Component Based Middleware-Synthesis for AUTOSAR Basic Software

Dietmar Schreiner[1a], Markus Schordan[2], Karl M. Göschka[1b]
[1]Vienna University of Technology
[a]Compilers and Languages Group and [b]Distributed Systems Group
Vienna, Austria
[2]University of Applied Sciences Technikum Vienna
Institute of Computer Science
Vienna, Austria
{schreiner@complang, k.goeschka@infosys}.tuwien.ac.at
{schordan}@technikum-wien.at

Abstract

Distributed real-time automotive embedded systems have to be highly dependable as well as cost-efficient due to the large number of manufactured units. To close the gap between raising complexity and cutting costs, upcoming software standards like AUTOSAR introduce a clear separation of concerns into their system architecture. An AUTOSAR application is built from components that deal with business logic only whereas infrastructural services are provided by standardized middleware. Unfortunately, this middleware tends to be heavy-weight due to its coarse-grained layered design. By applying a component based design to AUTOSAR's middleware, a custom-tailored version for each specific application and system node can be built to overcome this problem. This paper demonstrates how to automatically synthesize component based middleware via the Connector Transformation: Component connectors in platform independent application models are utilized to automatically assemble platform- and application specific middleware. As a result, AUTOSAR middleware becomes custom-tailored and hence light-weight and flexible. In addition, the described synthesis algorithm is capable of incorporating timing annotations via interface contracts at model level, and thus reflects upcoming ambitions to cover real-time constraints at middleware level within AUTOSAR. To prove our approach we successfully synthesized middleware for a demonstrator application and compared it to its conventional counterpart.

1 Introduction

State-of-the-art vehicles incorporate functionality like steer/break-by-wire, ambient-intelligence or multimedia functionality and thus contain an increasing number of electronic sub-systems. Hence, a modern automobile contains more than 70 Electronic Control Units (ECUs), typically connected by up to 10 diverse bus systems [7].

By considering the large number of automobiles manufactured every year—in 2006 a total of 69,1 million vehicles [12] have been produced worldwide—and their rather long life-cycles of more than 10 years, it becomes obvious that software for automotive electronic systems has become a key factor not only in cost and quality, but also in time-to-market of current vehicles. Therefore, major manufacturers, suppliers, and tool developers jointly founded the Automotive Open System Architecture (AUTOSAR) consortium [8] to standardize an open architecture for automotive electronic systems.

Initiated in 2002, AUTOSAR mainly aims at (i) increasing the quality of automotive software, its maintainability, and its scalability, and at (ii) optimizing cost and time-to-market. Therefore, the AUTOSAR standard prescribes a bipartite system architecture, separating application concerns from infrastructural ones.

While applications are built in accordance to the component paradigm [20, 4, 2], infrastructural concerns are covered by an underlying layered middleware architecture, called *Basic Software* (*BSW*). However, application components are not directly aware of the *BSW* but are physically connected to the AUTOSAR *Run-time Environment* (*RTE*). The *RTE* resides between the *Basic Software* and the application, and provides means of execution, inter-component communication and life-cycle management for the application components by utilizing the *BSW*. In accordance to the AUTOSAR methodology, the *RTE* is generated at compile time and interfaces the application components to the *BSW*.

However, the AUTOSAR *BSW* itself is specified as layered architecture that is only customizable on a coarse-

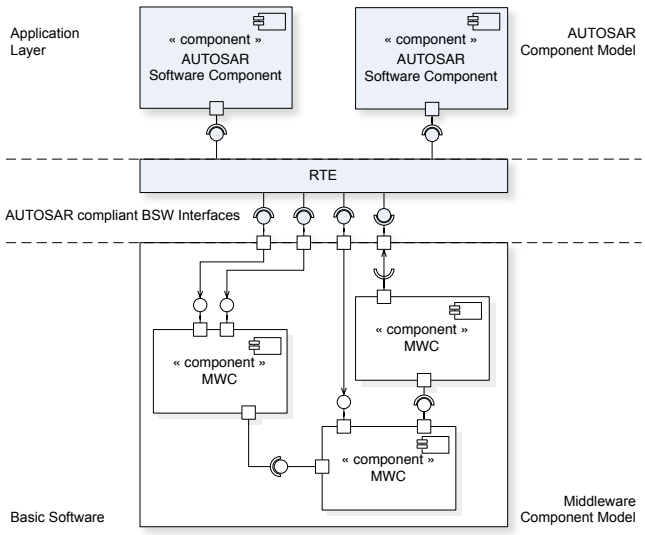

Figure 1. Component based AUTOSAR BSW

grained level, and thus tends to be heavy-weight and less flexible. This weakness can be overcome by applying the component paradigm not only at application level but also to the middleware itself as described in [17, 6] and schematically depicted in Fig. 1: the proposed component based *BSW* is assembled from middleware-components (MWCs), and provides all functionality required by the *RTE* and the application. To fully exploit the benefits of component based *BSW*, the process of building a custom-tailored version has to be automated. This paper contributes by defining a model transformation based algorithm that automatically synthesizes application- and node-specific communication middleware for AUTOSAR compliant *Basic Software*. The proposed synthesis algorithm transforms component connectors of platform independent models (PIMs) into platform specific middleware component architectures. These component architectures accurately incorporate the application's specific communication requirements. In addition, middleware component interfaces may be annotated with timing information that is considered by the synthesis algorithm. In that way, timing aware *BSW* can be synthesized. The proposed algorithm for component based middleware synthesis is exemplified for the AUTOSAR communication subsystem but can be applied to any other part of the *BSW*as well. To prove our algorithm, we designed a speed-aware door-lock application and synthesized the required application specific *BSW* that was finally compared to a conventional *BSW* stack.

1.1 Overview

The paper's structure is as follows: Section 2 describes an augmentation of AUTOSAR application models and a model driven development methodology, required for automatic middleware synthesis. It also defines explicit connectors, and contracts, to annotate constraints for the connectors. Section 3 provides the *Connector Transformation*, the algorithm used to synthesize the component based communication middleware from application models, middleware components and middleware patterns. In Sect. 4 we provide measurements for synthesized middleware for a proof-of-concept application. These measurements are compared to the same characteristics of a conventional *BSW*. Finally, the paper is concluded in Sect. 5.

2 Augmented Application Models

In accordance to the AUTOSAR methodology, an application's component architecture is described by platform independent models that are subsequently used to configure generic component middleware, and to generate the application's *RTE*. To synthesize communication middleware within a development process in line with the AUTOSAR methodology, the expressiveness of AUTOSAR PIMs has to be enhanced. The augmented models have to contain additional information that is required by our model transformation to automatically synthesize the communication middleware.

In well established component models like OMG's CORBA Component Model (CCM) [13], or Microsoft's Component Object Model (COM) [11], but also in AUTOSAR, all functionality related to component interaction and communication is implemented by component middleware. Matters of distribution or heterogeneity are completely transparent for the components. Connectors, as defined within these component models, are model level entities denoting the relation between associated component interfaces. Since they provide no implementation on their own, but represent attributes of interaction only, these connectors are referred to as implicit[1] connectors within this paper.

On the other hand, connectors represent middleware functionality, hence they can be associated with the implementation of specific middleware parts. Consequently, it is feasible to grant them first-class status within a component model. In contrast to implicit connectors that provide an abstraction of a system's middleware, these connectors resemble parts of the middleware. They make the system's middleware explicit, and thus are referred to as explicit connectors.

2.1 Explicit Connectors

Explicit connectors describe idiomatic patterns of functionality, provide mechanisms of data- and control-flow

[1] Connectors of this type implicitly denote component middleware that is transparent for the components.

Abbr.	Connector Name	Interaction	ECU	Address space
IPC	Intra Process	local	co-located	same
XPC	Extra Process	inter process	co-located	different
XNC	Extra Node	remote	distributed	different

Table 1. Connector Classification by Interaction Style

[9], and are associated with different architectural styles [19, 18]. Within application models, explicit connectors express interaction specific attributes and provide additional, fine-grained information on communication and interaction specific properties.

The implementation of explicit connectors may become rather complex, depending on communication style and deployment scenario. Thus, explicit connectors may also be constructed in a component based way by assembling middleware components.

Although explicit connectors resemble components, they differ in many aspects. In contrast to components, a connector changes its materialization during its life-cycle due to model transformations:

1. In platform independent models the explicit connector is an abstract representation of component interconnection, specifying interaction and communication specific properties.

2. In platform specific models the explicit connector is transformed into a set of "connector components", implementing the connector's interaction behavior and communication paradigm. These building blocks are composed structures, built from prefabricated middleware components. However, after successful transformation, all remaining connectors are implicit, local connectors only.

3. At deployment- and finally at run-time, explicit connectors are no longer visible. True middleware components, representing the explicit connectors' functionality, are deployed and executed.

2.2 Interaction Styles

All types of explicit connectors required for middleware synthesis can be classified by two dimensions:

- **Interaction Paradigm:** In accordance to the prevailing specification of AUTOSAR, any interaction within an application is mapped onto two basic interaction paradigms: the client-server paradigm and the sender-receiver paradigm. The functionality of these two paradigms has to be implemented by AUTOSAR communication middleware.

- **Component Deployment:** The second dimension is the outcome of how the connected components are deployed, relatively to each other (see Tab. 1). Two connected components may be deployed within the same address space (IP), on the same ECU, but within different address spaces (XP), or on different ECUs (XN).

As a consequence, the implementation of six types of explicit connectors has to be provided by AUTOSAR communication middleware.

2.3 Contracts

To strengthen the reliability and predictability of component based applications, guarantees, and requirements on properties and behaviors (e.g., worst-case execution time (WCET)) of application elements are formalized in contracts [10, 15, 5]. Contracts specify requirements and provisions of associated elements, in general a contract consists of two obligations:

1. The client, requiring functionality from another element, has to satisfy the preconditions of the provider.

2. The provider, that is the supplier of the required functionality, has to fulfill its post-condition, if the client's precondition is met.

When modeling a component architecture, contracts are denoted as model artifacts and are associated with the model element they refer to. For the purpose of automatic middleware synthesis, five main types of contracts are in use:

1. **Component-contracts** are associated with components and their instances. Typical component contracts deal with resource requirements or deployment restrictions like the components' memory footprint, required system resources, or the required ECU type.

2. **Interface-contracts** specify requirements and provisions for composition and interaction. Attributes within these contracts typically describe services and properties of the components' interfaces like operation signatures, accessible data elements or temporal properties like worst-case execution time at function level.

3. **Port-contracts** are associated with ports and their interfaces and deal with the relation between them, like

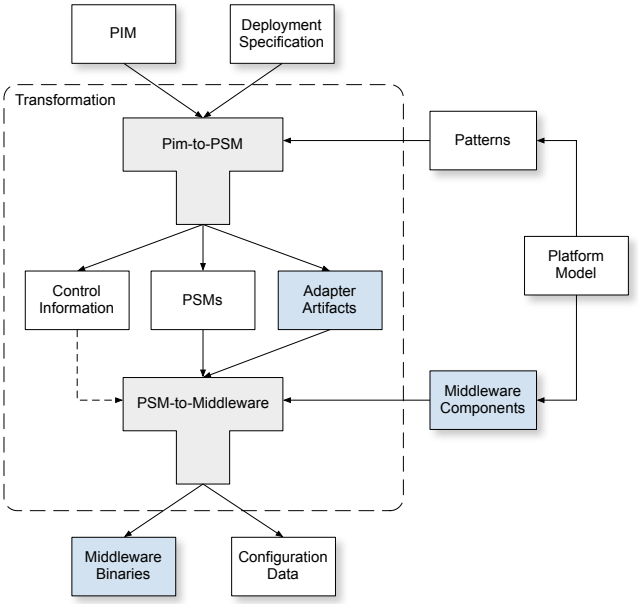

Figure 2. Connector Transformation (Concept)

message sequences, or the timing between port invocations. Behavior protocols, like described in [14], are typically contained within port-contracts.

4. **Connector-contracts** are associated with connectors at application level and contain constraints related to communication channels like worst-case propagation time or required communication style.

5. **Platform-contracts** specify properties of platform elements like ECUs or bus systems e.g. ECU type, available RAM, available ROM, or timing information. This type of contract is related to system nodes and hardware resources and thus is typically denoted within deployment models.

3 Connector Transformation

A model driven development process is mainly based on model transformations. This also applies to the methodology we propose within this paper: Custom-tailored communication middleware for AUTOSAR applications is synthesized from application models and prefabricated middleware components via our model transformation—the Connector Transformation—that is a two phase, guided transformation.

3.1 Concept

As specified in Fig. 2, within the first phase, the PIM-to-PSM transformation, we transform the platform independent application model into a set of platform specific models, a set of interface adapter components, and control information for the second phase, for each ECU. The transformation process is guided by annotated contracts within the PIM, the application's deployment model (DM) and architectural middleware patterns. The generated PSMs are component architectures, describing an ECU specific view of the distributed application, including the middleware's component architecture. All connectors remaining within the PSMs are local implicit connectors, hence are local procedure calls and shared-memory accesses.

In the second phase our approach refines the middleware's component architectures, contained within the PSMs, and then transforms each PSM into ECU specific middleware binaries: one for the communication part of the *BSW*, and one for the *RTE* parts, both related to interaction and communication. These generated binaries finally have to be linked and deployed with the remainder of the software system in accordance to the AUTOSAR methodology, to get an executable automotive application.

Figure 3 demonstrates how explicit connectors within a PIM are transformed into middleware components within PSMs: The upper half of the figure depicts a UML notation of two application components, *A* and *B*. *B* requires services, provided by component *A* via the interface *IA*. The connector, connecting the two associated interfaces, is depicted as usage-relation and constitutes an abstract entity that allows the annotation of communication and interaction specific attributes. This PIM, so far, is in line with conventional AUTOSAR application models. However, the PIM's semantic has to be augmented for the purpose of this thesis: The connector *C* between *B* and *A* is considered to be an explicit connector, and thus is granted first-class status within the model. As a result, *C* can be transformed into platform specific middleware components, contained within a PSM. The representation of an explicit connector within a PSM is a set of connector fragments, that are full-fledged, typically prefabricated middleware components. These connector fragments contain functionality and implementation of the system's component middleware. All communication and interaction specific properties can be specified as properties of the connector *C*, and its fragments.

Deployment Anomaly

Considering the given example, an additional characteristic of explicit connectors in distributed systems becomes visible. The explicit connector *C*, resides between the connected application components *A* and *B*. If *A* and *B* are deployed on the same single node, the connector's implementation has to be deployed there, too. However, if the two

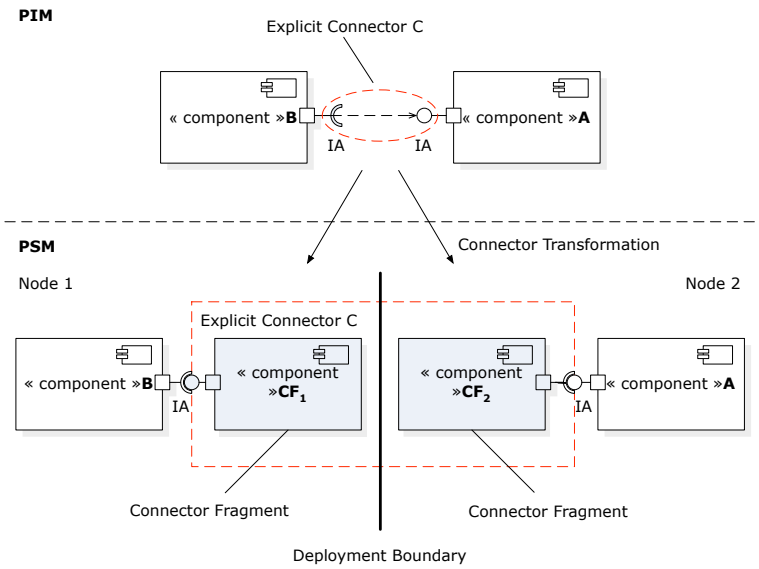

Figure 3. Transformation of Connector Artifacts from PIM to PSM

application components are placed on different nodes, as assumed in the given example, it is no longer trivial to deploy the connector. In fact, the connector element within the PIM has to be split into two pieces within the PSMs. These connector fragments subsequently have to be deployed along with their associated application components. CF_1 has to be deployed on the same node as *B* while CF_2 has to be co-located with *A*. As a result, the functionality of connector *C* crosses the deployment boundary. This characteristic is referred to as deployment anomaly in literature [3]: A connector, as one singular model element, may own an implementation, spread over multiple system nodes.

3.2 Algorithm

Figure 5 provides our algorithm for the *Connector Transformation* in pseudo-code. The used formalism denotes sets starting with a capital letter and ending with curly brackets as subscript. Elements are denoted starting with a minuscule.

To make the code more readable, tuples and positions within tuples are named. The following definitions provide those names, and must not be mixed up with the usage of tuples within the pseudo-code. Names within the algorithm's code denote elements (or sets) at the specific position within a tuple.

Definitions

The Connector Transformation proposed within this paper requires a platform independent application model and a deployment specification as input, and creates a platform- and node-specific component architecture for each affected system node. This platform specific models describe the custom-tailored communication middleware for each system node and can in consequence be transformed to executable code.

In detail, the input for the algorithm is specified as follows:

PIM: The application's component architecture is described by a platform independent model, denoted by a pair of sets of application components and of explicit connectors that connect the components. A connector is a tuple containing a client component, a server component, the interface used for the connection, the connector's interaction paradigm, and a contract for the connector, typically WCET annotations for the interface. Both, explicit connector and local connector, are isomorphic to the tuple *connector*.

$$\text{pim} = (\mathit{Components}_{\{\}}, \mathit{ExplicitConnectors}_{\{\}})$$
$$\text{explicitConnector} \cong \text{localConnector} \cong \text{connector}$$
$$\text{connector} = (\text{client}, \text{server}, \text{interface}, \text{iparadigm}, \text{contract})$$

$\text{DM}_{\{\}}$: The application's deployment model describes where the application components are deployed to. Valid locations for deployment are described by a *location* that is a pair of ECU and address space. The application's deployment model is represented by a set of pairs of one location and a set of components deployed to that location, such that there exists exactly one pair for each location of the target system.

$DM_{\{\}} = set\ of$ (location, *Components*$_{\{\}}$)
location = (*ecu*, *addressspace*)

ConnectorPatterns$_{\{\}}$**:** The set of connector patterns contains one architectural pattern for each interaction style (*distributionType* and *iparadigm*). The patterns do not prescribe specific middleware components but component classes of compatible components. The selection of a concrete implementation of a prescribed component class allows to flexibly adjust the resulting middleware to specific application needs. A connector pattern is a pair of connector fragments, one for the client- and the server-side. A connector fragment is a tuple containing a set of component classes (place-holders for concrete middleware components), a set of local connectors, connecting the middleware components, and an interface that denotes the fragment's generic interface type.

ConnectorPatterns$_{\{\}} = set\ of\ (distributionType,$
$iparadigm, connectorPattern)$
$connectorPattern =$
(clientFragment, serverFragment)
clientFragment $\cong$ serverFragment $\cong$
connectorFragment
connectorFragment = (*ComponentClasses*$_{\{\}}$,
LocalConnectors$_{\{\}}, isElementinterface$)

MWComponents$_{\{\}}$**:** The set of available middleware components provides the concrete building blocks, assembled to form the synthesized middleware. The set contains tuples of middleware components and associated contracts.

MWComponents$_{\{\}} = \{(component, contract)\}$

The algorithm's output is specified as:

PSM$_{\{\}}$**:** The *Connector Transformation* returns a set of platform specific models, one for each system node. These models contain all components (application and middleware) that are deployed to the PSM's location, and a set of local connectors, that connect the components.

psm = (location, *Components*$_{\{\}}$, *LocalConnectors*$_{\{\}}$)
PSM$_{\{\}} = set\ of\ psm$

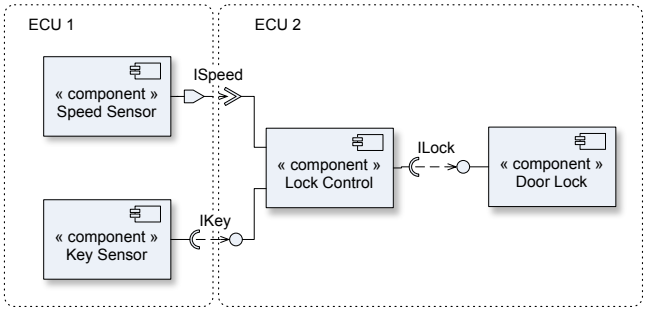

Figure 4. Demonstrator application

4 Evaluation

To prove our approach, we have applied it to an industrial implementation of an AUTOSAR *Communication Stack*—the *FlexRay Communication Services* stack. In the same way it can be applied to all *BSW* modules to gain fully component based *Basic Software*.

In a first step we have identified all middleware-component classes within an existing *Communication Stack* implementation via a static source code based analysis [17]. As a result, we gained 8 component classes and up to 4 implementation variants of distinct size for each class. The variants were constructed by reducing the capabilities of the originally identified component classes. Based on these building blocks, we designed 4 connector patterns in accordance to [16]. In a second step we designed a typical automotive demonstrator application: a speed-aware door-lock system (see Fig. 4). Finally we synthesized component based middleware (*Basic Software* and *RTE*) by applying our transformation.

The evaluation has been conducted on an ARM922T CPU running at 166 MHz and providing 16-bit access to an external ERAY 1.0 FlexRay communication controller. When comparing the created software system—application and ECU-specific middleware—to its conventional layered counterpart, the synthesized component based middleware proved to be more effective: In best case, the conventional communication stack, containing all functionality, required a total of 21.512 bytes, whereas a total of 6.884 bytes has been eliminated within the component based version. Thus we gained a reduction in size by more than 30%. In addition a run-time performance increase (10% less CPU usage) was gained. To examine the algorithm's limitations, we specified an application scenario where all types of interaction, requiring the full set of communication functionality, occurred on all ECUs and thus intentionally prevented optimization. However, in this worst case the synthesized middleware had exactly the same memory footprint and the same run-time performance than the conventional one.

```
1  ConnectorTransformation(pim, DM_{}, ConnectorPatterns_{}, MWComponents_{} ) : PSM_{}
2  begin
       // create and initialize PSM for each location
3      PSM_{} ← ∅;
4      foreach (location, Components_{}) ∈ DM_{} do add (location, Components_{}, ∅) to PSM_{};
       // iterate over all explicit connectors within the PIM
5      foreach (client, server, interface, iparadigm, contract) ∈ ExplicitConnectors_{} of pim do
           // localize PSMs of connected application components
6          foreach (location, Components_{}, LocalConnectors_{}) ∈ PSM_{} do
7              if client ∈ Components_{} then clientLocation ← location;
8              if server ∈ Components_{} then serverLocation ← location;
9          end
10         MiddlewareComponents_{} ← ∅; CFConnectors_{} ← ∅;
11         if clientLocation == serverLocation then      // connected components are co-located
12             distributionType ← LOCAL;
13             select psm ∈ PSM_{} | location of psm == clientLocation;
14             add(client, server, interface, iparadigm, contract) to LocalConnectors_{} of psm;
15         else
               // connected components are distributed
16             if ecu of clientLocation == ecu of serverLocation then
17                 distributionType ← INTERPROCESS;
18             else
19                 distributionType ← REMOTE;
20             end
               // select apropriate connector pattern
21             select (d, i, selectedPattern) ∈ ConnectorPatterns_{} | d == distributionType ∧ i == iparadigm;
               // compute fragments for client and server location
22             WorkSet_{} ← ∅;
23             add (client, clientLocation, clientFragment of selectedPattern) to WorkSet_{};
24             add (server, serverLocation, serverFragment of selectedPattern) to WorkSet_{};
25             foreach (appComp, cfLocation, connectorFragment) ∈ WorkSet_{} do
26                 (ComponentClasses_{}, CFConnectors_{}, cfInterface) ← connectorFragment;
                   // select optimal implementation for each component class due to specified contracts
27                 foreach part ∈ ComponentClasses_{} do
28                     select (mwc, contract') ∈ MWComponents_{} | mwc implements part ∧ contract' satifies contract;
29                     add mwc to MiddlewareComponents_{};
30                     if mwc comprises cfInterface then fragmentRoot ← mwc;     // for interface adapter
31                 end
                   // add interface adapter
32                 interfaceAdapter ←GenerateInterfaceAdapter(cfInterface, interface);
33                 add interfaceAdapter to MiddlewareComponents_{};
34                 add (appComp, interfaceAdapter, interface, iparadigm, contract) to CFConnectors_{};
35                 add (interfaceAdapter, fragmentRoot, cfInterface, iparadigm, contract) to CFConnectors_{};
                   // add middleware components and local connectors to PSMs
36                 select psm ∈ PSM_{} | location of psm == cfLocation;
37                 add MiddlewareComponents_{} to Components_{} of psm;
38                 add CFConnectors_{} to LocalConnectors_{} of psm;
39             end
40         end
41     end
42 end
```

Figure 5. Algorithm of Connector Transformation

5 Conclusion

Within this paper we proposed a model driven methodology for automatic middleware synthesis for the AUTOSAR standard. Our approach is based on a component based *BSW* architecture. It derives application requirements for the middleware from AUTOSAR compliant application models, and synthesizes *BSW* that covers these requirements in a custom-tailored way. Hence our approach introduces automatic generation of fine-grained custom-tailored *BSW* to AUTOSAR. The Connector Transformation requires an annotated platform independent application model, the application's deployment model, prefabricated middleware building blocks, and middleware patterns for each possible interaction style. To prove our approach, we have implemented a demonstrator application and have successfully synthesized light-weight communication middleware.

Ongoing research aims at an even more precise decomposition of the traditional *BSW*, leading to more but smaller middleware components, and hence to an even finer-grained component based middleware. In addition, we also focus on adding timing-awareness to the AUTOSAR *BSW* to support the construction of real-time capable middleware.

6 Acknowledgements

This work has been partially funded by the research project "Integrating European Timing Analysis Technology" (ALL-TIMES [1]) under contract No 215068 funded by the 7th EU R&D Framework Programme.

References

[1] *Project All-Times*. `http://www.all-times.org/`.

[2] AUTOSAR GbR. *Software Component Template 2.0.1*. `http://www.autosar.org/download/AUTOSAR_SoftwareComponentTemplate.pdf`.

[3] D. Bálek and F. Plasil. Software connectors and their role in component deployment. In *DAIS*, pages 69–84, 2001.

[4] B. Councill and G. T. Heineman. Component-based software engineering. chapter Definition of a Software Component and its Elements, pages 5–19. Addison Wesley, 2001.

[5] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House, 2002.

[6] T. M. Galla, D. Schreiner, W. Forster, C. Kutschera, K. M. Göschka, and M. Horauer. Refactoring an automotive embedded software stack using the component-based paradigm. In *Proceedings of the IEEE Second International Symposium on Industrial Embedded Systems (SIES 2007)*, IEEE, pages 200–208. IEEE, Jan 2007.

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

[8] H. Heinecke. AUTomotive Open System ARchitecture An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. In *Proceedings of the Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, 2004.

[9] S. Kell. Rethinking software connectors. In *SYANCO '07: International workshop on Synthesis and analysis of component connectors*, pages 1–12, New York, NY, USA, 2007. ACM.

[10] B. Meyer. The grand challenge of trusted components. In *ICSE*, pages 660–667, 2003.

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

[12] OICA - International Organization of Motor Vehicle Manufacturers. *2006 Production Statistics*, 2007. `http://oica.net/category/production-statistics/2006-statistics/` last visited on 21.11.2007.

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

[14] F. Plasil, S. Visnovsky, and M. Besta. Bounding component behavior via protocols. In *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings*, pages 387–398, Mar. 26 1999.

[15] R. H. Reussner and H. W. Schmidt. Using parameterised contracts to predict properties of component based software architectures. In S. L. Ivica Crnkovic and J. Stafford, editors, *Workshop on Component-based Software Engineering Proceedings*, 2002.

[16] D. Schreiner and K. M. Göschka. Building component based software connectors for communication middleware in distributed embedded systems. In *Proceedings of the 2007 ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications (MESA07)*, ASME/IEEE, 2007. CD-ROM.

[17] D. Schreiner, M. Schordan, G. Barany, and K. M. Göschka. Source code based component recognition in software stacks for embedded systems. In *Proceedings of the 2008 IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA08)*, Bejing, China, Oct 2008. IEEE. to appear.

[18] M. Shaw. Procedure calls are the assembly language of software interconnections: Connectors deserve first class. In D. Lamb, editor, *Studies of Software Design*, volume 1078 of *Lecture Notes in Computer Science*, pages 17–32, Baltimore, Md, May 1993. Springer-Verlag, Berlin.

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

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