Towards Consistent Specifications of Product Families[*]

Alexander Harhurin and Judith Hartmann

Institut für Informatik, TU München,
Boltzmannstr. 3, D-85748, Garching bei München, Germany
{harhurin|hartmanj}@in.tum.de

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, in particular, the functional variability already in the requirement engineering phase. In this paper, we precisely define the underlying concepts, such as the notion of individual services, the combination of services, inter-service dependencies, and variability. Thereby, we especially focus on establishing the *consistency* of the overall specification. To that end, we formally define conflicts between requirements and describe how they can be detected and resolved based on the introduced formal concepts.

1 Introduction

Today, in various application domains, e.g. the automotive domain, software plays a dominant role. The rapid increase in the amount and importance of different software-based 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. As a consequence there is a need for adequate modeling techniques for functional requirements. Prevalent approaches like UML Use Cases or 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 a formal foundation, discrepancies between conflicting functionalities can be detected and resolved already in the early phases of the development process. Furthermore, such a formal specification represents the first model in a model-based system development along different abstraction levels as introduced in [2]. It serves as a formal basis for the construction and verification of the models in the consecutive design phase. Consequently, we focus on the formal definitions of functional requirements and relations between them, and show how the upcoming service-oriented paradigm is used to handle the aforementioned functional intricacy.

[*] 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.

Our notation technique, the *Service Diagram*, informally introduced in [3] describes the system as a set of related functional requirements (*services*). Regarding product families, our approach includes concepts which allow for the formal specification of functional variability. Thereby, functional variability means that the specification includes alternative functional requirements. Each variant of the product family is required to satisfy at least one of the alternative requirements. Thus, the denotational semantics of our Service Diagram specifies the behavior of a product family as the behavior that can be delivered by at least one of its variants. Also, we precisely define the meaning of typical dependencies for product families, namely *excludes* and *requires*. These dependencies specify which requirements must and which ones must not be simultaneously satisfied by a variant of a product family.

In this paper, we especially focus on understanding how single services depend on and interfere with each other. Thereby, the main goal of our approach is to ensure the consistency of the specification, i.e. the absence of conflicts between services. Informally, there is a conflict between two services if they impose conflicting requirements on the behavior of a system which can not be simultaneously fulfilled. Giving formal definitions of these concepts, our approach can be used for a tool-supported analysis of the functional requirements and, in particular, for consistency checks between different variants.

1.1 Running Example

The concepts introduced in the remainder of the paper will be illustrated by a simplified example of a cruise control (cp. Figure 1). The cruise control comprises a manual cruise control (`MCC`) and an adaptive cruise control (`ACC`). The MCC specifies the acceleration/deceleration of the vehicle triggered by the acceleration/brake pedal (`Pedal`). Additionally, there is an option to control the speed via buttons on the steering wheel (`Steering Wheel`). The ACC comprises an automatic speed control (`Speed`), which controls the vehicle speed for a constant target speed. There exist two alternative variants varying in the way how the target speed is selected by the driver (target speed arbitrarily configurable (`Input`) or target speed set to the current vehicle speed when the ACC is activated (`Save`)). Furthermore, the ACC optionally comprises a follow-up control (`Follow-Up`) to automatically follow a target vehicle and a pre-crash control (`Pre-Crash`). There exist two variants of the pre-crash control, one which displays a warning (`Warning`) and one which actively brakes (`Brake`) as soon as a potential crash is detected. There are several dependencies between these functionalities to assure a correct interplay between them. The dependencies as well as all other relevant details will be described at the appropriate places.

1.2 Outline

The rest of this paper is organized as follows: In Section 2 the semantics of the *Service Diagram* is presented. In particular, we explain the formal specification of functional requirements by means of services, concepts for hierarchically structuring services, variability concepts, and concepts for modeling dependencies between services. In Section 3, we concentrate on the consistency of a service specification. To that end, we formally define conflicts and describe how they can be detected and resolved based on the introduced formal concepts. Contributions of our approach are listed in Section 4. Finally, we compare our service model to related approaches in Section 5 before we conclude the paper in Section 6.

2 Service Diagram

This section introduces the denotational semantics of the *Service Diagram*, a hierarchical model for the specification of the system functionality. This diagram gives a black-box specification of a system, i.e. the system behavior is specified as a causal relation between input and output messages. Thus, 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 subdivided services and four kinds of relationships between them, namely aggregation, functional dependencies, optional and alternative relations (cp. Figure 1). All these concepts will be introduced in the following subsections. A more detailed description of the basic concepts can be found in [4].

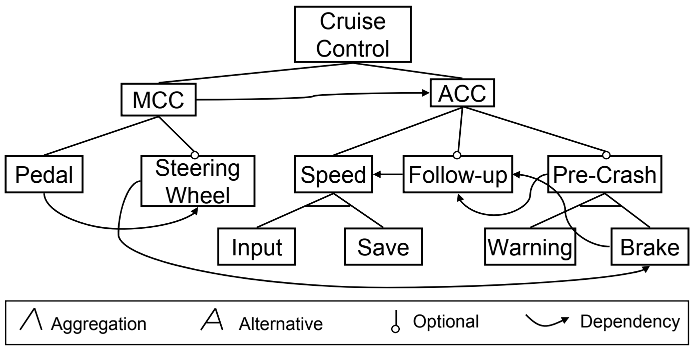

Fig. 1. Service Diagram for the Cruise Control

2.1 Single Service

The Service Diagram is based on the notion of a *service* [5] as the fundamental concept of the model. Intuitively, a service represents a piece of functionality by specifying requirements on the I/O behavior. More precisely, a service specifies a relation between certain inputs and outputs. Hence, the Service Diagram is a restrictive specification, where each service imposes a requirement on the system and, thus, further restricts the valid I/O behavior. Formally, a service is a (partial) stream-processing function which maps streams of input messages to corresponding streams of output messages. Here, a stream s of elements of type Data can be thought of as a function $s: \quad \mathbb{N} \rightarrow Data$.

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.

Table 1 depicts the syntactical interfaces of the atomar services of our example. Exemplary, the type of the port `currentSpeed` is $\mathbb{N}$ and the type of the port `speed` is $\{accelerate, decelerate, \varepsilon\}$. Note, that if necessary the type of a port includes the empty message ε to explicitly model no interaction.

Service	I Ports	O Ports
MCC/Pedal	brakePedal, accPedal	speed
MCC/Steering Wheel	brakeButton, accButton	speed
ACC/Speed/Input	currentSpeed, targetSpeed	speed
ACC/Speed/Save	active, currentSpeed	speed
ACC/Follow-Up	objectDetected, objectDistance, currentSpeed	speed
ACC/Pre-Crash/Warning	objectDetected, objectDistance, currentSpeed	warning
ACC/Pre-Crash/Brake	objectDetected, objectDistance, currentSpeed	speed

Table 1. Syntactical Interfaces of the Modular Services of the Cruise Control

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: $h: \quad P \rightarrow (\mathbb{N} \rightarrow Data)$. $\mathbb{H}(P)$ denotes the set of all such histories. $\mathbb{H}(I_s) \times \mathbb{H}(O_s)$ specifies the set of all *syntactically correct* I/O history pairs (x, y) for a service s with interface $(I_s \blacktriangleright O_s)$. For a history $h \in \mathbb{H}(P)$, we define its projection $h|P' \in \mathbb{H}(P')$ to be the history containing only streams which are attached to the ports in $P' \subseteq P$. Also, we denote a projection of a history x to the interface of a service s by x_s, i.e. $x_s = x|I_s$. The same goes for an I/O history pair: $(x_s, y_s) = (x|I_s, y|O_s)$. 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 time interval $t \in \mathbb{N}$.

Semantics To specify the behavior of a service, we use an assumption/guarantee notation (A/G) which consists of two predicates, namely, an assumption and a guarantee. The assumption specifies the domain of a service[1]. The guarantee characterizes the reaction of a service to its inputs if the inputs are in accordance with the assumption. Formally,

$$A : \mathbb{H}(I) \rightarrow \mathbb{B}ool, \qquad G : \mathbb{H}(I) \times \mathbb{H}(O) \rightarrow \mathbb{B}ool.$$

[1] 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.

By this, a service is a *restrictive* specification which restricts the set of all syntactically correct histories to a subset of *(semantically) valid* histories. An I/O history pair (x, y) is valid for a service if it fulfills the A/G of this service. We say, the behavior of a service is the set of all valid history pairs for this service. Formally, 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):

$$s : \mathbb{H}(I) \rightarrow \mathcal{P}(\mathbb{H}(O)), \qquad s(x) \equiv \{y | A_s(x) \wedge G_s(x, y)\}.$$

In our example, the variant `Input` of the speed control is specified as follows:

$$\begin{aligned}
&A(x) \equiv \forall t \in \mathbb{N} : x[\mathit{currentSpeed}](t) \in [20..220] \wedge x[\mathit{targetSpeed}](t) \in [40..200]\\
&G(x, y) \equiv \forall t \in \mathbb{N} :\\
&\quad x[\mathit{currentSpeed}](t) > x[\mathit{targetSpeed}](t) \Rightarrow y[\mathit{speed}](t+1) = \mathit{decelerate} \wedge\\
&\quad x[\mathit{currentSpeed}](t) < x[\mathit{targetSpeed}](t) \Rightarrow y[\mathit{speed}](t+1) = \mathit{accelerate} \wedge\\
&\quad x[\mathit{currentSpeed}](t) = x[\mathit{targetSpeed}](t) \Rightarrow y[\mathit{speed}](t+1) = \varepsilon
\end{aligned}$$

The assumption formalizes, that the behavior of the cruise control is only specified for current speed between 20 and 220 km/h and target speed between 40 and 200 km/h. In our terminology, a valid history x must contain a value between 20 and 220 on port `currentSpeed` and a value between 40 and 200 on `targetSpeed` within each time interval t. Otherwise, the behavior is not defined. The guarantee requires for each time interval, that the vehicle accelerates if the current vehicle speed is less, decelerates if the current speed is higher, and neither accelerates nor decelerates if the current speed is equal to the target speed.

2.2 Aggregation

The aggregation relation allows to arrange individual services into a service hierarchy. The semantics of a *compound service* (composed of several sub-services) is defined as being a container of all *concurrently* operating sub-services.

The interface of a compound service s_C composed of a set of sub-services S aggregates all I/O ports of all its sub-services. Its behavior is defined as the conjunction of the modular A/Gs of its sub-services. Formally,

$$A_{s_C}(x) \equiv \bigwedge_{s \in S} A_s(x_s), \qquad G_{s_C}(x, y) \equiv \bigwedge_{s \in S} G_s(x_s, y_s). \tag{1}$$

A more detailed description of the aggregation relation including illustrating examples can be found in [4].

2.3 Variability

The basic concept to model variability are *variation points* (VPs). Intuitively, a VP is a compound service composed of some mandatory, alternative, and/or optional sub-services (the latter two are also called *variants*). In the following, the syntactic interface and the behavior of a VP comprising alternative sub-services are introduced. Subsequently, we explain the semantics of optional services based on the definitions for alternative VPs. To understand the following definitions, it is important to keep in mind that our Service Diagram is a *restrictive* specification. Each service in the Service Diagram imposes a requirement on the I/O behavior which must be fulfilled by any valid I/O history. If a service is absent in a diagram (e.g. the service is not selected in a configuration[2]), the property specified by this service is not required. However, a valid I/O history is not prohibited from fulfilling this property.

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

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

Herewith, in combination with the aggregation relation, we can specify the interface of a product family. Mandatory and optional ports can be easily identified by means of the set-theoretical operations over the set-valued interface.

We call a history pair (x, y) *syntactically correct* for a VP if it conforms to the interface of one of its variants, i.e. if $\exists \ (I \blacktriangleright O) \in \mathbb{I}_{VP} : (x, y) \in \mathbb{H}(I) \times \mathbb{H}(O)$.

To be able to aggregate VPs, the definition of the history projection (see Section 2.1) must be adapted for set-valued interfaces. Since a VP comprises a set of interfaces, the projection to the interface of a VP results in a set of histories. For an I/O history pair (x, y) this projection is defined as follows:

$$(x, y)|\mathbb{I}_{VP} \equiv \{(x_s, y_s)|(I_s \blacktriangleright O_s) \in \mathbb{I}_{VP}\}. \tag{2}$$

Semantics Each alternative VP specifies a set of history pairs which are valid for at least one of its variants. Thereby, for the definition of the semantics, the syntactic interface must be taken into account. A history pair (x, y) defined over the interface $(I \blacktriangleright O)$ is valid if it fulfills the A/G specification of one of the variants with the same interface:

$$\begin{aligned} A_{VP}(x) &\equiv \exists s \in S_V : x \in \mathbb{H}(I_s) \wedge A_s(x) \\ G_{VP}(x, y) &\equiv \exists s \in S_V : (x, y) \in \mathbb{H}(I_s) \times \mathbb{H}(O_s) \wedge (A_s(x) \wedge G_s(x, y)). \end{aligned} \tag{3}$$

The assumption of a VP describes all input histories which are valid for at least one variant, in terms of sets: $\bigcup_{s \in S_V} \{x \in \mathbb{H}(I_s) \mid A_s(x)\}$. The guarantee of a VP describes all I/O history pairs which are valid for at least one variant, in terms of sets: $\bigcup_{s \in S_V} \{(x, y) \in \mathbb{H}(I_s) \times \mathbb{H}(O_s) \mid A_s(x) \wedge G_s(x, y)\}$. Note, according to Definition 1, the projection of a valid history pair of the compound service must fulfill the A/G of a VP if this VP is a sub-service of the compound service. According to Definition 2, the projection of a history pair to the interface of a

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

VP yields a set of history pairs. Thereby, a set of projected history pairs satisfies the specification of a VP if at least one of the pairs fulfills this specification.

In our example, the speed control `Speed` is a VP comprising two alternative variants (`Input` and `Save`). The specification of `Input` is given in Section 2.1. The specification of `Save` differs in the way how the target speed is selected. The target speed is set to the current speed if the speed control is activated. More precisely, let `Save` be active (value 1 on port `active`) exactly in the interval $[t_1..t_2]$. Then, in the interval $[t_1+1..t_2+1]$, the target speed for the speed instruction on port `speed` is equal to the value on port `currentSpeed` in t_1. The corresponding A/G formulas are similar to those of `Input` and, therefore, not explicitly specified here. The VP `Speed` conjoins the behaviors of both variants. So, `Speed` defines the set of all valid history pairs that fulfills the A/G of `Input` or `Save`. The syntactical interface $\mathbb{I}_{Speed}$ is obtained according to the definition of the set-valued interface: $\{(\{currentSpeed, targetSpeed\} \blacktriangleright \{speed\}), (\{currentSpeed, active\} \blacktriangleright \{speed\})\}$. The A/G of `Speed` is easily derivable according to Definition 3, but due to space limitation not presented here.

Optional Service Intuitively, an optional service s_o represents an alternative between the presence and the absence of this service within the Service Diagram. Consequently, it can be transfered into an alternative VP. This VP consists of two alternatives, namely the service s_o and *no service*. If the optional service is selected, a valid history must fulfill the requirements specified by the service s_o. If no service is selected, a valid history does not have to fulfill these requirements. Formally, no service is described by a special service s_Ω which has no ports ($I_{s_\Omega} \equiv O_{s_\Omega} \equiv \emptyset$) and is always fulfilled ($A_{s_\Omega}(x) \equiv G_{s_\Omega}(x,y) \equiv true$). Thus, s_Ω imposes no requirement on the I/O behavior of the system. Consequently, s_Ω acts as identity element concerning the aggregation relation, i.e. the aggregation of any service s and s_Ω results in s.

In our example, the service `Follow-Up` to control the speed based on the distance to a vehicle in front is optional. Thus, it can be transfered into a VP with syntactical interface $\mathbb{I} \equiv \{(\emptyset \blacktriangleright \emptyset), (I_{Follow-Up} \blacktriangleright O_{Follow-Up})\}$. According to Definition 3, this VP (i.e. the optional service `Follow-Up`) defines the set of history pairs (x, y) with either $(x, y) \in \mathbb{H}(\emptyset) \times \mathbb{H}(\emptyset)$ or $(x, y) \in \mathbb{H}(I_{Follow-Up}) \times \mathbb{H}(O_{Follow-Up})$ and (x, y) in accordance with the specification of `Follow-Up`.

2.4 Dependencies

By dependencies, we mean relations between services in a way that the behavior of one service influences the behavior of another one. 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. Dependencies between services can be explicitly given by functional requirements or they are introduced during the aggregation process to solve conflicts between services (see Section 3).

In the following, we introduce two kinds of dependency relations: *dependency predicates* and *dependency functions*. Dependency predicates formalize additional requirements on the I/O behavior and, thus, further restrict the set of valid I/O histories. Dependency functions, however, modify the user observable behavior of the influenced services without explicitly modifying their modular specifications. Having introduced these relations, we show how the behavior of a compound service composed of several sub-services is defined in consideration of the dependencies in-between. To simplify matters, we limit the following formal definitions to dependencies between two services. However, the extension to $m : n$ dependencies is straightforward.

Dependency Predicates A dependency predicate describes further restrictions on the inputs or outputs of the services. Formally, a predicate between the services s_1 and s_2 specifies a relation between messages on I/O ports of s_1 and s_2 in certain time intervals:

$$dPr : \ \mathbb{H}(I_{s_1}) \times \mathbb{H}(O_{s_1}) \times \mathbb{H}(I_{s_2}) \times \mathbb{H}(O_{s_2}) \rightarrow \mathbb{B}ool.$$

Dependency Functions A dependency function specifies a mapping from the original output histories (specified by the modular A/G specification) to new ones. This transformation of output histories greatly supports the modularity of our approach since single services can be specified without considering the interaction with other services. This is especially suitable in the context of product families where the context, i.e. the interaction with other services, may differ from variant to variant. Formally, a dependency function $dFct$ between an influencing service s_1 and an influenced service s_2 is a function of the form

$$dFct : \mathbb{H}(I_{s_1}) \times \mathbb{H}(O_{s_1}) \times \mathbb{H}(I_{s_2}) \times \mathbb{H}(O_{s_2}) \rightarrow \mathcal{P}(\mathbb{H}(O_{s_2})).$$

In our example, there is a dependency function between the services MCC and ACC. The application of the brake or accelerator (pedal or button) immediately deactivates the ACC. Whenever the MCC requires a nonempty speed instruction on port speed, the speed instruction calculated by the ACC is overwritten by those of MCC. This dependency is formalized as follows:

$$\begin{aligned} d(x_m, y_m, x_a, y_a) \equiv y'_a : \forall t \in \mathbb{N} : & y_m[speed](t) = \varepsilon \Rightarrow y'_a[speed](t) = y_a[speed](t) \\ & \wedge y_m[speed](t) \neq \varepsilon \Rightarrow y'_a[speed](t) = y_m[speed](t). \end{aligned}$$

Analogously, a further dependency between ACC and MCC determines that empty speed instructions of MCC are overwritten by those of ACC. Furthermore, there is a dependency function between the services Pedal and Steering Wheel. This dependency resolves situations where the services require different instructions on the common port speed, e.g. when the driver simultaneously presses a pedal and a button. In this case the output history of Steering Wheel is modified, i.e. Pedal overrules Steering Wheel. Analogously, the Follow-Up control overrules the Speed control. The formalizations of these dependences are very similar to the foregoing one and therefore omitted here due to the limitation of space.

Aggregation with Dependency For each kind of dependency relation, the behavior of the compound service s_C composed of two sub-services s_1 and s_2 and a dependency d in-between is defined in the following paragraphs.

If d specifies a dependency predicate between s_1 and s_2 which restricts the outputs of the services, the additional predicate must hold in the compound guarantee:

$$G(x, y) \equiv G_{s_1}(x_{s_1}, y_{s_1}) \wedge G_{s_2}(x_{s_2}, y_{s_2}) \wedge d(x_{s_1}, y_{s_1}, x_{s_2}, y_{s_2}). \tag{4}$$

If the dependency predicate affects the input histories, the compound assumption has to be modified analogously.

If d is a dependency function (s_1 influences s_2), the guarantee of the compound service is defined as:

$$\begin{aligned} G(x, y) \equiv \exists y' \in \mathbb{H}(O_{s_2}) : G_{s_1}(x_{s_1}, y_{s_1}) \wedge G_{s_2}(x_{s_2}, y') \\ \wedge\, y_{s_2} \in d(x_{s_1}, y_{s_1}, x_{s_2}, y'). \end{aligned} \tag{5}$$

In the compound service the assumption and guarantee of the influencing service s_1 must hold. Additionally, there must exist an output history y' which fulfills the guarantee of s_2 and which is transformable to y_{s_2} by the dependency d.

Obviously, if the respective compound services are optional, the dependencies must only be considered if the services are selected. Regarding product family dependencies, the effects on the syntactical set-valued interface must be considered in addition to the effects on the behavior (see the following subsection).

Requires and Excludes Dependencies Although there are a lot of methodological significant dependencies, here, we focus on typical dependencies for product families, namely *requires* and *excludes*. These dependencies specify that certain services must or must not be selected together in a configuration. Thereby, to select a service means that the valid I/O history pairs must fulfill the requirement formalized by this service. In the following, we introduce precise semantics of these relations by describing the corresponding dependency predicates.

A *requires* dependency between two alternative or optional services (t *requires* s) means that if t is selected in a configuration, s must be selected, too. Intuitively, a valid history is required to fulfill the requirement specified by the service s whenever it fulfills the requirement specified by t. Formally, a history pair (x, y) of the compound service is valid if its projections to the interfaces of the services t and s satisfy the condition $A_t(x_t) \wedge G_t(x_t, y_t) \Rightarrow A_s(x_s) \wedge G_s(x_s, y_s)$. However, this condition is only sufficient if the sets of valid histories specified by alternative services are disjunct. Otherwise, (x, y) – more precisely, respective projections of (x, y) – could fulfill the specification of more than one variant, e.g. t and t'. Since t' is allowed to be selected without s, a pair (x, y) which fulfills t and t', is valid even if s is not fulfilled. Thus, the correct meaning of *requires* is that all history pairs which *exclusively* fulfill the service t must fulfill the service s. Valid history pairs fulfilling t and another variant of the same VP do not necessarily have to fulfill s. Formally, the definition of *t requires s* is given by:

$$\begin{aligned}&(\nexists t' \in V_T \setminus \{t\} : (x_{t'}, y_{t'}) \in \mathbb{H}(I_{t'}) \times \mathbb{H}(O_{t'}) \wedge A_{t'}(x_{t'}) \wedge G_{t'}(x_{t'}, y_{t'}))\\ &\Rightarrow (A_s(x_s) \wedge G_s(x_s, y_s))\,,\end{aligned} \tag{6}$$

where V_T denotes the set of all variants of the VP comprising t. This means, a valid history pair (x, y) that fulfills the A/G of no variant of V_T except for t must fulfill the A/G of s. Although, satisfying t is not explicitly required in this definition, it is implicitly given since any valid history pair must fulfill at least one of the variants according to Definition 3. Here, this variant can only be t.

The dependency *t excludes s* means that the services t and s are not allowed to be selected simultaneously, i.e. if the satisfaction of the service t is required (i.e. t is selected), the satisfaction of the service s must not be required. Since at least one of the alternatives of a VP must be fulfilled, this implies that the satisfaction of one of the other alternatives must be required. Formally, the definition of the dependency *t excludes s* is given by:

$$\begin{aligned}&(\nexists t' \in V_T \setminus \{t\} : (x_{t'}, y_{t'}) \in \mathbb{H}(I_{t'}) \times \mathbb{H}(O_{t'}) \wedge A_{t'}(x_{t'}) \wedge G_{t'}(x_{t'}, y_{t'})) \Rightarrow\\ &(\exists s' \in V_S \setminus \{s\} : (x_{s'}, y_{s'}) \in \mathbb{H}(I_{s'}) \times \mathbb{H}(O_{s'}) \wedge (A_{s'}(x_{s'}) \wedge G_{s'}(x_{s'}, y_{s'})))\,,\end{aligned} \tag{7}$$

where V_T and V_S denote the sets of variants of the respective VPs. A valid history pair (x, y) that fulfills the A/Gs of no variant of V_T except for t must fulfill the A/G of a variant $s' \neq s$ of V_S.

Additionally to the behavior, the effects on the syntactical interface must be considered. The set-valued interface of a compound service only comprises interfaces which result from the aggregation including s_1 and s_2 or none of them if there is a *requires* dependency between them. If there is an *excludes* dependency between these services, the set-valued interface of their common compound service does not comprise the interfaces which originated from combinations including s_1 and s_2.

In our example, there is a *requires* dependency between `Pre-Crash` and `Follow-Up` which reflects technical prerequisites: the pre-crash control uses the sensors of the follow-up control, which are available in a vehicle only if the latter control is built in. Thus, their common compound service `ACC` defines a set of valid history pairs which obligatory satisfy `Speed` and fulfill `Follow-Up` if they fulfill `Pre-Crash`. Also, there is an *excludes* dependency between `Steering Wheel` and the variant `Brake` of the pre-crash control. If both services would be present in a configuration, there might be a conflict, e.g. if the pre-crash control demands the vehicle to slow down and, simultaneously, the driver presses the acceleration button. Hence, it was a marketing decision (non-functional requirement) to resolve this conflict by means of an *excludes* dependency. Consequently, if `Steering Wheel` is selected, `Warning` must be selected, too.

3 Consistency

The basic idea of our approach is that the overall specification is the combination of modularly specified sub-functionalities. Thereby, different services might be defined over the same I/O ports. Thus, the integration of different functions might cause unforeseen conflicts (known as *feature interaction*) and consequently lead to an inconsistent specification of the overall behavior. As a consequence, it becomes a central task during the functional integration to detect and resolve conflicts in order to assure the consistency of the overall specification. In the following sections, we precisely define what we mean by *conflicts* and show how the introduced formal concepts can be used to detect and to resolve conflicts between functional requirements. Regarding product families, we show how the compatibility of different variants can be analyzed.

3.1 Consistency of a Single System

A specification of a single product is *consistent* if there is no conflict neither between different modular services nor between services and dependencies. To allow tool-supported conflict detection and consistency checks we firstly introduce formal definitions of conflicts. Subsequently, we show how these conflicts can be detected and resolved.

Conflict Definitions We differentiate two kinds of conflicts, namely input and output conflicts. There is an *input conflict* between aggregated services and/or dependencies if there is no history $h \in \mathbb{H}(I_{s_C})$ that fulfills the assumption of their common compound service s_C:

$$\{x \in \mathbb{H}(I_{s_C}) \mid A_{s_C}(x)\} = \emptyset. \tag{8}$$

An input conflict shows that the assumptions of the sub-services of s_C (and potential dependencies between them) are contradictory.

The follow-up control of our example (`Follow-Up`) is designed for city traffic and consequently only defined for target speeds between 40 and 80 km/h. The pre-crash control, however, is designed for motorway traffic, e.g. for target speed between 100 and 200 km/h. Then, the aggregations of these services results in an input conflict as there exists no input history which satisfies the assumptions of both services on their common input port `targetSpeed`.

Analogously, there is an *output conflict* between aggregated services and/or dependencies if the history set defined by the guarantee of their common compound service is empty for a valid input history:

$$\exists x \in \mathbb{H}(I_{s_C}) : A_{s_C}(x) \wedge \{y \in \mathbb{H}(O_{s_C}) \mid G_{s_C}(x, y)\} = \emptyset. \tag{9}$$

An output conflict indicates that the guarantees of the sub-services of s_C (and potential dependencies between them) are not satisfiable simultaneously for at least one valid input.

In our example, the services `MCC` and `ACC` are output-conflicting. There are input histories which cause contradictory output histories, e.g. an input history where the brake pedal is pressed in a time interval in which the current speed is lower than the target speed. In this case, the `MCC` demands the message `decelerate` on the output port `speed`, whereas the service `ACC` requires the message `accelerate` within the same time interval.

The conflicts captured by the introduced definitions can be further classified according to their causes. We differentiate *service-service conflicts*, *dependency-service conflicts*, and *dependency-dependency conflicts*. Thereby, the definitions of I/O conflicts remain the same but the common compound service s_C is obtained in different ways (cp. Definition 1, 4).

Note, there are no conflicts between a dependency function and the service influenced by it. Nevertheless, there might be conflicts between the influenced service and other services or dependency predicates. These conflicts are also covered by the definitions introduced above (cp. Definition 5).

Conflict Detection and Resolution Obviously, two services are independently combinable if their sets of I/O ports are disjunct. Thus, methodically, we propose to start with an analysis of the syntactical interface to define the set of candidates for conflicting services. These services must be analyzed for service-service conflicts as described above. Subsequently, we take dependency predicates into consideration and check all affected services for dependency-service and dependency-dependency conflicts.

In order to get a consistent specification all detected conflicts have to be resolved. Therefor, we propose two methodical procedures. A conflict can be resolved by changing the modular specification of at least one of the affected services or dependencies respectively. In many cases, conflicts can be resolved easily by introducing nondeterminism in the modular specification. This resolution method is applicable to all kinds of conflicts. Moreover, changing the modular specification is the only way to solve input conflicts.

The input conflict between `Follow-Up` and `Pre-Crash` is solved by changing the modular specifications of both services. The assumption of both services is enlarged to target speed between 40 and 200 km/h. However, it is not further specified how the system reacts to the additional input histories, i.e. every output message is valid – both services are nondeterministic. This nondeterminism is resolved in the compound service `ACC` according to Definition 1.

For most of the output conflicts this procedure is not adequate since changing the modular specification accordingly to the behavior of another service implies a loss of modularity. Therefore, to resolve the source of output conflicts (namely, the service interaction) we propose to introduce additional dependency functions. A new dependency modifies the output histories in such a way that both interacting services always send the same message on the common ports. By this, we preserve the modularity of services and, furthermore, make functional dependencies explicit.

In our example, the output conflict between `MCC` and `ACC` is resolved by introducing the dependency function as described in Section 2.4. This dependency specifies that the service `MCC` overrules the service `ACC`, i.e. if conflicting the output of the service `ACC` is substituted by the output of the service `MCC`.

3.2 Consistency of a System Family

Next, we aim at ensuring the consistency of a product family. The specification of a product family is consistent if there is at least one consistent configuration of this family. In the following, we explain the meaning of conflicts between a service and a VP and sketch the methodology to analyze product families for conflicts.

Conflict Definitions Obviously, I/O conflicts between services and a certain variant of a VP are covered by the same definitions as conflicts between services of a single product. Based on these definitions, there is no conflict between a service s and a VP comprising a set of variants V if no conflict is detected between s and any variant $v \in V$. Particularly, the service s and the VP are *independently combinable* if s and each variant $v \in V$ are independently combinable.

Conflict Detection and Resolution In order to reduce the effort of the conflict detection we firstly analyze the syntactical interfaces. If s has no common port with the maximum interface of a VP[3], i.e. s has no common port with any variant of the VP, there is no conflict between s and any variant – no further analysis is necessary. Otherwise, a syntactical analysis of the single variants yields the variants which must be further analyzed (analogously to single services). To resolve conflicts we apply the already introduced procedures. Furthermore, we can eliminate conflicts by introducing *excludes* or *requires* dependencies.

To exemplify the procedure, we analyze the service `Follow-Up` and the VP `Pre-Crash` for output conflicts. An output conflict can not be excluded based on the syntactical analysis of the maximum interface of the VP. But the syntactical analysis of the single interfaces yields that `Follow-Up` and `Warning` have no common output port – they are independently combinable. `Brake` and `Follow-Up` use the common output port `speed` and a further analysis of their behaviors shows an output conflict between them. To resolve the conflict we introduce a dependency function which states that the service `Brake` has a higher priority. Note, that introducing an excludes dependency (`Brake` *excludes* `Follow-Up`) would also solve this output conflict, but would provoke a new dependency-dependency conflict because of the dependency `Pre-Crash` *requires* `Follow-Up`.

3.3 Tool Support

Thanks to the formal definitions of services, dependencies as well as conflicts, we can use a theorem prover (e.g. Isabelle [6]) to assure the consistency of a service specification. Thereby, all services (atomar as well as compound) are transformed into Isabelle functions. Then, for each compound service we have to prove two lemmata that claim that the sets of defined valid histories are not empty (negation of Definitions 8 and 9). However, the transformation to Isabelle is not scope of this paper – it is precisely addressed in [7].

[3] The maximum interface is the conjunction of the sets of all I/O ports of all variants.

4 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 have introduced a formal model with a well-defined semantics for specifying functionality. This has several advantages. Firstly, a formal model which formalizes (functional) requirements allows an analysis of the system already in the early phases of the development process. By this, discrepancies between conflicting requirements 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. The Service Diagram provides formal specification of the functional requirements which can be used for a (tool-supported) verification of the subsequent design models.

Functional Variability Furthermore, we have enlarged our approach to model whole families of related systems instead of single systems only. While traditional approaches mainly focus on structural aspects, we concentrated on the behavior and have precisely defined the behavioral meaning of variability. We especially focused on the consistency of the specification of a product family. By formally reasoning about the behavior conflicts between variants can be detected and resolved by introducing *excludes* and *requires* relations. By this, dependencies between variants which have not been realized during earlier engineering stages can be derived and made explicit.

5 Related Work

Formal Semantics The definition of a formal semantics for feature models – the main method to formalize variability in product families – is not new. In [8], Batory and O'Malley use grammars to specify feature models. The formalization of feature models with propositional formulas goes back to the work by Mannion [9], in which logical expressions can be developed using propositional connectives to model dependencies between requirements. Further formal semantics are compared in [10]. Another approach to specifying multi-functional systems is introduced by van Lamsweerde et al in [11]. The main deficit of all these approaches is a disregard for the behavior of single features. Moreover, approaches like FODA only provide a two-valued notion of 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" [12].

In [13], 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 approach, however, focuses on formalizing user requirements and their analysis in the early phases of the development process.

Our work is founded on 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.

Feature Interaction Using the formal foundation, a central task of our approach is to detect and resolve conflicts between single requirements (feature interaction) in order to assure the consistency of the overall specification. A large body of research [14] on feature interaction was caused by the the huge amount of software-based functions in telecommunication. The telecommunication-specific approaches to modeling feature interaction, like those by Jackson and Zave [15] or Braithwaite and Atlee [16], consider only telecommunication-specific features (functionality *additional* to the core body of software) and show how they can be combined in telecommunication systems. Thus, they are not directly applicable to other kinds of systems and for this reason can be barely compared to our work.

If we consider "feature" as a synonym of "function", we find further related work, e.g. approaches by Stepien and Logrippo [17] or Klein et al. [18]. All these approaches are comparable in the sense that they aim at explicit specification of feature behavior and at identifying feature interaction on the basis of behavior models. In our terminology, they look for interactions between services of a single product. However, they do not provide any means of variability.

To summarize, to the best of our knowledge, there is no approach to specify a product family, by formally describing the behavioral variability in requirements, and to detect conflicts between variants based on their behavioral specifications.

6 Conclusion and Future Work

In this paper, we have introduced and formally founded the underlying concepts of our service specification, which focuses on the modeling and structuring of functional requirements. Thereby, the concept of a service is used to model functional requirements in a modular fashion. In this paper, we especially concentrated on concepts to explicitly modeling inter-service dependencies. We have integrated the concept of behavioral variability which makes the Service Diagram suitable to formally capture functional requirements of a system family.

The formal specification of the functional requirements, their dependencies, and the behavioral variability already at an early stage of the development process allows to perform a formal (and therefore tool-supported) analysis of the functional requirements for conflicts. Since ensuring the consistency of the specification is one of the main goals of our approach, we have precisely defined the meaning of conflicts in the Service Diagram. Furthermore, we have described the detection and resolution of conflicts from a methodological point of view.

Regarding product families, we have shown how the compatibility of different variants can be analyzed.

Since the effort to perform consistency checks separately for all possible combinations of variants grows exponentially, we are currently working on concepts to reduce the effort of consistency checks by extracting commonalities between variants. Beyond this, our future work includes the development of a user-friendly syntax for the semantics introduced in this paper and the transition from the Service Diagram to the consecutive design models.

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. Harhurin, A., Hartmann, J.: A Formal Approach to Specifying the Functionality of Software System Families. Technical report, Technische Universität München (2007) http://www.in.tum.de/forschung/pub/reports/2007/TUM-I0720.pdf.gz.
5. Broy, M.: Service-oriented systems engineering: Modeling services and layered architectures. In: FORTE. (2003) 48–61
6. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
7. Spichkova, M.: Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle. PhD thesis, Technische Universität München (2007)
8. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Trans. Softw. Eng. Methodol. **1** (1992)
9. Mannion, M.: Using first-order logic for product line model validation. In: SPLC. (2002) 176–187
10. Trigaux, J.C., Heymans, P., Schobbens, P.Y., Classen, A.: Comparative semantics of feature diagrams: Ffd vs. vdfd. CERE **0** (2006) 36–47
11. van Lamsweerde, A., Letier, E., Darimont, R.: Managing conflicts in goal-driven requirements engineering. IEEE Trans. Softw. Eng. **24** (1998) 908–926
12. Schätz, B.: Combining product lines and model-based development. In: Proceedings of Formal Aspects of Component Systems (FACS 2006). (2006)
13. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE. (2005) 422–437
14. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. Comput. Networks **41** (2003) 115–141
15. Jackson, M., Zave, P.: Distributed feature composition: A virtual architecture for telecommunications services. IEEE Trans. Softw. Eng. **24** (1998) 831–847
16. Braithwaite, K.H., Atlee, J.M.: Towards automated detection of feature interactions. In: FIW. (1994) 36–59
17. Stepien, B., Logrippo, L.: Representing and verifying intentions in telephony features using abstract data types. In: FIW. (1995) 141–155
18. Klein, C., Prehofer, C., Rumpe, B.: Feature specification and refinement with state transition diagrams. In: Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems. (1997)