Proceedings of MESA07
2007 ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications
September 04-07, 2007, Las Vegas, Nevada, USA

DETC2007-34558

BUILDING COMPONENT BASED SOFTWARE CONNECTORS FOR COMMUNICATION MIDDLEWARE IN DISTRIBUTED EMBEDDED SYSTEMS

Dietmar Schreiner
University of Applied Sciences Technikum Vienna
Department of Embedded Systems
Höchstädtplatz 5, A-1200 Vienna, Austria

Vienna University of Technology
Distributed Systems Group
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
d.schreiner@infosys.tuwien.ac.at

Karl M. Göschka
Vienna University of Technology
Distributed Systems Group
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
k.goeschka@infosys.tuwien.ac.at

ABSTRACT

Interaction in distributed component based software-architectures can become a rather complex and error prone issue. As it is good practice to keep application concerns separated from infrastructural ones, component based applications typically rely on communication middleware to cope with matters of distribution and heterogeneity. Unfortunately, generic middleware tends to be monolithic, heavyweight software, which is unacceptable in resource constrained embedded systems. Communication middleware for distributed embedded systems has to be custom tailored to the application's interaction needs and therefore shall be as lightweight as possible. By applying the component paradigm to the communication middleware, a practical methodology can be defined, that allows the middleware's automatic generation from the application's architectural models and structural designs of explicit component connectors with a well defined set of prefabricated basic building blocks—so called communication primitives. This paper contributes by specifying the most common structural designs for explicit connectors within the automotive domain and thereby, in addition identifies a set of classes of automotive communication primitives. Thus this paper provides the sound foundation for automatic, model driven middleware synthesis by specifying all necessary basic modules.

1 INTRODUCTION

Driven by market demands the complexity of embedded systems applications has dramatically increased. Applications are no longer simple programs executed on one single electronic control unit (ECU), but are highly distributed, heterogeneous and often safety or mission critical software systems that have to be small, efficient and extremely reliable. State of the art automotive systems contain more than 70 electronic control units, typically connected by up to 10 diverse bus systems [1]. In addition automotive embedded systems provide very limited resources, as they are subject to mass production and therefore have to be as cheap as possible. This evolution of the embedded systems domain and the dramatic increase in software complexity consequently led to a longer time to market, higher development costs, and to an increasing number of erroneously deployed software. To overcome these problems and to master the process of cost-efficient software development for reliable distributed embedded systems, various existing approaches from classical software engineering have been adopted for the embedded systems domain.

One of these paradigms is that of component based software engineering (CBSE). Within the automotive domain, CBSE is defined to be the development paradigm for applications following the *Automotive Open System Architecture* (AUTOSAR) standard [2]. AUTOSAR is specified by a consortium of major auto-

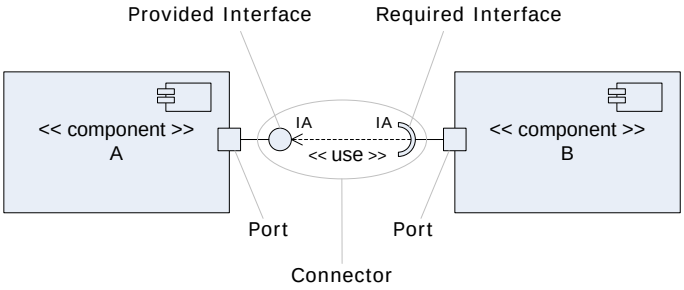


Figure 1: UML 2 Composition Sample

motive manufacturers and suppliers.

1.1 Component Based Software Engineering

In CBSE applications are built by assembling small trusted units of execution [3, 4], the components. These components interact only by their well defined interfaces according to contractual guarantees [5] and strictly contain no other external dependencies. Components apply to a component model, so they adhere to a composition and interaction standard [6] and can be independently deployed and composed without modification. As result components are well suited for reuse [7] and third-party composition.

Figure 1 shows a composition of two components in UML 2.0 notation [8]: Component *A* provides services by a provided interface of type *IA*, denoted as ball, while component *B* requires at least one service from that interface, thus it comprises *IA* as required interface denoted as socket. By associating *B*'s required interface and *A*'s *provided interface* (in UML 2 this can be done by joining the ball and the socket, or by drawing an arrow, representing a usage relation, from the required to the provided interface) the components are connected and hence may interact at runtime. Interfaces may be exposed by the component itself or by dedicated points of interaction, the port (as shown in Figure 1). Ports are typically used to (i) model distinct states of a component's interfaces, (ii) to express semantic relations between them or (iii) to provide a higher-level of abstraction for points of interaction. Interface states bound to ports play an important role in describing and verifying runtime behavior of composite structures [9], logic grouping is applied to increase a model's comprehensibility and higher-level abstraction is used within meta-models like the structural designs proposed within this paper. The connection between components is called connector. In contrary to component models like OMG's CORBA Component Model (CCM) [10] or Microsoft's COM [11], we consider a connector to be a first class architectural entity that represent interaction specific attributes and implementations in an explicit way [12]. Explicit connectors are composite architectures that describe idiomatic patterns of functionality associated with different architectural styles, as discussed in [13]. The building blocks of explicit connectors are so called communication primitives that are classified by their role.

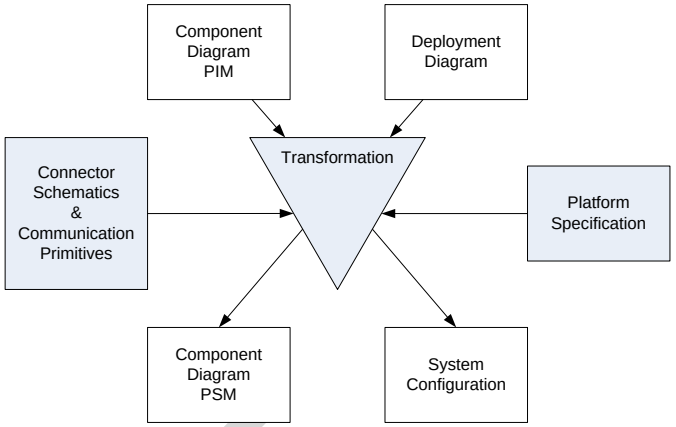


Figure 2: Model Driven Development Process

1.2 Model Driven Software Development

Component based software engineering aims at a significant reduction of time to market and development costs but also at an increase of software quality. When developing component based software, a proper engineering methodology is mandatory to reach that goal of better and cheaper software.

Model driven development (MDD) turns out to be a powerful engineering process when dealing with software components. In MDD different abstractions to the application and its context, so called views, are specified in various application and system models. They are used to express sundry aspects of an application in order to obtain a coherent total representation of the software's characteristic. By applying model transformations, models can be transformed into new, typically more specific ones. In addition, transformations can be used to extract model inherent information from source models. Figure 2 gives an example of a model transformation that is used to inject connector artifacts into UML component models: The application's architecture, the composition of its components, is specified within a platform independent model (PIM), to be more precise within a composition model. Composition models in UML 2.0 are expressed within component diagrams and in composite structure diagrams. The application's deployment, including the system's relevant physical structure, is described in a deployment model, specified within a deployment diagram. Using these source models and a target platform specification, the model transformation extracts information about the connectors' deployment and platform specific properties and as result generates a platform spe-

cific (PSM) composition model of the application, that later on can be transformed into ECU specific source code and system configuration data.

To specify models within this paper we will use OMG's Unified Modeling Language (UML 2.0) [8]. It provides a useful set of predefined classifiers that can easily be extended to meet domain specific requirements of architectural modeling [14]. We use component diagrams, containing components, explicit connectors and contracts, to denote component architectures, and composite structure diagrams, containing communication primitives and port connectors to specify connector schematics.

1.3 Motivation and Contribution

At runtime, components interact by their connected interfaces. Interaction implies inter-component communication that strongly depends on the physical system structure and the components' deployment scenario. In distributed, heterogeneous systems the process of interaction can become rather complex and difficult to handle. On this account, component based architectures typically utilize communication middleware to cope with the process of distributed interaction in a transparent way. Application components directly interact (at least seemingly), irrespective of their deployment scenario via this middleware. Thus they don't have to provide any implementation for means of distributed interaction. That keeps them focused on their actual purpose thereby staying small in size and less error prone. Due to the fact, that generic middleware, like e.g. middleware for the CORBA Component Model, has to provide implementations for all types of interaction that could occur within any exerting component architecture, this type of middleware typically tends to be heavy weight, monolithic software.

Embedded systems inherently lack on resources, so generic middleware can not be put to use. However, as the component paradigm is well suited for the embedded systems domain, and usage of communication middleware is a key concept within CBSE to keep application components small and reliable, the demand for light weight, optimized, yet custom tailored middleware arises.

Our approach aims at applying the component paradigm to the component middleware itself, thus gaining all advantages of CBSE for its middleware. As described in [15] custom tailored middleware can automatically be synthesized from a component architecture's composition and deployment specification by transforming its explicit connectors. The transformation substitutes model level connectors by appropriate composite structures that are assembled from prefabricated, component like, communication primitives according to the connector's functionality and well defined construction schematics.

The contribution of this paper is twofold. First, we identify two basic classes of explicit connectors according to the AUTOSAR standard and provide structural designs—connector schematics—for each of them. Second, we identify classes of basic communication primitives that can be plugged together in accordance to the provided schematics to form connector implementations for any required type of interaction. The presented schematics and the identified communication primitives provide the vital foundation for a sound synthesis of application specific, custom tailored middleware, supporting CBSE in automotive embedded systems but can easily be adopted to match other domains of distributed embedded systems.

1.4 Overview

This paper is structured as follows: Section 2 gives a short introduction to the basic concepts of distributed interaction within component based architectures. It defines classes of interaction—local, inter process, and remote—and types of communication primitives—active and passive ones. In Section 3 the concept of interaction patterns is summarized and two basic ones—Sender-Receiver and Client-Server—are described in detail. The structural templates—connector schematics—for both interaction types are provided in Section 3.1 and 3.2. In addition, classes of building blocks are specified within these sections. Finally we describe how our approach was applied to a typical embedded systems application from the automotive domain in Section 4 and conclude our work in Section 5.

2 Communication Primitives

Any communication within a computer system relies on one of two basic mechanisms: (i) asynchronous message passing that is typically used to transport information on bus systems and (ii) shared memory access that can yet be found in local procedure calls, sharing stack memory for passed parameters.

Besides, the process of interaction is characterized by coordination of the participants' flow of control (e.g., sequential or parallel execution), so the run-time behavior of the communicating entities. In our approach, the implementation of the whole process of interaction is located within the explicit connectors that are assembled from communication primitives, the basic building blocks of communication infrastructure. Communication primitives are alike application components, they mainly differ by their life-cycle. In contrary to application components, communication primitives do not exist within platform independent composition specifications, but are created during model transformation.

By taking the flow of control within a connector into consideration, we distinguish two types of communication primitives:

Passive Communication Primitives: Communication primitives of this type are executed within the thread of execution of the client, that is requesting their service. They do not provide infrastructural interfaces for discrete execution, but may require other infrastructural services, like time service

	Interaction	Connector Name	ECU	Address space
IPC	local	Intra Process Connector	colocated	colocated
XPC	inter process	Extra Process Connector	colocated	distributed
XNC	remote	Extra Node Connector	distributed	distributed

Table 1: Interaction Classes by Component Location

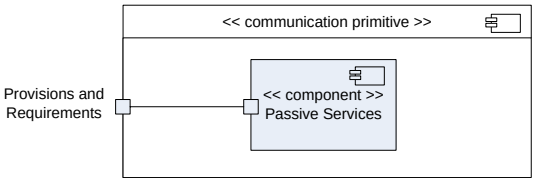


Figure 3: Passive Communication Primitive

or logging facilities. Figure 3 shows the composite structure of a passive primitive.

Active Communication Primitives: In contrary, active communication primitives own at least one thread of execution, thus have to provide at least one interface for activation by infrastructural services, like an operating system scheduler or an interrupt service routine (ISR). As shown in Figure 4, active communication primitives may contain passive sub-primitives, represented by the element labeled *Passive Services*. These passive sub-primitives again are executed within a client's thread of execution, and are exposed via the *Provisions* port. Those sub-primitives, containing services that are executed within the primitive's own thread of execution, are represented by the element labeled *Active Services*. They may naturally require and execute services of the passive sub-primitives by internal ports as well as of other (external) communication primitives via the *Requirements* port. Active primitives may provide services that operate asynchronous to a client's thread of execution. Thus the client has to provide a call-back interface for reception of results, or may pull results on its own by accessing the *Provisions* port.

It is obvious that the complexity of the overall process of interaction of composed components can vary from "low" in locally deployed architectures, to "high" in distributed heterogeneous systems. By taking this fact into account, three interaction classes, determined by the components' physical location at runtime, can be identified. Connected components may reside (i) within the same address space, (ii) in separated address spaces within the same ECU or (iii) within different ECUs. As denoted in Table 1, connectors between objects within the same address

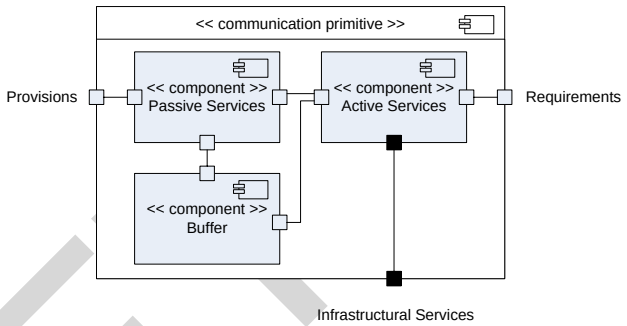


Figure 4: Active Communication Primitive

space are called *intra process connectors (IPCs)*, those between different address spaces but on the same ECU are called *extra process connectors (XPCs)* and those between different ECUs (which are also ones between different address spaces) are called *extra node connectors (XNCs)*.

3 Interaction Patterns for Explicit Connectors

In accordance to prevailing standards for embedded (automotive) communication middleware, we map interaction, occurring within distributed component based applications, onto two basic interaction patterns: (i) the client-server pattern and (ii) the sender-receiver pattern. Within this section we will provide schematics—structural designs—for explicit connectors implementing these patterns. The schematics are templates for the construction of the connectors, so the denoted model elements are meant to be meta-elements, placeholders for any real model element like components or connectors.

The placeholder for components is depicted as so called part, labeled with the components role and stereotyped as ≪ communication primitive ≫. In contradiction to the classification of application components, communication primitives are not classified strictly by their interfaces, but by their semantic functionality, so by their role within the schematic. Therefore both, active and passive versions of primitives, can be plugged in at the same place within the structural designs, assuming the connected primitives' interfaces match. If a part's functionality

is not required for a specific connector, a dummy primitive is plugged in, instead of a full-fledged one. These dummies satisfy the architecture's interface requirements, but provide only empty implementations or service indirections, if the primitive acts as connection between two other ones. During model optimization dummies may be eliminated, to reduce the connectors' size and increase their run-time performance.

Communication primitives are connected by "wiring" their associated ports. To keep the schematics as simple and flexible as possible, the wire is used as abstraction for the set of all explicit connectors between the primitives' connected interfaces. Wires between communication primitives are only valid, if the connected ports contain compatible, connectable interfaces. If the represented set is allowed to be empty—the connection is optional—the connector is drawn as dashed line. This is the case, if both of the assembled communication primitives do not expose interfaces that have to be connected.

3.1 Sender-Receiver Interaction Pattern

The first interaction pattern to specify is that of sender-receiver interaction, as client-server interaction in distributed systems can be built upon it.

In sender-receiver interaction a sender component emits information according to its interface specification. This information is received by one or more receiver components. Connectors, implementing the sender-receiver interaction pattern, provide means of non-blocking, one-to-many and many-to-one information distribution. Sender-receiver interaction is typically one-way communication. However, in dependable systems it is often feasible to implement communication with bidirectional data flow, to provide information (e.g., receipts and acknowledgements) about the communication status. Information, transmitted for interaction, can be processed in two different ways within the receiver. Therefore two types of sender-receiver interaction have to be distinguished:

State Transmission: The main purpose of this interaction type is to distribute plain data values. Thus this type typically implements a "last-is-best" semantic—only the last received data value is valid and accessible. Previous values are overwritten by current ones. Within the AUTOSAR standard the term *Data Distribution* is used for this interaction type.

Event Transmission: In event transmission, distributed data represents messages about occurred events, transferred from the sender to the receiver. Messages must not overwrite previously received ones, therefore receivers for this interaction type typically implement message queues.

Figure 5 shows the schematic for a sender-receiver connector, that can be used to assemble both, state transmission and event transmission connectors. The connector is made up of two fragments that independently have to be deployed aside with the component they are connected to: (i) the sender and (ii) the receiver. This so called deployment anomaly [16] has to be taken into account, when generating the connectors' deployment specification.

Depending on the type of connector that has to be constructed, full-fledged communication primitives or light-weight dummies can be plugged into the positions treated by the schematic's parts. However, at least one operational writer and one reader has to be plugged into a sound sender-receiver connector, therefore the placeholder for both of them is drawn with a thick black line. The full schematic for both connector fragments contains the following roles for communication primitives:

Reader and Writer: Communication primitives associated with one of this roles are responsible for the process of information dissemination. Writers put information on the information channel, while readers get it from there, whereas the nature of the channel depends on the components' deployment. Both, readers and writers have to be implemented for any interaction class. While readers and writers for local interaction may be as simple as shared memory accessors, they can also become rather complex primitives, implementing e.g., time-driven bus access or they may even be implemented in hardware. Passive readers typically store the last received data within an internal buffer (for example dual ported memory within the controller host interface (CHI)) that is overwritten by every later received value. If the received information has event semantic—it has to be stored within a queue, our schematics allow the usage of active readers that may invoke (passive) data storage primitives, to store the messages, or the usage of passive readers that get invoked by active data storage primitives. In both cases the active element has to be triggered by an infrastructural service (timer or receive interrupt) via its port for infrastructural services.

Encoder and Decoder: Encoders provide functionality that is primarily used by writers to manipulate data before it is emitted to the information channel. Therefore, this role is assigned to communication primitives that provide services of marshalling, data verification or encryption. Decoders provide services that are required by readers to manipulate data after it has been read from the information channel. The provided services correspond to that of the encoders. Decoder primitives are required only if encoders are used. For each type of provided service of an encoder or decoder a distinct interface has to exist within the primitive's port. In homogeneous systems encoders and decoders may be omitted, if they are only used for marshalling purpose.

Confirmation Handler: Confirmation handlers are communication primitives that keep control of the communication's status. They can provide distinct interfaces for various levels of confirmation (e.g., receipts for successful transmission

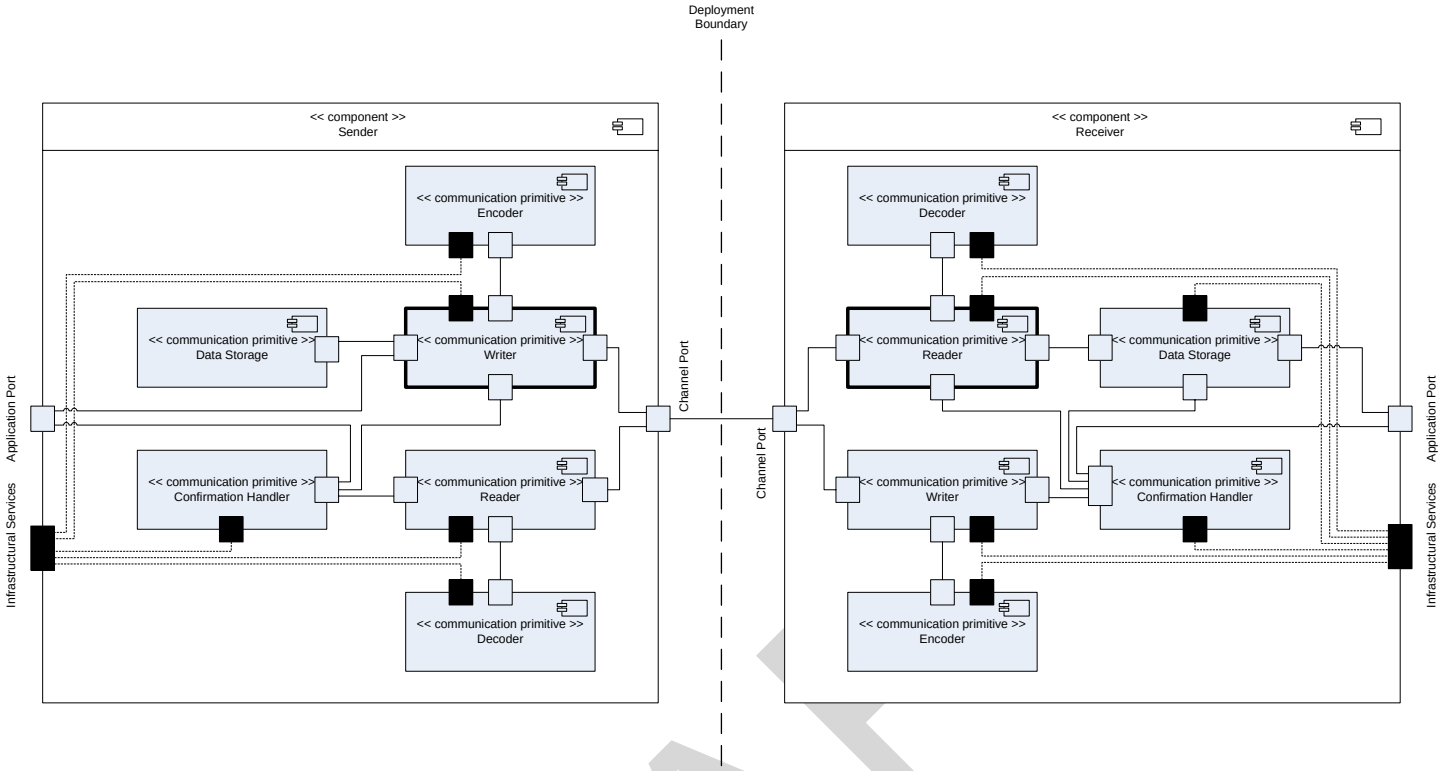


Figure 5: Sender-Receiver Connector Schematic

(TX), successful reception (RX) or successful execution by the receiving component). One has to keep in mind, that confirmed communication implies the presence of a writer primitive at the receiver fragment and a reader primitive at the sender fragment, except the desired confirmation is TX, which is a local confirmation.

Data Storage: If the receiver fragment has to store received messages, this can be achieved by any communication primitive that implements the data storage role. A message queue is a typical implementation of that communication primitive. In dependable systems, a writer may also store emitted messages to be able to retransmit them in case of a communication error.

Following the given schematic, very simple sender-receiver connectors for data distribution, but also complex ones for event distribution, can be built for each of the three interaction classes by eliminating dispensable parts.

3.2 Client-Server Interaction Pattern

A well known and often used communication pattern within distributed systems is the client-server pattern. The server provides a service, an operational functionality, that is required and used by the client.

In client-server communication the client initiates communication by requesting a service from the server. The server receives the request, performs the operation and dispatches a response to the client. Client requests can be (i) blocking, thus the client's thread of execution is stalled until the response from the server is received, or (ii) non-blocking, often referred to as asynchronous requests. A non-blocking client will issue a request to the server but will not stall its thread of execution waiting for the response. Therefore a non-blocking client has to provide a call-back interface, allowing the server to deliver its response.

Due to the nature of client-server interaction, its interfaces are procedural ones: Services are specified in terms of operations, that are identified by a unique function signature. This signature is made up of the operation's name and the number, sequence and type of its parameters. To connect the components' procedural interfaces with generic connectors, that are exposing generic interfaces only, the connectors' interfaces have to be wrapped by interface adapters to match the components' interfaces. These adapters can automatically be generated from proper interface descriptions of the components and the connector fragments.

Figure 6 shows the schematic for a client-server connec-

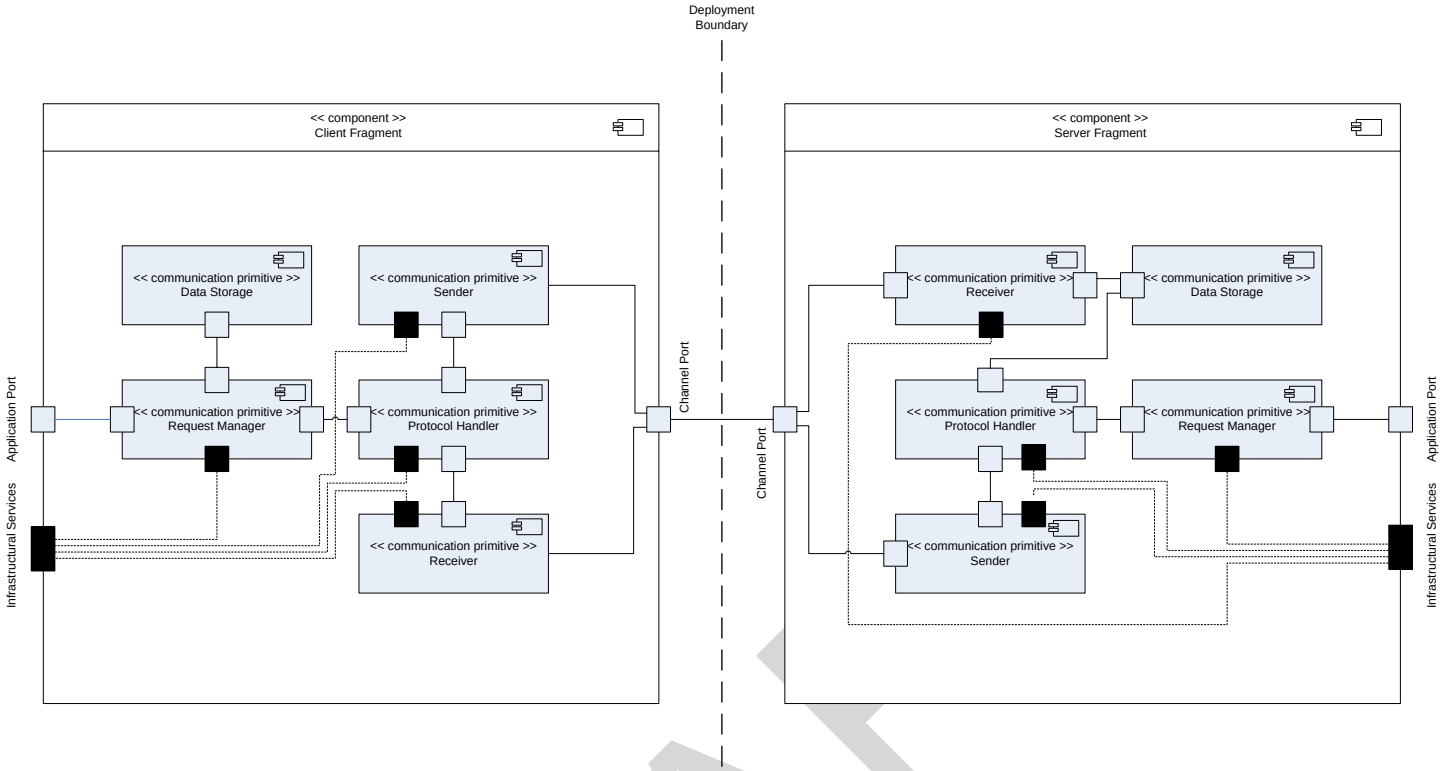


Figure 6: Client-Server Connector Schematic

tor. This connector type utilizes the sender and receiver fragments from Section 3.1 to realize inter-fragment communication via the connectors information channel. Thus all properties of sender-receiver interaction are also available for the sender and the receiver primitive within a client-server connector. Additional roles allow the creation of various specific client-server behaviors:

Request Manager: Within the client fragment, request manager primitives take care of client requests for services. Depending on the connector's needs and the primitive's implementation, various connector behaviors may be implemented. A request manager primitive might for example consider, if the communication is blocking or non-blocking, or if a request has timed-out and hard deadlines are exceeded (important for real-time systems).

Request manager primitives within the server fragment have to invoke the services of the connected server component in accordance to the messages received from the client fragment. If this primitive has to support concurrent server invocations, it has to be implemented as active primitive and also has to implement its own job-list for pending invocations.

Protocol Handler: Protocol handler primitives implement an arbitrary communication protocol for client-server interaction and are of great importance in dependable systems. As protocol handler primitives as well as request manager primitives only work in pairs, they have to appear in neither or in both fragments of a connector.

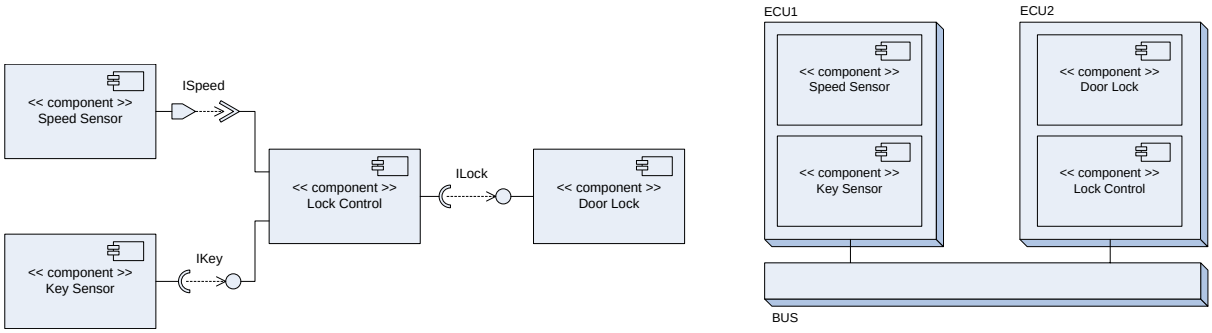
Data Storage: At the server-side fragment data storage primitives can be used to queue incoming but not-yet served requests. At the client-side fragment sent requests are stored for retransmission in case of error. The primitives on both sides are optional and are not related to each other.

Sender and Receiver: Those primitives are described in Section 3.1 and are used within the client-server pattern.

Using this schematic, a wide number of client-server connectors like simple procedure-calls (local and remote) or even real-time capable invocations can be constructed.

4 Proof of Concept

To prove our approach and to evaluate the gained benefit we built a small distributed embedded systems application from the automotive domain: a cruise-aware door lock system. The main purpose of the application is to lock or unlock the vehicle's doors. This should be done on user intervention by utilizing a key. In addition the application has to lock the doors automatically, if the



(a) Component Architecture

(b) Deployment Specification

Figure 7: Automotive Door-Lock Application

vehicle's speed exceeds a given limit. Once the door is locked, no unlock operation must be performed while the vehicle is moving.

The application as well as all used application components is quite simplified to fit this paper, but contains all relevant parts to demonstrate our approach.

Figure 7a shows the platform independent component diagram of our application containing all components and connectors. The door lock application is made up of four application components:

Speed Sensor: The speed sensor component provides the vehicle's current velocity by its interface *ISpeed*. The interface is a sender-receiver interface, the component will broadcast the vehicles current speed periodically. Consider the notation of the sender-receiver interface, which is done using a UML extension, proposed by the AUTOSAR [2] standard for automotive embedded systems.

Key Sensor: The key sensor component handles any key action set by the user. It reports a lock or unlock action to the lock control component by invoking the *lock()* or *unlock()* method of the *IKey* interface. The key sensor component requires the *IKey* interface from the Lock Control component.

Lock Control: The main application logic resides within the lock control component. This component receives the vehicle's current speed by its receiver interface *ISpeed* and all user key actions by its provided procedural interface *IKey*. The component controls the vehicle's door lock by invoking the *lock()* or *unlock()* method of the *ILock* interface provided by the door lock component.

Door Lock: This component locks or unlocks the vehicle's doors. It provides the *ILock* interface exposing the procedures *lock()* and *unlock()*.

Figure 7b depicts a deployment scenario for our application. We deploy our components on two ECUs of same type. Although the deployment diagram may contain platform specific information, we consider the deployment diagram at this stage to be platform independent. Remember that the explicit connectors are abstract entities at this moment, therefore they are not visible in the platform independent deployment diagram.

After transforming the PIM of the application's component architecture into a PSM, the three explicit connectors have been injected into the system.

Connector at ISpeed Interface: This connector is an extra node sender-receiver connector. It is constructed in accordance to the sender-receiver schematic. Only one reader and one writer communication primitive is required for this simple "data broadcast" connector.

Connector at IKey Interface: The connector for the IKey interface is a remote client-server connector. It is assembled in accordance to the client-server schematic. We use a simplified version of the RPC protocol for the protocol handler primitive, the request manager primitive is a dummy primitive that simply forwards all requests to the protocol handler. The sender and the receiver are of same type like the ones in the sender-receiver connector.

Connector at ILock Interface: This connector is a local client-server connector. By using the client-server schematic and dummy primitives for all parts, the final optimization removes all internals from this connector, exposing one simple procedure call.

The communication primitives used for the connector creation were built by slicing the AUTOSAR communication stack for FlexRay [17]. A detailed description of this process can be found in [18]. The communication middleware automatically gained by using our approach, showed a 30% reduction of the middleware's memory-footprint and a 10% reduction of runtime, compared to generic middleware.

5 Conclusion

Automotive embedded systems are highly distributed embedded systems that have to cope with very limited resources. As CBSE has attracted a lot of interest within the embedded systems community, well-suitable component middleware has to be built, to gain CBSE's benefits. This middleware has to be custom tailored to the application's specific needs. We propose to apply the CBSE paradigm in embedded systems not only at application level, but also at the system's component middleware, hence allowing automatic middleware generation from application models and prefabricated communication primitives. Therefore, this paper provides structural designs—connector schematics—for explicit connectors in component based distributed applications, that encapsulate a software architecture's communication needs, and a classification of basic building blocks—the communication primitives—for these connectors. Both, the schematics and the building blocks, are mandatory for automatic synthesis of an application's custom tailored communication middleware, that is part of the systems component middleware. The two types of explicit connectors treated within this paper, (i) the sender-receiver connector and (ii) the client-server connector, each adhere to one specific interaction pattern, occurring within automotive embedded systems as defined by the AUTOSAR standard. By applying our schematics, a broad variety of software connectors can be constructed to support the automatic generation of custom tailored, resource-saving, embedded communication middleware.

ACKNOWLEDGMENT

6 Acknowledgements

This work has been partially funded by the FIT-IT [embedded systems initiative of the Austrian Federal Ministry of Transport, Innovation, and Technology] and managed by Eutema and the Austrian Research Agency FFG within project COMPASS [19] under contract 809444.

REFERENCES

[1] Hansen, P., 2005. "New S-Class Mercedes: Pioneering Electronics". *The Hansen Report on Automotive Electronics*, **18**(8), Oct., pp. 1–2.

[2] AUTOSAR. *Automotive Open System Architecture*. http://www.autosar.org/.

[3] Meyer, B., 2003. "The grand challenge of trusted components". In Proceedings of ICSE 2003, pp. 660–667.

[4] Szyperski, C., 1998. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Jan.

[5] Meyer, B., 1992. "Applying "design by contract"". *IEEE Computer*, **25**(10), pp. 40–51.

[6] Councill, B., and Heineman, G. T., 2001. "Component-based software engineering". Addison Wesley, ch. Definition of a Software Component and its Elements, pp. 5–19.

[7] Nierstrasz, O., and Tsichritzis, D., eds., 1995. *Object-Oriented Software Composition*. Object-Oriented Series. Prentice-Hall, Dec.

[8] OMG, 2005. *UML 2.0 Superstructure Specification*. http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[9] Selic, B., 1999. "Protocols and ports: Reusable inter-object behavior patterns". In ISORC 1999: Int. Symposium on Object-oriented Real-time distributed Computing, IEEE Computer Society, pp. 332–339.

[10] OMG, 2006. *CORBA Component Model Specification Version 4.0*. http://www.omg.org/docs/formal/06-04-01.pdf.

[11] MICROSOFT, 2007. *COM (Component Object Model)*. http://msdn2.microsoft.com/en-us/library/ms680573.aspx.

[12] Schreiner, D., and Göschka, K. M., 2007. "Explicit connectors in component based software engineering for distributed embedded systems". In SOFSEM 2007: Theory and Practice of Computer Science, Proceedings, Vol. 4362 of *LNCS*, LNCS, Springer, pp. 923–934.

[13] Shaw, M., and Garlan, D., 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

[14] Robbins, J., Medvidovic, N., Redmiles, D., and Rosenblum, D., 1998. "Integrating architecture description languages with a standard design method". In Proceedings of the 20th International Conference on Software Engineering, IEEE Computer Society Press, pp. 209–218.

[15] Schreiner, D., and Göschka, K. M., 2007. "Synthesizing communication middleware from explicit connectors in component based distributed architectures". In Proceedings of the 6th International Symposium on Software Composition (SC 2007), LNCS, Springer. to appear.

[16] Bálek, D., and Plasil, F., 2001. "Software connectors and their role in component deployment". In DAIS, pp. 69–84.

[17] Führer, T., Hartwich, F., Hugel, R., and Weiler, H., 2003. "FlexRay – The Communication System for Future Control Systems in Vehicles". In Proceedings of the SAE 2003 World Congress & Exhibition, Society of Automotive Engineers.

[18] Galla, T. M., Schreiner, D., Kutschera, C., and Göschka, K. M., 2007. "Refactoring an automotive embedded software stack using the component-based paradigm". In SIES 2007: IEEE Second Symposium on Industrial Embedded Systems, Proceedings, IEEE. to appear.

[19] COMPASS. *Component Based Automotive System Software*. http://www.infosys.tuwien.ac.at/compass.