Multifunctional software systems: Structured modeling and specification of functional requirements

Manfred Broy *

Institut für Informatik, Technische Universität München, D-80290, München, Germany

A R T I C L E   I N F O

Article history:
Received 8 March 2009
Received in revised form 1 June 2010
Accepted 3 June 2010
Available online 19 June 2010

Keywords:
Functional specification
Multifunctional systems
Functional requirements
Service hierarchies
Interfaces

A B S T R A C T

This paper deals with the structured specification of interface behavior of multifunctional systems, which are systems that offer a variety of functions for different purposes and use cases. It introduces a theory and first concepts of a methodology for the identification, structured modeling, and formalization of functional requirements of multifunctional systems. Service hierarchies specify multifunctional systems in terms of their provided sub-functions called services together with their mutual relationships and dependencies. A service hierarchy describes the functionality of multifunctional systems in a structured way. Each service is specified independently and the specification is added to the service hierarchy. Modes help to specify the feature interactions and by that functional dependencies between the services. The approach is based on the Focus theory for modeling interface behavior and services.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

Traditionally, technical systems offer more or less only one function. The first telephones were just for issuing and receiving calls. The first photo cameras were just for taking pictures. Early radios were just for listening to radio stations. The first cars were just for driving. With the advent of embedded software more and more technical systems offer not just one function or a small number of related functions but a rich variety of functions for different purposes integrated into one device. Also software systems, such as for instance web browsers, operating systems, or business applications, typically offer many functions side-by-side integrated into one application. We speak of multifunctional systems.

In practice today, functional requirements and, in particular, specifications of multifunctional systems are generally not sufficiently precisely formulated (see [7,31]). One reason is that the documentation of requirements is not well supported by modeling techniques and formalisms available in practice, since useful structuring concepts are missing. In fact, the modeling approaches for requirements that have been proposed so far seem insufficient. For instance, UML offers little for documenting, modeling, and structuring functional requirements (and nothing for non-functional requirements) apart from its quite informal use case diagrams and its interaction and activity diagrams, which do not provide any help for hierarchical structuring (some structuring is achieved by high level message sequence charts—HMSCs, see [20]).

Also from a methodological point of view, there are severe deficiencies in the state of the art of systems engineering when dealing with functional requirements of multifunctional systems. One obvious problem, when dealing with the system functionality of multifunctional systems, is the absence of an appropriate, unambiguous and consistent terminology. There are many related (more or less synonymous) notions that address concepts and terms referring to the functionality of the systems such as “function”, “feature”, “service”, “scenario”, or “use case”. Each of these terms is used in a variety with quite
different meanings by different groups of researchers and engineers. In particular, the term “feature” is used in a broad spectrum of meanings. For instance, in the context of product line engineering the term feature is used in [1], which outlines “Our formal work on features is guided by the following informal definition: a feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder’s requirement, to implement and encapsulate a design decision, and to offer a configuration option”. We rather follow the Institute of Electrical and Electronics Engineers, who define the term feature in IEEE 829 as “A distinguishing characteristic of a software item (e.g., performance, portability, or functionality)” and speak in the following of “user functions” or “services” to refer to “functional features” (see also [12,25,26,28,24,27]). As pointed out in [8], in the telecommunications domain, the notion of feature is carefully studied and well established (see also [29]), however, the literature and even telecommunications standards blur the distinction between services and features. Functional features can be defined as “reusable, self-contained services” [23]; they encapsulate individual pieces of functionality of limited scope, typically used to structure the interfaces or internals of components.

Our informal definition of a service (or of a user function) is that it represents a sub-function of a multifunctional system. A service corresponds roughly to what is described in a use case (see [10,17]) and captured by scenarios modeled by a set of interaction patterns that describe how to use a system for specific purposes. Actually, use cases describe classes of using a system. This is typically done scenario-driven, giving characteristic examples of use. A systematic formal specification of multifunctionality is not addressed by use cases, in general.

In this paper, we aim at a theory supporting the structuring of the functionality of multifunctional systems, with the emphasis on the specification phase of requirements engineering. A service hierarchy provides a structured specification of the functional requirements of multifunctional systems in terms of their sub-services. To be able to support such a structuring, we suggest specifying these sub-functions independently in isolation and then combining them into an integrated system specification by making dependencies between services explicit. This approach leads to concepts of structuring the functionality of systems in terms of relations between their sub-functions.

We base our approach on the Focus theory that provides a modular approach to the logical description of distributed interactive systems (see [3]) by their interface behaviors given by the relation between their input and output streams. Systems are composed of interacting components working concurrently and exchanging messages via communication lines with a communication traffic modeled by data streams. Focus provides a modular technique for the specification of systems in terms of their interface behavior and for structuring systems that are composed of components. We introduce a formal model of comprehensive system functionality structured in terms of services along the lines of [4,8] where a theoretical basis for the concept of a service is introduced to form service-oriented architectures (see also [6]). This concept is taken as a basis for specifying functional requirements.

The overall goal of this paper is to work out semantic models of system functionalities of multifunctional systems in terms of services and their relationships. The goal of this theory is to provide a first basis for an engineering method for the design and specification of system functionalities.

The remainder of the paper is organized as follows. First, we give a brief introduction into mathematical models of system and service interface behavior and some basic techniques to specify systems and services by state machines using state transition tables. We briefly illustrate these techniques by simple examples. First we follow an analytical path and define concepts of how to decompose multifunctional systems into their sub-services by projection and how to relate these sub-services in terms of their dependencies and feature interaction using modes. Finally, we follow a constructive approach and show how to structure the specification of multifunctional systems by function hierarchies. We work out how these ideas relate to the composition of systems forming component architectures (see [22]). We use this composition to compose multifunctional systems from models of their sub-services.

2. Systems and services

In this section, we briefly introduce the syntactic and semantic notion of a system, its interface and that of a service. This theoretical framework is in line with [8].

2.1. Modeling systems and services

First, we briefly repeat the concepts on which we will base the theory for modeling multifunctional systems. We are dealing with models of discrete systems. We closely follow the Focus approach as described in [3]. A discrete system is a technical or organizational unit with a clearly specified boundary. It interacts with its environment over its boundary by exchanging messages in discrete events—in the case of Focus via channels through which it exchanges messages.

Systems have syntactic interfaces that are described by their sets of input and output channels attributed by the type of messages that are communicated over them. Channels are used for communication by transmitting messages and connecting systems. A discrete system has an interface behavior, which is modeled by a function mapping the streams of messages received on the system’s input channels onto streams of messages sent on its output channels. We call this the black box behavior or the interface behavior of discrete systems.

A (data) type T is a name for a data set, a channel is a name for a communication line, and a stream is a finite or infinite sequence of data messages. Let TYPE be the set of all data types. A set of typed channels is a set of channels with a type given for each of its channels.
Definition (Syntactic Interface). Let \( I \) be a set of typed input channels and \( O \) be the set of typed output channels. The pair \((I, O)\) characterizes the syntactic interface of a system. The syntactic interface is denoted by \((I \rhd O)\).

Fig. 1 shows the system \( F \) in a graphical representation by a data flow node with its syntactic interface consisting of the input channels \( x_1, \ldots, x_n \) of types \( S_1, \ldots, S_n \) and the output channels \( y_1, \ldots, y_m \) of types \( T_1, \ldots, T_m \).

In Focus, a system encapsulates a state and is connected to its environment exclusively by its interface given by its typed input and output channels.

Definition ((Non-timed) Streams). Given a set \( M \), by \( M^* \) we denote the set of finite sequences of elements of \( M \), by \( M^\infty \) the set of infinite sequences of elements of the set \( M \); infinite sequences are formally represented by functions \( s : \text{IN} \setminus \{0\} \rightarrow M \).

By \( M^\omega \) we denote the set \( M^* \cup M^\infty \), called the set of finite and infinite (non-timed) streams.

In the following we work with streams that include discrete timing information. Such streams represent histories of communications of data messages transmitted within a time frame. To keep the time model simple we choose a concept of discrete time where time is represented by an infinite sequence of finite time intervals of equal length.

Definition (Timed Streams). Given a message set \( M \) of data elements of type \( T \), we represent a timed stream \( s \) of type \( T \) by a function \( s : \text{IN} \setminus \{0\} \rightarrow M^* \).

In a timed stream \( s \) a sequence \( s(t) \) of messages is given for each time interval \( t \in \text{IN} \setminus \{0\} \). In each time interval an arbitrary, but finite number of messages may be communicated. By \((M^*)^\infty \) we denote the set of timed streams.

In all the examples of systems we consider in the following, we deal, for simplicity, only with streams carrying as elements sequences of messages that contain at most one element. Throughout this paper we work with a few simple basic operators and notations for streams, which are briefly summarized as follows:

- \( \langle \ \rangle \) empty sequence or empty finite stream,
- \( \langle m \rangle \) one-element sequence containing \( m \) as its only element,
- \( a^\hat{b} \) concatenation of the sequences or streams \( a \) and \( b \)
- \( s(t) \) \( t \)-th element of the stream \( s \) (which is a sequence in the case of timed streams),
- \( s \downarrow t \) prefix of length \( t \in \text{IN} \) of the stream \( s \) (which is a sequence of length \( t \) carrying finite sequences as its elements in the case of a timed stream).

A (timed) channel history for a set of typed channels \( C \) (which is a set of typed identifiers) assigns to each channel \( c \in C \) a timed stream of messages communicated over that channel.

Definition (Channel History). Let \( C \) be a set of typed channels; a (total) channel history \( x \) is a mapping (let \( IM \) be the universe of all messages)

\[
x : C \rightarrow (\text{IN} \setminus \{0\} \rightarrow IM^*)
\]

such that \( x(c) \) is a timed stream of messages of the type of channel \( c \in C \). \( \tilde{C} \) denotes the set of all total channel histories for the channel set \( C \).

For each history \( z \in \tilde{C} \) and each time \( t \in \text{IN} \) the expression \( z \downarrow t \) denotes the partial history (the initial communication behavior on the channels) of \( z \) until time \( t \). \( z \downarrow t \) yields a finite history for each of the channels in \( C \) represented by a mapping

\[
C \rightarrow (\{1, \ldots, t\} \rightarrow IM^*)
\]

\( z \downarrow 0 \) denotes the history with empty sequences associated with each of its channels.

The behavior of a system with a syntactic interface \((I \rhd O)\) is defined by a function that maps the input histories in \( I \) onto output histories in \( O \). This way we get a functional model of a system interface behavior.

Definition (Causal Behavior Function). For a function

\[
F : I \rightarrow \varphi(\tilde{O})
\]
we define the set

\[ \text{dom}(F) = \{ x : F(x) \neq \emptyset \} \]

called the service domain of \( F \). \( F \) is called total, if \( \text{dom}(F) = \mathbb{T} \), otherwise \( F \) is called partial.

The function \( F \) is called causal, if (for all \( t \in \mathbb{N} \) and all input histories \( x, z \in \mathbb{I} \)):

\[ x, z \in \text{dom}(F) \land x \downarrow t = z \downarrow t \Rightarrow \{ y \downarrow t : y \in F(x) \} = \{ y \downarrow t : y \in F(z) \} \]

\( F \) is called strongly causal, if (for all \( t \in \mathbb{N} \) and all input histories \( x, z \in \mathbb{T} \)):

\[ x, z \in \text{dom}(F) \land x \downarrow t = z \downarrow t \Rightarrow \{ y \downarrow t + 1 : y \in F(x) \} = \{ y \downarrow t + 1 : y \in F(z) \}. \]

Causality (for an extended discussion see [3]) indicates a consistent time flow between input and output histories in the following sense: in a causal function input messages received at time \( t \) only influence future output not before time \( t \); this output is given by messages communicated via output channels at times \( \geq t \) (in the case of strong causality at times \( > t \), which indicates that there is a delay of at least one time step before input has any effect on output).

**Definition (I/O-Behavior).** A causal function \( F : \mathbb{T} \rightarrow \wp(\mathbb{O}) \) is called an I/O-behavior. By \( IF[I \uparrow \mathbb{O}] \) we denote the set of all (total and partial) I/O-behaviors with a syntactic interface \((I \uparrow \mathbb{O})\) and by \( IF \) the set of all I/O-behaviors. \( \Box \)

We use I/O-behaviors to model both services and system interface behaviors. For systems we assume that the associated I/O-behavior is total. Behaviors \( F \) may be deterministic (in this case, the set \( F(x) \) of output histories has at most one element for each input history \( x \)) or nondeterministic.

### 2.2. Systems and their services

Systems and also services interact with their environment via the channels of their interface. We identify both systems and services by names. A system or a service (Fig. 2) named \( k \) has an interface, consisting of a syntactic interface \((I \uparrow \mathbb{O})\) and an interface behavior

\[ F_k : \mathbb{T} \rightarrow \wp(\mathbb{O}) \]

The behavior may be a combination of a larger number of more elementary sub-service behaviors. Then we speak of a multifunctional system.

Let \( \text{SID} \) be the set of service names. A service named \( k \in \text{SID} \) is called statically interpreted if for \( k \) a syntactic interface \((k \uparrow \mathbb{O}_k)\) is given only and dynamically interpreted if a behavior \( F_k \in IF[k \uparrow \mathbb{O}_k] \) is specified for \( k \), which is, in general, partial (for an extended introduction to services, see [8]).

From a methodological point of view, the concept of a service offered by a system is closely related to the idea of a use case (see [17]) as found in object-oriented analysis for illustrating one instance of using the system for a particular purpose (for example, using a mobile phone for taking a digital photograph). The instance describes an interaction scenario of a service (for instance the service of taking photos).

### 2.3. Describing services

Service behavior is modeled by interface behavior. To specify a service behavior we first choose a unique name \( k \in \text{SID} \) for it. In a practical application we may give an informal description of the service. This can be done by a use case description. In a use case description we illustrate in which way and which situation the service is triggered and how it reacts. Exceptional cases are identified and scenarios specify the reactions of the service. Message sequence charts model scenarios of interaction (see [5,15,19]) and dialogue between the service and its user and enhance use case descriptions.

Already in a use case description, events of sending and receiving messages are described that are part of the service dialogue. In our model the messages involved are specified by the syntactic interface. More precisely, the input and output channels are specified, and which types of messages are communicated over the channels and their meaning and purpose is explained. Since channels are distinguished into input and output channels (see [3]), the messages communicated over the channels are also classified as input and output messages.

Finally, the behavior of the services is specified. To do that there are three options: \( I/O \)-assertions (see [3]), state machines with input and output, or sets of message sequence charts (see [16]). We do not work, in the following, with specifying assertions but with tables that describe state transitions of state machines with input and output, since later we will compose and transform system behaviors in terms of operations on these tables. State machines with input and output describe systems as well as services in terms of states and state transitions.
For each state $\sigma \in \Sigma$ and each valuation $a : I \rightarrow M^*$ of the input channels in $I$ by sequences of input messages, every pair $(\sigma', \beta) \in \Delta(\sigma, a)$ defines a successor state $\sigma'$ and a valuation $\beta : O \rightarrow M^*$ of the output channels consisting of the sequences produced by the state transition. $(\Delta, \Lambda)$ is a Mealy machine with possibly infinite state space. If in every transition the output $\beta$ depends on the state $\sigma$ only but never on the current input $a$, we speak of a Moore machine.

State machines, as introduced, are nondeterministic, in general. Nondeterminism is essential, as we will see later, when studying projections for systems by hiding certain input actions, which may turn a deterministic machine into a nondeterministic one.

**Definition** (Interface Behavior of State Machines). Given a state machine $(\Delta, \Lambda)$ with syntactic interface ($I \triangleright O$) we define its interface behavior

$$F_{(\Delta, \Lambda)} \in IF[I \triangleright O]$$

as follows (let $\Sigma$ be the state space for $(\Delta, \Lambda), x \in \tilde{I}$)

$$F_{(\Delta, \Lambda)}(x) = \{ y : \exists \sigma : IN \rightarrow \Sigma : \sigma(0) \in \Lambda \land \forall t \in \tilde{I} : (\sigma(t+1), y(t+1)) \in \Delta(\sigma(t), x(t+1)) \}.$$ 

Note that in computations we start to count states by 0 such that the initial state is $\sigma(0)$, while we count the time intervals starting with 1 such that $x(1)$ and $y(1)$ are the sequences of messages exchanged in interval 1. □

By construction $F_{(\Delta, \Lambda)}$ is causal. If $(\Delta, \Lambda)$ is a Moore machine, then $F_{(\Delta, \Lambda)}$ is strongly causal.

State machines are described by state transition diagrams or by state transition tables. We prefer tables since we work with projections on system behaviors in the following, which in simple cases correspond to the elimination of columns in transition tables.

We start with a simple example of a service and its specification. We specify the service of a queue. It is a simple basic example out of a rich class of services for storing and retrieving data. To keep the examples simple, we consider only input histories with at most one (relevant) message per channel in each time interval. For sequences $s$ of messages of length $> 1$, only their last elements last($s$) are considered relevant.

The state space $\Sigma$ of a state machine is fixed by typed state attributes as in object orientation. Given a set $V$ of typed state attributes, for every state $\sigma \in \Sigma$ and every state attribute $\nu \in V$ of type $T$, $\sigma(\nu)$ denotes the value of the attribute $\nu$ in the state $\sigma$.

To specify systems, we work with specification templates. A service specification consists of a graphical description of a data flow node that specifies the name of the service, its state attributes including their initial values, and the input and output channels with their types. In addition, a table describes the state transitions.

**Example** (Queue Service). A Queue service allows us to store elements of type Data and to request them in a first in, first out (FIFO) fashion. A typical application of a device offering such a service might be a PDA that offers the option to store a queue of tasks or dates. We specify the data types involved as follows (req is the signal for an output request):

```plaintext
type QIn = {req} \cup Data

type QOut = Data.
```

Based on these data types we formulate the specification of the service Queue in Table 1.

The specification consists of a state transition table and a data flow node. The data flow node describes the input and output channels, their types, as well as the state attributes and their initial values. The entry “–” in the table indicates empty output (or input) for that channel. The rows in the table describe the state changes. $q'$ denotes the value of the state attribute $q$ after the state transition. For instance, the first row in Table 1 represents the state transition rule described by the following formula:

$$(\sigma', \beta) \in \Delta(\sigma, a) \iff \sigma(q) = s \land \text{last}(a(a)) = d \land \beta(b) = \langle \rangle \land \sigma'(q) = s'(d).$$

There may be different state machines that fulfill the state transition rules. With a table we associate the inclusion-least state transition function that fulfills the rules of the table.

---

**Table 1**

The system queue as a state transition table

<table>
<thead>
<tr>
<th>q</th>
<th>a</th>
<th>q'</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>d</td>
<td>s'(d)</td>
<td>-</td>
</tr>
<tr>
<td>{d}</td>
<td>s</td>
<td>req</td>
<td>s'</td>
</tr>
</tbody>
</table>

**System Queue**

Attributes

- q: Data
  - Initial: ⟨⟩

**a**: QIn

**b**: QOut
The given table, in fact, is an example of a specification of a partial behavior. If the input stream a has for instance the shape (let \(d_1, d_2 \in \text{Data}\))

\[ a = \langle \langle \text{req} \rangle \rangle \langle \langle d_1 \rangle \rangle \langle \langle \text{req} \rangle \rangle \langle \langle \text{req} \rangle \rangle \langle \langle d_2 \rangle \rangle \langle \langle \text{req} \rangle \rangle \ldots \]

then the transition rules for input stream a do not apply, since in the initial state \(q\) is empty and thus the set of output histories is empty. □

For details how a table describes a state machine and how a state machine description relates to a behavior see [3].

3. Structuring and relating services

In the following we study relationships as well as dependencies between services that are sub-services of multifunctional systems.

3.1. Relations between services

In this section we introduce the fundamental relation between services called sub-service relation.

3.1.1. Sub-typing of histories and services

To define the sub-service relation, we introduce some auxiliary notions. A fundamental notion is the sub-type relationship between syntactic interfaces. To define this relationship we introduce a sub-type relation on channel sets and on syntactic interfaces.

An input action of a system \(F\) with syntactic interface \((I \rightharpoonup O)\) is a pair \((m, c)\) where \(c \in I\) is an input channel and \(m\) is a message of the type of the channel \(c\). An output action of a system \(F\) with syntactic interface \((I \rightharpoonup O)\) is a pair \((m, c)\) where \(c \in O\) is an output channel and \(m\) is a message of the type of the channel \(c\). By \(\text{Act}(C)\) we denote the actions of a typed channel set \(C\).

A typed channel set \(C_1\) is called a sub-type of a typed channel set \(C_2\) if the following formula holds (we assume that all types stand for non-empty sets of elements):

\[ \text{Act}(C_1) \subseteq \text{Act}(C_2). \]

We write then

\[ C_1 \text{ subtype } C_2. \]

Thus, a sub-type \(C_1\) of a set \(C_2\) of typed channels carries only a subset of the channel identifiers from \(C_2\) and for each of the channels in \(C_1\) only a subset of the messages it carries in \(C_2\). The idea of sub-types is mainly used for relating services.

Sub-typing, as introduced, is extended schematically from channel sets to interfaces.

Definition (Sub-types Between Interfaces). If for syntactic interfaces \((I_1 \rightharpoonup O_1)\) and \((I_2 \rightharpoonup O_2)\) both \(I_1\) subtype \(I_2\) and \(O_1\) subtype \(O_2\) hold, we call the syntactic interface \((I_1 \rightharpoonup O_1)\) a sub-type of the interface \((I_2 \rightharpoonup O_2)\) and write:

\[(I_1 \rightharpoonup O_1) \text{ subtype } (I_2 \rightharpoonup O_2).\]

This means that \((I_1 \rightharpoonup O_1)\) includes only a subset of the input and output actions of \((I_2 \rightharpoonup O_2)\). □

This definition is chosen to study sub-behaviors working on subsets of input and output actions and is in contrast to sub-typing as defined in object-oriented languages. There a functional sub-type requires that its domain type may be increased while its range type may be decreased. Then the usage of functions with sub-types instead of original types does not lead to type errors.

3.1.2. Projections of histories and services

Based on the sub-type relation between sets of typed channels we define the concept of a projection of a history. It is the basis for specifying the sub-service relation.

Definition (History Projection). Let \(C\) and \(G\) be sets of typed channels with \(C\) subtype \(G\). We define for history \(x \in G\) its projection \(x|C\) \(\in C\) to the channels in the set \(C\) and to the messages of their types. For channel \(c \in C\) with type \(T\) we specify the projection by the equation

\[ (x|C)(c) = T \circ x(c) \]

where for a stream \(s\) and a set \(M\) we denote by \(M\circ s\) the stream derived from \(s\) by deleting all messages in \(s\) that are not in set \(M\). \(x|C\) is called projection of history \(x\) to channel set \(C\). □

To obtain the sub-history \(x|C\) of \(x\) by projection, we keep only those channels and types of messages in the history \(x\) that belong to the channels and their types in \(C\).
**Definition (Projection of Behaviors).** Given syntactic interfaces \( (l' \triangleright O') \) and \((l \triangleright O)\) where \((l' \triangleright O')\) subtype \((l \triangleright O)\) holds, we define for a behavior function \( F \in IF[l \triangleright O] \) its projection \( F\downarrow (l' \triangleright O') \in IF[l' \triangleright O'] \) to the syntactic interface \((l' \triangleright O')\) by the following equation (for all input histories \( x' \in I' \)):

\[
F\downarrow (l' \triangleright O')(x') = \{ y|O' : \exists x \in I : x' = x|l' \land y \in F(x) \}. \quad \square
\]

In a projection, we delete all input and output messages that are not part of the syntactic interface \((l' \triangleright O')\) and concentrate on the subset of the input and output messages of a system in its syntactic sub-interface \((l' \triangleright O')\). The idea is to derive less complex sub-behaviors that, nevertheless, allow us to conclude properties about the original system.

Projection may on one hand introduce and on the other hand reduce nondeterminism. If input messages are deleted that influence output messages that are not deleted, a projection \( F\downarrow (l' \triangleright O') \) may introduce non-determinism since this way we abstract away in projection \( x|l' \) some input messages of \( x \) that are needed to determine the output in \( F(x)|O' \). As a result, for some input histories \( x \in \text{dom}(F) \) the set

\[
F(x)|O'
\]

may contain fewer output histories than the set

\[
(F\downarrow (l' \triangleright O'))(x|l').
\]

This is the case if there are input histories \( x, z \in I \) with \( x \neq z \) which are identified under the projection such that \( x|l' = z|l' \) but

\[
F(x)|O' \neq F(z)|O'.
\]

We obtain with \( x' = x|l' = z|l' \) by the definition of projection

\[
F(x)|O' \cup F(z)|O' \subseteq F\downarrow (l' \triangleright O')(x').
\]

In fact, there are deterministic behaviors \( F \) with projections \( F\downarrow (l' \triangleright O') \) that are nondeterministic.

On the other hand, by abstracting away certain output messages, we may reduce nondeterminism. More precisely, if for some input history \( x \in \text{dom}(F) \), there are different output histories \( y, y' \in F(x) \) with \( y \neq y' \) for which \( y|O' = y'|O' \) holds, then \( F \) is nondeterministic since it may deliver either \( y \) or \( y' \) as output for \( x \) while \( F\downarrow (l' \triangleright O')(x') \) with \( x' = x|l' \) these two different outputs are identified to \( y|O' = y'|O' \).

In fact, there are nondeterministic behaviors \( F \) with projections \( F\downarrow (l' \triangleright O') \) that are deterministic.

The following definition characterizes projections that do not introduce additional nondeterminism, since the input deleted by the projection does not influence the output.

**Definition (Faithful Projection of Behaviors).** Let all definitions be as in the definition above; a projection \( F\downarrow (l' \triangleright O') \) is called faithful, if for all input histories \( x \in \text{dom}(F) \) the following formula holds:

\[
F(x)|O' = (F\downarrow (l' \triangleright O'))(x|l'). \quad \square
\]

In a faithful projection, the sets of histories produced as outputs on the channels in \( O' \) depend only on the messages of the input channels in \( l' \) and not on other inputs for \( F \) outside of \( l' \). A faithful projection is a projection of a behavior to a sub-function that forms an independent sub-behavior, where all input messages in \( I \) are included that are relevant for the considered output messages.

In general, of course, projections are not faithful. However, in any case the formula

\[
F(x)|O' \subseteq (F\downarrow (l' \triangleright O'))(x|l')
\]

is valid: If \( y' \in F(x)|O' \) then by definition there exists \( y \in F(x) \) with \( y' = y|O' \); then

\[
y' \in \{ y|O' : \exists x \in I : x' = x|l' \land y \in F(x) \}
\]

and thus \( y' \in (F\downarrow (l' \triangleright O'))(x|l') \).

A first simple example illustrating a system with at least two nontrivial projections is given in the following.

**Example (A System and its Projections).** We consider a system called SwStore with two input channels \( cx \) and \( cz \) and two output channels \( cy \) and \( cr \). The system allows storing and reading natural numbers and updating them. However, the storing and updating service can be switched off and on. As long as it is in the switched off mode, input to the store is ignored. The service communicates messages that are listed in the following table (here set\((n)\) is the message to set the state to the number \( n \in \mathbb{N} \):)

<table>
<thead>
<tr>
<th>Channel</th>
<th>Types of the channels</th>
<th>Messages sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>( cx )</td>
<td>Switch</td>
<td>{switch}</td>
</tr>
<tr>
<td>( cz )</td>
<td>AccData</td>
<td>{read} \cup {set(n): n \in \mathbb{N} }</td>
</tr>
<tr>
<td>( cy )</td>
<td>OnOff</td>
<td>{on, off}</td>
</tr>
<tr>
<td>( cr )</td>
<td>Ack</td>
<td>{done} \cup \mathbb{N}</td>
</tr>
</tbody>
</table>
Table 2
System SwStore as a State Transition Table ($j \in \text{IN}$).

<table>
<thead>
<tr>
<th>$m$</th>
<th>$v$</th>
<th>$c_x$</th>
<th>$v'$</th>
<th>$c_y$</th>
<th>$c_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>$j$</td>
<td>-</td>
<td>off</td>
<td>$j$</td>
<td>-</td>
</tr>
<tr>
<td>off</td>
<td>$j$</td>
<td>switch</td>
<td>-</td>
<td>on</td>
<td>$j$ on</td>
</tr>
<tr>
<td>off</td>
<td>$j$</td>
<td>switch</td>
<td>read</td>
<td>on</td>
<td>$j$ $j$</td>
</tr>
<tr>
<td>on</td>
<td>$j$</td>
<td>-</td>
<td>set($n$)</td>
<td>on</td>
<td>$n$ on done</td>
</tr>
<tr>
<td>on</td>
<td>$j$</td>
<td>switch</td>
<td>?</td>
<td>off</td>
<td>$j$ off</td>
</tr>
<tr>
<td>on</td>
<td>$j$</td>
<td>-</td>
<td>-</td>
<td>on</td>
<td>$j$ -</td>
</tr>
</tbody>
</table>

$\text{cx}: \text{Switch}$

System SwStore

Attributes
m: OnOff Initial off
v: Nat Initial 0

$\text{cr}: \text{Ack}$

$\text{cy}: \text{OnOff}$

Table 3
System Switch = SwStore$\dagger$Switching as a State Transition Table (let $s \in \{\text{on, off}\}$).

<table>
<thead>
<tr>
<th>$m$</th>
<th>$c_x$</th>
<th>$m'$</th>
<th>$c_y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>-</td>
<td>s</td>
<td>-</td>
</tr>
<tr>
<td>off</td>
<td>switch</td>
<td>on</td>
<td>on</td>
</tr>
<tr>
<td>on</td>
<td>switch</td>
<td>off</td>
<td>off</td>
</tr>
</tbody>
</table>

$\text{cx}: \text{Switch}$

System Switch

Attributes
m: OnOff Initial off
v: Nat Initial 0

$\text{cy}: \text{OnOff}$

Table 4
System Access = SwStore$\dagger$Accessing as a State Transition Table.

<table>
<thead>
<tr>
<th>$m$</th>
<th>$v$</th>
<th>$c_z$</th>
<th>$m'$</th>
<th>$v'$</th>
<th>$c_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>$j$</td>
<td>?</td>
<td>off</td>
<td>$j$</td>
<td>-</td>
</tr>
<tr>
<td>?</td>
<td>$j$</td>
<td>read</td>
<td>on</td>
<td>$j$</td>
<td>$j$</td>
</tr>
<tr>
<td>?</td>
<td>$j$</td>
<td>set($n$)</td>
<td>on</td>
<td>$n$</td>
<td>done</td>
</tr>
<tr>
<td>?</td>
<td>$j$</td>
<td>-</td>
<td>on</td>
<td>$j$</td>
<td>-</td>
</tr>
</tbody>
</table>

$\text{cz}: \text{AccData}$

System Access

Attributes
m: OnOff Initial off
v: Nat Initial 0

$\text{cr}: \text{Ack}$

The state of the service is given by two state attributes $m$ and $v$ with values of type OnOff and the type Nat of natural numbers. Table 2 defines the state machine (for simplicity we assume once more that there is at most one message in each time interval):

In this table, $m'$ and $v'$ denote the values of the state attributes after the state transition. By "-" we denote the empty sequence of messages. By "?" we denote arbitrary values (including "-").

We consider two sub-interfaces for the interface of system SwStore:

Switching = ($\{\text{cx}: \text{Switch}\} \uparrow \{\text{cy}: \text{OnOff}\}$)

Accessing = ($\{\text{cz}: \text{AccData}\} \uparrow \{\text{cr}: \text{Ack}\}$)

Let SwStore be the system behavior described above. We get by SwStore$\dagger$Switching a faithful projection with the simple table (since the state attribute $v$ is without influence on the output, its columns can be deleted) as shown in Table 3.

Note that the projection in our simple case means that we just eliminate the columns in the table and then simplify the resulting table.

By SwStore$\dagger$Accessing we get a projection that is not faithful, which therefore results in a nondeterministic behavior although SwStore is deterministic. SwStore$\dagger$Accessing is given by Table 4.

In states with $m = \text{off}$ a state transition may nondeterministically result in a state with $m = \text{on}$ or in a state with $m = \text{off}$. Similarly, in states with $m = \text{on}$ a state transition may nondeterministically result in a state with $m = \text{on}$ or in a state with $m = \text{off}$. Actually in each state transition the state of attribute $m$ may change nondeterministically.

We get two simple sub-services for SwStore. The first service SwStore$\dagger$Switching is a switch, which can be switched off and on. The second service SwStore$\dagger$Accessing is a store that can be written and read if it is in the mode $m = \text{on}$ whereas by every state transition nondeterministically the mode may be switched off and on.

The sub-service Access in the example above is dependent on the service Switch. To specify this dependency explicitly, we introduce the concept of a mode in the following.

3.2. Sub-services and their dependencies

In this section, we discuss the question, how a given service $F' \in IF[I' \uparrow O']$ that is to be offered by some system with interface behavior $F \in IF[I \uparrow O]$ where ($I' \uparrow O'$) sub-type ($I \uparrow O$), relates to the projection $F \uparrow (I' \uparrow O')$. This leads to the concept of a sub-service.
3.2.1. Unrestricted sub-services

A service behavior $F'$ is offered as a sub-service by a multifunctional system with behavior $F$, if in $F$ all the messages that are part of the service behavior $F'$ are as required in $F'$. This idea is captured by the concept of a sub-service.

**Definition (Sub-service Relation).** Given $(I' \triangleright O')$ subtype $(I \triangleright O)$, service $F' \in IF[I' \triangleright O']$ is a sub-service of a service $F \in IF[I \triangleright O]$, if for all histories $x \in I'$

$$F' = F \upharpoonright (I' \triangleright O')$$

We say that “service $F$ offers the service $F'$” and that “$F'$ is a sub-service of $F$”. We write $F' \leftarrow_{sub} F$. □

The sub-service relation forms a partial order. Since this is essential for the remaining sections, we give a proof for that proposition.

**Theorem.** The sub-service relation defines a partial order on the set of services.

**Proof.** Reflexivity and antisymmetry are obvious. Transitivity is shown as follows:

Let $(I_1 \triangleright O_1)$ subtype $(I_2 \triangleright O_2)$ and $(I_2 \triangleright O_2)$ subtype $(I_3 \triangleright O_3)$ hold and $F_1 \in IF[I_1 \triangleright O_1]$, $F_2 \in IF[I_2 \triangleright O_2]$, and $F_3 \in IF[I_3 \triangleright O_3]$ be given with

$$F_1 \leftarrow_{sub} F_2 \quad \text{and} \quad F_2 \leftarrow_{sub} F_3.$$  

Then by definition

$$\forall x \in I_1: F_1(x) = (F_2 \upharpoonright (I_1 \triangleright O_1))(x)$$

$$\forall x \in I_2: F_2(x) = (F_3 \upharpoonright (I_2 \triangleright O_2))(x)$$

holds. We get for all histories $x \in I_1$

$$F_1(x) = (F_2 \upharpoonright (I_1 \triangleright O_1))(x)$$

$$= (y|O_1: \exists x' \in I_2 : x = x'|I_1 \land y \in F_2(x'))$$

$$= (y|O_1: \exists x' \in I_2 : x = x'|I_1 \land y \in (F_3 \upharpoonright (I_2 \triangleright O_2))(x'))$$

$$= (y|O_1: \exists x' \in I_2 : x = x'|I_1 \land y \in \{y'|O_2: \exists x'' \in I_3 : x' = x''|I_2 \land y \in F_3(x'')\})$$

$$= (y|O_2|O_1: \exists x'' \in I_3 : x = x'|I_1 \land y \in F_3(x''))$$

This concludes the proof. □

Note that if $F' \leftarrow_{sub} F$ for $F' \in IF[I' \triangleright O']$ holds, this does not necessarily require that the projection $F \upharpoonright (I' \triangleright O')$ is faithful. Projection may introduce some additional nondeterminism; more precisely the choice of the output of $(F \upharpoonright (I' \triangleright O'))(x)$ may depend in $F$ on messages other than the messages included in the channels of $I'$; nevertheless $F' \leftarrow_{sub} F$ may hold as long as set $(F \upharpoonright (I' \triangleright O'))(x)$ is equal to $F'(x)$. Then messages outside of $I'$ may influence the choice of the output, which is correct in terms of $F'$, anyway.

A service may have many sub-services. The sub-service relation is significant from a methodological point of view, since it is the dominating relation for service hierarchies.

3.2.2. Restricted sub-services

The sub-service relation $\leftarrow_{sub}$ introduced so far is rather straightforward. Often, however, services are actually not sub-services but only somewhat close to that. Therefore we study weaker relationships between services $F' \in IF[I' \triangleright O']$ and the projection $F \upharpoonright (I' \triangleright O')$ of a multifunctional system with behavior $F$.

Let $(I' \triangleright O')$ subtype $(I \triangleright O)$ hold; in the remainder of this section we study situations in which the relation

$$F' \leftarrow_{sub} F$$

actually does not hold.

Nevertheless, even in such cases we want to say that the service $F' \in IF[I' \triangleright O']$ is offered by a super-system $F \in IF[I \triangleright O]$, if we restrict the input to $F$ to an appropriate sub-domain $R \subseteq I$ of $F$ that excludes the problematic input histories.
Definition (Restricted Sub-Service Relation). Given services $F' \in IF[I' \rightharpoonup O']$ and $F \in IF[I \rightharpoonup O]$ where $(I' \rightharpoonup O')$ subtype $(I \rightharpoonup O)$ holds, service $F'$ is called a restricted sub-service of service $F$ if there exists a subset $R \subseteq I$ such that

$$F' \leftarrow_{\text{sub}} F|R.$$

Here the partial service $F|R \in IF[I \rightharpoonup O]$ denotes as usual the restriction of function $F$ to subset $R$ of histories in $I$ with $(F|R)(x) = \emptyset$, if $x \notin R$, and $(F|R)(x) = F(x)$, if $x \in R$. If $R$ is the largest set for which the relationship $F' \leftarrow_{\text{sub}} F|R$ holds, then $R$ is called the service domain of $F'$ in $F$. □

Obviously, if $F' \leftarrow_{\text{sub}} F$ holds, then $F'$ is a restricted sub-service of $F$. The reverse does not hold, in general. The key question in the restricted sub-service relation of a service is, how to get a reliable access to the service $F'$ offered by $F$. To get access in $F$ to the service described by $F'$, we must not only follow the input patterns in $\text{dom}(F')$ but also make sure that the histories are in $R$. The restricted sub-service relation is a partial order, as well.

Theorem. The restricted sub-service relation defines a partial order on the set of services.

Proof. Reflexivity is obvious. The same holds for asymmetry. Transitivity is shown as follows:

Let $(I_1 \rightharpoonup O_1)$ subtype $(I_2 \rightharpoonup O_2)$ and $(I_2 \rightharpoonup O_2)$ subtype $(I_3 \rightharpoonup O_3)$ hold and $F_1 \in IF[I_1 \rightharpoonup O_1]$, $F_2 \in IF[I_2 \rightharpoonup O_2]$, and $F_3 \in IF[I_2 \rightharpoonup O_3]$ be given with

$$F_1 \leftarrow_{\text{sub}} F_2|R_2 \wedge F_2 \leftarrow_{\text{sub}} F_3|R_3.$$

Then we have

$$F_1 \leftarrow_{\text{sub}} F_3|R \text{ with } R = \{ x \in R_3 : x|I_2 \in R_2 \}.$$

This is shown as follows: for all $x \in I_1$ we get

$$F_1(x) = ((F_2|R_2)\circ (I_1 \rightharpoonup O_1))(x)$$

by

$$= \{ y|O_1 : \exists x' \in R_2 : x = x'|I_1 \wedge y \in F_2(x') \}$$

by

$$= \{ y|O_1 : \exists x' \in R_2 : x = x'|I_1 \wedge y \in ((F_3|R_3)\circ (I_2 \rightharpoonup O_2))(x') \}$$

by

$$= \{ (y|O_2)|O_1 : \exists x' \in R_2 : x = x'|I_1 \wedge y \in \{ x'|O_2 : \exists x'' \in R_3 : x'' = x''|I_2 \wedge y \in F_3(x'') \} \}$$

by

$$= \{ (y|O_2)|O_1 : \exists x' \in R_2 : x = x'|I_1 \wedge x' = x'\|I_2 \wedge y \in F_3(x') \}$$

by

$$= \{ (y|O_1)|O_1 : \exists x' \in R : x = (x'|I_2)|I_1 \wedge y \in F_3(x') \}$$

by

$$= \{ (y|O_1)|O_1 : \exists x' \in R : x = x'|I_1 \wedge y \in F_3(x') \}$$

by

$$= ((F_3|R)\circ (I_1 \rightharpoonup O_1))(x).$$

This concludes the proof. □

The restricted sub-service relation as introduced here is weaker and thus more flexible than the sub-service relation.

3.2.3. Independency relation between input actions and output

In this section we specify what it means for a system that an input action has some influence on certain output actions. For a multifunctional system with behavior $F \in IF[I \rightharpoonup O]$ and syntactic sub-interface $(I_1 \rightharpoonup O_1)$, projection $F\upharpoonright(I_1 \rightharpoonup O_1)$ provides an abstraction of $F$. If the projection is faithful, then there are no input actions in set $\text{Act}(I) \setminus \text{Act}(I_1)$ which influence the output actions of $O_1$ in $F$. Now we consider the case, where some input action $(m, c)$, with channel $c \in I$ but $c \notin I_1$ has influence on output actions of $F$ on channels of set $O_1$.

Definition (Independency of Projections of Messages). Let channel sets $I_2$ and $O_1$ as well as behavior $F \in IF[I \rightharpoonup O]$ be given with $I_2$ subtype $I$ and $O_1$ subtype $O$: the output actions of channel set $O_1$ are called independent of the input actions of $I_2$ within $F$ if for all input histories $x, x' \in I$ we have

$$x|I' = x'|I' \Rightarrow F(x)|O_1 = F(x')|O_1,$$

where $I'$ is the channel set with $\text{Act}(I') = \text{Act}(I) \setminus \text{Act}(I_2)$. □

If the projection $F\upharpoonright(I_1 \rightharpoonup O_1)$ is faithful then for each set of input channels $I_2$ with $\text{Act}(I_1) \cap \text{Act}(I_2) = \emptyset$ we get that for system $F$ channel set $O_1$ is independent of channel set $I_2$.

In our example for system SwStore channel set $\{cy\}$ is independent of channel set $\{cz\}$, whereas channel set $\{cr\}$ is not independent of channel set $\{cz\}$.
3.2.4. Dependency and independency of sub-services

In this section we specify, what it means that a sub-service is independent of another sub-service within a multifunctional system.

**Definition (Dependency and Indepency of Services).** Let sub-services \( F_1 \in IF[I_1 \rightharpoonup O_1], F_2 \in IF[I_2 \rightharpoonup O_2] \) of \( F \in IF[I \rightharpoonup O] \) be given with \((I_1 \rightharpoonup O_1)\) subtype \((I \rightharpoonup O)\) and \((I_2 \rightharpoonup O_2)\) subtype \((I \rightharpoonup O)\); service \( F_1 \) is called independent of the service \( F_2 \) in system \( F \), if the output actions of the channel set \( O_1 \) are independent of the input actions of \( I_2 \) within \( F \). If \( F_1 \) is dependent on the input actions in \( I_2 \) within \( F \) we write

\[
F_2 \to_{\text{dep}} F_1 \text{in } F.
\]

Dependency is not a symmetric relation. Service \( F_1 \) may be dependent on service \( F_2 \) in \( F \), while the service \( F_2 \) is independent of the service \( F_1 \) in \( F \). For example, sub-service \( F \uparrow \text{Switching} \) is independent of \( F \uparrow \text{Accessing} \) in \( F \) but not vice versa.

4. Structured specification of multifunctional services

So far we have followed an analytic approach to multifunctional systems. In an analysis of multifunctional systems we decompose them by projections into faithful or only weakly dependent sub-services. In this section we aim at using the introduced theory in a constructive way putting together multifunctional systems from given services.

4.1. Service hierarchies

A multifunctional system offers a family of services. The overall interface behavior of the system is modeled by service \( F \in IF[I \rightharpoonup O] \) where the sets \( I \) and \( O \) may contain many channels carrying a large variety of messages. In this section we show how the functionality and services offered by \( F \) are arranged into service hierarchies. In service hierarchies, names of sub-services are listed and syntactic interfaces are associated with them where each sub-service uses only a subset of the channels and messages of its super-service.

4.1.1. Syntactic service hierarchies

First we introduce a syntactic concept of service hierarchy that provides service names and syntactic service interfaces. Based on the concept of syntactic service hierarchy, we work out interpreted hierarchies where behaviors are associated with the service names.

To begin with, a hierarchy is a simple notion based on graph theory.

**Definition (Service Hierarchy).** Let \( \text{SID} \) be the set of service names. A service hierarchy for a finite set \( K \subseteq \text{SID} \) of service names is an acyclic directed graph \((K, V)\) where \( V \subseteq K \times K \) represents the sub-service relation. For every service with name \( k \in K \) the set \( \{k' \in K : (k, k') \in V\} \) of service names is called its syntactic sub-service family. The nodes in a service hierarchy without successor nodes are called the names of atomic services in the hierarchy. Their sub-service families are empty.

We denote the reflexive transitive closure of the relation \( V \) by \( V^* \). On \( K \) the relation \( V^* \) represents a partial order. Using specific names for the services of a hierarchy, we get an instance of a service taxonomy, which is a family of service names related by the sub-service relation.

**Definition (Rooted Hierarchy).** Hierarchy \((K, V)\) is called rooted, if there is a node \( r \in K \) called the root, such that for every node \( k \in K \) we have \((r, k) \in V^*\), which means that there is a path in \( V \) from the root \( r \) to the node \( k \).

A rooted service hierarchy specifies a sub-service structuring for its root service.

**Definition (Tree).** A rooted hierarchy is called a tree, if for each node \( k \) the path from the root to node \( k \) is unique.

A tree is a hierarchy without sharing of successor nodes, or – in terms of services – without services sharing sub-services.

**Definition (Syntactic Interface Service Hierarchy).** A syntactic interface service hierarchy is a service hierarchy \((K, V)\) with a syntactic interface \((I_k \rightharpoonup O_k)\) associated with each service name \( k \in K \) in the hierarchy such that for all service names \( k \in K \) we have: for every service \( j \) in the sub-service family of service \( k \) the relationship \((I_j \rightharpoonup O_j)\) subtype \((I_k \rightharpoonup O_k)\) holds.

If there is a path in a syntactic service hierarchy from node \( k \) to node \( j \), then \((I_k \rightharpoonup O_k)\) subtype \((I_j \rightharpoonup O_j)\) holds, since the subtype relation is transitive.

The following two properties characterize useful concepts for service hierarchies:

- A syntactic interface service hierarchy is called complete if for each non-atomic service name \( k \in K \) each input action in channel set \( I_k \) occurs as input action in at least one service of its syntactic sub-service family and each output action in channel set \( O_k \) occurs as output action in at least one service of its sub-service family.
A syntactic interface service hierarchy is called strict if for each non-atomic service name \( k \in K \) each input action in channel set \( I_k \) occurs as input action in at most one service of its sub-service family and each output action in channel set \( O_k \) occurs as output action in at most one service of its sub-service family.

In complete syntactic interface service hierarchies we only have to provide the syntactic interfaces for the atomic services and then the syntactic interfaces for the non-atomic services can be uniquely derived bottom–up from the atomic ones. In strict syntactic service interface hierarchies every input and every output is owned by exactly one atomic service.

Service hierarchies define the sub-service decomposition of services into sub-services. Syntactic interface service hierarchies associate channels and messages with each service.

4.1.2. Structuring service specifications by modes

In this section we introduce a technique to describe sub-services of a system in a modular way, even in cases where they are not faithful projections. We consider a system behavior \( F \in IF[I \triangleright O] \) and a sub-interface \( (I' \triangleright O') \) where \( (I' \triangleright O') \) is not faithful. Let for simplicity

\[
I = I' \cup I'', \quad O = O' \cup O''
\]

(where the sets \( I' \) and \( I'' \) as well as the sets \( O' \) and \( O'' \) are disjoint) and the types of the channels in \( I \) and \( I' \) as well as \( O \) and \( O' \) be identical. In other words, in the sub-interface \( (I' \triangleright O') \) we keep certain channels from \( (I \triangleright O) \) with the identical types. Let us furthermore assume that the projection \( F_{\mid (I' \triangleright O')} \) is faithful.

In this case, we cannot describe the sub-service offered by system \( F \) over sub-interface \((I' \triangleright O')\) exactly by projection. In fact, we can specify the unfaithful projection \( F_{\mid (I' \triangleright O')} \), but it does not give a precise description how the sub-service over sub-interface \((I' \triangleright O')\) behaves. To get a precise specification of the sub-service behavior as offered by system \( F \) over sub-interface \((I' \triangleright O')\) we need a way to capture the dependencies between the input actions in \( I'' \) that influence this sub-service, but are not in \( I' \), and the service over sub-interface \((I' \triangleright O')\).

One option to express the influence is the introduction of a channel \( cm \) between the over sub-interface \((I' \triangleright O')\) in \( F \) and the rest of \( F \) to capture the dependencies explicitly (see Fig. 3). Let the channel \( cm \) occur neither in channel set \( I \) nor in channel set \( O \). We define

\[
I^+ = I' \cup \{cm\}, \quad O^+ = O'' \cup \{cm\}.
\]

Our idea is to specify two behaviors \( F^+ \in IF[I^+ \triangleright O'] \) and \( F^\# \in IF[I'' \triangleright O^+] \) such that for all histories \( x \in I, y \in O \) the following formula is valid:

\[
y \in F(x) \Leftrightarrow \exists x^+ \in I^+, y^+ \in O^+: x^I' = x^+ \mid I' \land y^O'' = y^+ \mid O'' \land x^+(cm) = y^+(cm) \land y^+ \in F^\#(x^I' \land y^O')).
\]

This means that \( F^\#(x^I' \land y^O') \) provides on channel \( cm \) exactly the information needed from the input on channels in \( I'' \) to express for the sub-service in \( F \) on sub-interface \((I' \triangleright O')\) the dependencies on messages in \( I'' \).

We call \( cm \) a mode channel and the messages transmitted over it modes. In the following we explain the idea of modes in more detail. Later we study the more general situation where both projections \( F_{\mid (I' \triangleright O')} \) and \( F_{\mid (I'' \triangleright O^+)} \) are not faithful and mode channels in both directions are introduced.

Modes are a generally useful way to structure service behavior and to specify dependencies between services. Modes are used to discriminate different forms of operations for a service. Often mode sets consist of a small number of elements — such as enumerated types. An example would be the operational mode of a car being “moving_forward”, “stopped”, “moving_backward”. Nevertheless, arbitrary sets can be used as mode types. So we may have a mode Speed which may be any number in \([-30, \ldots, 250]\).

Formally a mode is a data element of a data type \( T \). \( T \) defines a set of data elements. Any type \( T \) can be used as a mode set. For a given type \( T \), we write Mode \( T \) to express that we use \( T \) as a mode type. We simply assume that type Mode \( T \) has the same elements as type \( T \). Each element of type Mode \( T \) is called a mode.

A mode type can be used for attributes of the state space as well as for input or output channels. For a service we may use several mode types side by side.

Example (Modes of a Mobile Phone). A mobile phone is, for instance, in a number of operating modes characterized by Mode Operation:

\[
\text{Mode Operation} = \{\text{SwitchedOff, StandBy, Connected}\}.
\]

Another mode set may reflect the energy situation:

\[
\text{Mode Energy} = \{\text{BatteryDead, LowEnergy, HighEnergy}\}.
\]

Both examples of modes are helpful to gain structured views of the services of a mobile phone. □

For services we use types that are designated as being modes to indicate which channels and attributes carry modes. We use modes in the following to indicate for sub-services of larger systems that do not correspond to faithful projections how the messages in the larger system influence the sub-service. This way we eliminate the nondeterminism caused by a non-faithful projection.

We use modes as follows:


Table 5
Service Access’ as result of service access enriched by channel \( m \) as input mode described as a state transition table.

<table>
<thead>
<tr>
<th>( v )</th>
<th>( cm )</th>
<th>( cz )</th>
<th>( v' )</th>
<th>( cr )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j ) off</td>
<td>?</td>
<td>( j )</td>
<td>( j )</td>
<td></td>
</tr>
<tr>
<td>( j ) on read</td>
<td>( j )</td>
<td>( j )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( j ) on set(( k ))</td>
<td>( k )</td>
<td>done</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( j ) on</td>
<td>( j )</td>
<td>( j )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6
System Switch’ as a state transition table (let \( s \in \{\text{on}, \text{off}\})

<table>
<thead>
<tr>
<th>( m )</th>
<th>( cx )</th>
<th>( m )</th>
<th>( cm )</th>
<th>( cy )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s )</td>
<td>--</td>
<td>( s )</td>
<td>( s )</td>
<td>--</td>
</tr>
<tr>
<td>( \text{off} ) switch</td>
<td>on</td>
<td>on</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>( \text{on} ) switch</td>
<td>( \text{off} )</td>
<td>( \text{off} )</td>
<td>( \text{off} )</td>
<td></td>
</tr>
</tbody>
</table>

- as attributes in state spaces to structure the state machine description of services — more precisely to structure the state space and also the state transitions; then we use state attributes with mode types called mode attributes. We speak of internal modes.
- to specify how services influence each other; then mode types occur as types of input or output channels called mode channels. We speak of external modes.

For mode channels we assume that in each time interval the current mode is transmitted.

External modes serve mainly the following purpose: they propagate significant state information from one service to other services of the system. If a service outputs a mode via one of its output channels, the service is called the mode master, if it receives the mode via one of its input channels the service is called a mode slave. Since in a system, each channel can be the output of only one sub-system, there exists at most one mode master for each mode channel.

**Example (Simple System Example (Continued)—Sub-Services Enhanced by Modes).** By replacing the attribute \( m \) in the example of the service Access as specified in Table 3 by an external input mode channel \( cm \), we make SwStore\textsuperscript{\dagger}Accessing deterministic again, as shown in Table 5.

Here \( m \) is no longer a local state attribute, but instead \( cm \) is an input mode channel. Accordingly we use the identifier \( m \) in SwStore\textsuperscript{\dagger}Switching not only as a state attribute but as an output channel, too.

To transform service Switch into service Switch’ channel \( cm \) is added as an external mode output channel as shown in Table 6. This way Switch’ becomes the mode master for the external mode channel \( cm \) for the mode slave Access’. 

Projection is used in the following sections to define a notion of sub-services and modes are used to capture the dependencies between sub-services.
We consider the system SwStore described in the example above. We extend SwStore by one further sub-service with additional mode channels ca and cb of type Nat denoting the set of natural numbers.

Here we have introduced a very simple additional sub-service. Using the state v as external input mode over channel cv the added sub-service called Mult is described by Table 8.

Here v is no longer a local state attribute, but an output channel for transmitting modes.

We get the service hierarchy shown in Fig. 4. We obtain a structured view onto system Demo in terms of a service hierarchy. It has three basic services that are all described independently by state transition tables and their dependencies are captured by modes. □

The key idea of the concept of a service hierarchy is that it is useful to decompose the functionality of the system into a number of sub-services that are specified and validated in isolation. Then dependencies are identified and specified by the horizontal dependency relation and labeled by modes that are used to specify the dependencies.

### 4.1.3. Interpreted service hierarchies

In this section we introduce service hierarchies where interface behaviors are specified for each service in the hierarchy. We speak of interpreted service hierarchies.

**Definition** (Interpreted Service Hierarchy). Given a syntactic interface service hierarchy $(K, V)$ where for each $k \in K$ the syntactic interface associated with $k$ is $(I_k \rightarrow O_k)$; an interpreted service hierarchy is a pair $(\langle K, V \rangle, \psi)$, where $\psi$ is a function $\phi : K \rightarrow \mathbb{F}$ that associates a service behavior $\psi(k) \in \mathbb{F}[I_k \rightarrow O_k]$ with every service name $k \in K$. The interpreted service hierarchy is called well-formed, if for every pair $(e, k) \in V$ the service behavior $\psi(k)$ is a restricted sub-service of $\psi(e)$. □

This form of service hierarchy does not indicate on which messages other than the input messages $I_k$ the restricted sub-service $\psi(k)$ depends. This information is included in an annotated service hierarchy.

**Definition** (Dependency Annotated Service Hierarchy). For an interpreted rooted service hierarchy $(\langle K, V \rangle, \psi)$ with root $r$ and a dependency relation $D \subseteq K \times K$, $(\langle K, V \rangle, \psi, D)$ is called annotated service hierarchy, if for service names $k, k' \in K$ with $(k, k') \notin V^*$ we have

$$(k, k') \in D \Leftrightarrow (\psi(k) \rightarrow_{dep} \psi(k')) \quad \text{in} \quad \psi(r). \quad \square$$

The relation $D$ documents all dependencies between services in the service hierarchy. If for a service $k$ there do not exist services $k'$ with $(k, k') \in D$, then service $k$ is required to be faithful. Note that there can be several dependencies for a service in a service hierarchy.

To give a more precise specification how in a service hierarchy a sub-service influences other services we use the concept of mode channels that allow us to specify the dependency of services in detail.

**Definition** (Service Hierarchy Annotated with Modes). For an annotated service hierarchy $H = (\langle K, V \rangle, \psi, D)$ the pair $(H, \psi)$, where $\psi : K \rightarrow \mathbb{F}$, is called service hierarchy annotated with modes if

- for each pair $(k, k') \in D$ a mode type $T_{k,k'}$ is given and a fresh channel $cm_{k,k'}$ with this type that serves as a mode channel
- the syntactic interfaces $(I_k \rightarrow O_k)$ of the services $\psi(k)$ are extended by the mode channels to syntactic interfaces $(I_k^+ \rightarrow O_k^+)$ of $\psi(k)$, where

$$I_k^+ = I_k \cup \{cm_{k,k'} : (k, k') \in D\}, \quad O_k^+ = O_k \cup \{cm_{k',k} : (k', k) \in D\}. \quad \square$$

In an annotated service hierarchy with modes, there is a mode channel $cm_{k,k'}$ for each dependency $(k, k') \in D$ from the service with name $k'$ to the service with name $k$. In the following section we describe, how to decompose the sub-services via their mode channels.

Relation $V$ is called vertical, relation $D$ horizontal for the hierarchy. An example of a horizontal relation in a service hierarchy is independency. In a horizontal relationship between two services $F_1$ and $F_2$ we do not deal with sub-service relations (neither $F_1$ is a super-service of $F_2$ nor vice versa) but with services that are either mutually independent or, where supposed to be, there exist specific feature interactions (for the notion of feature interactions see [9]) between these services that may be specified in terms of modes.

**Example** (Simple System Example (Continued)). We consider the system SwStore described in the example above. We extend service SwStore by one further sub-service with additional mode channels ca and cb of type Nat denoting the set of natural numbers.

Here we have introduced a very simple additional sub-service. Using the state v as external input mode over channel cv the added sub-service called Mult is described by Table 8.

Here v is no longer a local state attribute, but an output channel for transmitting modes.

We get the service hierarchy shown in Fig. 4. We obtain a structured view onto system Demo in terms of a service hierarchy. It has three basic services that are all described independently by state transition tables and their dependencies are captured by modes. □
4.2. Combiningservicesintomultifunctionalsystems

In this section we study sub-service based specifications of multifunctional systems aiming at a structured construction and description of the interface behavior of multifunctional systems from a user's and requirements engineer's point of view. A structured specification is essential in requirements engineering. The structuring is provided mainly in terms of relations between services.

4.2.1. Structuring multifunctional systems

Multifunctional systems incorporate large families of different, largely independent services. Services are formal models of use cases of systems. In this section we outline how to work out a multifunctional system in a sequence of development steps resulting in a service hierarchy as follows:

(0) Describe a set of use cases informally, identify all sub-services by introducing names and informal descriptions for them.
(1) Specify (a not interpreted) service hierarchy for the services identified in (0).
(2) Incorporate all the channels of the system and its services together with their types (to specify input and output actions) into the hierarchy extending it to a syntactic interface service hierarchy.
(3) Give behavior descriptions by interaction diagrams, by specifications through assertions or by state machines for each service; service behaviors are explicitly defined either for the atomic service names in the hierarchy or for their parent nodes; in the latter case the behaviors of the sub-services are derived by projection.
(4) Identify dependencies and introduce the horizontal dependency relation; define mode sets for each of the dependencies.
   Extend the service specifications for the modes.
(5) Combine the basic services via their modes into the overall system behavior.

The overall idea is to reduce the complexity of functional specifications of systems by describing each of its basic services independently by simple state machines. In a first step we do not take into account feature interactions. Only later do we combine the specified services into a service hierarchy and specify relationships between services by introducing modes to express how the services influence or depend on each other. Typically, some of the services are completely independent and are just grouped together into a system. Other services may depend on each other, with often just small, not very essential side effects on other services, while some services may heavily rely on other services that influence their behaviors in very significant and often subtle ways.

Understanding the overall functionality of a multifunctional system requires the understanding of its individual services, but also how they are related and mutually dependent.

4.2.2. Combiningservices

In principle, there are several ways to construct super-services out of given basic ones. First of all, we may combine given services into more elaborate ones. In the simple case of mutually independent services, we combine them directly into a multifunctional system.
Given two causal service interface behaviors

The combination of two services $F_1 \in IF[I_1 \triangleright O_1]$ and $F_2 \in IF[I_2 \triangleright O_2]$ is well defined only, if they are combinable; services are called combinable, if there exists a super-service, of which both services are sub-services. We denote the combination of combinable services $F_1$ and $F_2$ by

$$F_1 \oplus F_2 \in IF[I \triangleright O]$$

where $(I \triangleright O)$ is the least upper bound ("lub") of $(I_1 \triangleright O_1)$ and $(I_2 \triangleright O_2)$ in the subtype-relation. We define $F_1 \oplus F_2$ by the equation

$$(F_1 \oplus F_2)(x) = \{y \in \overset{\circ}{O} : y \in F_1(x|I_1) \land y \in F_2(x|I_2)\}$$

such that

$$F_1 \leftarrow \text{sub} \ (F_1 \oplus F_2) \land F_2 \leftarrow \text{sub} \ (F_1 \oplus F_2)$$

$F_1 \oplus F_2$ is called the independent service combination of the services $F_1$ and $F_2$. □

Note that for any channel sets $C_1$ and $C_2$ there is always a least upper bound for the subtype-relation, since sets of input channels are uniquely characterized by their sets of actions and the subtype-relation on sets of channels is equivalent to the subset-relation on the sets of actions. For the subset-relation on sets least upper bounds exist and thus also for the subtype-relation on sets of channels. For syntactic interfaces $(I_1 \triangleright O_1)$ and $(I_2 \triangleright O_2)$ the syntactic interface $(I \triangleright O)$ is the least upper bound where $I$ is least upper bound of $I_1$ and $I_2$ as well as the set $O$ is least upper bound of $O_1$ and $O_2$.

The simple diagram given in Fig. 5 illustrates independent service combination.

By service combination we put together multifunctional systems from elementary services. We combine a system $F \in IF[I \triangleright O]$ from a finite family $K = \{1, \ldots, n\}$ of independent combinable sub-services $F_k \in IF[I_k \triangleright O_k]$ of $F$ with $k \in K$; then

$$F = F_1 \oplus F_2 \oplus \cdots \oplus F_n.$$  

If we have for all $k \in K$

$$F_k = F|_{I_k \triangleright O_k}$$

then $F$ is the independent combination of its sub-services $F_k$.

Accordingly, for a given system with behavior $F$, we can ask whether this system can be combined from a family of independent sub-services.

4.3. Composing systems and services into architectures

In this section, a more general way to compose services and systems that mutually interact is defined, leading to component architectures (also called data flow architectures). First we introduce the composition of two systems or services. Then we extend the composition to families of systems or services forming architectures. Services and systems are composed by parallel composition with mutual feedback following [3].

**Definition (Composition of Services and Systems).** Given two causal service interface behaviors $F_1 \in IF[I_1 \triangleright O_1]$ and $F_2 \in IF[I_2 \triangleright O_2]$, with type consistent channels and where $O_1 \cap O_2 = \emptyset$, we define a composition of $F_1$ and $F_2$ involving the feedback channels

$$C_1 = O_1 \cap I_2 \quad \text{and} \quad C_2 = O_2 \cap I_1$$

by the expression

$$F_1 \otimes F_2.$$  

The behavior $F_1 \otimes F_2 \in IF[I \triangleright O]$ is defined as follows. Let

$$C = I_1 \cup O_1 \cup I_2 \cup O_2, \quad I = (I_1 \setminus C_2) \cup (I_2 \setminus C_1), \quad O = (O_1 \setminus C_1) \cup (O_2 \setminus C_2).$$

For $x \in I$ we specify:

$$(F_1 \otimes F_2)(x) = \{y \in \overset{\circ}{O} : \exists z \in C : x = z|I \land y = z|O \land z|O_1 \in F_1(z|I_1) \land z|O_2 \in F_1(z|I_2)\}.$$  

The channels in $C_1 \cup C_2$ are called internal for the composed system $F_1 \otimes F_2$. □
Fig. 6. Composition $F_1 \otimes F_2$.

Fig. 7. Refinement of two services to prepare for composition.

The composition of systems and services as defined above is graphically illustrated in Fig. 6.

The composition $\otimes$ works well, in particular, for strongly causal systems and also for composition without feedback (where $C_1 = \emptyset$ or $C_2 = \emptyset$). In a composed system $F_1 \otimes F_2$, the channels in channel sets $C_1$ and $C_2$ are used for internal communication. If $C_1$ and $C_2$ are empty, then composition is identical to combination:

$$C_1 = \emptyset \land C_2 = \emptyset \Rightarrow F_1 \otimes F_2 = F_1 \oplus F_2.$$  

The composition of systems with disjoint sets of input channels and disjoint sets of output channels is commutative

$$F_1 \otimes F_2 = F_2 \otimes F_1$$

as well as associative:

$$(F_1 \otimes F_2) \otimes F_3 = F_1 \otimes (F_2 \otimes F_3).$$

The proof of this equation is straightforward.

Composition is a partial function on the set of all system behaviors and the set of all services. It is defined only if the syntactic interfaces match, which means there are no contradictions or conflicts in their channel names and types. In this case, we can compose a set of given systems as follows. Given a finite, nonempty set $K = \{k_1, \ldots, k_n\}$ of system names and $

\psi : K \rightarrow IF$ such that the syntactic interfaces of $\psi(k)$ for $k \in K$ fit we write $\otimes \psi$ for the interface behavior of the architecture specified by

$$\otimes \psi = \psi(k_1) \otimes \cdots \otimes \psi(k_n).$$

This way we form a data flow network of systems with behaviors $\psi(k)$ for $k \in K$. The $\psi(k)$ then are called sub-systems or components of the network $\otimes \psi$. Note the significant difference between the concept of a sub-system and that of a sub-service. In general, $\psi(k)$ is not a sub-service and also not a restricted sub-service of component architecture $\otimes \psi$.

4.4. Service composition

Services that are to be combined might not be independent but actually may interfere with each other. This leads to the question of how to handle dependencies between services and still take advantage of their combination. We illustrate our idea of a systematic combination by Figs. 7 and 8.

Fig. 7 shows refinements of two services $F_1$ and $F_2$ by introducing additional mode channels. Fig. 8 shows how they are composed subsequently. Formally, we require that $F_1'$ and $F_2'$ offer the services $F_1$ and $F_2$ as sub-services — at least in a restricted form. To combine services from sub-services the channels in $C_1$ and $C_2$ carry only mode types.
We consider the system Demo from the example above. We get

\[
\text{Demo} = \text{Switch'} \otimes \text{Access''} \otimes \text{Mult}. 
\]

To prove this equation we have to refer to the tables specifying the behavior of the System Demo and the basic services Switch', Access'', and Mult. We form a table as shown by Table 10 for the services Switch', Access'', and Mult prepared with the entries found in Table 7 for Demo. Then we fill this table from the tables for the basic services Switch', Access'', and Mult. We start from the left by filling in the columns \(m', cm, cy\) for Switch' with the help of Table 6 specifying Switch'. Then we copy the output column \(cm\), which represents the modes on mode channel \(cm\), to the input column \(cm\) of Access'. Next we fill in the columns \(v', cr, cv\) for Access'' with the help of Table 9 specifying Access''. Then we copy the column \(cv\), which represents the modes on mode channel \(cv\), to the input of system Mult. Finally we fill in the column \(cb\) for Mult with the help of Table 8 specifying Mult. We obtain Table 11. Table 11 shows the overall service behavior of the system Switch' \(\otimes\) Access'' \(\otimes\) Mult. We compare Table 11 with Table 7 specifying System Demo. If we ignore the hidden mode channels \(cm\) and \(cv\) both tables are identical. This proves the equation.

Note that Table 7 is complete in the sense that all input patterns are covered in a complete case distinction (the same holds for the tables for Services Switch', Access'', and Mult). Note that Table 11 is not the result of a combination of independent services but of a composition with interaction via mode channels. □
Table 10
Table for the services Switch', Access'', and Mult with the entries as in Table 7 for system Demo.

<table>
<thead>
<tr>
<th>Service</th>
<th>Access''</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>j</td>
<td>a</td>
</tr>
<tr>
<td>off</td>
<td>-</td>
<td>a</td>
</tr>
</tbody>
</table>

Table 11
Table 10 from the tables for services Switch', Access'', and Mult.

<table>
<thead>
<tr>
<th>Service</th>
<th>Access''</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>j</td>
<td>a</td>
</tr>
<tr>
<td>off</td>
<td>-</td>
<td>a</td>
</tr>
</tbody>
</table>

As the example demonstrates, multifunctional systems can be specified by specifying their basic services in isolation and combining them into the overall system services interacting via mode channels. Accordingly, a service hierarchy \((K, V, \varphi, D)\) annotated with modes is called correct, if for each non-atomic node \(k \in K\) its interface behavior \(\varphi(k)\) is the composition of the interface behaviors \(\varphi(k')\) of the nodes \(k'\) in its sub-service family \(\{k' \in K : (k, k') \in V\}\) (see Fig. 9).

5. Summary and outlook

When dealing with typically complex, multifunctional software-intensive systems, the structured specification of their multifunctionality is a major goal in requirements engineering. This task is not sufficiently well supported by appropriate models, so far. In practice today, functional requirements are documented mainly by text. Models are not available and therefore not used. Use cases are applied but not formalized fully by models and not structured in hierarchies.

5.1. Related work

In telecommunications, service-oriented development is quite common. There the notions of feature and feature interaction are essential (see [9]). For instance, audio signaling is a service via the audio channel for signaling and user-interface issues. When features use audio signaling, and are assembled in a pipes-and-filters configuration, there is a potential of undesirable feature interactions. A way to describe features in telecommunication is Distributed Feature Composition (DFC), which is the invention of Michael Jackson and Pamela Zave [32] and relates feature-oriented descriptions to the needs of evolving systems, and explains how formal methods apply to both. The term “feature engineering” denotes the requirements-and-design process for feature-oriented systems.

Zave [33] describes the design in DFC of features for personal mobility, mail, switching and spontaneous conferencing. Zave [34] illustrates the design of a feature set consisting of selective call forwarding, blocking with urgent calling privilege, Outbound Messaging, and Inbound Messaging. The paper introduces a method for analyzing and managing the feature interactions that arise among this set.

Zave et al. [37] reports on the development of a voice-over-IP services centered on personal mobility explaining the functions of personal mobility, as well as its interactions with other major features. It shows how configurations and feature interactions are managed with the help of the coordination mechanisms in DFC. Zave [36] analyzes potential feature interactions. It proposes a method for eliminating some of them, as well as directions for future work on the remaining interactions. Zave [35] reports on their experience with using DFC to specify, build, and deploy the advanced features of voice-over-IP services.

Fig. 9 illustrates the approach based on modes and services in a service hierarchy as introduced in this paper, while Fig. 10 illustrates DFC. In Fig. 9 each service \(F\) in the hierarchy is given together with a diagram showing how it is composed of atomic services. In contrast to the service hierarchies and the combination of services, where the services are coordinated by modes (see Fig. 9), as described in the previous sections, in DFC the services are arranged in a pipeline (see [21]). Zave’s idea is to use a data flow architecture in a pipes-and-filters style (as illustrated in Fig. 10) to compose features, which, in our terms, is a composition of services into a specific architecture.
Use cases are a concept to describe instances of using systems and their features in scenarios. Use case diagrams (UCD), as advocated in UML (see [30]), can be understood as a technique to describe function families of multifunctional systems, but they do not address the concept of a function hierarchy. Also the possibilities to indicate the relationship between features are limited in UCDs.

In the context of product lines, a number of approaches exist along the lines of function hierarchies. A well known instance is the features in Feature-Oriented Domain Analysis (“FODA”) with its FODA trees (see [18]). The FODA methodology was founded on a set of modeling concepts and primitives used to develop domain products that are generic and widely applicable within a domain. However, FODA does not aim at a detailed specification of behavior.

In their book, Generative Programming, Czarnecki and Eisenecker (see [11] Section 4.4) describe how to build feature models consisting of a feature diagram plus semantic, rationale, and other attributes. There, feature models are used to drive design cycles, which eventually lead to manual or automatic assembly of configurations. For Czarnecki and Eisenecker a feature is “anything users or client programs might want to control about a concept” (see [11] Section 4.9.1). Actually, during feature modeling, they document not only functional features, but also implementation features, various optimizations, and alternative implementation techniques.

Also in [2] the concept of features is analyzed with respect to its usage in the context of product lines. As explained in [1] “a feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder’s requirement, to implement and encapsulate a design decision, and to offer a configuration option”. This is in contrast to our concept of services, which addresses the functional behavior of multifunctional systems at their interfaces in terms of their sub-services.

According to Gibson and Mery [14] features are units of “observable behavior”, and “requirements modules” serving as “units of incrementation as systems evolve”. This reference is also typical of the literature on features and services in defining a service notion that is localized to individual components. The interplay of components and services then emerges as an afterthought, rather than the starting point of service definition as advocated in our approach. The feature notion is similar to our service notion in the sense that functional aspects of the system are modeled in both approaches. Feature-oriented development is often employed in conjunction with object-oriented techniques. Our notion of service is completely independent of any programming style, however.
5.2. Concluding remarks and future work

Our service approach is a step towards model based requirements engineering of multifunctional systems. Service hierarchies annotated by dependency relations between the services in the hierarchy help to provide foundations for model driven requirements engineering for this type of systems.

What we included in this paper is only a first step towards a theory for the systematic development of multifunctional systems. We are interested in a simple and basic model of services and architectures just strong and rich enough to capture all relevant notions. It is our hope that services and systems provide appropriate foundations for model-orientation in the early phases of software development. We did not cover many of the relevant methodological issues. Open research issues include parts of the theory as well as issues of practice and engineering.

Although what we have presented is of rather theoretical character, our motivation is practical. So far there is not much modeling support for analyzing, representing and structuring the functionality of multifunctional systems. For such systems, we observe hundreds (in the case of mobile phones) or thousands (in the case of premium cars) of different functions (in our terminology services) which contribute to the overall functionality of a system and that are functionally dependent. Therefore, a decomposition of the overall functionality into sub-functions (in our terminology sub-services) and documenting their dependencies is a major goal for systematic requirements engineering.

Our current work in research aims at the application of the theory in a more pragmatic context. We hope that this will lead to a number of advantages such as:

- Deriving structured models for the usage view of systems free of any technical information, free of any classical architecture information, and just specifying the sub-services and their relations,
- Analyzing and specifying feature interactions between sub-services in the early phases of a system development,
- Tracing of the sub-services in the architectural decomposition,
- Establishing a kind of aspect oriented view onto the particular usage functions of multifunctional systems,
- Reuse of service hierarchies and their extension to product line approaches.

In the advanced engineering approaches used today in industry, a careful modeling of the architecture is mandatory. This is well understood and a guideline for the system development. However, till today a model-based approach to the service structure of multifunctional systems is missing but needed for a seamless model-oriented and model-driven development of software intensive systems.

Acknowledgements

It is a pleasure to thank Martin Deubler, Judith Thyssen, Elmar Jürgens, Alexander Harhurin, Philipp Neubeck, Andreas Rausch, Sabine Rittmann and Bernhard Rumpe for stimulating discussions and helpful remarks on draft versions of the manuscript. I am very grateful to the remarks of the anonymous referees including Pamela Zave on earlier versions of this paper that helped to improve and simplify the presentation considerably.

References

[22] D. Parnas, On the criteria to be used to decompose systems into modules, Communications of the ACM 15 (1972) 1053–1058.