Enabling semantic interoperation of dynamic plug-in services

Christoph Ainhauser, Reinhard Stolle, Jürgen Steurer

BMW Car IT
Petuelring 116
80809 München
{christoph.ainhauser,reinhard.stolle,juergen.steurer}@bmw-carit.de

Abstract: Dynamic HMI generation based on declarative descriptions solves the problem of dynamically integrating services into the HMI that are not known at design/deploy time of the HMI. [Hi07] describes this approach for individual services. In this paper, we show how we have extended this approach to enable HMI generation for compositions of services and their semantic interoperation.

1 Introduction

Mobile consumer electronics (CE) devices are continuously and quickly becoming more pervasive in everyday life. The range of functionality – telephony, music, navigation, web browsing, etc. – offered by such devices is growing rapidly, and the device owners expect to be able to access this functionality in virtually any situation. In this paper, we focus on the use of CE functionality in the situation of driving a car.

For safety reasons, automotive human machine interaction (HMI) systems are tailored toward the typical automotive use cases in order to minimize driver distraction. A promising way of allowing drivers to safely operate CE devices is to enable access to the CE device functionality through the automotive HMI. For example, an MP3 player's play-lists may be visualized on the car's central information display, the playback functions may be controlled via the BMW iDrive controller or the steering wheel buttons, and the audio output may be played over the car's stereo speakers.

The CE industry's innovation cycles are extremely short compared to a car's life cycle. This observation raises two problems concerning the integration of CE functionality into the automotive HMI [Hi07]. First, it is typically too time-consuming to manually implement and test the integration: by the time the integration is ready to be deployed, the CE device may be outdated. Second, customers want to operate new CE functionality in vehicles that were built before the CE device or its functionality existed. As a consequence, a proper solution must enable fast development of CE-HMI integrations without a need to update (and thus verify) the car's software. One solution to this problem, namely dynamic HMI generation for dynamic services based on declarative descriptions of service functionality and abstract interaction logic, was first presented in [Hi07]; it differs from traditional approaches by required key features like a) separation of HMI from application and b) common look and feel of dynamic plug-in services.

Given the ability to integrate services dynamically, several issues must be solved [Ai07]. The two main issues are resource management (priorities and permissions of services to access common resources) and service interoperation (exchange of data). This paper gives a solution to the interoperation problem. Typically, much of the added value of functionality comes from their composition. For example, a contact book on one device should be able to be linked with a navigation service on another device.[1] Manual implementation and verification will do the job for cross use cases with services that are known at development time of the HMI. In order to facilitate interoperation of new dynamic services, however, we have extended the HMI generation mechanism, which is defined in [Hi07] and summarized in the following section. The extension for interoperability is the topic of Section 3 of this paper.

2 HMI generation for plug-in services

Modern HMI systems are typically architected in layers, supporting a model-based development process [PS00]. The layered architecture of the HMI system that we are concerned with in this paper is shown in Figure 1.

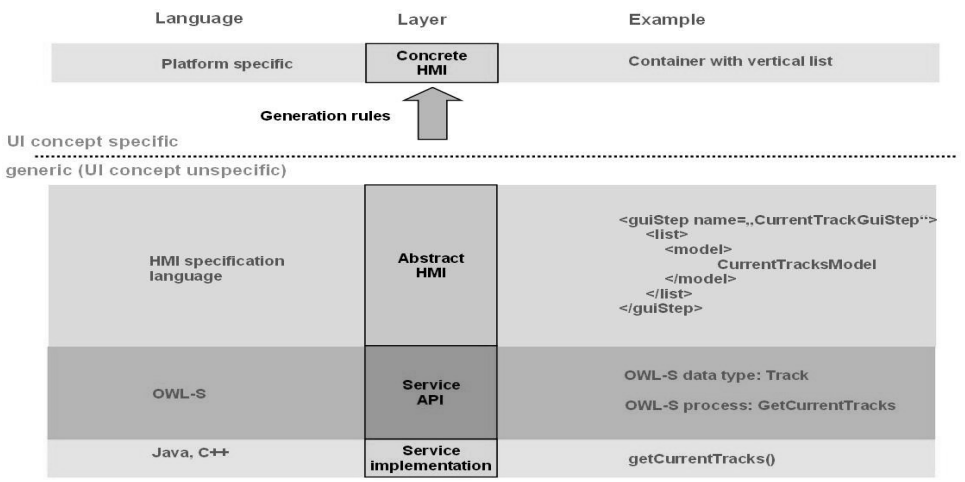

Figure 1: The layered HMI architecture. The bottom three layers describe the service independently of any particular UI. The HMI generator generates a concrete HMI following a particular UI concept ("look and feel") from the descriptions in the bottom three layers.

The four layers of the architecture are described in the following. The basic idea of dynamic HMI generation, which we summarize in this section, is to generate the top layer, namely the concrete HMI of a service, from the descriptions of the service's functionality and its abstract HMI. All knowledge about the specific user interface (UI) characteristics ("look and feel") is encoded in the generator's generation rules.

[1] Note that numerous solutions (standards and de-facto standards) are available to enable the functional interoperation of services. In this paper we are solely concerned with integrating and mediating the functional operation in the automotive HMI – and doing so for services that are unknown at deploy time of the HMI.

2.1 Service Implementation

The service implementation realizes the concrete service and provides several functions to the upper layers. The function getCurrentTracks() of the MP3 player is an example of such a service function.

2.2 Declarative description of the service functionality

The declarative description of the service's API hides the implementation details, such as the programming language, from the upper layers. We use OWL-S [OWL-S] to model the service API semantically. Each service function is expressed as a process with certain inputs and outputs that are in turn described semantically. The function getCurrentTracks(), for example, returns as result an array of objects of the type "Track." The main difference of this approach from a simple API description is that the data types are related to each other via a formal ontology [OWL].

2.3 HMI description

The HMI description comprises three parts. First, *models and actions* encapsulate the service's data structures that are maintained in the HMI and the possible actions (service function calls or HMI state changes). Second, *abstract HMI components* are typical HMI constituents, but without any concrete layout information. Examples are Lists, Activators ("buttons"), Input components and Labels. Abstract HMI components establish the link between user operations and actions (e.g., a button referring to an action) and between models and display components (e.g., a list of type "AbstractList" referring to the model "CurrentTracksModel" in order to fill the list with content). Third, *HMI states* describe sets of HMI components that are visible to the user in certain situations. A tree of HMI states defines how the user can navigate within the HMI.

2.4 Generated HMI

Given a particular UI concept, a rule-based generator now generates a concrete HMI from the declarative descriptions that accompany the service. In a first step, *special rules* can be used to control the exact layout of known interaction components, influencing the look and feel of the HMI by the UI designer. For abstract elements without special rule specification, *ontology rules* make use of semantic knowledge about the HMI components and their associated models and actions. For example, if an abstract Activator is linked to an action that is described as a "music play" action, the generator will map the activator to a concrete button with an appropriate "play" icon. Rules that use more-concrete data types in the ontology override less-concrete ones. Finally, *default rules* ensure that all abstract components are being mapped to concrete UI elements. This approach has proved feasible and useful in close-to-production projects.

3 Semantic interoperation of plug-in services

In order to facilitate the exchange of data between services, a system must know the kind of data that each service can produce or consume, and the states in which they can do so. In our approach, services describe this information declaratively in order to enable the HMI generator to allow the user to use data provided by one service in the context of another service. For example, in the state "destination input" of a navigation service, the HMI can allow the user to grab the destination address from an address book service.

Based on this information, service interoperation can be either semi-automatic (PULL, PUSH) or fully-automatic (see Figure 2). In the following, the two kinds of interoperation will be discussed in more detail.

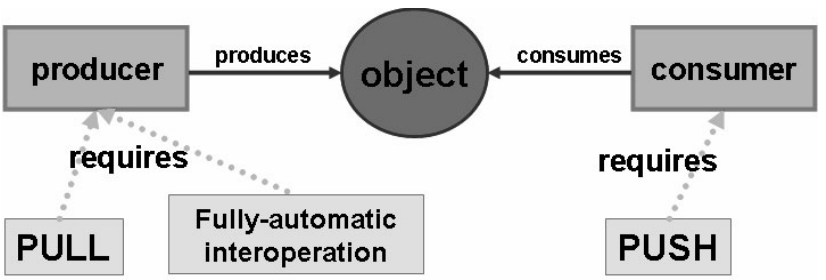

Figure 2: Scenarios for service interoperation

3.1 Semi-automatic interoperation

There are two kinds of interoperation that can be triggered by the user (e.g., by pressing the "Option" button). In a **"***PUSH***"** operation, the user processes the focused object within another service (e.g., while browsing a storage device, the user uses the selected MP3 file within the service "MP3 player"). In a **"***PULL***"** operation, the user imports data from another service to proceed (e.g., in the MP3 player, the user imports an MP3 file from a storage device).

In both cases, the system examines the current HMI state and identifies the *objects of interest*. In the PUSH case, the currently focused HMI component is analyzed: for example, lists or labels contain potential exchange objects. In the PULL case, the consideration of the HMI components is not enough, because the HMI state may also hint at the required objects. For instance, the state "start menu" of the phone service specifies that a contact object can be imported to provide a phone number.

Now the system has enough information provided by the services in order to recognize, what kind of object is either provided (in the PUSH case) or required (PULL case). After the relevant object types are detected, the signatures of the service APIs determine the services that can consume (in case of PUSH) or produce (in case of PULL) them.

The common inheritance hierarchy (ontology) of object types allows the HMI generator to resort to fallback reasoning if a service refers to new, unknown object types.

In addition to matching the producing and consuming services' object types we need to "align" the control flows of the producer and the consumer. To this end, every consumer service specifies an *action* for every object it can consume. When an object gets PUSHed to a consumer, the HMI state change requested by the associated action is going to determine the focus after the interoperation event. For PULL interoperations, we extended the service description by a dialog component type "Output Component". The new dialog component was required because there is more information required then an action can hold. First, producers for the PULL concept need to specify the HMI state that is focused for the interoperation process. Also, they specify where the exchange object is located after the interoperation. Here is an example of an output component:

```
<outputcomponent>
  <name>test</name>
  <ispublic>yes</ispublic>
  <contextentry>Addressbook</contextentry>
  <results>
     <string>Contacts#object</string>
  </results>
  <hmiaction>
     <jumpToGuiStep>OutputConctactAddress</jumpToGuiStep>
  </hmiaction>
</outputcomponent>
```

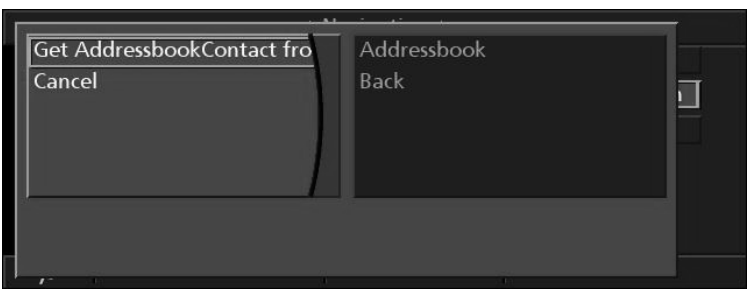

Figure 3: A (single) choice of producers in a PULL interoperation. The user is located in the navigation service and the system has recognized that a contact address is required.

Based on this information, the system offers the user a choice of producers that can provide the data requested by the PULL operation (see Figure 2). The following table shows the model elements that provide the required information in the respective cases.

Service Interoperation type	**Producer**	**Consumer**
PUSH	Focused HMI component	Action
PULL	Output component	Focused HMI component Focused HMI state

3.2 Fully-automatic interoperation

Fully-automatic interoperation happens without the user's involvement. Based on a service's description of its data needs (e.g., the current fuel level), the system automatically finds a service that offers the required information. Fully-automatic interoperation could be realized at various levels in the service architecture (e.g., exchange of models, dialog components). We chose the application layer as reasonable trade-off between semantic descriptions needed for an appropriate interoperation and generality of the concept. In our implementation, a service describes the required actions (and/or models). During the load process of the service into the HMI system, the system determines the missing objects and tries to find an appropriate producer. The remote model or action is inserted into the HMI of the consumer. If no producer is found, the required element is left blank; if multiple ones are found, one is chosen arbitrarily.

4 Results and outlook

We have implemented HMI generation for individual services [Hi07] at close-to-production level. The semantic interoperation mechanisms described in this paper have been implemented as a prototype [Ai07]. Our experience with the prototype indicates that the approach works well and is useful for many common infotainment use cases. The need for fully-automatic interoperation seems to be limited.

Our original aim of HMI generation was to enable HMI integration of third-party functionality with no or limited OEM involvement. We now consider the approach useful also for maintaining modularity when integrating OEM services or suppliers' services. However, for this case the developer needs more freedom to affect the HMI appearance of services. We are currently evaluating possible extensions to our approach that would let services bring along their own look and feel. This may result in an "HMI service integration development kit" with various extensions depending on the kind of changes that the developer is allowed to make to the HMI.

Bibliography

[Ai07] Ainhauser, C.: HMI-Generierung für dynamische Dienste und Anwendung in einem verteilten System. Master's thesis, Technical University of Munich, March 2007.

[Hi07] Hildisch, A. et al.: HMI generation for plug-in services from semantic descriptions. *Proceedings International Conference on Software Engineering (ICSE-2007), Workshop on Software Engineering for Automotive Systems*, Minneapolis, MN, May 2007.

[OWL] Web Ontology Working Group: Ontology Web Language, www.w3.org/TR/owl-ref/.

[OWL-S] OWL-S Web Ontology Working Group: Ontology Web Language – Services (OWL-S), www.daml.org/services/.

[PS00] Pinheiro da Silva, P.: User Interface Declarative Models and Development Environments: A Survey. LNCS Vol. 1946, pages 207-226, Springer, 2000, www.daml.org/services/.