MLCCA – Multi-Level Composability Check Architecture for Dependable Communication over Heterogeneous Networks

Michael Schulze, Georg Lukas
Department of Distributed Systems
University of Magdeburg Germany

E-mail: {mschulze, glukas}@ivs.cs.uni-magdeburg.de

Abstract

During the design of complex networked systems, it is cruical to ensure the composability of the deployed applications and network protocols. Special care has to be taken to provide non-functional requirements like bandwidth and latency. Existing solutions only tackle this problem during the design phase; later refactoring or added components are not covered, potentially causing QoS violations. We propose MLCCA, a multi-level architecture which complements the design-time composability checks with additional automatic checks performed at compile-time and at run-time. The required infrastructure is embedded into our communication middleware FAMOUSO, making it transparent to application developers. The architecture has been evaluated in a tele-operated mobile robot case study. If the QoS attributes could not be fulfilled due to refactoring or changed conditions, no communication was allowed by the middleware, ensuring that the application could enter a fail-safe state. No data was sent over insufficient channels. Thus, our combination of FAMOUSO and MLCCA enables the sustainable deployment of complex networked systems.

1 Introduction

Many complex systems with embedded components in the automotive field and industrial automation such as cars, autonomous vehicles and robots include multiple heterogeneous networks. These networks connect smart sensors and actuators and have specific attributes concerning bandwidth, delay, timeliness or dependability. To take an example from the automotive area, FlexRay [4], TTCAN [5] or multiple CAN buses [15] are currently used for time critical functions and demanding body electronics. The use of separate systems ensures local quality properties at adequate costs. However, firstly, this complicates the combination and exchange of information about network boundaries. This makes it hard to realise sensor sharing for high level services like driver assistance systems. Usually, address assignment is performed manually, requiring significant effort to synchronize the address mapping of different networks at the gateways. Secondly, to specify temporal properties of cross-network communication, no generic high level mechanisms are available. Instead, this has to be done explicitly at a low network implementation level, too.

In a multi-network system physical subnets are connected via gateways which route and filter information across network boundaries. The strength of this system structure, sometimes called a federated network, is the adaptation of the subnets to their specific needs and, from a safety point of view, the physical isolation between the networks concerning temporal and functional faults. As an alternative, integrated architectures have been proposed [16, 13]. Integrated architectures apply virtualisation techniques to emulate multiple networks on a single physical network. In that approach, a communication interface is provided to the application software which is equivalent to the interface in the respective sub networks. Thus, it is transparent for the application software that the underlying network is different and allows legacy communication software to be used unchanged. Beside other problems an integrated architecture is inappropriate in scenarios like cooperating mobile robots or tele-operating a mobile robot, where sensors and actors need a low-overhead communication bus, while tasks like fleet management require high-bandwidth Ethernet or Wireless LAN.

Our approach uses a federated architecture, allowing to preserve the advantages and physical separation of subnets. To tackle the problems of address assignment, cross-network gateways and filtering, we use our publish/subscibe middleware FAMOUSO. It hides the characteristics of the underlying networks by providing a common interface and a content-based

addressing scheme. Network identifiers are generated automatically on demand.

However, a straight-forward abstraction of different network protocols is not sufficient for applications with non-functional QoS or real-time requirements like a bounded latency or a certain guaranteed bandwidth. Such application end-to-end demands have to be considered, especially when crossing network borders. No communication link may be established over inadequate channels. Because a manual verification would be tedious and error-prone, we propose an automated approach to verify whether a potential network connection fulfills the application requirements. These composability checks are usually only performed during the design-time, leaving system reconfigurations after the initial design unverified. We are proposing MLCCA, the Multi-Level Composability Check Architecture, which extends these tests to the compile time and run time, closing the verification holes and allowing reliable end-to-end communication.

In section 2 we motivate the need for composability checks at different levels. Section 3 describes the communication middleware FAMOUSO, which we are using to hide the complexity of the checks from the application. The realization of MLCCA is presented in detail in section 4 and evaluated on a case study in section 5. Related work is regarded in section 6. Finally, the paper is concluded in section 7.

2 Multi-Level Composability Checks

In the design of complex systems it is very important to consider the requirements of the involved components with regard to their composability. Usually, this is done by specifying component attributes and performing manual or automatic composability checks.

It is also possible to specify communication attributes for components and network modules during the design time and to automatically verify the composability of those components with the desired networks. This can be made intuitive by graphical prototyping tools like Matlab Simulink or LabView, where data type safety is ensured by the design tool, disallowing connections between mismatching endpoints. In this work, we pick up the means of formal attributes to describe the QoS requirements of applications and the capabilities of network stacks, respectively. These attributes can be checked by a design tool, warning the user about QoS violations or even forbidding unsafe combinations.

However, it is not sufficient to implement these checks in the design tools. After the initial deployment of a multi-network system, it is possible that refactoring of the infrastructure takes place at the source code level, manually changing the auto-generated code or adding additional components. Such changes can lead to the invalidation of previous guarantees, causing component malfunctions. We propose a second level of checks at the compiler level to ease the network debugging and maintenance. When the source code is compiled for a certain component, it is possible to verify the QoS attributes again using compile time function evaluation and to generate compiler errors for invalid combinations. This is much more convenient than debugging invalid combinations in the field, especially when components are realized using embedded devices with limited debugging capabilities.

However, it is possible to combine different precompiled components on the same network, causing a run-time overload situation. To cope with this problem as well as to support network media with dynamic attributes (like wireless mesh networks with changing topologies), we propose a third level of checks performed at run-time of the network.

3 FAMOUSO

Our middleware FAMOUSO (**F**amily of **A**daptive **M**iddleware for autonom**OU**s **S**entient **O**bjects [19, 18, 17]) provides an event-based publish/subscribe communication over different network types. Objectives of FAMSOUSO are portability, adaptability, configurability and efficient resource usage to allow also the deployment on small resource-constrained embedded devices. The middleware supports a broad variety of different hardware platforms ranging from 8-bit micro-controllers up to 64-bit server systems and enables interaction over different communication media like CAN [15], 802.15.4 [21], AWDS [1] and UDP-multicast (Figure 1). Furthermore, FAMOUSO can be used from different programming languages (C/C++, Python, Java, .NET) as well as from engineering tools (Matlab/Simulink, Labview) simultaneously. Thus, the middleware enables the developers to individually choose their preferred combination of tools and languages.

In FAMOUSO the exchanged information is incorporated into events, which consist of three different parts:

1. a subject, represented by a 64 bit unique identifier (UID) that describes the content,
2. the content or payload itself (for instance the value of a distance measurement) and

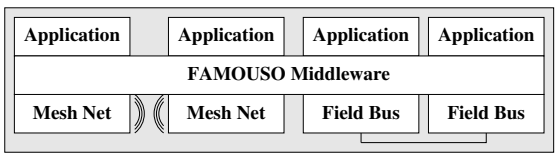

Figure 1. FAMOUSO middleware example

3. optional additional attributes (e. g. sensor position, context, time-stamp).

Subjects span a global address space, having a meaning across multiple networks. This global name space is exploited on gateway nodes for selective forwarding or filtering. For example, if a certain subject is not subscribed outside a subnet, it is not propagated by the respective gateway.

From the perspective of many applications, it is sufficient to define a subject and the according payload format to establish communication. However, networked embedded applications also often require non-functional properties like real-time delivery or reliability. To allow the definition of these properties, we introduce the notion of event channels. An event channel is a programming abstraction used to define the application requirements as well as to convey actual events between the application and the middleware. There are two distinct types of event channels used for publishing and subscribing information, which differ in the supported callback functions. The event channel definition resembles the event, also consisting of three parts:

1. the *subject* corresponds to the conveyed events,
2. *attributes* describe dissemination parameters like period, deadline, latency, reliability or bandwidth requirements,
3. *callbacks* specify which application functions have to be called for event processing. For the subscriber, there is a *notification handler* which is called on successful reception of events and an *exception handler* for cases when the specified requirements have been violated. For publishers only the *exception handler* exists. It is called if an event could not be published according to the specification.

The specification of QoS parameters for event channels allows the middleware to check whether demanded QoS parameters are admissible on the one hand, and on the other hand they are used to reserve local and/or network resources. During deployment, an instance of the FAMOUSO *event channel handler* must be running on every node. It integrates into the network, provides gateway services between subnets, and establishes the event channels requested by applications.

Our FAMOUSO middleware is a three-layered architecture. At the top-level, the event layer handles event channels and serves as an interface to the application. The next level, the abstract network layer encapsulates functionality that is independent of a specific network. For example, this layer offers a fragmentation protocol that adapts events to the maximum transportable payload of the lowest layer, the specific network layer. This layer contains the network specific aspects like configuration protocols or information how messages have to be transmitted. For a detailed description of the FAMOUSO architecture the reader is referred to [18, 17] as it is beyond the scope of this paper.

4 MLCCA – Realization

We extend the FAMOUSO middleware with an additional component, the Multi-Level Composability Check Architecture MLCCA. This allows us to hide the complexity of the composability checks on all layers behind a defined event+attribute interface. There is no change for legacy applications, while QoS applications are supported on all network layers specifying the supported attributes.

Still, two things must be considered for making the composability checking effective. First, the formal attribute definitions must be declared in a format which allows to check them during design-, compile- and run-time. To achieve this, we propose to use the XML format with the event channel syntax defined by [14] (example in figure 2). Second, the plethora of possible attributes must be standardized to allow matching applications against network layer modules.

However, the standardization of attributes is not trivial, because there are multiple ways to specify the same aspects of a network medium. To name an example, for periodic events the *deadline* and *period* attributes usually have the same meaning, requiring a conversion function. There is also a connection between *packet loss*, an attribute specifying the per-packet probability of loss, and *omission degree* which means the probability of a number of consecutive losses. However it is not trivial to convert between these two attributes, meaning that a network layer module should specify both of them, and the application can choose to use one or both according to its requirements.

We propose the following attributes to be specifiable by applications and network layers: *period* (packet rate and deadline for periodic applications), *latency* (the end-to-end delivery time for a packet), *jitter* (for inter-packet arrival times), *bandwidth* (specified in bytes/second), *packet loss* and *omission degree* (according to the previous paragraph).

In the following, we use the example of an emergency switch to demonstrate the principal approach across the layers. The emergency switch is one component of our mobile robot in figure 6. It is specified as a periodic application sending heart-beat signals with a *period* of $50\,ms$ with a maximum *omission degree* of $2\,packets$, resulting in a worst-case reaction time of $150\,ms$ after the switch is activated. These two constraints must be met by the network layer to allow successful deployment.

```
<EventChannel>
  <SubjectUID>0xa4efeecc3210b497</SubjectUID>
    <Description>emergency switch</Description>
    ...
    <Attributes>
      <Attribute>
        <Name>Omission</Name>
        <Value>2</Value>
      </Attribute>
      <Attribute>
        <Name>Period</Name>
        <Dimension>
          <SIUnit><Seconds>1</Seconds></SIUnit>
          <Magnitude>-3</Magnitude>
        </Dimension>
        <Value>50</Value>
      </Attribute>
    </Attributes>
    ...
  </EventChannel>
```

Figure 2. XML event channel attribute descriptions

4.1 Design Time

Complex systems like cars or mobile robots consist of many components and different sub-nets. For both the components and the network protocols their associated non-functional properties like bandwidth, latency and packet loss have to be defined. We use XML based object descriptions to specify these attributes in a way accessible to a variety of tools. Figure 2 shows an excerpt of the XML output generated by our `DescriptionCreator` design tool for the emergency switch described above. The example describes the QoS attributes of the emergency switch event channel and defines the omission and period QoS parameters for event transmissions.

One way to perform the composability check on a system consisting of different application and networking components is our `SystemChecker` tool. It works on the XML descriptions of all involved components and a description of the middleware configuration, which contains information about how the components are interconnected and is written in XML as well. It performs two types of checks by deploying XSL Transformations (XSLT):

completeness of specification, verifying that for every requested subscription there is also a publisher.

correctness, checking whether the QoS attributes match between publishers and subscribers and if they can be fulfilled by the network connection between the participants.

The early detection of mismatching configurations is especially desirable when developing complex systems consisting of many components. In such systems it is hardly possible to perform a manual verification of all attributes of every component. This is even more important when embedded systems are deployed, which do not provide suitable output facilities to signal problems at run-time.

In analogy to rapid prototyping software with automatic code generation, we developed a tool that generates C++ code from the XML based attribute descriptions. The created code contains all specified channels and application callback stubs, allowing the developer to concentrate on the program logic.

For a detailed discussion of the XML description format and the checking mechanisms the reader is referred to [9] and [8].

4.2 Compile Time

The second stage within our MLCCA is the compile-time checking. At first, this additional check might seem superfluous because potential errors would be uncovered at design-time. However, many systems undergo refactoring after a certain deployment time, often without using the original design tool. Such a refactoring can change the network connection between two components, invalidating their QoS attributes. In addition, it is possible to write applications from scratch without using rapid prototyping tools. These applications need the same means to detect unusable component combinations without having the help of a design tool. Again we follow the rule to detect problems as early as possible in the development process.

```
1  #include "Period.h"
2  #include "Omission.h"
3  #include "AttributeList.h"
4
5  // emergency switch event channel specification
6  struct EmergencySwitchSpec {
7      typedef AttributeList<
8                          Period<50>,
9                          Omission<2>
10                     >::type attributes;
11 };
```

```
12 #include "CheckAttributeAgainstAttributeList.h"
13 // check within a network layer component
14 typedef CheckAttributeAgainstAttributeList<
15                     EmergencySwitchSpec::attributes,
16                     Omission<5>
17                 >::type test;
```

Figure 3. C++ code that defines and checks QoS parameters

The challenge in providing compile-time checks for attribute constraints is that the application source code is syntactically correct, whereas the compiler shall refuse it with an expressive error message when the constraints are not met. Source code which contains matching attributes shall compile without any warnings.

Figure 3 shows source code generated from the XML description of the emergency switch event chan-

nel (lines 1-11) and the checking definition (line 12-17). This code deploys reusable templates for generic attributes like *Period* and *Omission*, which are parameterized by their respective values. The constraint checking logic is implemented using Template Meta Programming (TMP), allowing to use the compiler to interpret the attribute definitions and checks. Hereby, the compiler first instantiates the templates, replacing them with accordingly generated statements. In a second step, the post-processed source is compiled into machine code.

Our approach is embedded in the template instantiation phase, where the attribute types are matched against each other and their corresponding values are compared. Typically, compiler error messages caused during template instantiation are very verbose. To reduce confusion, we generate an additional customized error message that describes the encountered problem and appears at the end of the compiler output. Thus the developer is able to perceive the reason of the misconfiguration. The source code in Figure 3 generates such an error message due to the mismatch between the required *omission* of 2 and the provided *omission* of 5. Figure 4 depicts the error message, with the customized `ATTRIBUTES_MISMATCH` error in line 3, which is caused by line 17 of the mentioned source code. The error message clearly states that an attribute mismatch was caused between Omission$< 5u >$ and Omission$< 2u >$.

The code generation facilities during the template instantiation phase also allow us to automatically generate run-time accessible representations of the specified attributes in the ASN.1 notation [3]. These representations are used in the run-time checking system.

```
1 CheckAttributeAgainstAttributeList.h:78:
      instantiated from
      CheckAttributeAgainstAttributeList<list2<
      Period<50u>, Omission<2u> >, Omission<5u> >
2 main.cc:17:   instantiated from here
3 CheckAttributeAgainstAttributeList.h:60: error:
      no matching function for call to
      assertion_failed(*** (TestAttributes<
      Omission<5u>, deref<l_iter<list1<Omission<2
      u> > > > >::ATTRIBUTES_MISMATCH::***)(
      Omission<5u>, Omission<2u>))
```

Figure 4. Customized error message generation if QoS can not be mapped

It should be noted that the compile time checking code is not specific to a certain compiler but is absolutely C++ standard complaint, meaning that any C++ standard compliant compiler can be used.

4.3 Run Time

After a multi-component system which has passed the design- and compile-time checks is changed, it is still possible that the required attributes are violated. This can happen when two CAN-bus based components are separated onto different buses connected via an unreliable bridge, or when additional components are added, overloading the network medium. Because such reconfigurations can happen without using the design tool and the components are already finished and their program code compiled and deployed, the final composability check can only happen during runtime.

```
1: for req in subscriber.attrs ∪ publisher.attrs do
2:   avail := netlayer.attrs[req.name]
3:   if avail is null or
     avail.value < req.value then
4:     return "Error: req.name not fulfilled"
5:   end if
6: end for
7: return success
```

Figure 5. Run-time attribute verification

We embed the run-time checks into the abstract network layer (ANL) of the FAMOUSO middleware. This allows to hide the complexity both from the application and the network layer programmers, who only have to provide the required and supported attributes, respectively. These attributes can be autogenerated by the C++ compiler from the source code attribute specification, or specified manually. The ASN.1 syntax used in both cases allows to implement the checks even on low-end micro-controllers.

The run-time checks are performed for event channels as well as on gateways (algorithm pseudo-code in figure 5). The requirements provided by the subscriber and the publisher have to be matched to the ones supported by the network layer (or both network layers on a gateway). The check is performed by creating a union of the subscriber and publisher attribute lists and iterating over it (line 1). For each of the requested attributes, the according attribute of the network layer is looked up (line 2). If it is not specified or it does not provide the required quality (line 3), an error is returned. In that case, the subject is not subscribed and no event data is sent through the network, preserving bandwidth. The algorithm only succeeds (line 7) if all required attributes are defined by the network layer and provide sufficient quality. The consideration of already-existing connections has been removed from the pseudo-code for clarity, the actual algorithm is more complex.

For the emergency switch this means, if an unsuitable network bridge is deployed between the publisher (switch) and subscriber (actors), the heart-beat signals will not be relayed, preventing any activity which could cause harm. Critical applications should be designed to enter a fail-safe state when no communication is possible, e.g. by using the exception handler.

Some network layers allow varying quality depending on their current configuration. For example, the AWDS routing protocol provides higher bandwidth over short links because multi-hop connections accumulate network load. Also, the bandwidth requirements of the already-established event channels and other connections have to be considered [6].

For such dynamic scenarios the decision can not be made in the ANL, it has to be performed in the network layer on a case-by-case basis. Here, the application attributes and the desired end-points (publisher, subscriber or gateways) are passed down to the network layer, which calculates if the attributes can be granted and makes appropriate bandwidth/resource reservations. A dynamic network layer should define its static attributes based on a best-case situation, allowing to detect unrealistic combination at design- and compile-time. By also specifying a *dynamic* attribute, an appropriate display is possible in the design tools. The ANL can use this attribute to deploy the network-layer specific composability check function. This is used to make the final decision during run-time.

The combination of static checks (based on the pre-defined network attributes) and dynamic checks allows a reliable verification whether application demands can be satisfied at the network layer. Detected violations are communicated to the application's exception handler.

5 Case Study

To demonstrate the applicability of the multi-level composability check architecture, we show its actual implementation as part of a mobile robot case study. The mobile robot embeds a control laptop, different sensors as well as actors. It provides two video cameras for tele-operation and a built-in emergency switch (figure 6). There is a CAN bus interconnecting the laptop with the sensors, actors and the emergency switch. If the switch is triggered, the actors (motors) have to stop to prevent further damage. The cameras are attached via USB to the laptop, which acts as a gateway into the AWDS multi-hop network. A stationary PC is used by a worker to tele-operate the robot (as depicted in figure 7). The network connection is transparent to the FAMOUSO based control application, which runs on the PC and directs commands to the actors behind the laptop AWDS-CAN gateway. The sensors publish their values, which are transported by the gateway to the control application transparently (figure 8).

We will demonstrate the implementation of MLCCA based on the emergency switch and supplement it with a bandwidth-demanding application, the live video streaming.

The conceptual description of the emergency

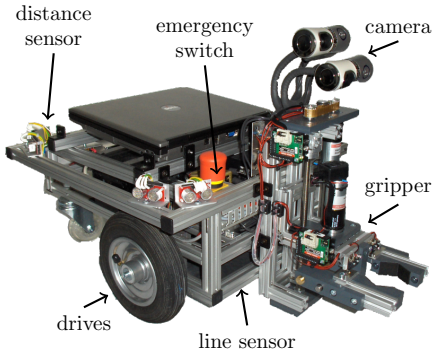

Figure 6. Volksbot platform

switch made in section 4 applies here as well. However, attempts to move the emergency switch from the robot to a stationary position near to the control PC should fail, because AWDS can not guarantee an omission degree less than 5, independent of its network topology. Thus, such a change would cause an error message in the design tool, if performed there. The generated code would also refuse compilation when replacing the CAN network layer with the AWDS instance manually, giving the error message in figure 4. When the switch is moved from the robot-local CAN bus to a second CAN bus behind the control PC, this change is completely transparent to both endpoints. It would not be detected by the compiler, because the CAN bus directly attached to the switch supports the required parameters. However, due to the run-time check performed on the gateway, the attribute mismatch would be detected and the heart-beat messages would be filtered. Without these messages, the motor controllers will not perform any action, staying in a fail-safe mode.

A more complex issue is presented by the live video stream. In addition to a latency requirement

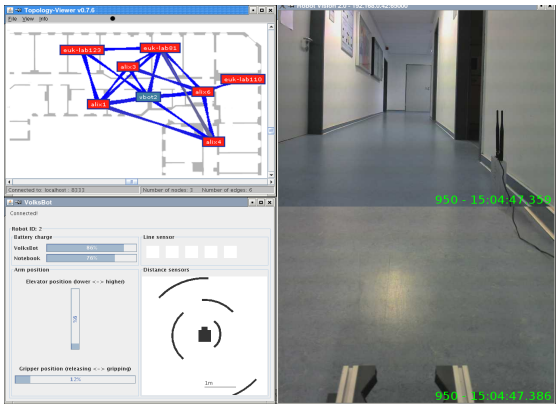

Figure 7. Tele-operation screen (left: network topology above distance sensors; right: live video)

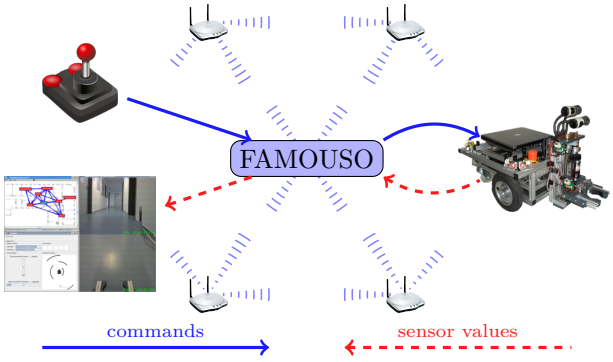

Figure 8. Case study: tele-operating a mobile robot

of $200ms$ for every frame, it imposes a high bandwidth demand. Depending on the video quality, either $100KByte/s$ or $400KByte/s$ have to be provided end-to-end. The sensors' CAN bus is obviously not suitable for this demand. However, even the AWDS network can not guarantee a successful transmission of the high-quality stream. As this question can not be answered during design- or compile-time (both the latency and the bandwidth demands can be fulfilled by AWDS in some topologies), these checks must pass. Only the run-time check can discriminate if the request can be granted or not. Here, we follow an adaptive strategy: the control PC subscribes to the high-quality stream. If this stream can be delivered, it is automatically routed from the robot. If a delivery is declined by the AWDS network layer, the application has to switch to the low-quality stream. When this is declined too, no tele-operation is possible any more.

The bandwidth-check is performed using a medium access time accumulation formula: every link of a multi-hop connection requires a certain amount of time to transfer the desired amount of data. This time is reciprocal to the link data rate, which the hardware adapts to the medium conditions. The available bandwidth on the link is the reciprocal of the sum of the times for all links on the path. If it is higher than the application requirement, the test is passed.

In a simplified example, consider the robot laptop A being connected via a router B to the control PC C. The link data rates are $R_{A\leftrightarrow B} = 24\text{MBit}/s$ and $R_{B\leftrightarrow C} = 11MBit/s$. The maximum throughput is then calculated with:

$$\frac{1}{R_{A\to B\to C}} = \frac{1}{R_{A\leftrightarrow B}} + \frac{1}{R_{B\leftrightarrow C}} \tag{1}$$

$$R_{A\to B\to C} = \frac{1}{\frac{1}{24MBit/s} + \frac{1}{11MBit/s}} \tag{2}$$

$$R_{A\to B\to C} \approx 7.54MBit/s \tag{3}$$

This is more than required for the high-quality video stream ($400KByte/s = 3.2MBit/s$) so it will be accepted by the network layer. However, if the data rate on one of the links drops or another station is introduced in between due to the robots mobility, the network layer re-evaluates the available end-to-end bandwidth and may disable the packet delivery, calling the subscriber's exception handler.

Of course, the realization of bandwidth guarantees requires an admission control scheme to prevent oversaturation of the medium. A possible mechanism has been presented in [6].

6 Related Works

Related work in the area of describing properties of components are manifold. Systems like WSDL [11], UPNP [7] or Jini [20] are only some representatives of systems that exploit descriptions for discovering services dynamically. However, due to lack of QoS attributes they are not suitable for our purpose and their complexity is also too heavyweight for use in networked embedded systems. Description systems like CANopen [2] or IEEE 1451 [10] are used in the industrial as well as automotive area to tackle this problem. However, most of the descriptions there are expressed in a very special and often proprietary way. Furthermore, the component descriptions explicitly define the underlying network medium instead of only specifying their requirements, making it infeasible to combine them with other network modules. Thus, these standards do not support interactions between different communication networks. Finally, in most cases the descriptions are not available in a form which can be automatically processed by checking and code generation tools.

Compilation-time checks of non-functional parameters comparable to the presented approach do not exist to the best of the authors' knowledge. Other compile time checks like type conformance or type conversion tests are usually automatically performed by the compiler. By doing additional checks on the attribute types and the QoS values and by automatically generating attribute mismatch error messages, we go a step further.

The most commonly used runtime system with support for attribute checking is CORBA [12]. The CORBA standard allows specification of QoS parameters for its interfaces. Attributes like reliable transport or deadlines can be requested, and the system only allows communication between participants if their specifications match. When an established specification is violated, the applications are notified and have opportunity to adapt to the new situation, as far as possible. However, the CORBA framework is not designed to scale down to low-end micro-controllers and their communication buses.

7 Conclusion

In this paper, we have presented MLCCA, a Multi-Level Composability Check Architecture. This architecture allows applications to specify non-functional requirements like QoS or real-time properties, which are automatically checked during the design-, compilation- and run-time of a project. With our approach, the application-specified requirements are never violated. If they can not be fulfilled by the network, no communication channel is established at all, preventing later damage. We have shown that MLCCA hides the complexity of the composability checking into the FAMOUSO middleware, providing a clean and easy-to-use interface to applications. This combination of communication middleware and enforcement of QoS requirements provides an ideal fundament for many applications. Both the FAMOUSO middleware and MLCCA are designed to scale from 8-bit micro-controllers up to large server systems, allowing simultaneous deployment of devices from the whole range. Different network technologies can be used, ranging from CAN to the IP-based UDP-multicast or Wireless Mesh Networks.

A case study has demonstrated how MLCCA can be successfully deployed to support the tele-operation of mobile robots. However, it has also shown an important detail which has to be handled in future work. Applications like video streaming, which can adapt their requirements to the available resources, have to do so manually in the current implementation. Future work will cover the support of *attribute sets*, where the application author specifies several alternative sets of attributes, each with a priority value. The run-time system then will have to perform a series of checks, trying to allow the set with the highest possible priority, and automatically switching between the sets according to the current situation.

References

[1] AWDS project. `http://awds.berlios.de`, 2009.

[2] CiA. *CiA 306 DS V1.3: Electronic Data Sheet Specification for CANopen*. CiA, CANopen, January 2005.

[3] O. Dubuisson and P. Fouquart. *ASN.1: communication between heterogeneous systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[4] FlexRay Consortium. *FlexRay Communications System Protocol Specification Version 2.1 Revision A*. 2005.

[5] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time Triggered Communication on CAN (Time Triggered CAN-TTCAN). *7th international CAN Conference*, 2000.

[6] A. Herms, E. Nett, and S. Schemmer. Real-time mesh networks for industrial applications. In *Proceedings of 17th International Federation of Automatic Control World Congress (IFAC'08)*, Seoul, Korea, July 6–11 2008.

[7] M. Jeronimo and J. Weast. *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play*. Intel Press, 2003.

[8] J. Kaiser and H. Piontek. CODES: Supporting the development process in a publish/subscribe system. In *Proceedings of the fourth Workshop on Intelligent Solutions in Embedded Systems WISES 06*, pages 1–12, Vienna, 30. June 2006. ISBN: 3-902463-06-6.

[9] J. Kaiser, S. Zug, M. Schulze, and H. Piontek. Exploiting self-descriptions for checking interoperations between embedded components. In *International Workshop on Dependable Network Computing and Mobile Systems (DNCMS 08)*, pages 41–45, Napoli, October 2008.

[10] K. Lee. Ieee 1451: A standard in support of smart transducer networking. In *Proc. 17th IEEE Instrumentation and Measurement Technology Conference IMTC 2000*, volume 2, pages 525–528 vol.2, 2000.

[11] J. J. Moreau, R. Chinnici, A. Ryman, and S. Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. Candidate recommendation, W3C, March 2006.

[12] OMG. *Data Distribution Service for Real-time Systems Version 1.2*. Object Managment Group, 1. January 2007.

[13] P. Peti, R. Obermaisser, F. Tagliabo, A. Marino, and S. Cerchio. An Integrated Architecture for Future Car Generations. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 2–13, Washington, DC, USA, 2005. IEEE Computer Society.

[14] H. Piontek. *Selfdescription mechanisms for embedded components in cooperative systems*. PhD thesis, University of Ulm, 2007.

[15] Robert Bosch GmbH. *CAN Specification Version 2.0*. 1991.

[16] J. Rushby. Partitioning for Avionics Architectures:Requirements, Mechanisms, and Assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.

[17] M. Schulze. FAMOUSO project website. online, `http://famouso.sourceforge.net`, 2009.

[18] M. Schulze. Famouso eine adaptierbare publish/-subscribe middleware für ressourcenbeschränkte systeme. *Electronic Communications of the EASST (ISSN: 1863-2122)*, 17, 2009. Workshops der Wissenschaftlichen Konferenz Kommunikation in Verteilten Systemen 2009 (WowKiVS 2009).

[19] M. Schulze and S. Zug. Exploiting the FAMOUSO Middleware in Multi-Robot Application Development with Matlab/Simulink. In *In Proceedings of the ACM/IFIP/USENIX 9th Internatinal Middleware Conference (Middleware2008)*, Leuven, Belgium, 1-5 December 2008.

[20] J. Waldo. *The Jini Specifications*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[21] ZigBee Alliance. *ZigBee Specification - IEEE 802.15.4*. 2003.