Managing Communication Paradigms with a Dynamic Adaptive Middleware

Tim Warnecke, Karina Rehfeldt, Andreas Rausch

Technische Universität Clausthal
38678 Clausthal-Zellerfeld, Germany
email: {tim.warnecke, karina.rehfeldt, andreas.rausch}@tu-clausthal.de

***Abstract*—Autonomous systems, such as self-driving cars, are an emerging but very complex class of systems designed to relieve people of many unpleasant tasks. However, in the industry software architectures and frameworks that have been developed for non-autonomous systems are often used for such autonomous systems. These architectures and frameworks are rigid, very close to the hardware platform and offer little room for extensions. In this paper, we present a dynamic-adaptive middleware for autonomous systems, which is based on a formal component model and supports several communication paradigms independent of the actually used communication technologies. The presented middleware has already been implemented as a prototype and tested with an electric model vehicle.**

Keywords–dynamic adaptive system; component model; decentralized configuration; middleware; communication paradigm

I. Introduction

Autonomous driving starting at level 3 of the Society of Automotive Engineers (SAE level 3) [1] is one of the biggest technological challenges of our current time. But, to be able to provide flexible and safe autonomous driving, a range of challenges have to be tackled. To scale new functions such as autonomous driving and to reduce the complexity of integrating new functions, the electrical/electronic architecture (E/EA) will change to fewer central (domain-specific) computers and increased data preprocessing directly in sensors and actuators. The development approach will shift from "new functions as Electronic Control Unit (ECU)" to "new functions as software".

The typical development process of an embedded system, or any system strongly dependent on hardware/ software, starts with the definition of functionalities. In a next step, these functionalities are divided in hardware and software requirements, which are used for the development of hardware and software solutions. Although, the development of the two domains is strongly interconnected, they often differ with regard to the time needed for their realization, their innovation cycles and their specific technological advancements [2] [3].

While developers may use a hardware platform for quite some time, innovations and new functionalities like autonomous driving are often driven by software improvements [4]–[6]. Additionally, it is desirable to re-use those new software solutions in form of components in a range of different hardware variants of a product line [7]. The changes in the vehicle architecture and the realization of new functions like autonomous fleet management also require to include back-end-systems and Car2X communication.

Another aspect, which is especially true for autonomous driving, is that a software system must maintain its functionality and safety in every situation [8]. Therefore, the software urgently needs the ability to adapt to context changes. In some circumstances, a reduction in quality is acceptable but the safety of the autonomous vehicle has to be guaranteed. To do this, safe adaptation during runtime even in uncertain environments and unforeseen situations must be possible [8], [9]. The best case is if both, proactive adaptation and reactive adaptation, are implemented within a safety critical system. Proactive adaptation is used to prevent failures from happening, while reactive adaptation is used to recover from changes or errors in the technical resources or context of the system.

The mentioned challenges have to be tackled to realize flexible and safe autonomous driving. Our solution is to change from rigid and inflexible systems to adaptive systems which adapt to the provisioned hardware and the context of the system. To build such an adaptive system exist different solutions which we will discuss in Section II. In our concept, we decided to construct a programming framework, which relieves the developer of the work of the adaptive mechanisms and optimizes various aspects of the system globally such as performance and communication.

As mentioned before, it will be a crucial part of the new vehicle architecture to be able to work on various hardware platforms and even migrate software components during runtime. Therefore, we are dealing with distributed systems connected by possibly diverse technical communication channels. Even nowadays exist a huge variety of communication busses, middlewares and paradigms within a vehicle [4].

In this paper, we will introduce a new middleware for autonomous systems which builds up on our experiences in adaptive component models and in software architectures. In Section II, we introduce the current state of the art of self-adaptive systems and communication paradigms and taxonomies. We describe our own communication taxonomy for distributed systems in Section III, followed by an introduction in Section IV to our dynamic adaptive middleware currently under development. Motivated by a small example, in Section V, we show which adaptive mechanisms and communication paradigms have to be considered for an adaptive middleware in the autonomous domain. Based on the previous Sections, we explain in Section VI how we integrated the presented communication paradigms into our middleware using fitting concepts. We conclude our paper in Section VII with possible future work and a conclusion in Section VIII.

II. Related Work

As motivated in the introduction, a middleware suitable for flexible autonomous driving scenarios has to deal with adaptive mechanisms and different communication middlewares and

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

paradigms. It has to provide the right tools for the developer for building a robust but flexible system. In this section, we give an overview of current self-adaptive system approaches and communication paradigms.

A. Self-Adaptive Systems and Frameworks

A self-adaptive system is capable of monitoring its operating environment and automatically adapt to changes [10]. It provides self-x properties like self-configuration, self-optimization, self-healing or self-protection [11]. A comprehensive and exhaustive study by Krupitzer et al. has attempted to structure the field of self-adaptation with the help of their own taxonomy [8]. Krupitzer et al. ask a total of six questions related to self-adaptation and map these to five different properties of a self-adaptive system shown in Table I. In their study, they shed light on the individual dimensions of adaptability in detail and evaluate a large number of self-adaptive approaches.

TABLE I. ADAPTION TAXONOMY [8]

Question	Dimension of taxonomy
When to adapt?	Time
Why do we have to adapt?	Reason
Where do we have to implement change?	Level
What kind of change is needed?	Technique
Who has to perform the adaptation?	N/A because of self-adaptation
How is the adaptation performed?	adaptation Control

Self-adaptation can be achieved by many different means and procedures. Adaptation can be managed internally or externally, it can be reactive or proactive, centralised or decentralised and many other criteria play a role. In the field of large-scale systems, component-based development is a solid and state-of-the-art approach [12]–[14]. One example for component based development are middlewares, which not only define services and establish an infrastructure, but also specify a formal component model [15]. In our approach, we opted for a component model and middleware solution that frees the application developer from the task of self-adaptation and is able to hide the underlying specific platform and provide a unified high-level interface.

One well known component model is the CORBA Component Model (CCM) [16] from the Common Object Request Broker Architecture (CORBA) [17], a component based middleware. Building up on CORBA, *dynamicTAO* [18] introduces a reflective Object Request Broker (ORB) to support dynamic reconfiguration by maintaining an explicit representation of the internal structure of the system. Another well-known framework is the Rainbow framework [19] which divides the self-adaptive system in an architecture layer and a system layer with the managed resources. To realize adaptation, an external manager is used, which exploits the architecture model to monitor the running system and in case of constraint violations it performs adaptation accordingly.

In recent years, attempts have also been made in the automotive domain to introduce adaptability as a development approach. One example is the Dynamically Self-Configuring Automotive Systems (DySCAS) project [20], in which a policy-driven middleware was developed. The introduced policies describe a desired high-level behavior, which is then executed by the underlying middleware. This procedure makes it possible that the policy writer does not have to be a self-adaptive expert. One of the focal points of DySCAS is the limited resources of the ECUs and the resulting resource-optimized performance.

Becker et al. describe a model-based extension of the AUTomotive Open System ARchitecture (AUTOSAR) that introduces reconfiguration mechanisms at the architectural level [21]. They present a toolchain, which allows the developer to describe configuration alternatives and transitions between different configurations. These models are translated into executable code. The system's reconfiguration capabilities are limited by the developer's specifications created at design time, but can be verified because of the model-based approach.

B. Communication Paradigms

In our middleware, the focus is not only on adaptivity but also on communication. The application developer should be provided with exactly the right communication paradigms to develop a good architecture. For this reason, we will take a closer look at various well-known communication paradigms in this section.

For communication in distributed systems exist a variety of different paradigms, each suited for special cases. Tanenbaum and van Sten discussed various communication paradigms in [22]. They structured these in four categories: Remote Procedure Call (RPC), message-oriented communication, stream-oriented communication and multicast communication. RPC was further broken down into synchronous and asynchronous RPC, whereas message-oriented communication was separated in transient and persistent communication. Examples for transient communication are Berkeley Sockets or Message-Passing Interfaces. Persistent communication is realized through message broker architectures or message queues.

According to Tanenbaum and van Sten [22], stream-oriented communication is mainly used for continuous media. Important aspects of stream-oriented communication are the quality of service, and, in case of multiple streams, the time synchronization.

Multicast communication includes all communication from one sender to a group of receivers [22]. This could be done by directly addressing a group and sending the message to all of them at once or by using gossip-based protocols where each node receiving a message shares the message with a group of other nodes until all nodes have received the messages.

Another comprehensive summary of communication paradigms is found in Schill and Springer [23]. They structure communication paradigms similar to Tanenbaum and van Sten in Remote Procedure Calls, message-oriented and stream-oriented communication but also include data-based communication. Unlike Tanenbaum and van Sten, they do not have clear sub-categories but highlight individual aspects of communication.

In RPC, they particularly emphasize the synchronous bidirectional call [23]. For message-oriented communication, they focus on unidirectional communication channels such as those used in Publish/Subscribe systems. In the case of stream-oriented communication, they mainly consider periodic, synchronous data streams that can be either uni- or bidirectional.

Data-based communication is divided into mobile objects and shared space or information sharing [23]. Mobile objects

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

are known, for example, as a part of Java RMI (Remote Method Invocation). These refer to data that is packed into an object and exchanged between sender and receiver instead of using messages for data exchange. They are used mainly for communication between a limited number of participants. Shared space allows a number of components to write and read structured data in a collective space. Another suitable term for this kind of communication, which is more common in embedded systems, is global variables.

III. Communication Taxonomy for Distributed Systems

The taxonomy of Tanenbaum and van Sten [22] lacked the data-based communication that is often used in the automotive domain, while the taxonomy of Schiller and Spring [23] did not go into as much detail as Tanenbaum and van Sten and lacked the inner structure. That is why, we derived our own taxonomy based on these two, which is shown in Figure 1. In our opinion, we have combined the best of both and supported it with references to well-known software patterns.

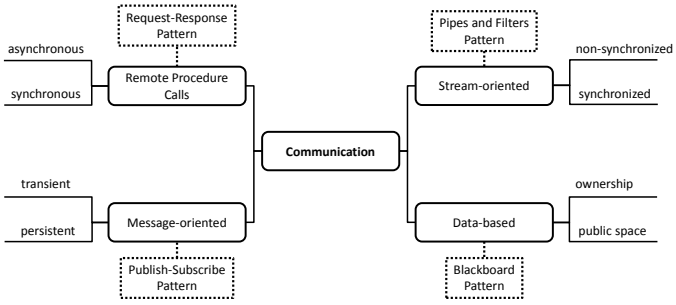

Figure 1. Adapted Communication Taxonomy for distributed Systems based on [22] and [23]

Our taxonomy has four categories: Remote Procedure Calls, message-oriented, stream-oriented and data-based. RPC maps to the classic request-response or client-server pattern [24]. We distinguish it like Tanenbaum and van Sten in asynchronous, non-blocking and synchronous, blocking calls. RPC is mostly used when data has to be transferred irregularly on a request-response (bidirectional) basis.

For message-oriented communication we assign the publish-subscribe [24] pattern. Just like RPC, we follow Tanenbaum and van Sten in our message-oriented communication and further classify messages according to how they are transmitted. In the persistent case, the transmission is carried out using a broker which is able to store messages temporarily, for example. For the transient transmission of messages, we follow the channel pattern [24] that allows the transmission of messages using topics or special channels. An important aspect of topics is that the recipients and senders do not know each other. In both cases, persistent and transient, message-oriented communication is used for either periodic or sporadic data. The transmission is always unidirectional.

In stream-oriented communication, we distinguish between synchronized and non-synchronized communication. By synchronized streams we mean anything where two or more streams have to be synchronized with each other, e. g. internet telephony or video conferences. Non-synchronized communication is used for sensor data streams or continuous media streams. Overall, stream communication maps to the Pipes and Filters pattern [24].

Our last category is based on the taxonomy of Springer and Schill. Data-based communication describes the exchange of structured data that either belongs to a system participant (ownership) or that can be read and processed by everyone in a collective space (public space). Data-based communication matches the blackboard pattern [24]. This communication method is useful when structured data has to be shared between a number of components on a regular basis.

We use the presented taxonomy to divide means of communication. Based on this, we discuss which communication methods should be supported by a dynamic adaptive middleware. In the next section we present our former middleware for dynamic, adaptive systems, which only supported synchronous RPC, followed by the expansion of our middleware to include asynchronous RPC, transient message-oriented and data-based ownership communication.

IV. Dynamic Adaptive System Infrastructure - DAiSI

Before we take a closer look at our new concepts for the development of a dynamic-adaptive middleware for autonomous driving, we explain in this section our previous work on a dynamic-adaptive middleware for information systems called Dynamic Adaptive System Infrastructure (DAiSI) [25]. DAiSI is a middleware based on a formal component model.

The model defines each software component in a system as a dynamic adaptive component (*DynamicAdaptiveComponent*), which consists of several component configurations (*ComponentConfiguration*). Each configuration describes in detail which services a component requires (*RequiredServiceReferenceSet* - RSRS) and which services it provides (*ProvidedService* - PS). The PS and RSRS match on service descriptions in the *DomainInterface*. All *DomainInterfaces* are bundled in the *DomainArchitecture*, which describes possible services of a whole domain like autonomous driving.

The provided and required services match via domain interfaces that specify which synchronous method calls the service instances offer. Each component implements the MAPE loop [11]. Each component observes changes to the system and its environment or context (Monitoring), analyses them (Analyze) and plans (Plan) and executes (Execute) necessary adaptations. How these processes work in detail can be found in [25]. For our purposes, a short introduction is sufficient.

A fundamental characteristic of DAiSI is that each component tries to resolve its dependencies in order to be able to offer its services. Each element, briefly explained in the following, implements its own state machine. If a *ProvidedService* is required to run, either because it is needed by the environment (for example GUI or interface to external systems) or because another component in the system has requested the service, this PS informs all *ComponentConfiguration* by which it is provided. The *ComponentConfiguration* then changes to the RESOLVING state. As a result, all RSRSs declared by the *ComponentConfiguration* will also switch to RESOLVING state. This process reflects, that to provide a service, a number of dependencies on other services have to be resolved.

The RSRS is aware of all services available in the system. Either via a central instance, such as a broker or a registry,

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

or via a service discovery. For resolving its dependencies, the RSRS inquires the usage of available matching services. When a PS changes to the RUNNABLE or RUNNING state - meaning its dependencies are resolved - it can be used by an RSRS. After the successful negotiation and connection between PS and RSRS, the RSRS change to the RESOLVED state. When all RSRS of a *ComponentConfiguration* are resolved, the *ComponentConfiguration* itself is considered resolved. The *ProvidedServices* either change to the RUNNABLE state, when they are not needed, or to RUNNING, when they are.

In DAiSI, only one configuration can be active at a time. Therefore, if several configurations are potentially activateable (in RESOLVED state), the ”best” one is activated. Each component strives to resolve its ”best” configuration to provide the ”best possible” service.

Over the last five years, we introduced various new concepts to our component model. The local optimality of the self-adaptive system is complemented in [26] by a global system design that specifies which rules/objectives the overall system should follow. This limits the system configuration resulting from the adaptation process. Likewise, in Klus et al. [26] interface roles were introduced. These extend the concept of domain interfaces with an additional statement about the role of a *ProvidedService* at runtime.

Based on the InterfaceRoles, a quality concept is introduced in [27]. The ability to compare the quality of two interfaces allows to make a more differentiated selection of services for the component. This quality may also include properties at runtime, such as current load, communication latencies or the location of services, due to the runtime evaluation of the InterfaceRole.

Up to this point, the interconnection was based purely on syntactic compatibility. However, in systems developed by several developers, such as in the domain of IoT, it is no longer possible to rely solely on syntactic compatibility. In [28] an additional aspect of interconnections in dynamic adaptive systems was considered - semantic compatibility with syntactic differences.

A semantic description language, on which the developers agree, is used for this purpose. By this, in addition to the syntactical description of the interfaces, a semantic description is given. At runtime, the middleware automatically generates adapters that create syntactic compatibility between interfaces which are semantically compatible.

Although different adaptive concepts have been developed, so far only a few communication paradigms have been considered apart from the synchronous RPC. In the following sections, we will describe our work on further communication paradigms that fit into the DAiSI adaptive component model. First of all, we use an example to motivate which paradigms are essential for the autonomous domain.

V. Motivating Example

In this section, we introduce a motivating example for our adaptive middleware which we keep as simple as possible but still shows the challenges for an adaptive middleware in the autonomous driving domain. The example involves an autonomous electric car, which is equipped with diverse sensors and actuators and some software components as shown in Figure 2. The architecture shown is a combination of hardware components abbreviated with <<HW-C>> and software components abbreviated with <<SW-C>>.

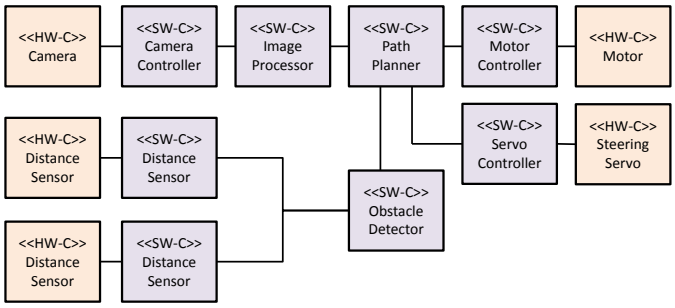

Figure 2. Example Architecture

The task for the car is to follow a lane on the ground and to halt in case of obstacles. A video camera is attached at the front of the car to record a video of the field of vision. The *Camera Controller* component fetches the individual images of the video made available by the camera and the *Image Processor* handles the images to identify the lane. The coordinates of the lane are used by the *Path Planner* to plan the trajectory. The velocity of the motor and the steering angle are controlled by the components *Motor Controller* and *Servo Controller*. These two software components accept the values calculated by the *Path Planner* and control both the motor and the steering servo.

The second simple task for the car is to stop if an obstacle is in front of it. In order to detect obstacles, two distance sensors are attached to the left and right side of the camera. The two software components *Distance Sensor* are responsible for fetching the distance from each sensor. The component *Obstacle Detector* uses these two values to decide whether an obstacle is in front of the car. This in turn is used as an input for the *Path Planner* to determine if the car must halt.

In the next section, we will use this small example to discuss the new communication paradigms in DAiSI.

VI. Extending DAiSI for Autonomous Driving

As we have already indicated in Section III, there are several communication paradigms used in distributed systems. The previous concepts in DAiSI only supported synchronous RPC. It is clear that a modern automotive middleware should offer the application developer more communication methods. Using our taxonomy, we identified three communication paradigms which we included in our component model to account for the needs of the automotive domain. Namely they are asynchronous RPC, ownership data-based communication and persistent message-oriented communication realized through topics. Therefore, we demonstrate how the former component model of DAiSI (see Section IV) can be extended by these three new communication modes. The complete extended model can be seen in Figure 3.

At this point, we will use the example of the autonomous electric car presented in Section V to illustrate our component model.

The elements *DynamicAdaptiveComponent* and *ComponentConfiguration* remain unchanged and still form the basis of the model. The only enhancement at this point is that we have

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

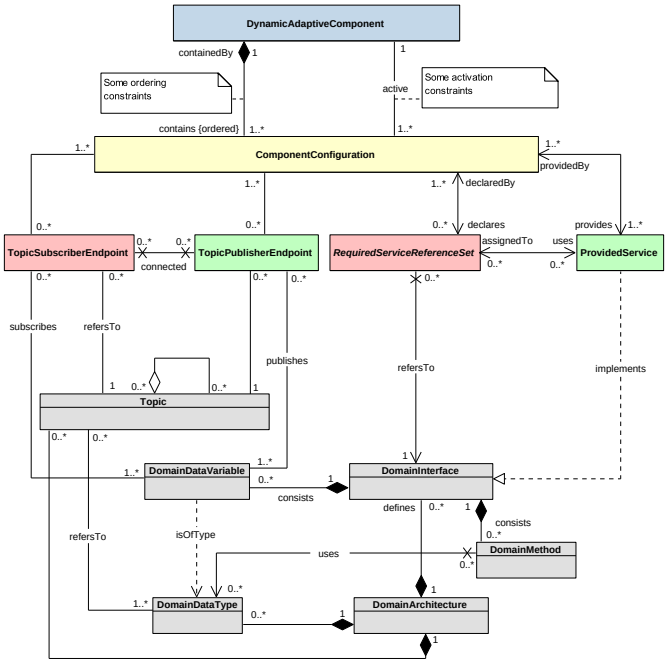

Figure 3. Extended Daisi component model

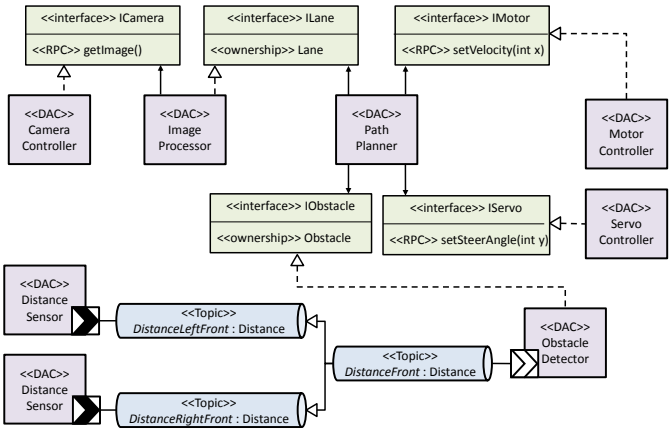

Figure 4. Example Architecture with communication paradigms

granted the possibility that several component configurations can be active at the same time. This should make it easier for developers to combine different configurations.

Each software component shown in the example in Figure 4 corresponds to a *DynamicAdaptiveComponent* annotated with the abbreviation <<DAC>>. However, the communication between the components is now specified by the paradigms RPC, ownership and topics. During runtime, the components can be started and stopped. Thanks to the adaptive concept of the middleware they will connect to the system as shown above. Of course, for every component different *ComponentConfigurations* might exist. The scope of this paper, however, is on the communication paradigms and therefore these aspects are not further examined here.

In the example shown, the *Path Planner* component uses two different RPCs to control the vehicle based on given sensor information. The component uses on the one hand the interface *IMotor* of the component *Motor Controller* to control the speed of the car and on the other hand the interface *IServo* of the component *Servo Controller* to adjust the steering angle. Both values are set using a call when necessary.

RPCs have always been part of DAiSI. The *RequiredServiceReferenceSet* and *ProvidedService* with the corresponding *DomainInterface* therefore remain identical. The *DomainInterface* is structured in *DomainMethod* and *DomainDataVariable*. *DomainMethod* are (asynchronous) callable methods within the interface. *DomainDataVariable* is introduced for the ownership paradigm.

To determine the speed and steering angle, *Path Planner* uses data from the *Image Processor* and *Obstacle Detector* components using the ownership paradigm. The *Image Processor* provides the *Path Planner* with the *Lane* object of its *ILane* interface. The *Path Planner* also uses the object *Obstacle* of the *IObstacle* interface, which is implemented by the *Obstacle Detector*. Using the ownership paradigm, the *Path Planner* can subscribe to changes of the object, while it still knows exactly to which service the object belongs to.

The ownership communication paradigm is illustrated in the component model in Figure 3 at the bottom left. An ownership object is part of a domain interface, using the *DomainDataVariable* with a data type given by the *DomainDataType*. A domain interface can contain any number of ownership objects and methods.

Transient message-oriented communication is realized using hierarchic topics. In the example, a topic hierarchy is shown in the lower part of the component model. The left distance sensor of the car publishes its data on the topic *DistanceLeftFront* and the right distance sensor correspondingly on the topic *DistanceRightFront*. The naming of the topics generates a certain semantic meaning, in this case a localization of the installation of the sensors. The topic *DistanceFront* merges the data of topics *DistanceLeftFront* and *DistanceRightFront* to describe the distance to objects in front but without further direction. This topic is used by the *Obstacle Detector* component to evaluate whether there is an object in front of the car. In our example, the topics could be used by more than one distance sensor and it would make no difference for the *Obstacle Detector*, as long as the semantic meaning of *DistanceFront* is preserved. In other words, the *Obstacle Detector* does not need to know the exact data sources.

In open and commercial middlewares, it is a common practice for the publisher to define the topic name and data type for the middleware to create a topic based on these definitions. This means that a consumer needs to know which topics are created by the publishers. Although it is often practiced, it is not a good choice in terms of the overall system architecture. Publishers can publish whatever and however they want, and consumers must adapt accordingly.

We propose to decouple the creation of topics from the publishers and to give these rights to the architect on the basis of a formalized description of the topics. For this purpose, we present the concepts *TopicSubscriberEndpoints* and *TopicPublisherEndPoints* in combination with *Topics*. *Topics* are named and typed transport channels for data sent by a publisher. In our concept the middleware itself is responsible for the creation of these topics via a predefined specification by the architect.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

The data types that can be published on a topic are determined by the *DomainDatatatype*. Further freedom is given to the architects through the self-aggregation of *Topic*. This relationship allows to design topic hierarchies, that is, one can create topics with names and permitted data types that are inherited from each other, as seen in the example.

In the following sections, we explain the realization of the three used paradigms in details and give a brief introduction to our implementation using code snippets.

A. Remote Procedure Call Paradigm

An RPC can be used in the extended DAiSI, in contrast to its predecessor, both in the synchronous and the asynchronous version.

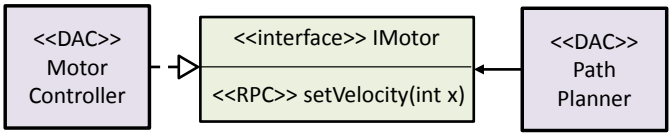

Figure 5. Example RPC

In our example of the electric car, several Remote Procedure Calls have been modeled. At this point, we explain the RPC between the component *Motor Controller* and *Path Planner*, which is shown in Figure 5, in detail. In this case, the component *Motor Controller* is the provider of the interface *IMotor* with the method *setVelocity(int x)* contained in it. It sets the motor speed using the parameter *x*.

```
public class MotorController extends
    DynamicAdaptiveComponent{

  public MotorController() {
    ProvidedService motorSpeed =
    new ProvidedMotor();

    ComponentConfiguration config =
    new ComponentConfiguration() {
      @Override
      protected void
          notifyStateChanged(ConfigurationState
          state) {
        // react to different states
      }
    };
    config.addProvides(motorSpeed);
    this.addConfiguration(config);
  }

  class ProvidedMotor extends ProvidedService
      implements IMotor {
    @Override
    public void setVelocity(int x) {
      // control motor
    }
  }
}
```

Listing 1. Example RPC Callee as Code

In the following, both the *Motor Controller* and *Path Planner* components are shown as simplified Java code. The *Motor Controller* in Listing 1 initially inherits from the *DynamicAdaptiveComponent* of our component model. This allows the application developer to make use of the adaptive mechanism in the middleware. In all of the following examples, the components always inherit from *DynamicAdaptiveComponent*.

To be able to offer a service to other components, a component must first create its own *ProvidedService*. A *ProvidedService*, like *ProvidedMotor* in our example, is created by extending the abstract class *ProvidedService* and implementing a *DomainInterface*, like *IMotor*, with the corresponding methods. In a final step, the service needs to be added to the component configuration to be usable by other components.

A component configuration is created in our middleware using the class *ComponentConfiguration*. The method *notifyStateChanged(ConfigurationState state)* has to be implemented, which informs the component about state changes, e.g. when all dependencies are resolved.

Once the configuration has been created, both provided and required services can be added to it. In the example of *Motor Controller*, the service *ProvidedMotor* is added to the configuration *config* using the method *addProvidedService(ProvidedService ps)*. To add a configuration to a component, the method *addConfiguration(ComponentConfiguration cc)* is used. In the example, *config* is added to the component.

The *Path Planner* component (see Listing 2) wants to use the service *IMotor* and must therefore create a *RequiredServiceReferenceSet* typed with the desired interface. In addition, a RSRS must be given the minimum and maximum number of service instances needed. In the example, both values are set to 1, meaning exactly one service is needed. The RSRS is added to the created configuration using the method *addRequires(RSRS rsrs)*. The process for generating a required service is therefore very similar to the process for creating a provided service. Additionally, the processes for creating a component configurations and adding *ProvidedServices* and *RequiredServiceReferenceSets* to these is the same for each communication paradigm and is therefore not shown again in the following sections on ownership and topics.

```
public class PathPlanner extends
    DynamicAdaptiveComponent{

  public PathPlanner() {
    RequiredServiceReferenceSet<IMotor> m=
    new RequiredServiceReferenceSet(
      IMotor.class, 1, 1);

    ComponentConfiguration config =
    new ComponentConfiguration() {
      @Override
      protected void
          notifyStateChanged(ConfigurationState
          state) {
        // react to different states
      }
    };

    config.addRequires(m);
    this.addConfiguration(config);
    // calculate x
    m.getService().setVelocity(x);
  }
}
```

Listing 2. Example RPC Caller as Code

Once the dependency has been resolved, the service can be used. The RSRS has a *getService()* method, which returns a bound service instance of the given interface *IMotor*. As usual in Java, the methods such as *setvelocity(int x)* can be called here either synchronously or asynchronously using Futures.

B. Data-based Communication Paradigm - Ownership

The biggest disadvantage of the classic, asynchronous or synchronous RPC is the periodic access to data. Each time a component requires a value from another component, an RPC must be sent, a return value calculated and the value returned to the sender. Since the periodic access to data is common in embedded and automotive systems, we have extended our middleware by the communication paradigm ownership.

In our concept, components can create their own data objects and make them available to other components via their *DomainInterfaces*. By making the data available via interfaces, it is clear to the consumer of the data from which component he receives the data. This is extremely important in many cases, as it makes a big difference whether a component receives data from a specific and known sensor or from an arbitrary sensor.

A consumer can subscribe to an ownership object and is thus informed of changes to it. The producer decides when a consumer is informed about data changes. Technically, the consumer is informed about the changes to the object via callback methods of the middleware.

```
public class PathPlanner extends
    DynamicAdaptiveComponent{

  public PathPlanner() {
    private RequiredServiceReferenceSet<ILane> l;
    l = new RequiredServiceReferenceSet(ILane.class,
        1, 1);

    // add required service to config as in Listing 2

    ProvidedService s = l.getService();
    DataObject d = s.getDataObject();
    d.addCallback(new LaneCallback());
  }

  class LaneCallback implements Callback<Lane> {
    @Override
    public void update(Lane lane) {
      // react to new input
    }
  }
}
```

Listing 3. Example Ownership Consumer as Code

We give the producer further freedom in designing the ownership object by allowing to choose between two different types of ownership objects: *ConsumerImmutable* and *ConsumerMutable*. If the producer chooses a *ConsumerImmutable*, he determines that consumers are only allowed to read the data of the ownership object but not to change it. This is especially useful if, for example sensor data is made available to consumers. At this point, it would not be useful or even dangerous if the consumer could change the data. A changed sensor value could lead to fatal consequences for other consumers. By selecting a *ConsumerMutable*, the producer determines that he also allows consumers to change data. Such a setting could be used for example for configuration parameters that are optimized at runtime by different parties.

In relation to our running example Figure 6 shows the components *Image Processor* and *Path Planner*, which exchange their data via ownership communication. As already mentioned, the *Image Processor* offers an object of type *Lane* which stores the exact location. The component updates this object each time it receives a new image from the *Camera Controller*. The component *Path Planner* uses the interface *ILane* to get the ownership object *Lane* from the *Image Processor* and subscribes to it.

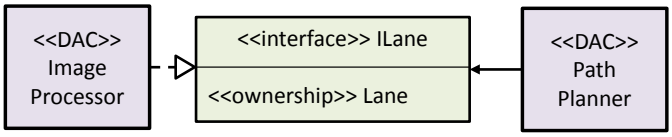

Figure 6. Example Ownership

Based on the simple, small example of the Path Planner and the Image Processor, the two components are shown in Listing 3 and Listing 4 as simplified Java code. In order for the *Image Processor* to be able to offer its *Lane* object to other components, as shown in Listing 4, it must first define such an object and have it inherited by either *ConsumerImmutable* or *ConsumerMutable*. In the example shown, the lane may not be changed by the consumer and therefore it inherits from *ConsumerImmutable*. In the class *Image Processor*, the lane is added within the provided service *ProvidedLane* which implements the DomainInterface *ILane*. The object can be retrieved using the method *getDataObject()*. A consumer is not informed about changes until the producer calls the *publish()* method within the object. This allows the developer to publish updates only for specific events.

```
public class ImageProcessor extends
    DynamicAdaptiveComponent{

  public ImageProcessor() {
    ProvidedService laneLocalisation = new
        ProvidedLane();
    // add provided service to config
  }

  class ProvidedLane extends ProvidedService
      implements ILane {
    private final Lane lane;

    public ProvidedLane() {
      lane = new Lane();
    }

    @Override
    public Lane getDataObject() {
      return lane;
    }
  }
}

public class Lane extends ConsumerImmutable {
  // define lane
}
```

Listing 4. Example Ownership Producer as Code

To be able to use the *Lane* object offered by the *Image Processor*, the *Path Planner*, shown in Listing 3, must first create a *RequiredServiceReferenceSet*, which requires the *ILane* interface. Based on this, the *Path Planner* gets the possibility to access the services of the interface and the method *getDataObject()*, which returns the *Lane* object. A callback is added to this object which is called as soon as the *Lane* is updated by the *Image Processor*.

C. Message-oriented Communication Paradigm - Topics

Topics are the core paradigm for many-to-many communication in DAiSI. We combine already existing concepts for

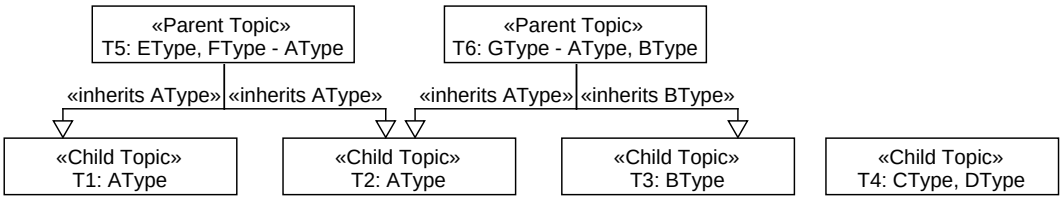

Figure 7. Example Topic Hierarchy

topics with new ideas to a solid and well-structured topic hierarchy. First we use the concept of named and strongly typed topics, which is also already used in middlewares like the Robot Operating System (ROS) [29]. In the case of strongly typed topics, a publisher has to publish data with the correct data type defined by the topic; otherwise the topic refuses transmission. The topics are also defined by a name, so that there can be several topics of the same data type for different purposes.

As already mentioned, we want to further expand the structure of topics so that it is possible to design topics in advance like any other communication and component in the system. We achieve this with a concept that we call *Topic Hierarchy*. The Topic Hierarchy exploits the inheritance relationship of object-oriented programming in such a way that it is possible to combine several smaller topics into larger ones.

Before we delve into the example, we formally define what we mean by the term Topic Hierarchy. First, we define DT as a set of all possible data types which can be used in topics.

A topic hierarchy is a tuple $TH = (CT, PT, IE, src, trg)$ which is defined as a directed acyclic, but not necessarily connected, graph with

- CT (called child topic) is a set of nodes and every node has a label n, a set of own data types $ODT \subseteq DT$ and an arbitrary number of incoming edges
- PT (called parent topic) is a set of nodes and every node has a label n, a set of own data types $ODT \subseteq DT$, a set of inherited data types $IDT \subseteq DT$ and an arbitrary number of incoming and outgoing edges
- IE (called inheritance edge) is a set of directed edges
- $src : IE \rightarrow PT$ is a function which maps the source of an inheritance edge to a parent topic
- $trg : IE \rightarrow PT \cup CT$ is a function which maps the target of an inheritance edge to a parent topic or a child topic

Furthermore, label n is the name of a topic, $ODT \subseteq DT$ is a set of data types that can be published and subscribed and $IDT \subseteq DT$ is a set of inherited data types that can only be subscribed but not published. This restriction was made because parent nodes should aggregate the data of their child topics but can also introduce additional data types. The own data type could also be identical to the inherited data types.

The topics T1 and T2 represented in Figure 7 are both child topics which support data type AType, but have different names and therefore different semantic meanings. Any publisher wishing to publish the data type AType may choose to publish on both topics. In contrast, T5 is a parent topic inheriting the data types from T1 and T2 by an inheritance edge - AType - and introducing EType and FType as own data types. On the other hand T6 introduces only one additional data type GType and inherits the data type AType from T2 and BType from T3. The last topic T4 is not connected to other topics of the hierarchy, which is also allowed.

In our example of the electric car in Figure 8, the Topic Hierarchy is used to provide data from distance sensors. The Topic Hierarchy is initially described by two child topics *DistanceLeftFront* and *DistanceRightFront*. These are used by both distance sensors individually. In addition, both topics only allow the transmission of data of type *Distance*, i. e. no data of other types can be published on these topics. In order to determine if there are objects directly in front of the vehicle, no matter whether they are to the left or right of it, another channel called *DistanceFront* is used. *DistanceFront* is a parent topic, which subscribes to its two child topics and thus receives all the data sent to them.

As already mentioned, the distance sensors gain access to the two child topics (black arrow in white box) via their *TopicPublisherEndpoints*. The *Obstacle Detector* component subscribes to the parent topic *DistanceFront* using a *TopicSubscriberEndpoint* (white arrow in white box).

```
public class DistanceSensor extends
    DynamicAdaptiveComponent {

  @TopicPublisher(type="topic",
  instance = {"name", "DistanceLeftFront",
     "ownType", "Distance"})
  public TopicPublisherEndpoint publishEndpoint;

  public DistanceSensor() {
    Distance d = new Distance();
    publishEndpoint.addObjectToPublish(d);
    // change d
  }
}

public class Distance extends
    ConsumerImmutableDataObject {
  // define distance
}
```

Listing 5. ExampleTopic Publisher as Code

In Listing 5, the architecture described above for the distance sensor was implemented with the help of our middleware. To gain access to the child topic *DistanceFrontLeft*, the component declares a *TopicPublisherEndpoint*. To specify which topic it has to connect to, it is annotated with @TopicPublisher. This annotation defines two attributes: *type* and *instance*. First of all, the type describes which medium the

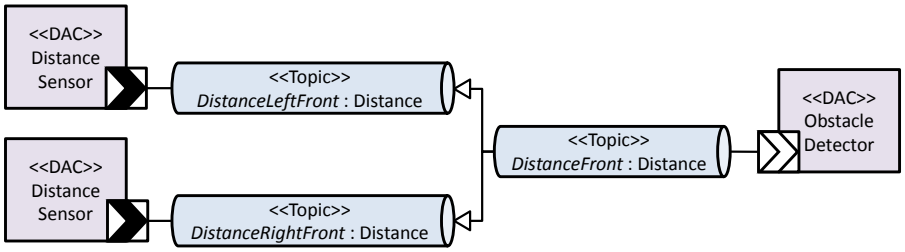

Figure 8. Example Topics

TopicPublisherEndPoint should connect to, which in this case is a topic.

The instance attribute specifies to which concrete instance of the specified type the *TopicPublisherEndpoint* is supposed to be connected. An instance is described by key-value pairs. In this example, a topic named *DistanceLeftFront* which offers the type *Distance* is searched.

```
public class ObstacleDetector {
  @TopicSubscriber(type="topic",
  instance = {"name", "DistanceFront",
  "ownType", "Distance"})
  public TopicSubscriberEndpoint subscriberEndpoint;

  public ObstacleDetector() {
    subscriberEndpoint.addCallback(
      Distance.class,
      new DistanceCallback())
  }

  class DistanceCallback implements
      Callback<Distance> {
    @Override
    public void update(Distance d) {
      // react to new input
    }
  }
}
```

Listing 6. Example Topic Subscriber as Code

After defining the *TopicPublisherEndpoint*, the distance object created can be added to the topic using the method *addObjectToPublish()*. The *TopicPublisherEndpoint* subscribes to this object like a consumer in the Ownership paradigm in the previous section and is informed about updates of this object. The *TopicPublisherEndpoint* automatically sends these updates to the connected topic *DistanceLeftFront*.

To obtain the data provided by the distance sensors, the *Obstacle Detector* component must declare a *TopicSubscriberEndpoint* (see Listing 6). The *TopicSubscriber* annotation is attached to it and uses the same attributes type and instance as seen in the *TopicPublisher* example. However, the *Obstacle Detector* defines that it wants to be connected to the parent topic *DistanceFront*.

Notification of new data on the subscribed topic is handled in the same way as in the ownership paradigm. A callback handler can be added to a *TopicSubscriberEndpoint*, which is called whenever an updated object has been received via the defined topic.

This concludes the presentation of our new component model and middleware. In the next section, we show possible future work within our dynamic adaptive middleware.

VII. Future Work

During interviews with experts in the domain of automotive engineering it quickly became clear that a single communication middleware, such as ROS, would not be sufficient to meet all the requirements of a modern vehicle. Nowadays, most vehicles are equipped with diverse communication middlewares for different purposes and therefore they exchange data with each other by means of adapters. With our middleware DAiSI we would like to offer the possibility that dynamic adaptive components, which run on different DAiSI instances with different communication middleware, can communicate with each other. In order to achieve this goal, a first concept has already been conceived, which will be integrated in a next iteration of DAiSI.

In our interviews, we have also noticed that we need some kind of control mechanism for our adaptation concept, because in safety-critical parts of the system we dont want components to interconnect with each other without a global goal in mind. In this system parts it is often better to give the middleware some kind of blue print on how some of its components have to be connected to guarantee a more deterministic behavior. In this case, we will extend the concept of templates [26] in our new middleware version of DAiSI to have better control over the structure of those critical system parts.

In this paper we have already presented three different types of communication in details: RPC, ownership and topics. However, we noticed that we lacked two crucial communication paradigms for autonomous driving, namely streams and blackboard. We included them in our taxonomy but did not yet implemented them in our DAiSI concept. Streams are used for the transmission of continuous sensor data, e. g. for the transmission of the camera image, since the communication methods described above are not suitable for this type of communication.

Another feature we would like to add to our middleware is a filtering mechanism for the communication methods ownership and topics. In the presented version of these two methods, a consumer of data is informed each time updates were made to the ownership object or new data is published using a topic. We are therefore planning to introduce a filter mechanism that will allow consumers to specify after which kind of changes they would like to be informed. We are still working on a good concept where filtering should take place, whether on the consumer or producer side. Both options have their own advantages and disadvantages in terms of performance.

We are currently persisting the structure of the Topic Hierarchy with an XML document (Extensible Markup Language). We intend to further expand this approach by supplementing

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

semantic information with the help of ontologies. Also, we want to extend the semantic concepts done in [28] to our middleware. With this new feature, we will evaluate whether it is possible to use ontologies effectively and usefully in an autonomous driving scenario. We expect that in very large systems it can make sense to distribute the information about a Topic Hierarchy or general data relation under a common upper ontology. These would be merged at runtime if new components or DAiSI instances are installed in the system and additional topic or data information would be available. We estimate that a semantic reasoner should be able to build the entire Topic Hierarchy.

VIII. Conclusion

In this paper, we have introduced a new kind of dynamic-adaptive middleware for autonomous systems that breaks with many paradigms of classical embedded systems. Components in our middleware are clearly separated from the actual communication technology. The components connect to executable systems on the basis of specified configurations and can be added to or removed from the system at runtime. We have presented three different types of communication: RPC, ownership and topics, which are needed for different requirements of autonomous systems. In addition, we have expanded the classical understanding of topics to structure them into hierarchies.

We are firmly convinced that in a world where autonomous systems such as vehicles, robots, drones or machines are becoming increasingly common, new software architectures and models are also needed. These architectures and models must allow autonomous systems to receive updates and new functions at runtime without having to be installed in workshops by the manufacturer. In this paper we showed a possible way to lay the foundation for dynamic, self-adaptive and autonomous systems with the help of our middleware.

References

[1] SAE J 3016, "Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles," p. 30, 2016.

[2] C. Brink, E. Kamsties, M. Peters, and S. Sachweh, "On hardware variability and the relation to software variability," in Proceedings - 40th Euromicro Conference Series on Software Engineering and Advanced Applications, 2014, pp. 352–355.

[3] O. Oliinyk, K. Petersen, M. Schoelzke, M. Becker, and S. Schneickert, "Structuring automotive product lines and feature models: an exploratory study at Opel," Requirements Engineering, vol. 22, no. 1, 2017, pp. 105–135.

[4] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in Future of Software Engineering, 2007, pp. 55–71.

[5] M. Broy, "Challenges in automotive software engineering," in Proceedings of the 28th international conference on Software engineering, vol. 2006, 2006, pp. 33–42.

[6] K. Grimm, "Software Technology in an Automotive Company - Major Challenges," in Proceedings of the 25th International Conference on Software Engineering, 2003, pp. 498–503.

[7] B. Hardung, T. Kölzow, and A. Krüger, "Reuse of software in distributed embedded automotive systems," in Proceedings of the fourth ACM international conference on Embedded software - EMSOFT '04, 2004, p. 203.

[8] C. Krupitzer, F. M. Roth, S. Vansyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," Pervasive and Mobile Computing, vol. 17, no. PB, 2015, pp. 184–206.

[9] M. Bernard et al., Mehr Software (im) Wagen : Informations- und Kommunikationstechnik (IKT) als Motor der Elektromobilität der Zukunft *English Translation: More software in the car: information and communication technology as the motor of electromobility for the future*. ForTISS GmbH, 2011.

[10] P. Oreizy et al., "An architecture-based approach to self-adaptive software," IEEE Intelligent Systems and Their Applications, vol. 14, no. 3, 1999, pp. 54–62.

[11] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," Computer, vol. 36, no. 1, 2003, pp. 41–50.

[12] C. Szyperski, Component Software: Beyond Object-Oriented Programming (2nd Edition), 2nd ed. Addison-Wesley Professional, 2002.

[13] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry," Harvard Business School Technology & Operations Mgt. Unit Research Paper, 2007.

[14] B. Councill and G. T. Heineman, "Definition of a software component and its elements," Component-based software engineering: putting the pieces together, 2001, pp. 5–19.

[15] P. A. Bernstein, "Middleware: a model for distributed system services," Communications of the ACM, vol. 39, no. 2, 1996, pp. 86–98.

[16] N. Wang, D. C. Schmidt, and C. O'Ryan, "Overview of the CORBA Component Model: Component-based Software Engineering," in Component-based software engineering, G. T. Heineman and W. T. Councill, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 2001, pp. 557–571.

[17] Object Management Group - OMG, "CORBA Component Model Specification," 2006.

[18] F. Kon et al., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," IFIP/ACM International Conference on Distributed systems platforms, 2000, pp. 121–143.

[19] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture- Based Self-Adaptation with Reusable Infrastructure," in Proceedings of International Conference on Autonomic Computing, 2004, pp. 46–54.

[20] R. Anthony and C. Ekelin, "Policy-driven self-management for an automotive middleware," in Proceedings of the 1st International Workshop on Policy-Based Autonomic Computing (PBAC 2007), 2007, p. 7.

[21] B. Becker, H. Giese, S. Neumann, M. Schenck, and A. Treffer, "Model-based extension of AUTOSAR for architectural online reconfiguration," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 6002 LNCS, 2010, pp. 83–97.

[22] A. S. Tanenbaum and M. van Steen, Distributed systems: principles and paradigms. Prentice-Hall, 2007.

[23] A. Schill and T. Springer, Verteilte Systeme Grundlagen und Basistechnologien. Berlin; Heidelberg: Springer Vieweg, 2012.

[24] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture, A System of Patterns: Volume 1 (Wiley Software Patterns Series). Wiley, 2013.

[25] H. Klus and A. Rausch, "DAiSI A Component Model and Decentralized Configuration Mechanism for Dynamic Adaptive Systems," International Journal On Advances in Intelligent Systems, vol. 7, no. 3 and 4, 2014, pp. 27–36.

[26] H. Klus, D. Herrling, and A. Rausch, "Interface Roles for Dynamic Adaptive Systems," in Proceedings of the Seventh International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE), 2015, pp. 80–84.

[27] D. Herrling, A. Rausch, and K. Rehfeldt, "Extending Interface Roles to Account for Quality of Service Aspects in the DAiSI," International Journal on Advances in Software, vol. 9, no. 1 & 2, 2016, pp. 37–49.

[28] Y. Wang, D. Herrling, P. Stroganov, and A. Rausch, "Ontology-based automatic adaptation component interface in DAiSi," in Proceedings of the Eighth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2016), 2016, pp. 51–58.

[29] M. Quigley et al., "ROS: an open-source Robot Operating System," in ICRA Workshop on Open Source Software, vol. 3, 2009, p. 5.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1