Integrating Service Specifications on Different Levels of Abstraction

S. Rittmann, A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, D. Wild
Software & Systems Engineering
Technische Universität München
85748 Garching b. München, Germany

Abstract

The service-oriented paradigm is a promising approach to handle the growing complexity of software systems. This paper introduces a methodology for a stepwise refinement of service specifications on different levels of abstraction. Moreover, it deals with the integration of service- and architecture-specifications. Underlying concepts for a formal service specification are motivated from a methodological point of view and precisly given in this paper. Furthermore the application of these concepts is demonstrated within a case-example.

The presented methodology stems from the RoFaSoft[1] *project, where both aforementioned development paradigms are consolidated and integrated.*

1 Introduction

The emerging interoperability among software systems, especially across operational boundaries, results in challenging problems. The heterogenity of distributed systems and the complex interactions between those have to be dealt with. A promising approach to handle this intricacy is the upcoming service-oriented paradigm. Here, a service is a *piece of functionality* like for example the opening of the power windows of a car. Therefore, the focus lies on the system *behavior*, and not on the system *structure* (as it is the case with the traditional component based approach).

Having their roots in the area of telecommunication, services have started also to conquer other domains, e.g. web-services in the internet, look-up or naming-services in the field of middleware. However, most of the work in these areas focus on implementation technologies like WSDL (cf. [2]), SOAP (cf. [9]) and CORBA (cf. [3]). The only abstraction is achieved by defining services by a syntactic list of procedures which are called by a client. This resembles to the idea, that a service simply is a function or (at least) can be decomposed hierarchically into functions at the most fine-grained level.

However, simply specifying a service by its syntactic interface is not sufficient. The problem is that complex interactions and the interplay of functions, which is necessary to provide a thorough understanding of the system, are lost sight of. Local views on pieces of the system functionality do not allow for an overview of the various relationships between system entities. This is especially true for domains where systems are characterized by a high amount of interactions between functional modules. It is therefore inevitable to have a service notion that can cope with complex dependencies between functions. We need an *abstract* understanding of the term *service* that can be used in the overall development process.

A suitable service-oriented development methodology should therefore offer concepts, to start with abstract service specifications, and to refine those specifications into more concrete ones on various levels of abstraction (which finally lead to implementation). Furthermore, modeling techniques should be used to specify services by clearly stating the modeled service-aspects and the abstracted aspects on each level, respectively.

In this paper we make following contributions:

- We show how complex interactions between functional system entities are not lost sight of during the overall development process.
- We provide a service notion that can be used on different levels of abstraction. Additionally, we clearly state all modeling aspects.
- We show how an abstract service specification can be refined into a concrete one by introducing a service-oriented development methodology.
- We clearly separate the notion of services from the notion of functions and show, how these parts of a system specification fit together.

[1]The work is partially funded by the Bavarian Government under grant number IuK 188/001.

The paper is structured as follows: In the following, we describe a running example which we use in order to illustrate our concepts. In section 3 we introduce the different levels of abstraction starting from the most abstract and leading to the most concrete one. Section 4 relates our approach to other work. In section 5 we give a conclusion and list future work.

2 A Running Example: The Power Windows System

In order to introduce our concepts of different service abstractions, we will make use of a running example from the automotive domain - the functionality of automotive power windows. The considered system has to fulfill the following requirements:

- **Opening and closing.** By pressing a toggle switch (open/close), the power windows are moved into the corresponding direction as long as the switch is pressed or until the end position is reached.
- **Comfort opening and comfort closing.** A short tap on the switch (open/close) leads to a complete aperture/closure of the window. The press or tap of the switch in any direction causes the windows to stop. If a clamp is detected during the closure, the window is first stopped and then moved down for two seconds.
- **Child safety lock.** The operation of the back windows can be prohibited by activating the child safety lock (switch located on the drivers side).

Due to the limitation of space further requirements such as shut-protection, repetition-lock or block-detection are not considered in this paper.

3 The Service Methodology

In the following we introduce our service-oriented methodology. To that end we describe the different levels of service abstraction that serve as a basis for our approach. Starting from abstract service specifications we become more and more concrete on the levels beneath until we obtain implementation-close specifications. Notice, that the concretizations made on one level are also present on the proceeding levels.

The subsections are organized as follows: We first *motivate* each abstraction level. Then we give a *survey of concepts* used on each level. Additionally, we present a *formal specification* of the concepts introduced. The *case example* illustrates our ideas and shows how the concepts on each level can be modeled with the help of design techniques. The end of each subsection summarizes the aspects we abstract from on the respective level. These *abstraction aspects* are: ***service relationships, time, states (and system transitions), additional behavior***, and ***actors/entities***.

For reasons of simplicity, we do not take care of the aspect ***data*** in this paper.

3.1 Abstraction Level 1: Interaction between System and System Environment

Motivation. Systems - in particular: multi-functional systems - can be very complex. In order to handle the complexity resulting from a high degree of functions and dependencies between those, we start our methodology as follows: We first do a scoping of the system under consideration by capturing its black box behavior.

This is done by describing *exemplary interaction sequences* containing the activites performed *between the system and its environment*. It is important that these sequences are manageable effectively and therefore are not too large. They give a first idea of the main services that are visible to the outside of the system. Each interaction sequence, or a set of related interaction sequences, can be seen as a service or a sub-service (depending on the level of granularity).

Survey of Concepts. On this level of abstraction we consider *actions*; to be more precise:

- *input actions* by a user (where user does not only refer to human users but to the whole system environment), and
- *output actions* of the system which are a response to the user stimuli and which deliver a visible output to the environment.

Furthermore, we take into consideration both *causal* and *temporal dependencies* between the actions performed. Also, we look at *service relationships*.

Formal Specification. For modeling actions we introduce *interaction sequences*. They are capable of capturing *causal* and *temporal properties between interactions* (alternative and parallel execution, and abortion of actions). Furthermore, *time specifications* can be expressed.

The structure of interaction sequences can be defined by a grammar. In the following we use the Backus-Naur-Form (BNF, cf. [8]) to represent the grammar syntactically. We will construct the BNF for our interaction sequences step by step, bottom-up.

Activities Interaction sequences are based on *activities* (Act) which consist of

- input actions ($InAct$) performed by a user (which - as already mentioned - refers to the whole system environment) and
- output actions ($OutAct$) performed by the system as response to input actions.

In Backus-Naur-Form[2]:

$< Act >$ ::= $< InAct > \mid < OutAct >$
$< InAct >$ ::= $Iact1 \mid Iact2 \mid ... \mid IactN$
$< OutAct >$ ::= $Oact1 \mid Oact2 \mid ... \mid OactN$

$InAct$ and $OutAct$ must be disjoint in order to imply which action is performed by the system and which by a user.

Timed Activities Each activity is performed for a certain duration which can be arbitrarily short (even nearly punctual) or arbitrarily long. Consequently, we introduce *timed activities* $TAct$:

$< TAct >$ ::= $(< Act >, < Time >) \mid$ (1)
$(< Op >, < Time >, < Act >) \mid$ (2)
$< Act >$ (3)
$< Op >$ ::= $< \mid > \mid =$
$< Time >$ ::= $< Real > sec$ (4)

The semantic interpretation of the derivation rules is as follows:

- $Real$ is the set of real numbers; sec stands for "seconds" (cf. (4)).
- $(act, [2sec])$ denotes that activity act is performed for 2 seconds (cf. (1)).
- $(<, [2sec], act)$ expreses that activity act starts less than 2 seconds after the preceeding action ended (cf. (2)).[3]
- It is also valid to omit a time specification (cf. (3)), e.g. when it is of no relevance.

Interaction Sequences Now, we are able to give the grammar for our interaction sequences ($< S >$):

$< S >$ ::= $< S > \mapsto < S > \mid (< S > \mid\mid < S >) \mid$
$< S > : < S > \mid \{< S >, < S >\} \mid < TAct >$

Again, we explain the semantic interpretation of the derivation rules:

- $\mapsto$ denotes the (causal and temporal) ordering of actions. For example, $act1 \mapsto act2$ means that $act1$ is performed prior to $act2$. Activities can be comprised of other activities, e.g. $act1 \mapsto \{act2, act3\} \mapsto (<, [3sec], act4)$.[4]
- $act1 : act2$ denotes that activity $act1$ is aborted by activity $act2$.
- $\{act1, act2\}$ denotes that either activity $act1$ or $act2$ is performed.[5]
- $(act1||act2)$ denotes that activites $act1$ and $act2$ are performed in parallel.

Case Example. We illustrate the use of our interaction sequences by specifying the interaction behavior of our running example. First we determine the input and output actions: Our switch has five positions: `open` (O), `close` (C), `comfortOpen` (CO), `comfortClose` (CC) and the rest position. The switch can be pressed ($_P$) and released ($_R$). We obtain the following input actions (terminal symbols):

$< InAct >::= CO_P|O_P|O_R|C_P|C_R|CC_P|Block|Clamp$

We do not have to specify the release of the comfort positions as they do not have any impact on the functionality. Analogously, the rest position is not modeled either.

The output actions of the system can be the closing (W_{UP}), stopping (W_{STOP}) and opening (W_{DOWN}) of the window[6]. Therefore:

$< OutAct >::= W_{UP}|W_{STOP}|W_{DOWN}$

Usually, the input and output actions are not known from the start but elaborated when developing the interaction sequences. However, in this simple example we give them right from the start.

The interaction sequences of our comfort closing service look like follows:

ComfortClosingBlock: $CC_P \mapsto W_{UP} : Block \mapsto (<, [0.1sec], W_{STOP})$. The semantics are: When the position `comfortClosing` is selected, the window is moved up. When a block is detected, the motion of the window is aborted and the window is stopped (within less than 0.1 seconds).

ComfortClosingCmd: $CC_P \mapsto W_{UP} : \{C_P, O_P, CO_P\} \mapsto W_{STOP}$. Again, on moving the switch in the `comfortClosing` position, the

[2] As usual (cf. [8]), nonterminals are put in angle brackets.

[3] Analogously, $(>, [2sec], act)$ and $(=, [2sec], act)$ express that activity act starts at least 2 seconds, or exactly 2 seconds after the preceeding activity, respectively.

[4] Do not confuse our ordering symbol ($\mapsto$) with the derivation symbol ($\rightarrow$) typically used in grammars (e.g. $S \rightarrow a \rightarrow bc \rightarrow ... \rightarrow Terminals$).

[5] Notice, that we use BNF and not Enhanced BNF (EBNF, [8]). In EBNF the expression { ... } has another meaning!

[6] The special case that no action is performed is already covered by the grammar rules.

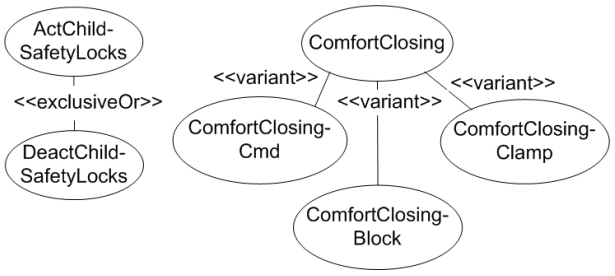

Figure 1: Relationships Between Interaction Sequences (Modified UML Use Case Diagram)

window is moved up. When another position is switched, the window stops.

ComfortClosingClamp: $CC_P \mapsto W_{UP} : Clamp \mapsto (<, [0.1sec], W_{STOP}) \mapsto W_{DOWN}[2sec] \mapsto W_{STOP}$. This sequence stops the motion of the window (within less than 0.1 seconds) after a clamp has been detected. The window is moved down for two seconds and stopped again.

Here, each sequence describes one variant (or version) of the comfort closing service (comfort closing until a block occurs, another service is called, or a clamp is detected). The sequences can be seen as exemplary interaction behavior, respectively. The total of the three variants can be called a service.

For simpler services it might be sufficient to give one sequence for the service (and not to specify several ones).

On this abstraction level, we also capture associations between our interaction sequences. We suggest the following ones:

- *variant*: Two interaction sequences are variant if they contain the same (main) subsequences. (E.g. the same start and end sequences.)
- *exclusive*: If two interaction sequences are exclusive, only one can be performed at a time.
- *independent*: Mutally independent interaction sequences.

In Figure 1 the different associations between our actions are shown graphically by means of a UML Use Case-like notation. Each oval circle represents an interaction sequence (except `ComfortClosing` which is the union of the service variant `ComfortClosingBlock`, `ComfortClosingCmd`, and `ComfortClosingClamp`). The stereotypes <<`variant`>> and <<`exclusiveOr`>> indicate the relationships. No connection between two sequences means that they are mutually independent.

Interaction sequences are a powerful, yet simple, way to specify the interaction behavior of a system under consideration. However, they have to be enhanced in the future to also capture system behavior in a more elaborate fashion.

Abstraction Aspects. The aspects *time* and *service relationships* are considered right from the start. On the very first level, we already capture different (causal and temporal) relationships between services and therefore do not abstract from these aspects.

As this is the first step performed to gather the system functionality, we do not speak from *additional behavior* on this abstraction level. *States* (and *transitions*) and *actors/entities* are not part of this level either and therefore are abstracted from, too.

3.2 Abstraction Level 2: Integrating Behavior

Motivation. The aim of our methodology is the specification of the overall system behavior. So far, we specified the services separately by introducing an interaction sequence for each service or service variant, respectively. This has two advantages:

- Single services (service variants) can be reused.
- The complexity is reduced, as only one service (variant) is designed at a time.

In order to obtain the overall system behavior, the *closure of the interaction sequences* has to be created; i.e. the sequences have to be combined or *integrated*. The aim of this level is not to specify the services separately (as in the previous level), but to specify the *overall system behavior with all its interactions and dependencies*.

When combining the behavior, contradictories can be detected and eliminated. Additionally, we have to delete non-determinism which might be a result of the service combination.

Survey of Concepts. The following concepts are introduced on this level of service abstraction:

- *states* being places at which the sequences are "glued together" and
- *transitions* partitioning actions into *constraints* and *(fired) events*.

The identification of states and transitions is a genuine design decision.

Formal Specification. We make use of *Moore automata* (cf. [5]) to model our combined services (service variants). As result, we want to obtain *one* Moore automaton specifying the overall functionality of the system under construction (which can refer to the whole system or only a sub-system). *Hierarchical automata* help us to reduce the complexity at this point.

To reach our aim, we construct the automaton step by step. First we create small automata each specifying one or several services. Then we combine these automata to get larger automata until the overall automaton is obtained.

The question that we face at this point is whether to create one automaton per interaction sequence or one automaton for capturing the behavior described by several interaction sequences. If we recognize common behavior (identical sub-sequences), it might be a good idea to integrate them into one automaton. Another way is to combine automata that have common states. In our terminology we call the integrated Moore automaton *Interaction Automaton* which is formally given by a 5-tupel:

$$\widetilde{IA} = (States, TInAct, TOutAct, \lambda, \delta, S_0)$$

- $States$ is a finite set of control states.
- $TInAct$ is a finite set of incoming timed interactions.
- $TOutAct$ is a finite set of outgoing timed interactions.
- $\lambda : States \rightarrow TOutAct$ is a output relation. The semantics of $(s, act) \in \lambda$ is that if the system is in state s, the action act is performed.
- $\delta : States \times InAct \rightarrow States$ is a nondeterministic transition relation. The semantics of $(s_1, act, s_2) \in \delta$ is that if the system is in state s_1 and the action act happens, the system switches to state s_2.
- $S_0 \subseteq States$ stands for the starting states of different services.

Case Example. In the example of our comfort closing service, we easily recognize that the sequences start and end with identical parts, respectively. This makes sense as the sequences are variants of each other (cf. stereotype <<`variant`>> in Figure 1). Therefore, we create one automaton for the overall service. The graphical representation can be seen in Figure 2. It can also be specified formally as follows:

$\widetilde{ComfortClosing}$ = ({`NoMove`, `TimeToStop`, `Closing`, `ClampDetected`, `Stopped`}, {CC_P, C_P, O_P, CO_P, $Clamp$, $Block$, $\perp$}, {W_{UP}, W_{STOP}, W_{DOWN}}, {(`NoMove`,CC_P,`Closing`), ...}, {(`Closing`, W_{UP}), ...}, `NoMove`).

The service variants (`ComfortClosingBlock`, `ComfortClosingCmd`, and `ComfortClosingClamp`) can be seen as *paths through the automaton*.

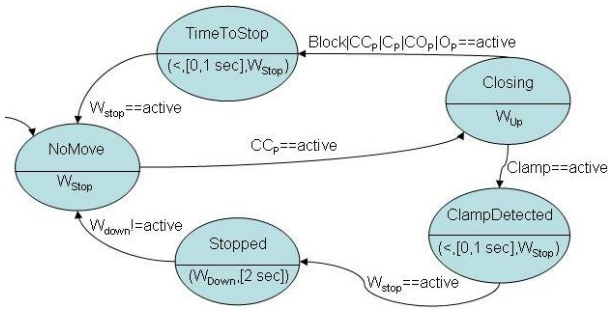

Figure 2: `ComfortClosing` Service (Automaton)

We proceed like this for the `ChildSafetyLocks` service (cf. Figure 3). As the system can either be in the activated or deactivated state (cf. stereotype <<`exclusiveOr`>> in Figure 1), we need two separate states between which we can switch.

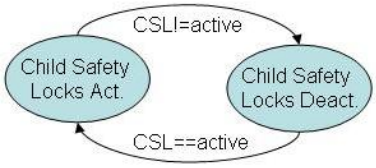

Figure 3: `ChildSafetyLocks` Service (Automaton)

As a next step, we combine the automata. Figure 4 shows the overall integrated behavior - and thus: integrated automaton - of our running example. As the `ComfortClosing` and the `ChildSafetyLocks` services are mutually independent (no connection in Figure 1) we simply compose the automata with help of an AND-state.

When specifying the system behavior, it is necessary to also *refine* the services. Services have to be created that are not visible to the outside, but needed to establish the system services which can be accessed by the user. For example, the controling of the window motors is needed by all power window services although not directly callable by the user. On this level of abstraction (and also on the proceeding ones) the behavior - and therefore: automata - are

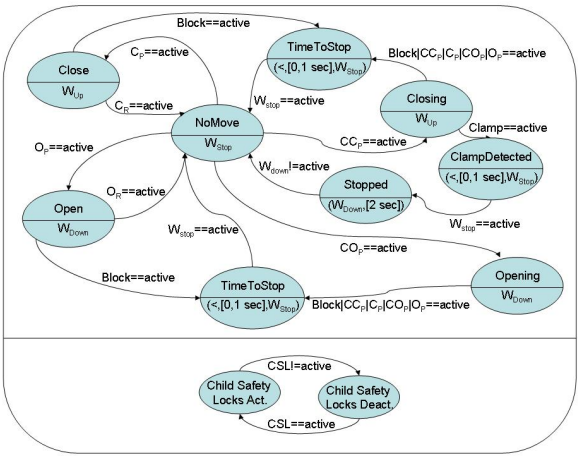

Figure 4: Power Window Service - Integrated Behavior (Integrated Automaton)

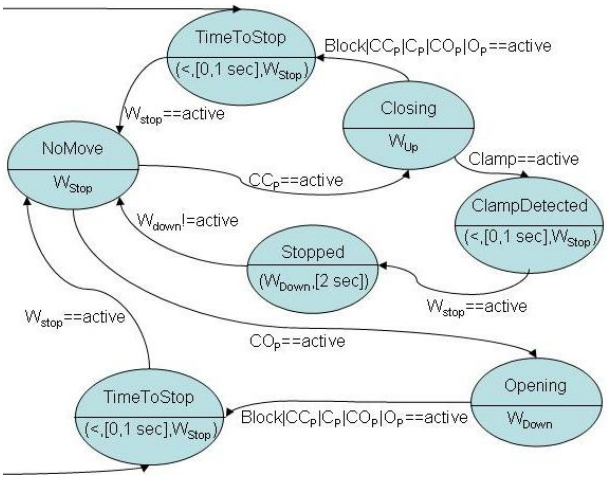

Figure 5: Comfort Functionality (Partial Automaton)

also refined. For the sake of simplicity we do not show this step in this paper. Notice, that when taking care of behavioral refinement the automaton has to be enlarged to $\widetilde{IA} = (States \cup States', InAct \cup InSys, OutAct \cup OutSys, \lambda', \delta', S_0' \subseteq States')$; where $InSys$ and $OutSys$ contain the actions which are not visible to the outside of the system but needed for the service refinement; $States'$ denote the states that are added in context of the refinement; S_0' represents the new set of starting states (if neccessary) and λ' and δ' are the new relations.

Abstraction Aspects Still the *service relationships* and *time* are looked at on this level. Additionally, we now take care of different *states* the system can be in and of respective *transitions*. Furthermore, *additional behavior* is identified as automata are refined.

The *actors/entities* do not play a role.

3.3 Abstraction Level 3: Partitioning Behavior into Functions

Motivation. At this stage of our methodology, we have a functional network of services containing all the complex interactions and dependencies. However, this network can not be mapped to a processor as one processor can not handle the whole functionality. Therefore, before we can map pieces of the functionality to pieces of hardware (processors), we have to decompose - or: *partition* - our service network *into functions*.

A question at this point is according to which aspects the partion is done. Some ideas are: logical clustering, minimum of communication effort, maximum degree of parallelism (mutually independent functions should be composed in parallel), reuse from a logical perspective.

When partitioning our service network into a set of functions we have to be aware of the following fact: Usually, not all of the input/output actions (that are relevant for the whole service network/the system) are relevant for each function. That means, that we have to define those *input/output actions which are of relevance for one function*, respectively.

Survey of Concepts. We use the concepts of *partial functions* and of *interfaces of partial functions* on this abstraction level. A partial function is a function which does not have a defined output for each input.

Formal Specification. We make use of partial automata (cf. [5]) each specifying one partial function. A partial automaton is characterized by a partial transition relation δ. This means that there exist arguments for which λ and δ are not defined, respectively (formally: $\exists s_1 \in States, act \in InAct$: $\forall s_2 \in States$ $(s_1, act, s_2) \notin \delta$; λ analogously). To stress the fact that we carefully look at all relevant input/output actions for each function we call our partial automata *Partial Interface Interaction Automata*.

As result we obtain a *set* of Partial Interaction Automata that - together - perform exactly the same behavior as the Interaction Automaton specifying the service network (cf. section 3.2).

Case Example. For our power windows, we could have decided to realize the comfort functionality (comfort closing and comfort opening) by one function. The result would look like Figure 5. Here, we simply cut out the respective states and transitions that are needed by the comfort functionality.

The next step is to determine the interface of the partial function (Partial Interface Interaction Automaton). We easily see, that the commands CC_P, CO_P, O, C, *Clamp*, and *Block* are input data to this automaton. But are there any further inputs which have an influence on the comfort function? The (de)activation of the child safety locks does not affect the comfort functionality of the driver's window (which we consider in this paper). Therefore the inputs `activateCSL` and `deactivateCSL` do not have to be taken into consideration for the comfort function. However, we face another problem: The `Opening` and `Closing` do affect our comfort mechanisms. If the `open button` is pressed, the system has to "decide" whether to open the window (if no comfort function is being performed currently) or if the comfort (un)locking service has to be stopped (if a comfort service is being performed). For that end, we would have to insert a synchronisation logic.[7]

Abstraction Aspects. The answering of the question which behavior goes in which function is a genuine design decision. By choosing functions we already make a step towards thinking of *actors/entities* as functions will later be mapped to hardware entities. Therefore, we no longer abstract from actors/entities. Also, as mentioned in subsection 3.2, we refine behavior on this level and thus obtain *additional behavior*.

Another design decision on this level is to determine the necessary input and output actions.

3.4 Abstraction Level 4: Totalization of Functions

Motivation. So far, we have specified our system in terms of partial functions (modeled in terms of Partial Interface Interaction Automata). However, it is necessary to specify what the system has to do for each possible input in each possible situation. Therefore, we need to refine our functions.

Survey of Concepts. On this abstraction level, we have *total functions*. A total function is a function which has defined output data for each input data. The interface descriptions of the previous level give us a helping hand.

Formal Specification. We make use of total automata each specifying one total function. A total automaton is characterized by a total transition relation δ. This means that: $\forall s_1 \in States, act_1 \in InAct$: $\exists act_2 \in OutAct, s_2 \in States\ (s_1, act_1, act_2, s_2) \in \delta$.

The automaton is given by:

$\widetilde{IIA} = (IInAct, IOutAct, States', S_0', \delta')$

[7] Omitted here because of simplicity.

The set of total automata specifies our overall system in terms of functions.

Case Example. In order to turn our partial automata into total automata, we have two possibilities: Either

- we add missing transitions (so that in each state it is defined what output should be generated for each possible input), or
- we use automata accordingly to the Harel semantics (cf. [4]).

In the latter case, the system does nothing in case an unspecified input data occurs for the current state. Due to reasons of simplicity, we make use of the second variant for our running example.

On this level we add more information about behavior and therefore become less abstract.

As a result of our methodology, we now have *total functions* which together establish the system services by collaborative interworking. The total functions can be mapped to *structural system pieces* (e.g processors).

Abstraction Aspects. On the very last abstraction level, we refine our automata and get *additional behavior*.

4 Related Work

As already mentioned in the introduction, most of the work on service-orientation deals with technical issues and does not consider different levels of abstraction. To the best of our knowledge, we do not know of any work that investigates *services on different levels of abstractions*.

In [1], the authors introduce a service notion that is based on the notion of streams. This notion can be used to specify a system merely in terms of services and functions. However, services are only looked at using one level of abstraction. In the future, we will investigate how this stream-based notion can be used in our methodology (e.g. on more concrete abstraction levels).

In [7], interaction-based services are presented. These services are spread across structural system entities (e.g. components, packages, classes); i.e. that a service is established by the interplay of several structural entities. They are similar to our services in regard that our services can also be partitioned into functions which in turn are mapped to structural system entities. However - again - this work does not consider different abstraction levels.

In [6] service refinement is introduced. However, refinement in this context is understood as the stepwise adding of more informational data. E.g. first a basic printer service is described; when a user specifies the type and the resolution of the document to be printed, more information about the printer service (e.g. time when printing will finish, costs, etc.) is added - and therefore the service is refined (according to the authors' understanding of refinement).

5 Conclusions and Future Work

In this paper we introduced a service-oriented development methodology resting on precise modeling concepts to represent services. Using various abstraction levels and our notions of services and functions, we come to the following conclusions:

Separation of services and functions. Due to the clear separation of considered aspects the conceptual separation of services and functions is comprehensible.

Abstract specification of services. In contrast to other approaches it has proven to be useful to specify services already in an abstract manner.

Domain-Independency. Despite the approach was originally developed in the automotive domain, we claim, that it is also valuable in other domains.

Logical relationships between services are caught on the abstract level in an overall functional service network. As the distribution of services to functions is explicit (by partitioning the service network into functions) the logical relationships between functions are also well-understood.

Design decisions. The specification steps on every abstraction level represent design decisions and can not be automated. This is especially true for the transition from the service network to partial functions.

Figure 6 gives an overview about our abstraction levels, the aspects we abstract from, and the modeling techniques we made us of, respectively.

In our current work we study the extension of our methodology to manage the integration of non-functional services. Additionally, we take care of the aspect *data* which was omitted in this paper. In particular we investigate how input and output interactions can be mapped to data.

On the formal level we are working on a service specification language which represents the descripted service aspects. Last but not least we are working on a theory to manage the feature-interaction problem on different levels of abstraction.

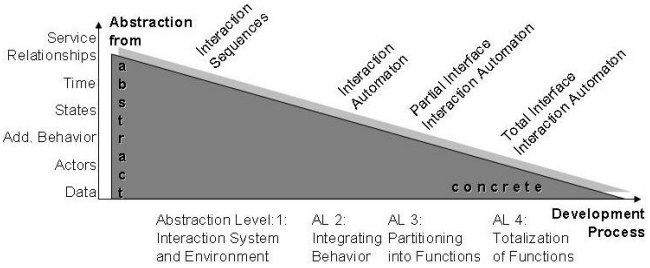

Figure 6: Levels of Service Abstraction

References

[1] M. Broy and I. H. Krüger. Services and service-oriented software architectures - methodological foundations, 2004.

[2] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl). *W3C*, (Note 15), 2001. Available at: `www.w3.org/TR/wsdl`.

[3] O. M. Group. Common object request broker: Architecture and specification, revision 2.2. *Online Sites of Object Management Group*, 1998.

[4] D. Harel. A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[5] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Number ISBN 0-201-44124-1. Addison Wesley, 2001.

[6] M. Klein, B. König-Ries, and P. Obreiter. Stepwise refinable service descriptions: Adapting DAML-S to staged service trading. In *Proceedings of the First International Conference on Service-Oriented Computing, ICSOC*, LNCS 2910. Springer, 2003.

[7] I. H. Krüger. Service specification with MSCs and roles. In *Proceediungs of IASTED International Conference on Software Engineering*, 2004. Available at: `http://www.cs.ucsd.edu/~ikrueger/publications/iasted_SE_04.pdf`.

[8] Meyers Lexikonredaktion. *Duden Informatik - Ein Fachlexikon für Studium und Praxis*. Number ISBN 3-411-05233-3. Bibliographisches Institut & F.A. Brockhaus AG, Mannheim, Germany, 2001.

[9] W3C. Simple object access protocol (soap). *W3C*, (Note 8), 2000. Available at: `http://www.w3.org/TR/soap/`.