Source Code Based Component Recognition in Software Stacks for Embedded Systems

Dietmar Schreiner[1] and Markus Schordan[1] and Gergö Barany[1] and Karl M. Göschka[2]
Vienna University of Technology
[1]*Compilers and Languages Group and* [2]*Distributed Systems Group*
Argentinierstrasse 8, A-1040 Vienna, Austria
{*schreiner,markus,gergo*}*@complang.tuwien.ac.at*
{*k.goeschka*}*@infosys.tuwien.ac.at*

Abstract

Current trends in embedded systems software 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 the embedded systems domain. Unfortunately most existing embedded systems applications are realized as coarse-grained layered or even monolithic software that can hardly be reused. This paper demonstrates how to recognize reusable and exchangeable components within existing typically monolithic or stacked embedded systems software via a semi-automatic analysis of the system's source code. The complexity of the proposed analysis is kept linear to code size by utilizing expert-knowledge on the application-domain, and deployment specific configuration data. To prove our approach, a functional decomposition for an existing automotive middleware stack is calculated and is finally compared to a human designed one.

1. Introduction

State-of-the-art embedded systems software has reached a level of complexity that led to a noticeable increase not only in time- and costs-to-market but also in maintenance costs. As a result, different software engineering methodologies have been adapted and introduced into the embedded systems domain to address this issue.

One of these methodologies is that of Component Based Software Engineering (CBSE) [12, 4, 11]. Various component models have been proposed by academia, some of them have even found their way into industrial usage [13, 14].

In CBSE an application is built by assembling small prefabricated, typically black-boxed, building blocks—the components. Components are trusted units of execution that encapsulate specific functionality and are well defined by a precise specification of their points of interaction, the so-called interfaces. Interfaces may be of type *provided* if a component provides a set of operations or data holders, or of type *required* if a component requires operations or access to data holders.

Components interact with their environment via their well defined interfaces only. As a consequence, components are free from side-effects and thus are potentially reusable and exchangeable. In addition, component based development introduces a higher level of abstraction for system developers and a clear separation of concerns by breaking away system- and component-development. Upcoming industry-standards like the Automotive Open System Architecture (AUTOSAR) [2, 8] heavily rely on CBSE as the engineering paradigm for their future applications.

However, existing embedded applications are typically designed as layered or sometimes even monolithic software architectures, and thus do not comply to the component paradigm. To keep CBSE's promises of reduced costs and increased quality, existing, well-tested, but non-component-based code-bases have to be made accessible for component based software development.

This paper contributes a solution to this remit, by proposing a methodology on how to recognize potential software components within layered or monolithic embedded systems source codes, and thus making these existing codes available for CBSE.

The paper's structure is as follows: Section 2 provides an overview on coupling within software for embedded systems. Section 3 describes the proposed methodology by defining an algorithm for component recognition, based on static analysis. To prove the proposed methodology, in Section 3.2 the semi-automatically generated decomposition of an automotive software stack (AUTOSAR Basic Software [3]) is compared to a manually calculated one. Section 4 provides an overview of related work in the domain of static analysis. Finally, the paper is concluded in Section 5.

2. 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 memory locations that in total provide specific functionality exposed by the component's interfaces.

When trying to find those related operations and memory locations 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 algorithm presented within this paper 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 specific locations like designated memory cells, coupling via data-flow can be observed by examining accesses to variables and structure fields. Two distinct functions are strongly coupled via data-flow if both access the same memory location, 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 strongly coupled if they are contained within distinct components (horizontal coupling).

2. Two distinct operations may be strongly coupled if they are contained within one component (vertical coupling).

3. Component Recognition

To decompose existing non-component-based programs into a set of reusable, exchangeable and thus independent components, the proposed algorithm gathers information on horizontal and vertical coupling by performing a static analysis of the program's source code. Utilizing the analysis results, all functions of the program are classified into sets of decoupled, independent components.

The analysis is performed at compile-time and is independent of any test cases. The recognized components are determined w.r.t. all possible program runs. The algorithm is flow-insensitive (independent of intra-procedural control-flow) and performs a single pass on the input program. It does not consider the different calling contexts of functions and is therefore context-insensitive.

To determine the vertical and the horizontal coupling, the algorithm considers three categories of information that is gathered by traversing the program's abstract syntax tree (AST):

Functions. The functions of the input program are the basic ingredients for component recognition and in the end are assigned to one specific component each, recognized by the algorithm's decomposition.

Function calls. Any function call means coupling by control-flow. Thus function calls imply a semantic relation between the calling and the called function, except if the called function is marked as irrelevant (see Section 3.1).

Types of accessed structure fields. To gather information on coupling by data-flow, our algorithm relies on type-information without the consideration of any concrete instances of data. As any function that uses a specific type has a semantic relation to all other functions also using this type, type based analysis turned out to be sufficient for the algorithm's purpose.

However, basic data types typically do not imply any semantic relations. As consequence only user defined data types are considered during analysis. In detail, the collected type-usage information consists of structure types and field names.

3.1. 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.

In detail, our algorithm presented in Figure 2 requires the following work-flow:

1. **Annotate function pointers.** Any function pointer within the input program P has to be associated with exactly those function(s) it will point to at run-time. In embedded systems software this is typically exactly one function, that is not known at compile-time but is subject to system configuration and specified at link- or deployment-time.

2. **Mark relevant functions.** Most software utilizes common helper functions or compiler-built-ins for e.g. memory manipulation. To avoid assigning these functions to program specific components,

```
struct S1{
  S2 a;
} s;

struct S2{
  int b;
};

struct S1 *y=&s;
x=y->a.b;
```

Figure 1: Example Source Code

all relevant functions have to be marked by a domain expert. In practice, the set of unwanted functions is smaller than the set of relevant ones. Therefore, the algorithm takes a set of irrelevant functions as input value ($\overline{F_M}$). This set is later on used to calculate the set of all relevant (and thus marked) functions F_M (Fig. 2, line 2).

3. **Mark characteristic structure fields.** The most important task of a domain expert within our workflow is the identification of relevant data structures and their respective relevant fields. Some structures are closely related to component functionality, but other ones are used for internal purpose only. By taking only relevant fields into account, our algorithm is sensitive to dedicated component functionality only. Again, normally the set of irrelevant structure fields is smaller than that of relevant ones. Consequently a set of fields that have to be ignored ($\overline{D_M}$) is passed to the algorithm, that calculates the set of all relevant and therefore marked fields D_M (Fig. 2, line 3).

All following steps are part of the algorithm's implementation and may therefore be executed automatically.

4. **Calculate the call graph.** Both, call graph and usage graph, are gained by iterating over all expressions within all functions of P (Fig. 2, lines 5,7). A call graph, $G_C = (N_C, E_C)$, is computed from P's AST where functions of the program are represented by nodes $n \in N_C$, and calls are represented by edges $e \in E_C$ between the calling and the called function (Fig. 2, lines 17–19).

5. **Calculate usage graph.** A usage graph, $G_U = (N_U, E_U)$, is computed where functions and accessed data fields are represented by nodes $n \in N_U$, and the usage of a specific data field in the respective function is represented by an edge $e \in E_U$, 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 (Fig. 2, lines 9–16). Figure 1 provides a C-code snippet and Figure 3 depicts an AST snippet representing the last line of the code. Our algorithm traverses the program's AST and finds all occurrences of field accesses in user defined data structures. In the given example, two structure types (*S1* and *S2*) with one data field each (a and b) are defined. As a result two data field accesses ($(S1, a)$ and $(S2, b)$) are collected by storing the structure-type of the left-hand-side, and the field name of right-hand-side of the dot- and the arrow-operator.

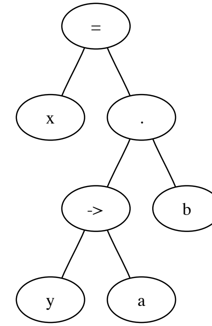


Figure 3: Example AST

6. **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 usage graph. It unites all gathered information on coupling via control-flow and on coupling via structure type based data field usage. (Fig. 2, line 26)

7. **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.

3.2. Proof of Concept

To prove the proposed algorithm it was applied to an automotive communication stack, namely an AUTOSAR Basic Software (BSW) implementation. The same implementation was manually decomposed as described in [7], 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 our component recogni-

Input:

P : program to be analyzed
$\overline{F_M}$: set of functions to be ignored
$\overline{D_M}$: set of structure fields to be ignored

Output:

C_P : set of component-sets

```
1  begin
2      F_M ← functions(P) − F̄_M;   // determine relevant functions
3      D_M ← structure_fields(P) − D̄_M;   // determine relevant structure fields
4      N_C ← ∅; E_C ← ∅; N_U ← ∅; E_U ← ∅;
       // iterate over all marked functions in a program
5      foreach f ∈ functions(F_M) do
6          N_C ← N_C ∪ {id(f)};  // id(...) provides a unique identifier
           // iterate over all expressions and subexpressions in a function
7          foreach exp ∈ expressions(f) do
8              switch exp do
                   // collect structural data usages
9                  case lhs.rhs or lhs −> rhs   // dot or arrow expression
10                     s ← structure type in lhs;
11                     d ← structure field name of rhs;
12                     if (s,d) ∈ D_M then
13                         N_U ← N_U ∪ {id(f)} ∪ {id((s,d))};
14                         E_U ← E_U ∪ {{id(f) , id((s,d))}};
15                     end
16                 end
                   // collect call graph data
17                 case f_c(...)  // function-call-expression
18                     E_C ← E_C ∪ {{id(f) , id(f_c)}};
19                 end
20                 otherwise  // any-other-expression
21                     skip;  // no information is extracted
22                 end
23             end
24         end
25     end
       // build undirected graph
26     G_P ← (N_U ∪ N_C, E_U ∪ E_C);
       // extract components via reachability
27     C_P ← ∅
28     while G_P ≠ (∅,∅) do
29         n ← choose_node(G_P)   // choose some node n
30         G_S ← reachable_subgraph(n, G_P)
31         G_P ← G_P − G_S   // remove subgraph G_S from G_P
32         C_P ← C_P ∪ {nodeset(G_S)}
33     end
34 end
```

Figure 2: Component Recognition Algorithm

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

tion 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 and their respective 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.

Although our algorithm is applied to one specific implementation of the AUTOSAR communication stack, its outcome—a decomposition of parts of the AUTOSAR BSW into components—is applicable to other implementations of the same standardized stack, as the chosen implementation is a reference implementation that exposes standardized functions only and that implements all functions strictly according to the standard. All non-standardized functions are not considered within our analysis and are thus kept hidden within the components.

Figure 4 provides a visualization of the component graph of the analyzed source code where circles represent functions and squares represent data fields. Figure 4a and Figure 4b show *P*'s component graph containing all functions and data fields. Figure 4b in addition depicts the manually identified components, represented as clusters. Figure 4c depicts the component graph as seen by our algorithm. Only nodes and edges marked as relevant are considered for calculation, all irrelevant edges and nodes are filtered out. As depicted, all recognized components match the manually identified ones, again denoted as clusters. No edges cross the boundaries of the defined clusters, so no strong coupling exists between distinct components.

4. Related Work

Lee et al. [10] describe a methodology to recognize components within existing object-oriented source-codes considering class cohesion. They propose to calculate coupling by message passing and data usage, and in addition consider coupling by class association, composition and inheritance. In contrary, our algorithm is intended to work on object-oriented but also on non-object-oriented source-codes. To keep their analysis effort small, their approach also relies on domain knowledge, mainly extracted from UML use-cases and architectural descriptions.

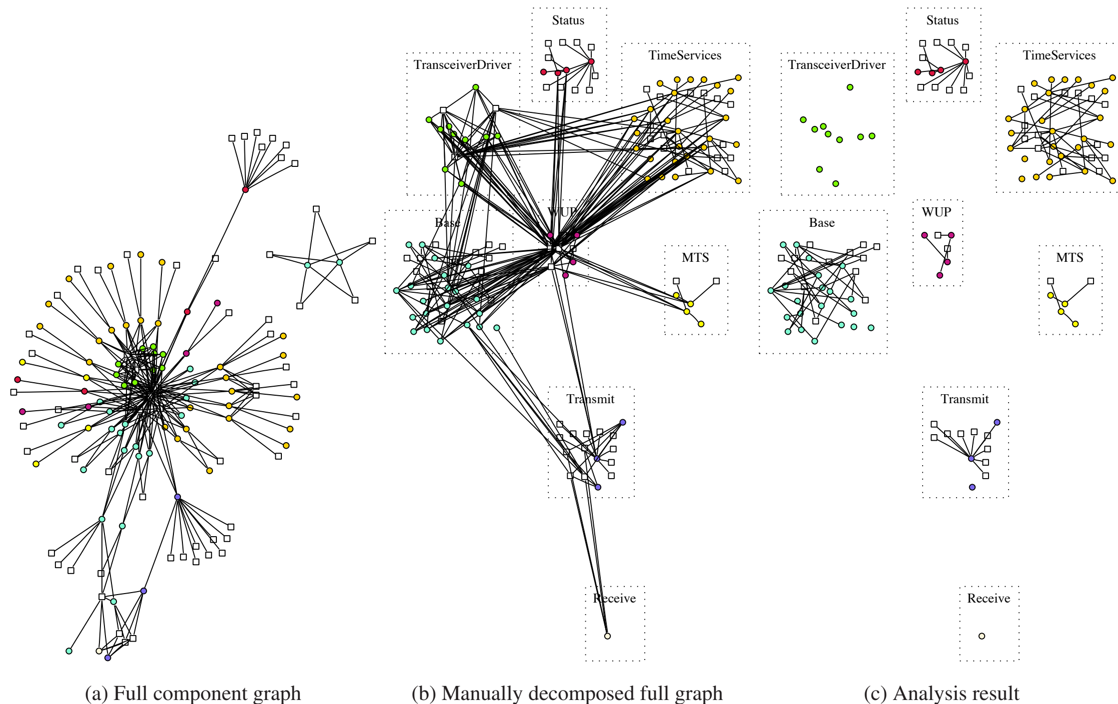
Emami, Ghiya, and Hendren [6] present a context-sensitive approach that generates a graph representing all invocation paths (in the absence of recursion) and precisely handles indirect calls through function pointers in C. They focus on analysis of stack-directed pointers, and collect alias information in the form of points-to relationships. They claim through empirical evidence that exponential behavior is not seen in practice and suggest the use of a memorization scheme to avoid redundant analysis. Our algorithm is linear in the size of the program, but sufficient to establish the required properties of the input program.

Hind, Burki, Carini, and Choi give experimental results of comparing flow-sensitive and flow-insensitive flow analysis algorithms in [9] based on memory locations associated with names. The call graph is constructed while alias analysis is performed.

Diwan, McKinley, and Moss evaluate three alias analysis algorithms based on programming language types. The most precise of these three is a flow-insensitive analysis that uses type compatibility and additional high-level information such as field names [5]. They use redundant load elimination to demonstrate the effectiveness of the algorithms in terms of opportunities for optimization.

5. Conclusion

This paper provides an algorithm for component recognition within monolithic or layered software architectures. The algorithm performs static source code analysis in a flow- and context-insensitive single pass. To keep its complexity linear to code size of the input program, domain expertise is used to confine the algorithm's search space. The decomposition of an industrial automotive communication stack, calculated with our algorithm, was successfully compared to a manually created one and showed to be valid, and to be even more precise than the human made one.



(a) Full component graph (b) Manually decomposed full graph (c) Analysis result

Figure 4: Component Graphs

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. *Automotive Open System Architecture*. http://www.autosar.org/.

[3] AUTOSAR GbR. *Layered Software Architecture 2.0.0*. http://www.autosar.org/download/AUTOSAR_LayeredSoftwareArchitecture.pdf.

[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] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 106–117, Montreal, Canada, 17–19 June 1998.

[6] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, 29(6):242–256, June 1994.

[7] 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.

[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] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[10] J. K. Lee, S. J. Seung, S. D. Kim, W. Hyun, and D. H. Han. Component identification method with coupling and cohesion. In *Proceedings of the Asia-Pacific Software Engineering Conference 2001*, pages 79–86, 2001.

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

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

[13] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, pages 78–85, Mar. 2000.

[14] M. Winter, T. Genßler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, P. Müller, C. Stich, and B. Schönhage. Components for embedded software — the PECOS approach. In *Proc. Second International Workshop on Composition Languages*, 2002.