Adaptation of Automotive Infotainment Interfaces using Static and Dynamic Adapters

Thomas Pramsohler[1] and Uwe Baumgarten[2]
[1]BMW Forschung und Technik GmbH, München, Germany,
[2]Technische Universität München,
Lehrstuhl für Betriebssysteme,
Garching bei München, Germany,
thomas.pramsohler@bmw.de
baumgaru@in.tum.de

Abstract: Software adaptation is a promising approach to building flexible interfaces for variable software systems. In this contribution we present a novel adapter modeling approach. The key idea is to split the adapter model into two parts. The first, the static adapter model, addresses the most usual adaptations in an easy way. The second, the dynamic adapter model, may be used for complex behavioral adaptations including timing and parallelism. Additionally we show how both models are merged in order to obtain and validate the overall adapter behavior. We apply the approach to an in-vehicle infotainment system and show a practical realization of the models.

1 Introduction

Automotive software systems form a complex network of highly interconnected components. Thus, exchanging or evolving a single component will likely affect the surrounding components. Additionally, product customization by the customer combined with the diversity of product lines, requires flexible software components. There are a range of different approaches, which tackle this challenge. One is to make each component implement several interfaces, which makes the software interface unnecessarily big. A second option may be to provide different implementations of one component for each environment. Such a solution, however will increase the development and test effort.

In this contribution we address the problem using software adaptation. An adapter enables the composition of components with mismatching interfaces [MPS12]. Such an approach has a small impact on the client and server implementation by adding a mediator component in between [Wie92]. Adaptation is not completely new in the field of automotive software engineering. Current gateway electronic control units (ECUs) for instance act as mediator and can be configured for signal adaptation using parametrization at runtime. Furthermore, the AUTOSAR [AUT] runtime environment supports basic adaptation at design time.

The selection of a suitable adaptation technique depends on the target component and interface model. In this contribution we focus on the inter-process communication in a GENIVI compliant [GEN] in-vehicle infotainment (IVI) system. Usually, the IVI-system

is the most advanced ECU and allows for dynamic changes of the running software. This enables the introduction of dynamic adapter mechanisms. An adapter for infotainment components has to fulfill several requirements which arise from the domain specific properties.

In contrast to the specific properties of the electrical system, which mainly rely on cyclic real-time communication, the interaction between infotainment components implements complex protocols and data. Thus, a suitable adapter model has to provide solutions for rich behavioral adaptations.

Classical adaptation approaches, which use a finite state machine for this task, have two major disadvantages. First, even simple adaptations like method renaming will lead to additional states and transitions which makes the state machine unnecessarily big. Second, the execution of such state machines usually causes some overhead and may be time and memory-consuming, specially for bigger models and without proper code optimization. As a result of limited hardware resources, we should minimize the usage of such models.

In this contribution we present a modeling approach for interface adaptation which fits these requirements. We extend our previously defined adapter architecture [PSB13] with the usage of timing information and parallelism. We give a practical view of the adapter model implementation by extending the GENIVI interface modeling infrastructure. In our approach we split the adapter model into two parts. The static adapter model describes simple but frequent adaptations. Static adaptations are applied globally and can be implemented without using a state machine. The second part of the model, the dynamic adapter model, describes context-sensitive adaptations which depend on the current communication state. Splitting the adapter in such a way will cause additional synchronization effort between both models. For this reason we define the necessary modeling concepts for synchronization and show how the overall adapter behavior can be synthesized. The resulting model provides concepts for the adaptation of syntactical (signature) and behavioral (protocol) mismatches and is precise enough for automated adapter-code generation [PSB13]. In order to show the application of the approach we present an implementation of the model together with an automotive use case. The approach is not only applicable to automotive infotainment interfaces and may be adapted to different target domains.

The remainder of this paper is structured as follows; Section 2 presents our running example and the models used for interface description. Section 3 introduces our adapter model including static and dynamic adaptations. In section 4 we show how the two adapter models can be validated. Section 5 gives practical information on which elementary adaptation scenarios can be solved using our approach. Section 6 compares our approach to related work and section 7 will conclude the paper.

2 Interface Model and Running Example

In this section we present our interface model and two mismatching components. These components will be used as example throughout the paper in order to illustrate our approach.

Our example defines an interface which may be used to implement a parking assistant (ParkA). The ParkA helps the driver to identify obstacles around the car and is responsible

to provide a reliable link in order to obtain up-to-date values from the corresponding distance-sensors. Different clients may communicate with the ParkA in order to generate auditory or visual output for the driver.

We use the approach presented in [PSB13] for interface modeling. In this contribution we will give a more practical view of the interface model showing a concrete implementation of the models. Interfaces are modeled with three sub-models, the syntax model, the event model and the behavior model. The syntax model is used for signatures and type definitions. The event model implements our interpretation of UML [OMG09] transition triggers with corresponding signal events and guard conditions. Furthermore, the behavior model describes the communication protocol using the defined signal events.

2.1 Syntax model

The syntactical interfaces are defined using the Franca interface description language (IDL) [FRA] which is the current standard for specifying GENIVI compliant infotainment interfaces. The syntax model includes information about the interface name, the interface version, methods, broadcasts, parameters and type definitions. Figure 1 shows an example using our case study.

```
package com.bmw.demo

interface Parka {
 version { major 1 minor 0 }

 method startup { }



 method shutdown { }


 broadcast sensorValues {
  out {
   UInt8 front_left
   UInt8 front_mid
   UInt8 front_right }}


 broadcast started { }
}
```

(a) ParkA_1.0 interface

```
package com.bmw.demo

 interface Parka {
 version { major 2 minor 0 }

 method connect {
  in { Boolean retry
     Int8 retryCount }
  out { am_error result
       Int8 connectionID }}
 method disconnect {
  in { Int8 connectionID }
  out { am_error result }}
 method getSensorsAndStatus {
  out { am_error result
       UInt16 frontLeft
       UInt16 frontMidleft
       UInt16 frontMidright
       UInt16 frontRight }}
 enumeration am_error { ERROR
                        OK }}
```

(b) ParkA_2.0 interface

Figure 1: Two variants of the syntactical ParkA interface

In our scenario the ParkA component exists in two versions, ParkA_1.0 and ParkA_2.0. ParkA_1.0 provides two methods, one for the component startup and the other for component shutdown. Additionally ParkA_1.0 defines a broadcast (`started`) for signaling the ready state, and one broadcast (`sensorValues`) for sending sensor values. ParkA_2.0 defines three methods. Two for startup and shutdown and one for sensor values retrieval.

2.2 Event model

The event model extends the syntax model with the definition of *signal events*. We use a signal event in order to describe an occurrence of a method call, method response, or broadcast with a concrete parameter constraint. Figure 2 shows the textual notation of the event model using our case study.

The first line imports the Franca IDL file. The next statement references the target interfaces from the Franca file. After that, we support the definition of multiple events for each method or broadcast defined in the corresponding interface. A signal event could be either a call event, a response event or a broadcast event. The signal event can be specified with the `constraint` attribute which defines a logical expression using the parameters of the referenced method. The call event can be constrained using the input parameters of the method and the response- and broadcast events can be constrained using the output parameters of the method or broadcast. In our implementation of the model we support basic arithmetic and logical expressions.

```
import "../franca/ParkA1.fidl"

Interface Parka {
  CallEvent start {
    methodRef startup
  }

  ResponseEvent start{
    methodRef startup
  }

  CallEvent shutdown{
    methodRef shutdown
  }
  ResponseEvent shutdown{
    methodRef shutdown
  }
  BroadcastEvent sensorValues{
    methodRef sensorValues
  }
  BroadcastEvent started{
    methodRef started
  }
}
```

(a) ParkA_1.0 event model

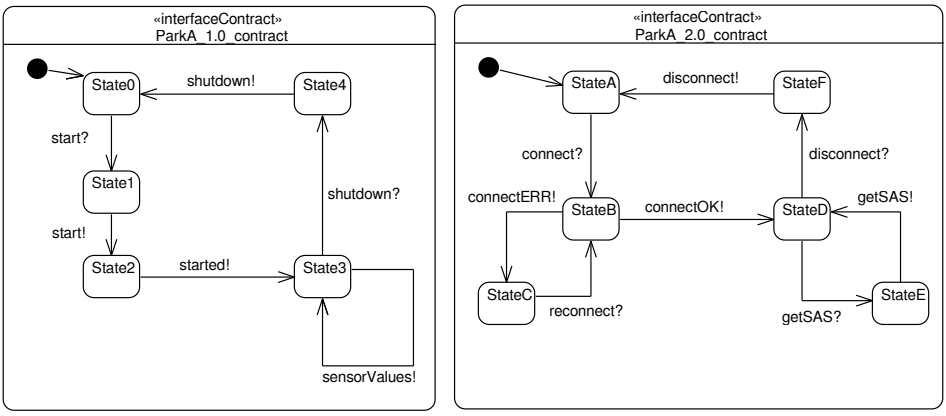
```
import "../franca/ParkA2.fidl"

Interface Parka {
  CallEvent connect {
    methodRef connect
    constraint {retry == false}
  }
  CallEvent reconnect {
    methodRef connect
    constraint {retry == true}
  }
  ResponseEvent connectOK{
    methodRef connect
    constraint {(result == OK)
      && (connectionID != -1)
    }
  }
  ResponseEvent connectERROR{
    methodRef connect
    constraint {(result == ERROR)
      && (connectionID ==-1)
    }
  }
  CallEvent disconnect{
    methodRef disconnect
  }
  ResponseEvent disconnect{
    methodRef disconnect
  }
  CallEvent getSAS{
    methodRef getSensorsAndStatus
  }
  ResponseEvent getSAS{
    methodRef getSensorsAndStatus
  }
}
```

(b) ParkA_2.0 event model

Figure 2: Two variants of the ParkA event model

2.3 Behavior Model

In order to correctly adapt the behavior of a component, it is necessary to specify the protocol of a service. We use a subset of UML state machines for behavioral description. We restrict the expressive power with a UML profile and add the elements needed for seamless integration in our modeling approach. The behavior model depicted in figure 3 uses the signal events defined in the event model (see figure 2) as triggers. The behavior model of the ParkA_1.0 service is shown in figure 3a. First, the `startup`-method has to be called. After the method return, the server sends a `started`-broadcast. After this broadcast the ParkA_1.0 service will provide cyclic sensor values until proper shutdown by the client.



(a) ParkA_1.0 (b) ParkA_2.0

Figure 3: Behavior models of the ParkA_1.0 and the ParkA_2.0 services. Client calls are denoted using ? and server responses and broadcasts are denoted using !

Figure 3b shows the behavior of the the ParkA_2.0 service which differs in various ways from the ParkA_1.0. The ParkA_2.0 gets started with a `connect`-method and does not send the `init`-broadcast. Additionally the server may reply to the `connect`-method with an error. In this case the client has to invoke the `connect`-method again. In contrast to the ParkA_1.0 service the ParkA_2.0 does not provide the sensor values with a cyclic broadcast. The client has to call the `getSensorsAndStatus`-method in order to get the current values.

3 Adapter Model

Most of the adaptations which occur in the daily work are syntactical interface changes. On the one hand we want an adapter model which addresses these changes in an easy way. But on the other hand we do not want to completely forgo behavioral adaptation.

For this reason we developed an adaptation approach which consists of two separate models. The first, the *static adapter model* (global adapter), describes global adaptations which get applied anytime. Whereas the *dynamic adapter model* (local adapter) specifies context-sensitive adapter behavior depending on the current state of the communication between client and server (communication state). Static adaptations may be implemented without using a state machine.

It depends on the target system which information is modeled in the static adapter and which information in the dynamic adapter. If, for instance, the adapter should address AUTOSAR software components, the timing information is an important part of such components. The modeler may want to change the timing information of a port globally in the static adapter model. Thus, the approach presented in this section may be adapted to a different target component model.

In this contribution we extend our previous approach [PSB13] with timing information and parallelism. Figure 4 shows the architecture of the adapter. The client proxy communicates with the adapter calling methods and receiving broadcasts. Each method call will be processed by the adapter depending on the modeled adaptation behavior. The static adapter gets the method call and applies the static mapping. Every interaction with the client or the server is forwarded to the dynamic adapter and may cause the dynamic adapter to trigger additional actions. While the dynamic adapter is active, the static adapter will pause the communication until he gets activated again.

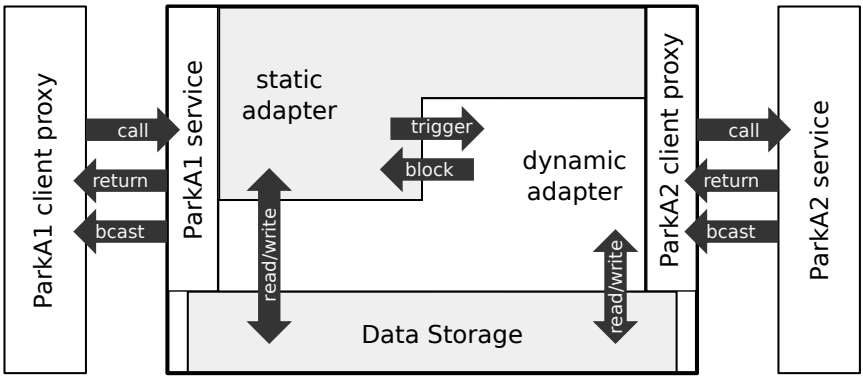


Figure 4: Architecture of the static and dynamic adapter

3.1 Static Adapter Model

The static adapter model provides a mapping of methods and parameters. We implement this model using a textual notation. Figure 5 shows the static adapter model for the ParkA use case. The static adapter model consist of a method mapping and a parameter mapping.

```
import "../franca/ParkA1.fidl"
import "../franca/ParkA2.fidl"

interface Parka.1.0 {
  static adapter ParkA_1_to_2{
    server interface Parka.2.0

    method mapping{
      Parka.1.0.startup = Parka.2.0.connect
      Parka.1.0.shutdown = Parka.2.0.disconnect
    }
    parameter mapping{
      Parka.1.0.sensorValues.front_left = 0
        | Parka.2.0.getSensorsAndStatus.frontLeft;
      Parka.1.0.sensorValues.front_mid = 0
        | (Parka.2.0.getSensorsAndStatus.frontMidleft +
          Parka.2.0.getSensorsAndStatus.frontMidright
        )/2;
      Parka.1.0.sensorValues.front_right = 0
        | Parka.2.0.getSensorsAndStatus.frontRight;
      Parka.2.0.disconnect.connectionID = 0
        | Parka.2.0.connect.connectionID;
      Parka.2.0.connect.retry = false;
      Parka.2.0.connect.retryCount = 0;
    }
  }
}
```

Figure 5: Static adapter model which maps a ParkA_1.0 client interface to a ParkA_2.0 server interface

- *Method Mapping:* Client methods can be mapped to one or more server methods and server broadcast can be mapped to one ore more client broadcasts. Such a mapping will cause the static adapter to call the mapped methods or broadcasts in the specified order. In our example we have two method mappings. One for the `startup` method and one for the `shutdown` method.
- *Parameter Mapping:* Each output parameter of the adapter (parameters of server calls, client responses, client broadcasts) can be defined in a parameter mapping. A parameter mapping consists of a mandatory default value and an optional arithmetic expression of input parameters of the adapter. The default value is needed, because the adapter may not have received all the values of the parameters contained in the arithmetic expression. The parameter mapping depicted in figure 5 defines the `front_mid` return parameter as average value of `frontMidleft` and `frontMidright` with a default value of `0`. The default value will be sent if the `frontMidleft` or the `frontMidright` attribute was not yet returned by the server.

3.2 Dynamic Adapter Model

The dynamic adapter model defines local adapter behavior which depends on the current communication state. This may be an adaptation which is not performed each time a method gets called but only in a specific state. Additionally the dynamic adapter model may describe more complex adaptations than the static adapter model like parallelism and timing. We use the same notation as for the interface behavioral model (see figure 3) but add the concepts of actions, timing and parallelism. Figure 6 shows the dynamic adapter of the ParkA use case.

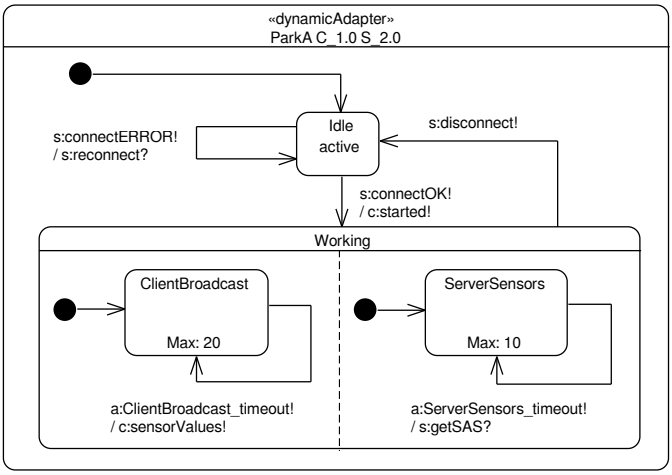


Figure 6: Dynamic adapter model which maps a ParkA_1.0 client interface to a ParkA_2.0 server interface. The active-flag is only shown if it is set to true. The Max-argument is only visualized if the timeout is set for a specific state.

The state machine consists of initial states, states, join states, transitions and parallel regions. The transitions use the signal events defined in section 2 as triggers and actions. The states have an additional `active`-flag. This flag causes the dynamic adapter to remain active after a transition was triggered and prevents the static adapter to continue with the static adaptation. Additionally the states have a `Max`-attribute. It can be used to define the maximum active time of a state. If the Max-attribute of a state is set to some value, the state machine will start a timer while entering the state. Once the timer elapses a timeout signal will be generated which may be used as trigger.

In our example the dynamic adapter gets first activated when a `connectERROR` or a `connectOK` signal event gets processed. In case of `connectERROR`, the dynamic Adapter will send a `reconnect` method call to the server. Additionally the dynamic adapter will remain active until the server returns the connect call matching the `connectOK` signal event. In this case `connectOK` the adapter will send a `started`-broadcast to the client. The next state has the `active`-flag set to false and will unblock the static adapter. Therefore the static adapter will continue with the static mapping of the startup method

returning the `connect` call to the client.

In the next state, the `Working`-state the dynamic adapter will continue with a parallel region. We use the Max-argument of our model in order to define cyclic behavior. The first region sends a broadcast every 20 milliseconds to the client with sensor values. The second region retrieves the current sensor values from the server every 10 milliseconds with the `getSensorsAndStatus` method call. The sensor value parameters and the disconnect method are mapped with a static mapping.

In the dynamic adapter model the modeler has to specify how a `reconnect`-signal can be generated. This can be done in our signal event definition. We added an `emulation` field which contains all the necessary information for calling or returning a method and sending a broadcast. Thus, each input parameter of a method call or each output parameter of a method response or broadcast is defined inside the emulation field. Figure 7 shows the first part of the Parka_2.0 interface event model. For the `reconnect` method call the `retry` parameter will be set to `true` and the `retryCount` will be increased each time the `reconnect` signal gets emulated. If an emulation gets executed and at the same time a static mapping exists for one emulated parameter, the dynamic mapping will override the static one.

```
import "../franca/ParkA2.fidl"
Interface Parka {
  CallEvent connect {
    methodRef connect
    constraint {retry == false}
  }
  CallEvent reconnect {
    methodRef connect
    constraint {retry == true}
    emulation {
      retry = true
      retryCount = 1 | connect.retryCount+1
    }
  }
  ResponseEvent connectOK{
    methodRef connect
    constraint {(result == OK) &&
      (connectionID != -1)
    }
```

Figure 7: Signal event mapping with emulation definition

4 Validating Static and Dynamic Adapters

Splitting the adapter into two parts makes it hard for the modeler to identify whether the adapter model is correct for a specific client and server behavior. Thus, the modeling environment has to facilitate adapter validation.

In this contribution we show, how the behavior model of the client service, the static adapter and the dynamic adapter can be joined to one behavior model. This enables us to validate

the adapter comparing the joined model with the behavior model of the server.

Figure 8 shows the overall process of the validation. The first step is to apply the static mapping. The second step is to insert the behavior defined in the dynamic adapter.

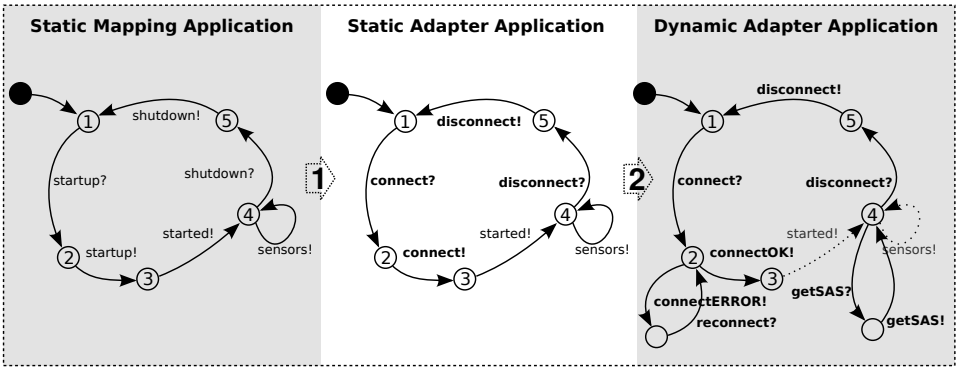


Figure 8: Validation of the adapter applying the static and dynamic adaptations to the behavior model of the client. Deleted transitions are denoted using dashed lines.

- Static adapter application: Each occurrence of a mapped method or broadcast will be replaced by the mapped method calls. For the ParkA use case the `startup` and `shutdown` signal events will be replaced by `connect` and `disconnect` signal events.
- Dynamic adapter application: Both graphs get traversed starting from the initial node. If a trigger of the dynamic adapter model fires at a certain transition in the client behavior model, this trigger will be exchanged by the dynamic adapter trigger, all the signal events denoted as server calls will be inserted in the client behavior and all the signal events denoted as client broadcasts will be deleted. It depends on the `active`-flag of the target state in the dynamic adapter model whether the next transitions will also be inserted.

  In our example we insert the dynamic adapter shown in figure 6. The `connect`-response in the client behavior model gets substituted by the triggers `connectOK` and `connectERROR` and the `reconnect`-action gets inserted. The `started`-broadcast will be sent by the adapter and will be deleted in the client behavior model. Then, the `getSAS`-call and response will be inserted and the `sensors`-broadcast will be deleted.

The resulting graph should now be compatible with the server behavior (see figure 3b). This means, that every client call gets accepted in a certain state of the server and every server response gets accepted by the client.

5 Expressive Power of Static and Dynamic Adapters

In order to better understand the expressive power of our modeling approach we define elementary activities which can be performed with an adapter. This could be adding a parameter to a method or performing an additional method call after some event. Gierds et al. [GJW08] specify behavioral adapters with transformation rules for elementary activities. That is, an adapter is composed exactly of this activities. But with the approach presented in this contribution it is hard to define elementary activities which do exactly match the expressive power of the adapter.

Table 1 shows a collection of elementary activities which can be addressed with our approach. Everything which can be done with the static adapter, may also be modeled using the dynamic adapter. For instance, a global renaming of a method may be modeled with the dynamic adapter using a parallel region which gets triggered by the method call and performs an action with the renamed method call. The difference may only be the performance of the generated code. But in general, global changes should be modeled using the static adapter and local changes using the dynamic adapter.

Some activities can only be performed with the usage of the dynamic adapter. This is closely related to the used modeling approach. Our dynamic adapter model includes timing definition and parallelism. Therefore such adaptations may only be implemented using this model.

Elementary activity	**Static Adapter** (global changes)	**Dynamic Adapter** (local changes)
change parameter type	●	●
delete parameter	●	●
add parameter	●	●
rename parameter	●	●
change parameter order	●	●
change method return type	●	●
rename method / broadcast	●	●
remove method / broadcast	●	●
do additional method calls / broadcasts	●	●
delay method call / broadcast	○	●
wait for method call / broadcast	○	●
do parallel actions	○	●

Table 1: Elementary activities which can be performed using the static- or the dynamic adapter model. ○ denotes that an elementary activity may not be realized using the model. ● denotes that the model allows for the description of the elementary adaptation.

6 Related Work

The main goal of component based software engineering (CBSE) is the planned reuse of software artifacts. One mayor problem while integrating components in a new environment is architectural mismatch [GAO95] [GAO09]. In this context, component adaptation is a promising approach to extend the reuse of software components even if the interfaces of the target environment do not match exactly. The detection of mismatches is usually based on a formal definition of the interface syntax and semantics.

Interface mismatches may be defined at several levels. Examples for a classification are presented in [BBG+06] and [CMP06]. Becker et al. define five hierarchical levels of software adaptation. Our approach addresses the first four levels: signature mismatch, assertion mismatch, protocol mismatch, and quality attributes mismatch. The first three are covered in detail but the quality attributes mismatch includes only the adaptation of timing.

Most of works dealing with component adaptation use formal service- and adapter-descriptions [BBC05, BP06, GJW08, GMW12, CPS08, MP09]. The abstract nature of the models enables for formal adapter validation and semi-automated adapter model generation. In contrast to these approaches our main goal was the practical applicability and code-generation for real software-systems.

Our Static Adapter Model describes a simple mapping between methods and parameters and may be compared to the approaches presented in [BBC05, GJW08, GMW12]. Gierds et al. specify elementary activities which include message creation, multiplication, deletion, transformation, message splitting, merging and recombination. Dynamic adaptation is often addressed by non-deterministic elementary activities. Bracciali et al. use dedicated non-deterministic actions [BBC05] but no insight is given of how the adapter chooses which rule to take in a certain state.

Martin and Pimentel [MP09] present a modeling approach which describes services in the mean of methods and parameters. The main goal was the automated generation of adapter models. The approach relies on the presumption that method parameters are already mapped between the services. Such a mapping is realized by our static adapter model. Canal et. al. [CPS08] also present an approach for adapter generation. They use finite-state machines with labeled transitions to describe the behavior of components. The formalism focuses on the behavioral aspects and is limited to the usage of events which are represented as strings.

Cubo et al. [CSC+08] present a solution for the adaptation of Windows Workflow Foundation. The method mapping is implemented using vectors and the modeler may extend the adapter with C# code snippets.

Our approach uses many concepts of the presented approaches but extends the models in various ways. First, we propose to split the monolithic adapter model in a static and a dynamic part. In current adaptation approaches this may be implemented with an additional post-processing step. A simple way to achieve this would be to identify and extract elementary static mappings in the monolithic behavioral adapter model. Second, we add the concepts needed for simple timing adaptation and parallelism. This enables the adapter to perform event-triggered actions such as delaying messages or to sending cyclic messages.

And finally, we specify the model in such detail, that direct code-generation for our target system was possible [PSB13]. In this contribution we also showed how the defined model may be implemented using UML profiling aside with a textual interface and event notation.

7 Conclusion and Future Work

In this contribution we presented an approach for the adaptation of mismatching interfaces. We cover both signature-level mismatches and behavioral mismatches. Additionally, we introduce a model extension for timing and parallelism.

In contrast to previous approaches we split the adaptation model into a static and a dynamic model. This enables us to define simple but frequent adaptations in a separate model. These static adaptations are applied globally and executed without using a state-machine. Furthermore we use the dynamic adapter model for timing and parallel actions. The dynamic adapter model is used for the description of local adaptations, which will only be executed in a certain communication state.

Our target system was the inter-process communication of an automotive infotainment system. We use the current tools and models from the GENIVI project as the basis for our implementation. A practical realization of the models was shown using a textual notation for the static adapter model and a UML profile for the dynamic adapter model.

One perspective of our work is the semi-automated adapter model generation and completely automated adapter model validation. Currently, the validation only addresses the control flow but not timing constraints and parallel regions. A second perspective is the formal definition of our models, which will enable us to formally define the algorithms needed for validation.

References

[AUT] AUTOSAR. AUTomotive Open System ARchitecture. Online: 28.06.2013. http://www.autosar.org.

[BBC05] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, January 2005.

[BBG+06] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an Engineering Approach to Component Adaptation. In RalfH. Reussner, JudithA. Stafford, and ClemensA. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 193–215. Springer Berlin Heidelberg, 2006.

[BP06] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In A. Dan and W. Lamersdorf, editors, *Service-Oriented Computing - ICSOC 2006*, volume 4294 of *Lecture Notes in Computer Science*, pages 27–39. Springer Berlin Heidelberg, 2006.

[CMP06] C. Canal, J. M. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1):9â31, 2006.

[CPS08] C. Canal, P. Poizat, and G. Salaun. Model-Based Adaptation of Behavioral Mismatching Components. *IEEE Transactions on Software Engineering*, 34(4):546 –563, August 2008.

[CSC+08] Javier Cubo, Gwen Salan, Carlos Canal, Ernesto Pimentel, and Pascal Poizat. A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. *Electronic Notes in Theoretical Computer Science*, 215:39–55, June 2008.

[FRA] FRANCA. The Franca Interface Description Language. Online: 28.06.2013. http://code.google.com/a/eclipselabs.org/p/franca/.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: why reuse is so hard. *Software, IEEE*, 12(6):17 –26, November 1995.

[GAO09] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse Is Still So Hard. *Software, IEEE*, 26(4):66 –69, August 2009.

[GEN] GENIVI Alliance. The GENIVI Alliance. Online: 28.06.2013. http://www.genivi.org.

[GJW08] C. Gierds, Mooij A. J., and K. Wolf. Specifying and generating behavioral service adapter based on transformation rules. Preprint CS-02-08, Universität Rostock, Rostock, Germany, August 2008.

[GMW12] C. Gierds, A.J. Mooij, and K. Wolf. Reducing Adapter Synthesis to Controller Synthesis. *IEEE Transactions on Services Computing*, 5(1):72–85, 2012.

[MP09] J. A. Martn and E. Pimentel. Automatic Generation of Adaptation Contracts. *Electronic Notes in Theoretical Computer Science*, 229(2):115–131, July 2009.

[MPS12] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. *IEEE Transactions on Software Engineering*, 38(4):755 –777, August 2012.

[OMG09] OMG. UML Version 2.2 Superstructure. Technical report, OMG, 2009.

[PSB13] T. Pramsohler, S. Schenk, and U. Baumgarten. Towards an Optimized Software Architecture for Component Adaptation at Middleware Level. volume 7957 of *Lecture Notes in Computer Science*, pages 266–281. Springer Berlin Heidelberg, 2013.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38 –49, March 1992.