Applying the Component Paradigm to AUTOSAR Basic Software

Dietmar Schreiner

Vienna University of Technology
Institute of Computer Languages, Compilers and Languages Group
Argentinierstrasse 8/185-1, A-1040 Vienna, Austria
{schreiner}@complang.tuwien.ac.at

1 Introduction

Current trends in embedded systems software for the automotive domain aim at an increase of reusability, exchangeability and maintainability, and thus at a significant reduction of time- and costs-to-market. One way to reach these goals is the adaption of Component Based Software Engineering (CBSE) for resource constrained embedded systems. The Automotive Open System Architecture (AUTOSAR) [1, 2], an upcoming industry standard within the automotive domain, reflects this fact by constituting CBSE as development paradigm for automotive applications: Application concerns are covered by software components, while infrastructural ones are handled within layered component middleware—the AUTOSAR Runtime Environment (RTE)[3] and the Basic Software (BSW) [4].

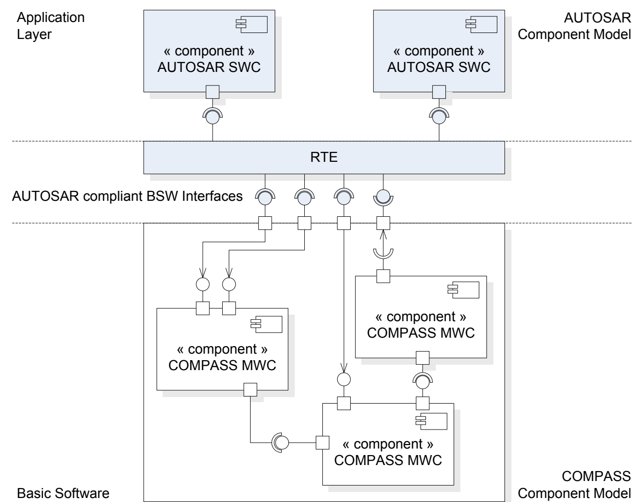

Fig. 1. Component based Basic Software

However, the AUTOSAR *Basic Software* itself is specified as layered architecture that is only customizable on a coarse-grained level, and thus tends to be heavy-weight and less flexible. Therefore, this paper contributes by applying the component paradigm to AUTOSAR *Basic Software* , to improve the capabilities of AUTOSAR compliant software systems, as conceptually depicted in Fig. 1: The redesigned BSW externally provides all interfaces to the RTE prescribed by the AUTOSAR standard, whereas the BSW's internal architecture is fully component based.

2 AUTOSAR Basic Software Components

In CBSE, the most common entity is called component. Unfortunately, literature shows many, very often contradictory [5], definitions of that term. So the first thing to do when dealing with CBSE, is to clarify the semantic meaning of this appellation, and to provide a clear vocabulary as basis for the remainder of this paper.

In accordance to the work of [6–8] we define components as follows:

> Components are trusted architectural elements of execution that interact only by their well defined interfaces, according to contractual guarantees, and strictly contain no other external dependencies. Components conform to a component model, so they adhere to a composition and interaction standard, and can be independently deployed and composed without modification. As a result, components are well suited for reuse and third-party composition. A set of well composed components is referred to as component architecture, while the term component model denotes the framework and standards, a component has to adhere to.

When building component based middleware for AUTOSAR, two things have to be taken into consideration:

- Middleware components cannot rely on component middleware. Therefore, the AUTOSAR component model [9] cannot be applied at *Basic Software* level.
- Middleware components have to provide standardized AUTOSAR functionality. Hence, the component model for middleware components has to be designed in line with the AUTOSAR standard, especially when it comes to the type system, to allow a seamless integration of component based middleware into the AUTOSAR environment.

A detailed specification of a component model, designed to meet these prerequisites within the AUTOSAR context, is provided in [10] and is used for the remainder of this paper. The so called COMPASS component model is compatible to the type system of AUTOSAR's C-language binding. It defines middleware components to be encapsulated units of execution, that interact via function calls within one global address space, and via shared memory access. The COMPASS component model specifies the *Basic Software* component classes, defines their minimal interfaces, and prescribes all means of composition and interaction for BSW components.

3 Component Recognition for AUTOSAR BSW

As our primary goal was to redesign AUTOSAR *Basic Software* in a component based way, we analyzed an exiting, layered BSW implementation to identify basic groups of functionality that could serve as base-line for BSW components: All functions, specified by AUTOSAR, which are either coupled via function calls, or which are coupled via shared memory accesses, have been marked as candidates for the same BSW component class. In addition, functions that are semantically related to each other, but are not directly coupled, have manually been assigned to the appropriate BSW component classes by domain experts. In that way we identified a set of BSW components that completely resemble the functionality and all external interfaces of the standard's layered BSW. Using the COMPASS component model and the identified *Basic Software* building blocks, it is now possible to build a component based AUTOSAR BSW, that on the one hand provides a fine-grained, function-based partitioning, enabling the creation of custom-tailored BSW, and that on the other hand highly supports reuse and exchangebility of BSW components.

3.1 Coupling within Software Components

One of the important properties of software components is that of encapsulation and separation. A well designed component contains a set of semantically related operations and data holders, that in total provide specific functionality exposed by the component's interfaces.

When trying to find those related operations and data holders within a global set of functions and data structures contained within monolithic or coarse-grained layered software, the coupling between all functions and all data structures has to be examined. For the task of component recognition two types of coupling are of interest:

Coupling via Control-Flow. Control-flow refers to the path of execution of a program. Two distinct functions within a program are strongly coupled via control-flow, if at least one of them passes control over to the other one. This is typically done by invoking the other function via a function call.

Coupling via Data-Flow. Data-flow refers to the flow of information during the execution of a program. As information is typically stored within data holders like memory cells, coupling via data-flow can be observed by examining access to data holders. Two distinct functions are strongly coupled via data-flow if both access the same data-holder, no matter if the type of access is read or write.

When taking these two types of coupling into account, two rules have to hold when performing automated component recognition:

1. Two distinct operations must not be coupled if they are contained within distinct components (horizontal coupling).
2. Two distinct operations may be coupled if they are contained within one component (vertical coupling).

3.2 Recognition Algorithm

When developing an algorithm based on static analysis, various options regarding complexity and thus execution time exist. Our algorithm was developed with respect to scalability, hence its complexity is kept linear to the size of the analyzed source code. In addition, our algorithm incorporates configuration data, especially data on late bound function pointers and on domain specific properties, to provide linear complexity and to find sufficient solutions quickly.

1. **Calculate the call graph.** A call graph, $G_C = (N_C, E_C)$, is computed from P's AST where functions of the program are represented by nodes, N_C, and calls are represented by edges, E_C, between the calling and the called function.

2. **Calculate usage graph.** A usage graph, $G_U = (N_U, E_U)$, is computed where functions and accessed data fields are represented by nodes, N_U, and the usage of a specific data field in the respective function is represented by an edge, E_U, between the function node and the data field node. To identify field accesses, the occurrence of arrow- and dot-expressions within the AST is analyzed. Our algorithm traverses the program's AST and finds all occurrences of field accesses in user defined data structures.

3. **Calculate component graph.** A component graph, $G_P = (N_P, E_P)$, can be calculated by creating a set union of the call graph and the data usage graph. It unites all gathered information on coupling via control-flow and on coupling via structure type based data field usage.

4. **Extract components from component-graph.** The algorithm's final step is the extraction of all disjoint, connected sub-graphs, the components, from the component graph. Our algorithm performs the extraction via a reachability calculation. The algorithm's output is a set of sets of nodes where each set of nodes represents one component. A domain expert can further group subsets of the automatically computed components into single components if desired.

4 Results

To prove our algorithm, it was applied to a full fledged implementation of an AUTOSAR communication stack, which is a subset of AUTOSAR *Basic Software* . The same implementation was manually decomposed as described in [11], which allows a reliable validation of the algorithm's results.

The analyzed source code is full-fledged C-code that implements the *FlexRay Interface- and Driver-Layer* as specified by the AUTOSAR standard. The source code can be characterized as shown in Table 1 and in Table 2. The generated AST contained 291794 nodes, which were traversed only once by our algorithm. The full execution of

FlexRay Interface			
	# of files	LOC	kB
Header	4	1620	59
Implementation	15	4192	135
FlexRay Driver			
Header	13	1660	88
Implementation	27	7142	222

Table 1. Source Code Characteristics

Function Definitions	107
Function Call Expr.	431
AddressOf Op.	12
Ptr Deref. Exp.	241
Arrow and dot Op.	457
Cast Expr.	4924

Table 2. Program Characteristics

our component recognition algorithm took less than 5 seconds and used approx. 80MB of memory on a 64-bit Intel PC.

The manual decomposition identified 8 components for the FlexRay Interface- and the FlexRay Driver-Layer. They were called *Base*, *Transmitter*, *Receiver*, *Time Services*, *Status*, *MTS*, *WUP*, and *TransceiverDrv*.

To perform our automatic component recognition, a set of 87 functions has been marked as relevant. 12 data structures respectively their data fields have been marked as irrelevant. On that basis our algorithm was able to recognize not only one but a set of valid decompositions. This is due to the fact, that some of the manually defined components consist of multiple, uncoupled sub-components that may be combined randomly, as they do not interfere. However, the manually created decomposition was contained within the calculated set, proving our algorithm valid in terms of imposed requirements.

5 Acknowledgments

This work has been partially funded by the European Community under the FP7 IST project All-Times, contract 215068.

References

1. Heinecke, H.: 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)
2. AUTOSAR: Automotive Open System Architecture. `http://www.autosar.org/`.

3. AUTOSAR GbR: Specification of RTE Software 1.0.1. `http://www.autosar.org/download/AUTOSAR_SWS_RTE.pdf`.
4. AUTOSAR GbR: Layered Software Architecture 2.0.0. `http://www.autosar.org/download/AUTOSAR_LayeredSoftwareArchitecture.pdf`.
5. Broy, M.: A uniform mathematical concept of a component (appendix to M. broy et al: "What characterizes a (software) component?". Software - Concepts and Tools **19**(1) (1998) 57–59
6. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (January 1998)
7. Heineman, G.T., Councill, W.T., eds.: Component-Based Software Engineering. Addison Wesley (2001)
8. Meyer, B.: The grand challenge of trusted components. In: ICSE. (2003) 660–667
9. AUTOSAR GbR: Software Component Template 2.0.1. `http://www.autosar.org/download/AUTOSAR_SoftwareComponentTemplate.pdf`.
10. Schreiner, D., Göschka, K.M.: A component model for the AUTOSAR Virtual Function Bus. In: COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference. Volume 2., IEEE (2007) 635–641
11. Galla, T.M., Schreiner, D., Forster, W., Kutschera, C., Göschka, K.M., Horauer, M.: 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, IEEE (Jan 2007) 200–208