# Improving the Flexibility of In-Vehicle Infotainment Systems by the Smart Management of GUI-Application Binding Related Information

Ran Zhang and Tobias Altmüller

Robert Bosch GmbH, Daimlerstrasse 6, 71229, Leonberg, Germany
{Ran.Zhang,Tobias.Altmuller}@de.bosch.com

**Abstract.** This paper introduces an approach to build a new system application addressing the smart management of binding related information for in-vehicle infotainment systems. The system application is based on a client-server model using Web technologies and provides message oriented middleware to drive bidirectional GUI-Application communication. Additionally, it also allows GUI-Application binding at runtime and supports the same GUI to be bound with the applications located on different devices. The result shows that this approach improved the reusability and the adaptability of binding related information, and also increased the flexibility and the scalability of IVI systems.

**Keywords:** in-vehicle infotainment, GUI-Application binding, SOA, runtime binding, WebSocket, middleware.

## 1 Introduction

In comparison to foretime, today‘s car is not only the means of transportation, but is also integrated with interactive computational environment allowing an internet connection and multifarious applications e.g. Browser and social community. This interactive computational environment, also called in-vehicle infotainment (IVI) system [1], is an integration of in-car information system and entertainment system. The in-car information system consists of the applications which are responsible for the exchange of information between the user and the vehicle as well as the traffic environment, e.g. a navigation program. The entertainment system provides the user entertainment related functionalities, such as radio or media player.

The graphical interactive components of IVI systems can be described abstractly with three layers: graphical user interface (GUI) layer, application layer and additional middleware layer binding the GUI and application components. Figure 1 shows a classic IVI system based on the above described layers and the event-driven communication among them. The GUI layer can be specified by the presentation elements which define the visible information (e.g., layout, color, position or text) as well as the control logic elements driving the screen transition. The application layer can be described by its business logic and functional units (e.g. C++ functions and Java methods) which implement the functional capabilities of applications. In the development process of today‘s automotive software products [2], it is common that the GUI and the application components of IVI systems are developed separately and then bound together at development stage using a middleware layer which is also called binding layer. Currently, the binding layer is based on a data pool, an event observer and an event handler. The data pool is implemented as a list of variables which can be changed by the events initialized by the GUI and the application. The event observer monitors the data pool and catches the change events of the data pool. If a change event of the data pool is caught by the event observer, the event handler will drive the update of the GUI. In the development of automotive software products, HMI specifications and applications of IVI systems are individual in each target product. Therefore, the binding related information, which is exchanged between the GUI and the application, is dependent on binary data whose name and value are product- and manufacturer-specific. Therefore, the binding related implementation of previous development cannot be directly used for present and future development. The major part of the middleware needs to be re-implemented for each new project and this reimplementation leads to additional development cost, as well.

The reason for this low flexibility is the lack of a mechanism which can automatically adapt binding related information in the middleware layer each time there are updated requirements. There is still no generic solution for how to increase system flexibility by improving the reusability and adaptability of such information. To face the above mentioned issue, this paper introduces an approach to build a new system application addressing smart management of binding related information for IVI systems. The objective of the system application is to support the runtime GUI-Application binding and also binding a common GUI with applications of different implementations, platforms and devices. If these objectives can be reached, the flexibility of GUI-Application binding for IVI systems will be improved, and the development time and costs of IVI systems will be reduced.

## 2 Related Work

In this section, we introduce several approaches which are related to GUI-Application binding and were already used in the automotive domain.

The EB Guide [3] is a development environment for the model-driven HMI (human-machine-interface) development of IVI systems. Within the EB Guide, the events are used to drive the transition of views linked to state charts. The data-binding is realized with the aid of a data pool implemented as properties of widgets. The property value of a widget can be changed by events initialized by the GUI and the application.

Another well-known development tool for modeling and generating the HMI of IVI systems is CGI Studio [4]. Courier is the interaction framework used

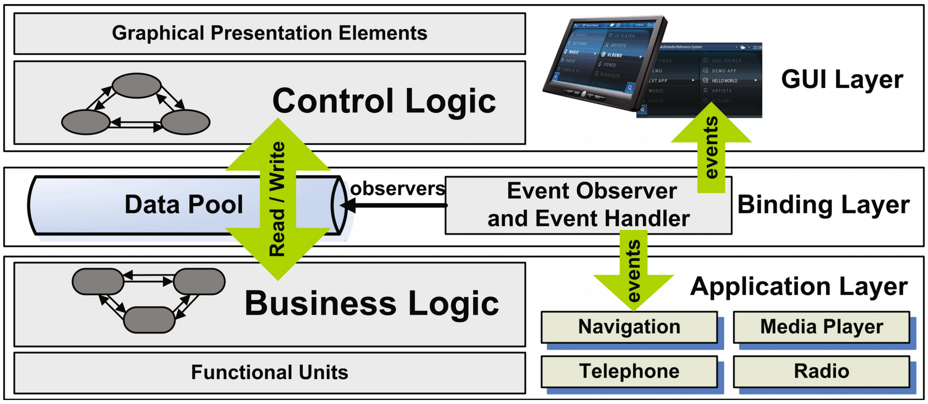

**Fig. 1.** GUI-Application binding of a classic IVI system

within CGI Studio and focuses on message handling and data binding with the application. This framework implements Model-View-Control architecture pattern.

To formally define the API for IVI systems, GENIVI Alliance[1] published a framework called Franca [5]. The core of this framework is Franca IDL, which is a formally-defined and textual interface description language (IDL). In Franca, an API can be declared by three basic elements: attributes, methods and broadcasts. The dynamic behavior of interfaces is described by using contracts, which are based on protocol state machines.

A flexible in-vehicle HMI architecture based on Web technologies was developed in [6]. This approach provided functions being handled as a Web service. The HMI was implemented using HTML and JavaScript and rendered in the browser. This approach supported the integration of external devices in the form of plug-in services in IVI systems and generated the HMI for this service at runtime.

Addressing the semantic description of binding related information, [7] introduced an approach to generate the HMI for plug-in services from semantic descriptions. This approach was based on four layers: service implementation, service functionalities and API description, abstract HMI description and concrete HMI with generation rules. Compared with other solutions, this approach used ontology for the reuse of knowledge and provided higher machine readable description of binding related information.

## 3 Methodology for Identifying Solutions

Addressing the objectives defined in Section 1, we deduced those requirements which could improve the flexibility of GUI-Application binding and fulfill the

[1] `http://www.genivi.org/`

domain-specific requirements for developing IVI systems. After this, we purposed several solutions which fulfilled each requirement. Then, we identified the final solutions for each requirement regarding the dependency and conflicts among these solutions.

## 3.1 Requirements Elicitation

Figure 2 shows the technical requirements for building the new system application. The requirements are based on two top aspects: the flexibility aspect and the quality aspect, which are outlined as the roots of two trees. Its corresponding leaves are numbered from F1-7 and Q1-3.

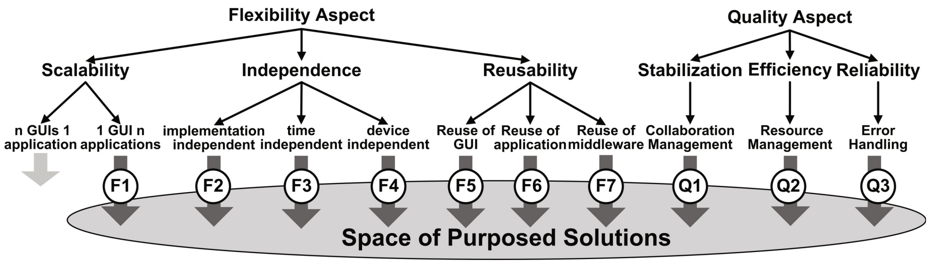

**Fig. 2.** Requirements elicitation

The flexibility aspect addresses the requirements for improving the scalability, independence and reusability for GUI-Application binding. In this context, scalability means that two types of extensions should be supported in an IVI system. The first extension allows generating GUIs from different design concepts for an application. In another extension, a GUI can be bound with multiple applications, which are located in different platforms or devices or are implemented differently. The requirement of independence can be described in three sub requirements. A full flexible GUI-Application binding should be independent of implementation (e.g., platform or programming language, the time when the binding happens, and device, on which the target GUI or application is located). Another important requirement of the flexibility aspects is the reusability which can be refined by the reuse of GUI, application and middleware.

Besides the flexibility aspect, how to keep continual stabilization, efficiency and reliability of IVI systems by improving its flexibility is also a challenge. Compared with mobile devices, the instable system behavior and the failures in the application and the GUI of IVI systems lead not only to low user experience but also to the user being distracted from driving, and can even cause accidents. For this purpose, a collaboration management which enables the stable collaboration among the components, especially by using a loosely coupled system architecture, is needed. In this paper, we concentrated only the requirement on efficiency under the same condition of hardware. The efficiency requirement of

IVI systems can be concreted as resource management e.g. thread assignment. Besides stabilization and efficiency, error handling is also required for a reliable system behavior.

## 3.2 Identifying Solutions

In our work, we have focused on binding multiple applications with the same GUI. Therefore, the requirement on generating multiple GUIs for a single application will not be considered in following work.

**Solutions to Requirement F1, F2, F3, F5 and F6.** The pre-condition for binding multiple applications to a common GUI (F1) as well as the reuse of GUI and applications (F5 and F6) is that these applications have the same abstraction but are implemented differently. This means that the APIs used to invoke these applications should be in a standard form. For independent implementation (F2), two of the most popular technologies service-oriented architecture (SOA) and object-oriented architecture (OOA) could be applied. A well-known example of OOA for middleware is CORBA [8], which is based mainly on a request-response mechanism. However, IVI systems prefer an asynchronous function invocation, which has more performance benefits in comparison to a synchronous function invocation. Compared with OOA, SOA has the benefit not only in implementation independence and loose coupling, but also in efficiency of function invocation with the aid of additional support for the asynchronous function invocation. For this reason, we have chosen the SOA to realize the independency of implementation. In order to enable the GUI-Application binding during the development time, and also at runtime (F3), the new system application must provide a mechanism supporting the GUI to be bound with a reference instead of binary data. The benefit of using a reference is that the binding can be executed after functions are being called. This also provides the possibility to swap the sources of applications under the same GUI at runtime. Based on the above described requirements, we chose Web service which allows application abstractions, supports the SOA and provides reference for function invocation at runtime.

**Solutions to Requirement F4.** For device independence, there are two possible technologies which could be applied in our work. The first possibility was the approach based on the server-client model. The idea behind this approach is to develop a structure supporting the GUI layer, the binding layer and the application layer located in different devices. Another possibility was to use MirrorLink which is also called Terminal Mode [9]. This solution supports mapping the GUI from mobile phones into IVI systems but does not provide any benefits for the other requirements. Therefore, we chose the server-client model with extension of Web technologies.

**Solutions to Requirement F7.** In order to improve the reusability of the middleware, it was anticipated that the binding layer can self adapt on new changes in the GUI layer and the application layers e.g. adding a new button or changing the source of application. Bidirectional communication, which is driven not only by the GUI, but also by the application, should be allowed to avoid the high manual reimplementation for new development requirements. Combined with F4, we chose WebSocket to realize the bidirectional communication.

**Solutions to Requirement Q1.** Collaboration management is an indispensable requirement for loose coupled systems. There should be a definition on how these independent components communicate with each other. The possibility was using either a central message broker or standalone solution, which requires an additional message mechanism for every component. This leads to high implementation and maintenance cost. For this reason, the central message broker was more suitable for our solution.

**Solutions to Requirement Q2.** To improve the resource management of a server-client model, which bases principally on a synchronous function invocation, we would extend the server-client model with additional asynchronous function invocation such as callback functionality in our solution.

**Solutions to Requirement Q3.** There existed two possibilities of the conception of error handling: developing a separate component for error handling or integrating error handling in the participants of message communication. For the purpose of reducing system complexity, we preferred to integrate error handling in the components which initialize message communication.

## 4 System Design

Based on the identified solutions, we used a top-down method to design the system application. Figure 3 shows the system architecture at runtime.

First, we defined a layer model based on the GUI layer, the binding layer and the application layer. Based on the server-client model, the GUI layer was refined by the GUI client and the GUI server. The implementation of the GUI is located in a GUI server, loaded and rendered in a GUI client at runtime. The binding layer and the application layer respectively contain binding server and application server(s). The application server is a computational environment of a device, on which the applications and the Web services are located and invoked. Between the GUI client and the application server, the binding server processes the request from the client and invokes the Web service of the application server. The client and servers can be located in different devices or in the same device, e.g., the GUI client and server, as well as the binding server can be a part of IVI systems.

Second, we refined the components for each layer. The GUI was described by the control logic elements, the presentation elements and the GUI message handler. The control logic elements define the control logic e.g. the menu flow of

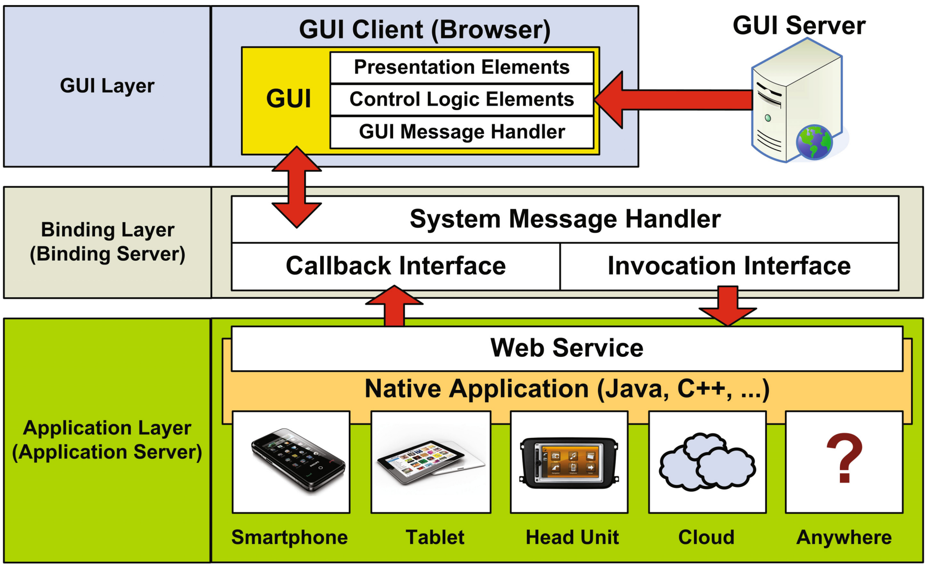

**Fig. 3.** System architecture of the developed system application

the GUI according to the user behavior. The presentation elements represent the presentation information e.g. color or position of widgets. Additionally, a GUI message handler was developed to handle the interaction-related message.

On the server side, we have also defined a message handler called server message handler. These two message handlers are the central components for the communication between GUI and application. They manage the following binding related information: the initialization object of communication, the content of communication and the target object of communication. This information was described in the form of a message with sender, content and receiver. Additionally, two types of interface were defined: invocation interface and callback interface. The application server provides Web service which abstracts the functional capabilities of the local applications. It can be located in a Smartphone, a Tablet, a Head Unit of car, in Cloud or anywhere, if the connection with the binding server is established.

In addition, we have defined three participants of message communication in GUI-Application binding: GUI, Web service and system service. The system service represents the basic functions of the operation system (OS) located on the binding server. Based on these three participants, we categorized the binding related messages in nine types and defined the routers for handling different message types (see Table 1). Initially, the messages which are initialized by user interaction via the GUI, are processed in the GUI message handler. If these messages are related to invoke a function of application or OS function, it will be forwarded to the system message handler. The system message handler has a mapping table, which helps to match the GUI events to the corresponding functions. Otherwise, the GUI message handler will directly update the target GUI element. The messages, which are initialized by a Web service or a system service and related to update the GUI, will be handled in the system message handler and then forwarded to the GUI message handler, which is responsible for updating the GUI. As our work focuses mainly on the message communication between GUI and application, the message communication between the application and the OS, as well as the intercommunication of the OS were not regarded in this paper.

**Table 1.** The routers for handling different message types

| **Requester** | **Responder** | **Router for message handling** |
|---|---|---|
| GUI | GUI | GUI message handler |
| GUI | Web service | GUI message handler → system message handler |
| GUI | system service | GUI message handler → system message handler |
| Web service | GUI | system message handler → GUI message handler |
| Web service | Web service | - |
| Web service | system service | - |
| system service | GUI | system message handler → GUI message handler |
| system service | Web service | - |
| system service | system service | - |

In this paper, three types of message casting were defined. In unicast, there is only one receiver, which can be a component of the GUI, the Web service and the system service. Multicast allows at least two receivers, but not all available components. In broadcast, all available components of the GUI, the Web service and the system service can receive the sent message.

Figure 4 shows the work flow of the GUI message handler. In the left figure, a GUI event was initialized by user interaction in the GUI logic control block. If this event is relevant for invoking a function, a message object based on the implementation language will be created and sent to the binding server. If this event is also related to an update of the GUI, the GUI message handler will send the message to the related GUI component(s) and perform decomposition, if it is necessary. On the other figure, the GUI message handler has received a message object from the system message handler located in the binding server. In this case, the GUI message handler is responsible for decomposing multicast or broadcast messages and for updating the target GUI component(s).

## 5 Prototype

Based on the developed system architecture (see Figure 3), we implemented a prototype in a Java platform. The presentation elements (style and layout) of the GUI were implemented using CSS and HTML5. The control logic elements

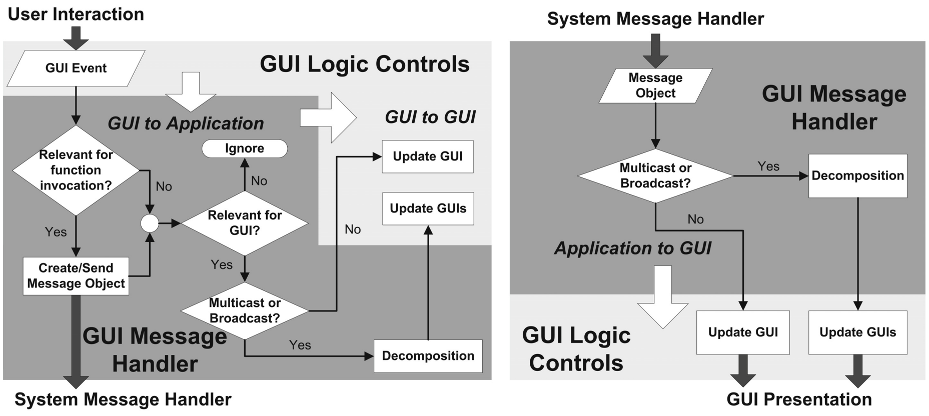

**Fig. 4.** Workflow of the GUI message handler

and the GUI message Handler were realized using JavaScript. The system message handler was implemented in servlet and used WebSocket to connect to the GUI message handler. Additionally, JSONObject was used for formalizing the messages. The callback interface and invocation interface connected with the Web service used a SOAP engine based on Axis 2.

To demonstrate the developed prototype, we implemented two media player applications using Java and C++ and named them AP1 and AP2. Additionally, we developed a simple GUI for the media player. Both applications were implemented in different algorithms but the common abstraction in the form of Web service interface was used. To evaluate our work, we defined the following criterions: (1). Is GUI-Application binding at runtime successful? (2). Is binding a common GUI with applications of different implementations, platforms and devices successful?

Based on above criterions, we have performed four tests (results are shown in Table 2). Device A and B were desktop PCs running Linux and device C was a Tablet running Windows 7. The following components were involved in testing: both applications AP1 and AP2, the binding server (BS), the GUI server (GS) as well as the GUI client (GC). We have combined these components into four groups and allocated them on the devices (see Table 2). In every test, we have successfully bound the GUI with applications at runtime and could flexibly swap

**Table 2.** Results of evaluation

| Test Nr. | Device A | Device B | Device C | 1. criterion | 2. criterion |
|---|---|---|---|---|---|
| 1 | all components | - | - | Yes | Yes |
| 2 | AP1, AP2 | BS, GS | GC | Yes | Yes |
| 3 | AP1, AP2 | BS | GS, GC | Yes | Yes |
| 4 | AP1 | BS, AP2 | GS, G C | Yes | Yes |

the sources of the applications under the GUI. The results show that our work enabled the GUI layer, the binding layer and the application layer located in separate devices. Additionally, our solution allowed not only a GUI-Application binding but also to change the source of applications at runtime.

## 6 Conclusion

In this paper, a new approach has been developed which addresses the smart management of binding related information for IVI systems. This approach was based on the solutions derived from the requirements on improving the flexibility and the requirements of automotive software products. On this basis, we have developed a system application which is able to bind the same GUI with multiple applications at run time, which were implemented in different programming languages, located in different devices and running on different platforms. The results show that our solution provided a high scalability of IVI systems at runtime. In our work, we have reached an improvement of the flexibility of GUI-Application binding for IVI systems by increasing the reusability and the adaptability of binding related information.

## References

1. Mause, D., Klaus, A., Zhang, R., Duan, D.: GUI Failure Analysis and Classification for the Development of In-Vehicle Infotainment. In: Proc. of VALID 2012, pp. 79–84 (2012)
2. Hess, S., Gross, A., Maier, M., Orfgen, M., Meixne, G.: Standardizing Model-Based IVI Development in the German Automotive Industry. In: Proc. of the 4th International Conference on Automotive User Interfaces and Interactive Vehicular Applications (2012)
3. Fleischmann, T.: Model based HMI specification in an automotive context. In: Smith, M.J., Salvendy, G. (eds.) HCII 2007, Part I. LNCS, vol. 4557, pp. 31–39. Springer, Heidelberg (2007)
4. CGI Studio factsheet, `http://www.fujitsu.com/emea/services/microelectronics/software/cgistudio/` (last visited: February 28, 2013)
5. Franca User Guide Release 0.1.3, `https://code.google.com/a/eclipselabs.org/p/franca/` (last visited: February 28, 2013)
6. Eichhorn, M., Pfannenstein, M., Steinbach, E.: A flexible in-vehicle HMI architecture based on web technologies. In: Proc. of the 2nd International Workshop on Multimodal Interfaces for Automotive Applications, pp. 9–12 (2012)
7. Hildisch, A., Steurer, J., Stolle, R.: HMI generation for plug-in services from semantic descriptions. In: Proc. of the 4th International Workshop on Software Engineering for Automotive Systems, vol. 4 (2012)
8. Birman, K.P.: Corba: The common object request broker architecture. In: Guide to Reliable Distributed Systems, pp. 249–269 (2012)
9. Bose, R., Brakensiek, J., Park, K.: Terminal Mode:Transforming Mobile Devices into Automotive Application Platforms. In: Proc. of the Second International Conference on Automotive User Interfaces and Interactive Vehicular Applications, pp. 148–155 (2010)