assumption: the messages on the ports originally not present in the interface of a sub-service are not subject to any restrictions. Formally, the set specified by the common assumption is defined as $\bigcap_{s \in S_v} \{x \in \mathbb{H}(I_{max}) | A_s(x|I_s)\}$.

The *common guarantee* assures the common behavior of all variants for certain time intervals. Given a valid input history, it guarantees an unambiguous reaction for each common output port $o \in O_{min}$ and each time interval in which all variants cause the same reaction. As the common specification focuses on modeling commonalities rather than differences, this guarantee assures nothing (i.e. totally non-deterministic behavior) on a port $o \in O_{min}$ for a time interval in which two variants specify different reactions. Thus, the valid output histories of a VP are only predefined within time intervals in which all variants produce the same message. The messages within other time intervals as well as the histories of $o' \notin O_{min}$ are not subject to any restrictions. Formally, the common behavior of a VP combining a set S_V of different alternative services is defined for $x \in \mathbb{H}(I_{max})$ and $y \in \mathbb{H}(O_{max})$ by

$$
\begin{aligned}
&A_{comm}(x) \equiv \bigwedge_{s \in S_V} A_s(x|I_s) \\
&G_{comm}(x, y) \equiv \forall t \in \mathbb{N}, o \in O_{min} : \\
&\quad y[o](t) \in \{v \in type(o) \mid \forall s \in S_V : \exists y_s \in s(x|I_s) : y_s[o](t) = v\}.
\end{aligned}
$$

Whenever an input history is in accordance with this assumption, the projection to the interface of any variant is a valid input history for this variant. Whenever all variants produce the same message on a common output port within a time interval, also the common guarantee assures this message on this port within this interval. Otherwise, the behavior is totally non-deterministic. As a consequence, no projection of an I/O history pair that violates the common guarantee of a VP, fulfills the guarantee of one of its variants. Thus, inconsistencies of the common specification and, consequently, of the specifications of each single configuration can be detected and the consistency of the common behavior can be assured. Additionally, the common specification defines the minimum requirements that the environment must necessarily fulfill to be able to process the outputs of all possible configurations. However, these requirements are not sufficient to assure the correct interplay with all variants.

In our example, the VP `Heating` comprises the alternatives manual heating `Manual` and automatic heating `Automatic` (defined in Section 2.2). The projections of all enlarged input histories that fulfill the common assumption of `Heating` fulfill the corresponding assumptions of `Manual` and `Automatic`. Regarding the guarantee, the manual and automatic specifications show the same reaction

– if the switch is turned off,
– immediately after the switch is turned on, or
– if the switch is turned on and the instruction calculated by the automatic variant equals the temperature requested by the manual variant.

In all other cases, both variants behave differently. Consequently, the common specification requires nothing and consequently accepts any arbitrary message. Thus, if an I/O history pair violates the common guarantee of `Heating`, its projections neither fulfill the guarantee of `Manual` nor of `Automatic`. E.g., the pair $\begin{pmatrix} 1 \;\; 1 \;\; 0 \;\; 0 \;\; 1 \;... \\ 25\;25\;25\;25\;25\;... \\ 25\;26\;25\;23\;23\;... \end{pmatrix}$, $\left(\varepsilon\;\varepsilon\;25\;\varepsilon\;\varepsilon\;25\;...\right)$ violates the common specification of the VP `Heating` (because of the second ε on the port `heater`) and, consequently, each projection to the interface of the service `Manual` (first two lines) or `Automatic` (all three lines) violates the specification of the respective sub-service.

Note that the specification of each variant is a behavioral refinement of the common specification. We can easily integrate a new variant into this VP as long as it does not violate the common specification, i.e. as long as the new variant guarantees the common behavior. Since we demand no specific message if the existing variants already behave differently, in this cases the new variant can behave completely different to all existing variants.

Consistency As mentioned before, the specification of a product line is consistent if each its possible configuration is consistent. However, the effort to perform the consistency checks for all possible configurations separately would grow exponentially. Thus, we use the common specification to reduce the effort and perform consistency checks more efficiently. Based on the common specification, the consistency of the common behavior can be assured and must be checked only once. If there is a conflict between the common specification of a VP and another service, there is definitely a conflict between any variant of the VP and the other service. However, since the common specification only describes the common behavior, a conflict between a variant of the VP and the other service might remain undetected based on this specification. Thus, the disjunctive behavioral specification is needed to assure the correctness of the whole product line.

Assuring the consistency of product line specifications is one of the main purpose of our approach and a major part of our current and future work. However, due to space limitation we only present underlying concepts for this analysis in this paper.

3 Contributions

Having introduced the formal foundation of the underlying concepts in the previous sections, we shortly sketch the potential of our approach in the following.

Formalization of requirements In contrast to pure informal approaches like FODA, we introduce a formal model with a well-defined semantics for describing the functionality already at an early stage of the development process. This has several advantages: Firstly, a formal model which formalizes (functional) requirements allows an automatic analysis of the system already in the early phases of the development process. By this, discrepancies between conflicting functionalities can be detected and resolved. Secondly, since implementation models will build upon this functional specification, it supports bridging the formal gap between functional requirements and design models.

Modularity The complexity of multi-functional systems requires to design a system in a modular fashion by splitting up the system into an appropriate set of different sub-services and generating the overall system as a combination of them. Consequently, for construction as well as for verification issues, this modularity can ease certain problems immensely.

Functional variability While traditional approaches mainly focus on structural aspects, we give a detailed specification of the behavioral variability. The presented specifications of behavioral variability open up new possibilities, e.g.:

- predictions about a product line without knowing the concrete configuration,
- consistency checks/conflict detection between variants,
- deriving unknown dependencies between variants based on their behavior,
- seamless integration of new variants.

4 Related Work

The approach presented in this paper introduces a formal model for specifying functionality and functional variability of system families. Thus, our related work can be found in the area of the formalization of feature models – the main method to formalize the variability in product lines.

The definition of a formal semantics for feature models is not new. In [6], Batory and O'Malley use grammars to specify feature models. In [7], Czarnecki et al. argue that cardinality-based feature models can be interpreted as a special class of context-free grammars. Sun et al. define a formal semantics for the feature modeling language using first-order logic in [8]. The formalization of feature models with propositional formulas goes back to the work by Mannion [9], in which logical expressions can be developed for the model, using propositional connectives to model dependencies between requirements. Another approach to specifying multi-functional systems is introduced by van Lamsweerde et al. In [10] they propose formal techniques and heuristics for detecting conflicts from specifications of goals (requirements) and their interactions specified in LTL.

The main deficit of these approaches is a disregard for the behavior of single features. In [11], Czarnecki and Antkiewicz recognize that features in a feature model are "merely symbols". They propose an approach to mapping feature models to other models, such as behavior or data specifications, in order to give them semantics. However, this approach only focuses on assets like software components and architectures. Our work focuses on formalizing user requirements and their analysis in the early phases of the development process.

The closest approach to our work is a theoretical framework introduced by Broy [5] where the notion of a service behavior is formally defined. This framework provides several techniques to specify and to combine services based on their behaviors. However, this approach does not cover several relevant issues such as techniques for the specification of functional variability and of inter-service dependencies.

To summarize, to the best of our knowledge, there is no approach to specify a product line, by formally describing the behavioral variability in requirements.

5 Conclusion and Future Work

The presented concepts can be roughly summarized as follows. We introduced and formally founded the underlying concepts of our service specification, which focuses on the modeling and structuring of functional requirements without any further technical details. Thereby, the concept of a service is used to model single functionalities in a modular fashion. Together with a well-defined meaning for inter-service relations we are able to reduce the specification of the overall system behavior to the specification of individual sub-functionalities. This helps to master the complexity of multi-functional systems.

With respect to product lines, we integrated the concept of behavioral variability which makes the Service Diagram suitable to formally capture the functional requirements of a system family. Thereby, we focused on explicitly modeling behavioral commonalities and differences between variants.

The formal description of the functionality and the behavioral variability already at an early stage of the development process allows to perform a formal (and therefore automatic) analysis of the functional requirements, i.e. to detect conflicting functionalities. Inter-service conflicts and consistency checks as well as the analysis of interactions between different variants are in the focus of our approach and major part of our current and future work.

An other concern of our future work is the development of a user-friendly syntax for the semantics introduced in this paper.

References

1. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, SEI, CMU, Pittsburgh (1990)
2. Gruler, A., Harhurin, A., Hartmann, J.: Modeling the functionality of multi-functional software systems. In: Proceedings of ECBS07. (2007)
3. Gruler, A., Harhurin, A., Hartmann, J.: Development and configuration of service-based product lines. In: Proceedings of SPLC07. (2007)
4. Schätz, B.: Combining product lines and model-based development. In: Proceedings of Formal Aspects of Component Systems (FACS 2006). (2006)
5. Broy, M.: Service-oriented systems engineering: Modeling services and layered architectures. In: FORTE. (2003) 48–61
6. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Trans. Softw. Eng. Methodol. **1** (1992)
7. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice **10** (2005) 7–29
8. Sun, J., Zhang, H., Wang, H.: Formal semantics and verification for feature modeling. In: Proceedings of ICECCS05. (2005) 303–312
9. Mannion, M.: Using first-order logic for product line model validation. In: SPLC. (2002) 176–187
10. van Lamsweerde, A., Letier, E., Darimont, R.: Managing conflicts in goal-driven requirements engineering. IEEE Trans. Softw. Eng. **24** (1998) 908–926
11. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE. (2005) 422–437