A Hierarchical Model for Structuring Functional Requirements and Their Dependencies

Andreas Vogelsang

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, D-85748 Garching, Germany
vogelsan@in.tum.de

Abstract. Almost every software system nowadays can be considered as a multifunctional system in the sense that it offers the user a multitude of functions and use cases. The increasing number of functions and their complex interplay (feature interaction) is becoming a major cause of system failures and unintended behavior. We propose a hierarchical model that structures the functional requirements of a system and supports behavior composition of modular services. This model provides a formal system specification, which reveals and handles functional dependencies.

1 Introduction and Problem Statement

Most software systems designed today do not only serve one purpose but combine a lot of functional features. A modern luxury car for example consists of hundreds of user functions [4]. However, this increasing number of functions and their complex interplay is becoming a major cause of system failures [11]. This fact originates from the following problems:

Imprecise system specifications. Capturing the requirements of a system in a precise system specification is crucial to a successful development. However, current approaches employed in practice (e.g., UML [3] or DOORS\(^1\)) lack the ability to structure requirements in a way that they provide a modular system specification. Additionally, they have no formal semantics, which makes it impossible to detect inconsistencies or reveal unwanted behavior automatically. This does not only result in requirement descriptions that do not precisely cover the requirements but also hinders the development of an appropriate solution. As a consequence of imprecise and ambiguous descriptions of the requirements, system specifications are often vague, inconsistent and incomplete.

Component-orientation instead of function-orientation. Most of the established modeling techniques for structuring software systems (e.g., UML Component Diagrams [3], Math Works Simulink\(^2\), ASCET-SE\(^3\)) are component-based and

\(^1\) http://www.ibm.com/software/awdtools/doors/
\(^2\) http://www.mathworks.com/products/simulink/
\(^3\) http://www.etas.com/de/products/ascet_se_software_engineering.php
focus on the inner structure of a system. Thus, they are not capable of abstracting from the solution domain (the component architecture) to model user functions observable at the system boundary (the problem domain). Typically, user functions are realized by a number of interacting components. For requirements analyses, however, it is important to focus on functions rather than on interacting components, since components represent the inner structure of a system and not the functions that the system should provide.

**Complexity of feature interaction.** Behavior often arises from functions interfering with each other, both intentionally and unintentionally. This fact has been carefully studied in the telecommunication domain where dependencies between functions are called feature interaction [22]. An example for intentional feature interaction is the telephone function of a mobile phone interrupting the messaging function due to incoming calls. An example for unintentional feature interaction is the air conditioning system of a parking car requiring a higher engine speed, whereas increasing the engine speed may release the electronic hand brake to enable a comfortable driveaway. Thus, the air conditioning system influences the electronic hand brake. This dependency is obscure and may lead to unwanted behavior. Especially, unintended feature interaction is a major cause of system failures [2,11], but nevertheless is disregarded by existing approaches (e.g., [13,16,23]).

**Missing integration of modeling techniques.** Even if different modeling and specification techniques are used to develop different aspects of a system, they often cannot be related to another due to a missing semantical foundation [7]. This prevents proper reuse and comprehensive analyses of models (cf. [4]).

## 2 Proposed Solution

We propose a method that models the functional requirements of a system in a hierarchical model. The basic building blocks of this model are services. A service is a functional feature in terms of “a distinguishing characteristic of a software item” [15]. A service encapsulates a set of functional system requirements. We describe a service by a syntactic interface that consists of input and output channels. We further assign a behavior to that interface, which defines the reaction of the service in terms of output values for given input values. This notion is in line with [5], which gives a formal description of services, syntactic interfaces, and interface behaviors.

An example of a service that we modeled for an avionic case study is depicted in Fig. 1. The behavior of the service is given by a table specification that maps values of the input channel to values of the output channel.

This formal notion of a service allows us to retrieve functional specifications of entire systems by a parallel composition of services. We can visualize this service composition by a hierarchical model called a service hierarchy. The behavior of each service in this model is a projection of the behavior of the entire system.
Fig. 1. The service alertPilot models two functional requirements: If the system detects an alerting situation, the system should warn the pilot and If no alerting situation is present, the pilot should not be warned. The syntactic interface of the service consists of an input channel Radar that transmits information about the airplane’s vicinity and an output channel Warning that controls a warning sign in the cockpit. The tabular specification defines the service behavior.

represented by the root node. That means, services slice the functionality of a system in different chunks. The behavior of a service is determined either by the parallel composition of its sub-services or by a behavior specification. There are several ways to provide a behavior specification, both executable (e.g., table specifications, state machines) or descriptive (e.g., contracts). Behavior specifications are only given for atomic services that are not further decomposed and thus are the leaf nodes in the model.

Figure 2 shows an example of a service hierarchy for the avionic situation awareness system that we modeled as a case study.

Fig. 2. Service hierarchy for a situation awareness system that monitors the airspace around an airplane with the help of a radar system and recognizes objects in the airplane’s vicinity. Solid arrows indicate the sub-service relationship and dashed arrows indicate functional dependencies, where w stands for warning mode, p stands for perspective, z stands for zoom level, and o stands for the current object to display.

Service hierarchies not only model the sub-service relationship (solid arrows) but also dependencies between services (dashed arrows). Service dependencies
indicate feature interaction, both intended and unintended. We introduce so-called mode channels to integrate these dependencies into our formal notion of a service. A mode channel is a channel that transmits values of an abstract system state (a mode). An example for a mode of a car might be an Operation mode that can take the values Off, Driving, or Standing. Modes are used in a service behavior specification to encapsulate certain context situations. To capture these situations in a behavior specification, mode channels are handled as additional input or output channels in the syntactic interface of a service. Figure 3 shows the behavior specifications of two services from the situation awareness example where the alertPilot service affects the mode W, which influences the changePerspective service.

<table>
<thead>
<tr>
<th>alertPilot</th>
<th></th>
<th>changePerspective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radar</td>
<td>W</td>
<td>Warning</td>
</tr>
<tr>
<td>alert</td>
<td>true</td>
<td>On</td>
</tr>
<tr>
<td>¬alert</td>
<td>false</td>
<td>Off</td>
</tr>
<tr>
<td>PButton</td>
<td>W</td>
<td>P</td>
</tr>
<tr>
<td>?</td>
<td>true</td>
<td>Object</td>
</tr>
<tr>
<td>2D</td>
<td>false</td>
<td>2D</td>
</tr>
<tr>
<td>3D</td>
<td>false</td>
<td>3D</td>
</tr>
<tr>
<td>Object</td>
<td>false</td>
<td>Object</td>
</tr>
</tbody>
</table>

Fig. 3. Feature interaction is expressed by a mode channel between two services. The service alertPilot changes the mode W (Warning), whereas the service changePerspective uses the values of the mode as input. This models the functional requirement: If an alert is detected, the display shall switch to the object perspective, no matter which perspective button is pressed by the user.

This concept of abstract system modes allows a separate specification of services that are refined by modes to capture functional dependencies. This results in a modular way of specifying service behavior.

3 Expected Contributions

We believe that modeling the functional requirements by means of the introduced model enhances the development of complex systems for several reasons:

Modular requirements management. Functional requirements are structured according to services. This does not only increase clarity and understandability of the requirements but also allows specifying system functions more or less independent from each other. The composition of services—and thus the composition of the functional requirements—is formally defined and enables assertions about the composed behavior.
Explicit feature interaction. Functionality that results from the interplay of different features can explicitly be modeled and detected by modes transmitted over mode channels. These dependencies between services do not only serve as visual dependency links but capture the impact on the composed behavior entirely. This allows to check assertions about the system behavior including behavior that arises from feature interaction.

Unambiguous behavior specifications. The formalization of natural language requirements results in precise and unambiguous behavior specifications. This entails benefits for testing and simulating a system. The introduced model of the functional requirements also serves as a test model, which is authoritative for the design and implementation of the system. If the behavior is modeled by executable specifications like state machines or tabular specifications, the functional model represents a functional prototype of the system.

Early detection of inconsistent requirements. A formal model of the requirements is a necessity to detect non-trivial inconsistencies between them. The introduced model especially detects inconsistent requirements that originate from the composition of requirements and thus have no obvious correlation. Depending on the specification technique used to describe the service behavior, this can be achieved for example by model checking or SMT solving.

Each of the mentioned benefits is received already in a very early phase of development. A model of the requirements can be derived without any knowledge of the design or implementation of the system.

4 Evaluation and Validation

The evaluation of our approach will be mainly driven by industrial case studies. In order to cope with large case studies and to benefit from automated analyses that are enabled by the approach we will implement it in the academic CASE-tool AutoFocus3\(^4\), which already provides an environment for the model-based development of systems.

We will evaluate the advantages and disadvantages along the lines of the Goal Question Metric approach (GQM) [1] by identifying the overall goals of our methodology, deriving questions to characterize the abstract goals, and finally giving metrics on how to assess the degree of achievement. The goals reflect the mentioned expected contributions, that are: modular requirements management, explicit feature interaction, unambiguous behavior specifications, and early detection of inconsistent requirements. In the following we give a list of research questions and case study setups that we want to use to evaluate our approach:

RQ1: Does the approach detect inconsistent requirements? Setup: Given a document of system requirements, we formalize and structure these requirements according to our approach. We will use different techniques (e.g.,

\(^4\) http://af3.fortiss.org/
model checking or SMT solving) to analyze the model. If inconsistencies are detected, we will discuss them with the developers.

**RQ3:** Does the approach detect unintended feature interaction? **Setup:** Given a model of the requirements, we will confront the developers with the feature interactions that result from mode dependencies within the model. To what extent were the developers aware of these dependencies?

**RQ4:** How much effort does it take to model the functional requirements with our approach? **Setup:** Given a document of system requirements, we will measure how much effort it takes to come up with a model of the requirements. To what extent is the effort related to the number of requirements?

We will compare our method with existing approaches from academia and industry by means of these questions.

5 Related Work

The term *feature interaction* itself and service-oriented development in general is very common in the telecommunication domain [9]. For example, Jackson and Zave [16] introduce *Distributed Feature Composition (DFC)* as a modular, service-oriented architecture for applications in the telecommunication domain. DFC relies on the notion that a user service request can be composed of a set of smaller features, which are arranged in a *pipes-and-filters* architectural style. Feature interaction is an integral part of this architecture since it describes control and data flow between features [24]. In contrast, our approach aims at covering only services that are directly observable by the environment and thus completely abstract away realization details. These services are provided concurrently by the system and feature interaction is only used to steer the behavior of a service.

Our notion of a service can roughly be related to the notion of a *Use Case* in UML [3] or SysML [21]. Use Case Diagrams summarize and relate Use Cases to another, thus describing the family of functions of a multifunctional system. In contrast to Use Case Diagrams, our approach is more flexible and expressive, since Use Case Diagrams do not support hierarchical structuring, and relations can only describe *include* or *extend* relations between Use Cases. Additionally, behavior specifications for Use Cases in UML and SysML are rather informal.

The *KAOS approach* [10, 18] introduces a hierarchical model of system goals and requirements. According to that approach, abstract goals, which a system shall fulfill, are decomposed to more concrete goals and finally to system requirements. Goals and requirements can be enriched with constraints that allow detecting inconsistent or conflicting goals and requirements. In contrast to our approach, which focuses on a formal model of the functional requirements, the KAOS approach especially facilitates the acquisition of system goals and requirements.

The concept of features is also widely used in domain analysis and product line management. Kang et al. [17] introduce *Feature-Oriented Domain Analysis (FODA)* as a method to identify functional commonalities and differences of
applications in a domain. Functional features are organized in a hierarchical model that distinguishes between optional, mandatory, and alternative features and allows annotating features with a set of dependencies such as excludes or requires. In contrast to FODA, our approach does not focus on variability aspects or product families but rather on a precise model of features and especially their dependencies. Therefore, our approach provides a more general notion of dependencies (feature interaction).

There are already some approaches that structure functional requirements by hierarchical models. Rittmann [19] started structuring functional requirements by services arranged in a service hierarchy. She characterizes service dependencies by a set of activation/deactivation mechanisms. We extend this approach by a more flexible concept of dependencies represented by modes. Harhurin [14], in contrast, is more concerned about conflicting services. He resolves these conflicts by additional prioritization services. We model these conflicts by additional mode dependencies, which resolve the conflicts depending on the current mode. Schätz [20] uses two operators for composing functions: conjunctive (parallel) and disjunctive (alternative) composition. Furthermore he distinguishes between control flow and data flow between functions. Our approach captures all aspects by parallel composition and data flow.

6 Preliminary Work and Current Status

Our approach is based on FOCUS [8], a framework for the formal specification and stepwise development of distributed systems. Broy et al. [6] adapted this framework to services and service composition [5]. Based on this formal framework we derived a method to specify services independently by means of lightweight table specifications and applied it to two case studies from the avionic and the automation domain [12]. Both case studies show that it is possible to model the functional requirements with the introduced methodology. However, the case studies also show that the methodology as well as the formal framework still demands further extensions and improvement. As next steps we will perform further case studies to evaluate our research questions as well as to improve the approach. Further research will address the relation between services and components of the system architecture to provide a tracing of requirements in the system architecture.

7 Acknowledgement

The author wants to thank Manfred Broy, Martin Feilkas, Bernhard Schätz, Benedikt Hauptmann, Maximilian Junker, and Jakob Mund for inspiring discussions and helpful comments on the paper.

References