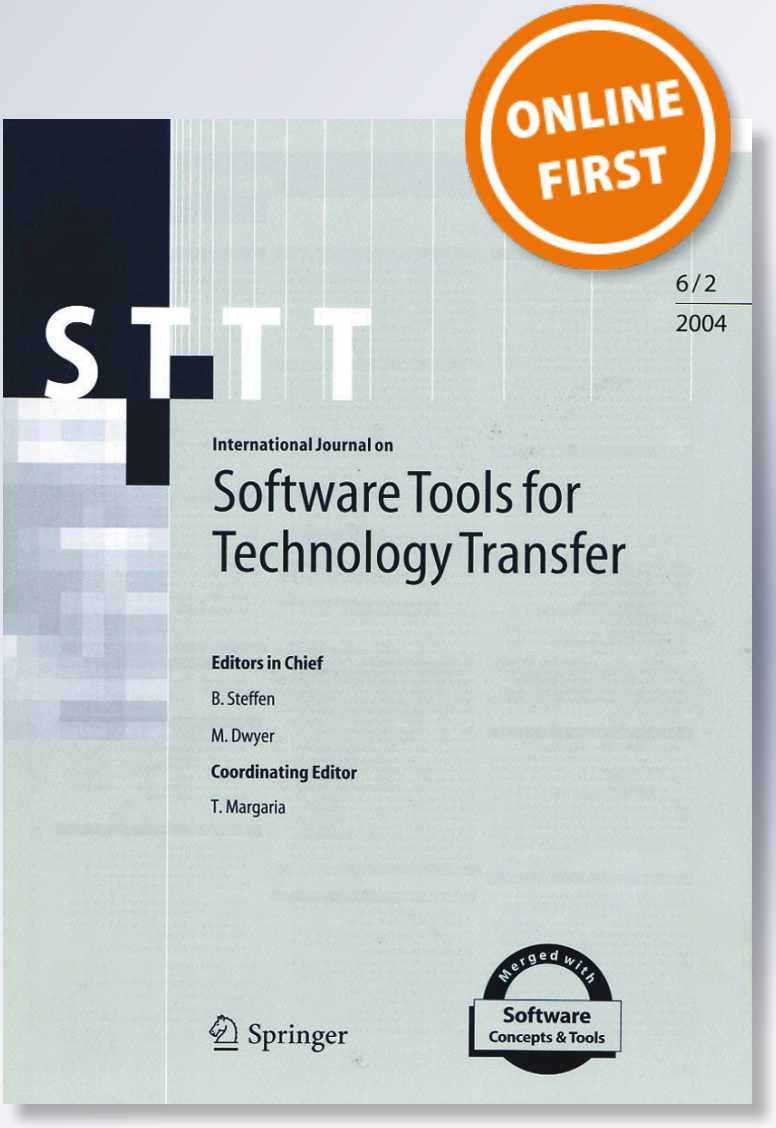
Comparison of type-based and alias-based component recognition for embedded systems software

Dietmar Schreiner, Gergö Barany, Markus Schordan & Jens Knoop

International Journal on Software Tools for Technology Transfer

ISSN 1433-2779

Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-012-0251-0



Springer

Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-012-0251-0

ALL-TIMES

Comparison of type-based and alias-based component recognition for embedded systems software

Dietmar Schreiner · Gergö Barany · Markus Schordan · Jens Knoop

Abstract Component-based software engineering has found broad acceptance within the embedded systems community over the last years. However, to fully exploit its potential in terms of reusability and cost-efficiency, existing code-bases have to be refactored in a component-based way. To support refactorization, static analysis techniques can be used to identify components within coarse-grained layered or even monolithic legacy software for embedded systems. We present an approach for semi-automatic extraction of components from automotive software and compare two different versions, one type-based component-recognition analysis of linear complexity with a more precise version based on a points-to analysis of almost linear algorithmic complexity. Both analyses are applied to an industrial implementation of an automotive communication stack. Each analysis is evaluated with two sets of additional manually created annotations of distinct size and precision. Thus, both analyses are fully evaluated in terms of execution-time, memory consumption and analysis precision, and its impact on the number of recognized components. We show that the analysis with higher precision allows the use of a smaller user-provided filter set and obtain a proper component recognition.

D. Schreiner · G. Barany · J. Knoop
Compilers and Languages Group, Institute of Computer Languages, Vienna University of Technology, Vienna, Austria
e-mail: schreiner@complang.tuwien.ac.at

G. Barany
e-mail: gergo@complang.tuwien.ac.at

J. Knoop
e-mail: knoop@complang.tuwien.ac.at

M. Schordan (✉)
Institute of Computer Science, University of Applied Sciences Technikum Wien, Vienna, Austria
e-mail: schordan@technikum-wien.at

Published online: 15 July 2012

Keywords Static analysis · Component recognition

1 Motivation

The ongoing growth in complexity of real-time embedded systems software over the last years led to a tremendous increase in development costs and time-to-market, but also to a loss in software quality and reusability. As a consequence, industry had to change the way embedded systems software has been built over many years, in particular as timing aspects are considered. Well-established software engineering methodologies like component-based software engineering (CBSE) have been adapted to the domain's needs, to reduce costs and time-to-market, and to increase software quality and reliability [1].

Unfortunately, most embedded systems software is not the result of a component-based design methodology, and even if it is, it is difficult to verify that the components indeed meet all the required constraints of data dependencies to allow a component-based composition. To fully exploit the potential of CBSE in terms of software reuse, existing legacy code bases, typically coarse grained layered or even monolithic, have to be preserved by refactoring them in a component based way. Refactorization of existing code bases requires a huge amount of domain expertise as well as knowledge on the legacy code itself, and is thus an intricate and expensive process. By applying static analysis techniques to those legacy code bases, valuable guidance for a manual decomposition, or even an analysis-based semi-automatic decomposition technique can be obtained [2,3].

In this article, we present an approach for semi-automatic extraction of components from automotive software that was developed in order to demonstrate and evaluate the capabilities of our supporting analyses and their binding into the ALL-TIMES tool chains. The project ALL-TIMES

was initiated by academia and by leading European industry to strengthen competitiveness of European worst-case execution time (WCET) analysis tools. It aims at an increase of interoperability between timing analysis tools and at the creation of common supporting analyses for embedded systems.

This article contributes by investigating the impact of the precision of two kinds of analyses on the quality of component decomposition. The motivation for this work is to improve the developer's productivity by reducing the amount of required manual annotations (so called "filters") for supporting our semi-automatic analysis method. We investigate the trade-off between precision of the analysis, its run-time and memory consumption, and the recognized set of components, when applied to the source code of embedded software stacks.

Therefore, we compare the impact of two different analyses on component recognition: a type-based analysis technique and an alias-based analysis. The type-based analysis has been presented in [3]. The alias-bases analysis is one of SATIrE's [4] core analyses, and has been integrated in our approach as well. Our semi-automatic extraction method also requires the user to provide a set of filter rules. This work can be tedious, but with our new alias-based method, we can show that the number of required rules to obtain a similar result in terms of component recognition as in [3], can be reduced by 22 %. We consider the number of rules as measure for the productivity when applying the presented approach. The new alias-based analysis is more precise than the type-based analysis, and therefore allows to reduce the filter set, but achieve a similar result.

The type-based analysis is flow-insensitive and has linear complexity w.r.t. to the size of the program. As alias analysis allows for a more precise analysis and requires more implementation effort than the linear type-based analysis, alias-based component recognition should yield more precise results to be considered profitable. The run-time of the alias analysis is higher, but still reasonable, because of being an almost linear alias analysis.

Both analyses are benchmarked and directly compared in terms of execution time, memory consumption, and the precision of the calculated results. We show in detail in the following sections that the precision of our different analyses directly influences the number of manual annotations that are necessary to extract components. The identified components then offer interfaces which are even suitable for high-level WCET annotations [5]. The overall goal is to reduce the required work for annotating programs.

2 Static cohesion analysis

Over the last years, several methodologies to identify functional subsystems within existing software have been developed and specified [6] within the reverse-engineering

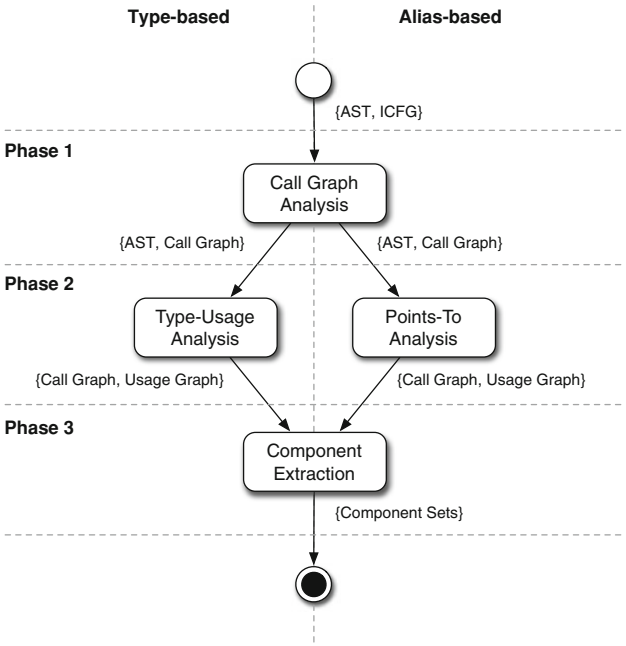


Fig. 1 Overview of cohesion analysis

community. Within this article, we compare two cohesion analyses—a type-based one [3] with a new alias-based version—that identify component candidates in accordance to call- and data-dependencies within legacy code under analysis. Any function-pointers that are not statically resolvable because their value is not set in the analyzed code, have to be annotated to ensure proper decomposition; pointers to variables are not of interest within the type-based analysis, but are fully covered within the alias-based version.

Our analysis is a three phase analysis (see Fig. 1), which (i) calculates a undirected call-graph (G_C) based on flow-analysis, (ii) calculates a usage-graph (G_U) based either on a type-based analysis or on an alias-based analysis, and finally (iii) extracts component candidates from a component graph, which is the union of the undirected call-graph and the usage-graph.

Definition 1 (*Undirected call graph*) Let a program P be a set of functions and global variables. An undirected call graph G_C of a program P is a pair of sets N_C and E_C, where the graph's nodes $n \in N_C$ represent marked functions of a program P, and the undirected edges $e \in E_C$ denote calls between these functions.

$$F_P = \{f \mid \text{f is a function} \quad \wedge \quad f \in P\} \tag{1}$$

$$\begin{aligned} G_C = (N_C, E_C) \mid\ & N_C = \{f \mid f \in F_P \wedge \\ & f \text{ is marked}\}, \\ & E_C = \{\{a, b\} \mid a, b \in N_C \wedge \\ & a \text{ calls } b\} \end{aligned} \tag{2}$$

For the purpose of our cohesion analysis, we are interested in properties of coupling associations rather than in call

directions, hence we consider edges as *undirected*. Therefore, edges are represented by a set of two elements rather than by an (ordered) pair. To keep the analysis focused on the program code itself, helper-functions are held off from the undirected call-graph by considering relevant—manually marked—functions only.

In the embedded systems codes of interest no recursion is allowed. The second phase of the analysis calculates a usage-graph that represents coupling—a usage relation—between functions and data.

Definition 2 (*Usage graph*) A usage graph G_U is represented by a pair of sets N_U and E_U. The graph's nodes $n \in N_U$ represent an abstraction of data usage—N'_{SF}—as well as marked functions that use specific data—N'_U. Edges $e \in E_U$ connect functions and data usages and thus represent usage of data by specific functions.

$$U_P = \{s \mid s \text{ is a data usage abstraction } \in P\} \tag{3}$$

$$\begin{aligned} G_U = (N_U, E_U) \mid N_U = N'_{SF} \cup N'_U, \\ N'_{SF} = \{ n \mid n \text{ is marked } \wedge \\ n \in U_P\}, \\ N'_U = \{ m \mid m \text{ is marked } \wedge \\ m \in F_P \ \wedge \\ \exists\, u \in U_P \text{ where} \\ u \text{ occurs in } m\}, \\ E_U = \{ (a, b) \mid a \in N'_U \ \wedge \\ b \in N'_{SF} \ \wedge \\ b \text{ occurs in a}\} \end{aligned} \tag{4}$$

While the set of marked functions N'_U remains the same for both versions of the analysis, "data usage" has a distinct meaning for both of them. Therefore, the set of data usages N'_{SF} and consequently the set of edges E_U differ in the two versions. However, by providing a common carrier set U_P for both versions of the analysis, and by specializing this set for each of them (see Sects. 2.1, 2.2, respectively), a unified algorithm can be used. As a result, the analyses become directly comparable.

The third and final phase of the cohesion analysis is the fusion of the results of the previous two phases, and the extraction of component candidates therefrom. Fusion is accomplished by creating a component graph.

Definition 3 (*Component graph*) A component graph G_P is denoted by a pair of sets N_P and E_P. Its nodes $n \in N_P$ represent the program's data usage abstractions, and the program's functions. Its edges represent calls between the program's functions, or usage of data by a function.

$$\begin{aligned} G_P = G_C \cup G_U = \\ = (N_P, E_P) \mid N_P = N_C \cup N_U, \\ E_P = E_C \cup E_U \end{aligned} \tag{5}$$

The component graph is calculated by creating a set union of the program's undirected call graph and its usage graph. It unites all gathered information on coupling via control-flow and on coupling via data usage.

The component graph contains disjoint sub-graphs—*connected components* in a graph-theoretic sense—that represent coupled, self-contained functionality, and thus serve as candidates for components in the software engineering sense. These component candidates are extracted via a reachability calculation. The output of the analysis is a set of sets of nodes where each set of nodes represents one component. In addition, a domain expert can further merge any automatically computed components into single components if desired.

2.1 Type-based analysis

As described in Sect. 2, one integral part of the cohesion analysis is the calculation of a program's abstract data usage. Within the type-based analysis, data types represent a proper abstraction of data. All functions that operate on the same data type are semantically related to each other, and hence are coupled within the usage-graph. It is obvious that this approach tends to couple too many functions if applied to all data types found within an arbitrary program. Especially, the basic data types typically do not imply semantic relations between their enclosing functions. In consequence, the type-based usage analysis (type-usage analysis) has to be enhanced by manually identifying those data types that are of relevance for meaningful decomposition. For our type-usage analysis basic data types are disregarded, only structure types are considered.

Definition 4 (*Type-based field usage*) A type-based field usage is a pair (s, d) where s denotes the structure type of the used data structure, and d denotes the field name—not its type—of the used data field.

$$\begin{aligned} U_T = \{(s, d) \mid s \text{ is a structure type } \in P \ \wedge \\ d \text{ is field of } s \ \wedge \\ s \text{ and } d \text{ are marked}\} \end{aligned} \tag{6}$$

Following this definition, a specialized version of a usage-graph can be defined:

Definition 5 (*Type-based field usage graph*) The type-based field usage graph G_U for a program P is represented by a pair of sets $G_U = (N_U, E_U)$ in the sense of Definition 2 with the data usage abstraction set $U_P = U_T$.

Hence, the result of the type-usage analysis is a graph that contains function nodes and field usage nodes as data usage abstraction. Function nodes are connected to type-based field usage nodes if the structure type's field is accessed within the function.

2.2 Alias-based analysis

We used a flow-insensitive context-insensitive points-to analysis in the spirit of Steensgaard [7] to refine the analysis of data accesses. This analysis computes a partitioning of the memory used by the program into a set of abstract locations with points-to relationships between them. Locations correspond to sets of program variables or dynamically allocated memory regions, with one abstract location for each static call site of an allocator function. For the purposes of resolving the possible targets of function pointers, functions are also represented by special locations.

Each location may have one outgoing points-to edge. These edges model points-to information: an edge from location s to location t represents the information that one of the objects represented by s may point to one of the objects represented by t. If at some point the analysis finds that s may also point to some other location t', the two possible targets t and t' are unified to give a new target location for s's points-to edge. Unification merges the object sets of t and t', and recursively unifies their respective points-to targets if necessary.

The result of the analysis is a points-to representation that can be used to compute may aliasing information: Two objects may be aliased iff they belong to the same abstract memory location. Conversely, objects represented by different abstract locations are definitely not aliased.

The basic flow-insensitive context-insensitive algorithm can be implemented in almost linear time, requiring a single pass over the input program. Using a fast union/find data structure [8] to represent classes of unified memory locations, the algorithm's overall time complexity is $O(n \cdot \alpha(n, n))$. Here, n is the size of the input program and α is an inverse of Ackermann's function, which increases very slowly—the value of $\alpha(n, n)$ is always less than 5 in practice.

Our analysis treats each array as a single memory location; structures are treated as locations that are divided into distinct sub-locations. Casting between pointers is mostly ignored as it does not change which objects a pointer may point to; however, the analysis catches cases where a structure may be accessed through a pointer of inappropriate type. This type of access may invalidate any of the structure's fields, so in this case, the structure is 'collapsed' into a single object by unifying it with all of its fields.

The points-to effects of functions that are not present in the source code or possess special memory effects (such as allocation functions) may be abstracted by implementing function summaries, which are essentially small plug-ins to customize the analysis. Such summaries have the additional advantage that summarized functions can be treated in a context-sensitive way; for instance, the summary for `memcpy` will not cause aliasing between the function's arguments at different call sites. We have implemented summaries for a number of standard C and POSIX functions as was necessary for our experiments.

The points-to analysis safely ignores pointer arithmetic, assuming that arithmetic is not used to compute addresses for memory accesses outside the original object's bounds. (Such accesses are deemed undefined by the C standard.) As is common in the literature, our analysis also ignores obscure corner cases such as writing a pointer to a file and retrieving it again.

The results of the points-to analysis can be used to replace the type-based data usage analysis with a more precise one. The following definition adapts the concept of a reference to a structure field to the results of the points-to analysis.

Definition 6 (*Abstract alias-based field usage*) An abstract alias-based field usage is a pair (l, d) where l is an abstract memory location (computed by points-to analysis) of some marked structure type s, and d is a marked field of s. The set U_A is the set of all abstract alias-based field usages for a given program.

$$\begin{aligned} U_\mathrm{A} = \{(l, d) \mid\ & l \text{ is an abstract location of type } s \ \wedge \\ & s \text{ is a structure type } \in P \ \wedge \\ & d \text{ is field of } s \ \wedge \\ & s \text{ and } d \text{ are marked}\} \end{aligned} \tag{7}$$

The set U_A is a refinement of U_T for any given program that does not contain casts between pointers to structures: for any $(s, d) \in U_\mathrm{T}$ there can be several different $(l, d) \in U_\mathrm{A}$ with structure type s as the type of location l, and some field d. Those different pairs refer to field d in different instances of the structure.

The following definition instantiates the concept of the usage graph with the alias-based field usage abstraction.

Definition 7 (*Alias-based field usage graph*) The alias-based field usage graph for a program P is a graph $G_\mathrm{U} = (N_\mathrm{U}, E_\mathrm{U})$ in the sense of Definition 2, with the data usage abstraction set $U_\mathrm{P} = U_\mathrm{A}$.

As the alias-based field usage graph has the same structure and interpretation as the type-based field usage graph, the component recognition analysis can proceed in exactly the same way for both. The following section compares these two variants of the analysis; other variants on different kinds of usage graph are also possible.

2.3 The cohesion analysis

Based on the formal definitions provided so far, Algorithm 1 denotes the algorithm of the Cohesion Analysis for component recognition. Performing the analysis presumes the following work-flow:

1. **Annotate function pointers.** Any function pointer within the input program P has to be associated with

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

Algorithm 1: Type-based cohesion analysis

exactly the function(s) it will point to at run-time. Within the analyzed AUTOSAR framework, function pointers are often used to implement configuration-time late binding. The address of a function pointed to by the function pointer is manually set to the address of an available function after compile-time as part of the system's configuration. As this address is not known at compile-time, and thus is not available at analysis-time, manual annotation with points-to information, or static assignments of the addresses of configured functions to the respective function pointers at source-code level, is mandatory for the algorithm to calculate valid results.

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, all functions of interest (within this article also referred to as relevant functions) have to be marked by a domain expert. In practice, the set of irrelevant 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 (Algorithm 1, line 3).
3. **Mark characteristic structure fields.** The most important task of a domain expert within the proposed workflow is the identification of relevant data structures, respectively, their relevant fields. Some structures are closely related to exposable component functionality, but other ones are used for internal purpose only. Fields that can easily be identified and be flagged as irrelevant are for example pointers to global resources like timers, logging facilities, or system function tables that are commonly used but do not indicate cohesion. By taking only relevant fields into account, the algorithm is sensitive to dedicated component functionality only. 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,

which calculates the set of all relevant and therefore marked type/field-name pairs D_M via set-subtraction of $\overline{D_M}$ from the set of all type/field-name pairs of P (S_P)(Algorithm 1, line 4).
All three steps so far imply manual work by domain experts. However, as result we obtain the input data, required by the Cohesion Analysis algorithm: the annotated source code of the program under analysis, a set of irrelevant functions, and a set of irrelevant structure fields. All following steps are part of the algorithm's implementation and are therefore executed automatically.

4. **Calculate the undirected call graph.** An undirected call graph, $G_C = (N_C, E_C)$, is computed from the AST of program P 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 (Algorithm 1, line 16–17).
5. **Calculate usage graph.** A usage graph, $G_U = (N_U, E_U)$, is computed where functions and type-based field usages are represented by nodes, N_U, and field access within a specific function is represented by an edge, E_U, between the function node and the type-based field usage node. To identify field accesses, the occurrence of arrow- and dot-expressions within the AST is analyzed (Algorithm 1, line 10–15).
6. **Calculate component graph.** A component graph, $G_P = (N_P, E_P)$, is calculated by creating a set union of the undirected 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 (Algorithm 1, line 20).
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. The algorithm performs the extraction via a reachability calculation. Its output C_P is a set of components, where each component is represented by a set of functions and structure fields contained within the component (Algorithm 1, line 21–26).

Finally, a domain expert may further group multiple of the automatically computed decoupled components into single components, if desired. This step becomes rather handy, if the analysis identifies many small components. Although large numbers of very small components imply high potential for optimization during automatic middleware synthesis, it is feasible to combine some into larger components for reasons of economy in maintenance and versioning.

In Algorithm 2, we illustrate the salient difference between the type-based and alias-based variants of the cohesion analysis. The algorithm fragment shows only the part of the algorithm concerned with identifying field usages. In fact, the only difference to Algorithm 1 is on line 11: Rather than partitioning structure expressions only by type, this version uses the finer equivalence classes computed by alias analysis. This is the only change needed to refine the cohesion analysis to take the results of alias analysis into account.

```
// collect structural data usages
10 case lhs.rhs or lhs -> rhs   // dot or arrow expr.
11 |  s ← abstract location of the object denoted by lhs
12 |  d ← structure field name of rhs
13 |  if (s, d) ∈ D_M then
14 |  |  N_U ← N_U ∪ {f} ∪ {(s, d)}
15 |  |  E_U ← E_U ∪ {{f, (s, d)}}
```

Algorithm 2: Fragment of alias-based cohesion analysis (only the difference to Algorithm 1 is shown (see line 11))

```
struct S { int x; };
struct S s1, s2;

void foo() {
    s1.x = 23;
}

void bar() {
    s2.x = 42;
}

void sna() {
    /* ... */
}

int main() {
    sna();
    foo();
}
```

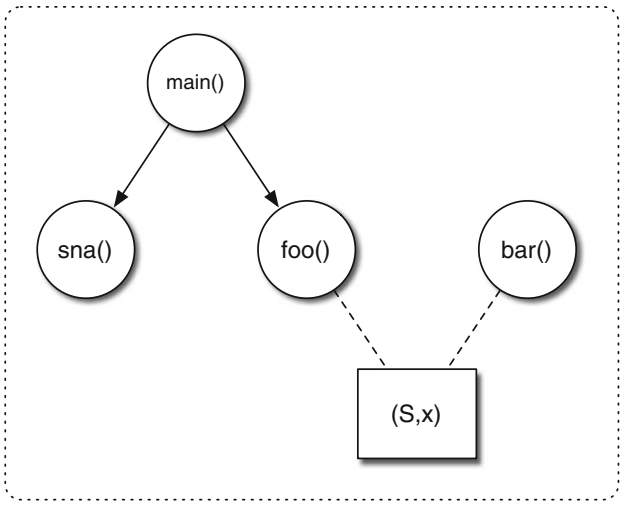
Fig. 2 Example source code for comparison of analysis variants

2.4 Example: type-based and alias-based analysis

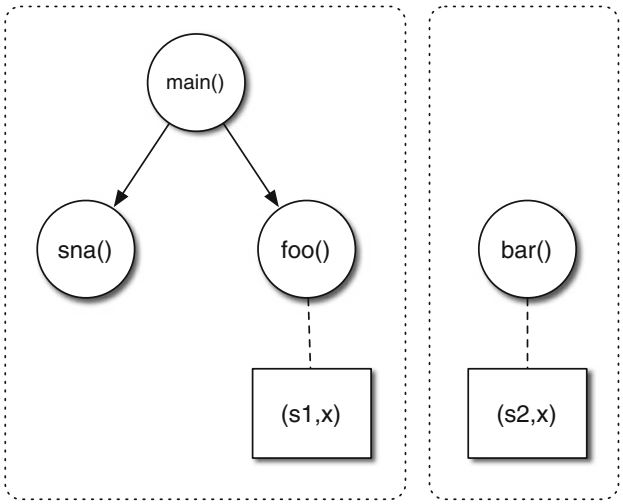
We illustrate the differences between the two variants of analysis with a basic example. Figure 2 gives a small program, and Fig. 3 shows the results for the different kinds of analysis.

In Fig. 3, functions are represented by circles, data usages as rectangles; call graph edges are drawn as arrows, usage graph edges between functions and data usages as dashed lines. The dotted boxes show the manually achieved reference decomposition. The analysis computes a valid solution if all connected components fit into exactly one dotted box (which is the case for our analysis). The ideal solution has exactly one connected graph in each dotted box—hence, a smaller number of graphs in a dotted box represents a more precise result. The example assumes that all functions and structure fields are marked as relevant, none are ignored.

The program's `main` function calls both `sna` and `foo`; they must therefore be grouped together in the same component. The differences between the analyses arise with the handling of structure field accesses: In the results of type-based analysis, both assignments to structure fields are



(a) Result of type-based analysis



(b) Result of alias-based analysis

Fig. 3 Results of analysis variants for example source code

modeled as a single pair consisting of structure `S` and field `x`. Thus both functions that access this field, `foo` and `bar`, must also be placed in the same component, as is shown in Fig. 3a. In contrast, alias-based analysis keeps structure instances `s1` and `s2` apart. This removes the inaccurate data coupling between `foo` and `bar`. This is an example of a case where the alias-based analysis results in a more refined decomposition into two components as shown in Fig. 3b.

3 Type-based versus alias-based analysis

The type- and alias-based component recognition analyses were compared using a total of four different variants of the general analysis on a real-world software system. Two parameters of the analysis were varied: The sets of abstract data usages were computed using either type- or alias-based analysis, and both of these analyses were tested with a larger and a smaller set of marks (see Definition 1) for relevant structure fields and functions. The marks were annotated in the form of filter rules (negative marks), hence are referred to as filters or filter sets for the remainder of this paper. For the given source code under analysis, two different sets of marks were defined: for the large set 87 functions were marked as relevant, while 50 fields within 12 data structures were filtered out, therefore were not marked; for the smaller set of marks the same 87 functions were marked, while only 39 fields within 8 structures were filtered out. Accordingly, the amount of manual annotations (marks) was reduced by about 22 %.

Both versions of the cohesion analysis operate on the same data structures—a call graph, a usage graph, and a component graph—thus they can be directly matched for comparison. To get a sound comparison of the type-based and the alias-based cohesion analysis, both versions were applied to the same source code, and were guided by the same manual annotations and marks. The analyzed source code is an industrial implementation of an AUTOSAR [9] communication stack. It is real-world C-code characterized as shown in Tables 1 and 2. The program's abstract syntax tree (AST) contained 291,794 nodes, which were traversed only once by both versions of the analysis.

In addition, we manually decomposed the AUTOSAR communication stack implementation as described in [2], to obtain at least one solution that must be found by both versions of the cohesion analysis. The manual decomposition identified 8 components for the FlexRay Interface- and the FlexRay Driver-Layer. They were called *Base*, *Transmitter*,

Table 1 Source code characteristics

	# files	LOC	kB
FlexRay interface			
Header	4	1,620	59
Impl.	15	4,192	135
FlexRay driver			
Header	13	1,660	88
Impl.	27	7,142	222

Table 2 Program characteristics

Source code construct	# occurrences
Function definition	107
Function call expression	431
Address-of operator	12
Pointer dereference expression	241
Arrow and dot operator (field access)	457
Cast expression (explicit and implicit)	4,924

Table 3 Different instances of structures as identified by alias-based analysis

Static type and instance characteristics	Count
Structure types	16
Structure instances	25
Singleton structures (only one instance)	12
Max instances per type	7

Receiver, *Time Services*, *Status*, *MTS*, *WUP*, and *TransceiverDrv*. We consider that this decomposition is near optimal. Consequently, a good cohesion analysis algorithm has to provide similar results in terms of coupling and cohesion. All of these were found by the analyses and are thus depicted in Fig. 4 in Sect. 3.2, which compares the results of the four analysis variants.

The effects of points-to analysis on the example application are summarized in Table 3. These numbers provide an intuition about the results to be expected from alias-based component recognition in comparison to the type-based analysis: For structure types with many independent instances at run time, the points-to analysis should ideally identify several instances of the structure and thus eliminate false coupling edges between functions accessing distinct instances. In contrast, if some structure has only one run time instance, pointer analysis cannot improve the analysis that simply checks types of field accesses.

In fact, of the 16 structure types in the program, points-to analysis found only one instance each for 12 structure types. In some cases, this could be attributed to the simplicity of the very fast pointer analysis, but several structures do indeed only have a single global instance accessed from many functions. Points-to analysis successfully identified several independent instances for four structure types, with no fewer than seven instances of one of these structures. Overall, these numbers indicate that it is reasonable to expect some improvements in alias-based component recognition over the type-based variant. The next two sections provide a detailed comparison of these two approaches on the example application.

3.1 Measurements

An analysis run starts by parsing the input program and building appropriate intermediate representations for subsequent analysis stages. These parts are always identical regardless of the details of the actual analysis. Execution time and memory usage data for these phases are listed in Table 4. The input is parsed using the commercial frontend from Edison Design Group (EDG) [10]. The intermediate representation of the program is the AST provided by the ROSE program analysis and transformation framework [11]. The interprocedural control flow graph (ICFG) computed by SATIrE [12] is used as the basis for the call graph.

Table 4 Execution characteristics for parts shared between analysis variants.

Preparation phase	Time (ms)	Memory (MB)
EDG frontend	50	10
ROSE AST construction	1,360	41
ICFG construction	310	37

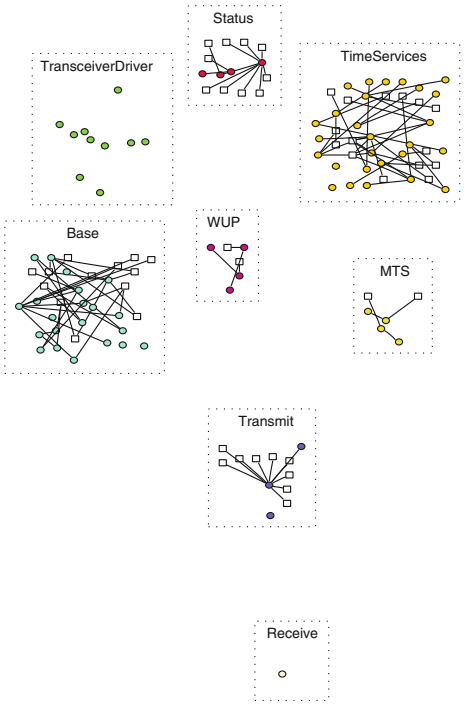
Table 5 Execution times for analysis variants (in ms)

Type-based analysis		Alias-based analysis	
Large filter set			
Collection	15.9	Collection	55.3
Extraction	8.1	Extraction	8.7
Total	24.0	Total	64.0
Small filter set			
Collection	15.9	Collection	55.3
Extraction	9.1	Extraction	10.0
Total	25.0	Total	65.3

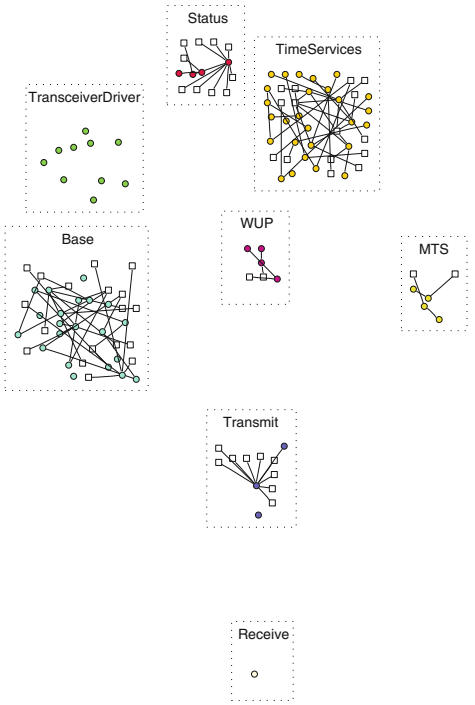
Table 5 summarizes execution time measurements for the four analysis variants, identified by their use of filter rules and the kind of usage analysis. Two analysis phases are distinguished: collection of field usages and component extraction; the latter includes construction of the component graph. Times reported are in milliseconds and were obtained from measurements of the total time for 100 runs of each phase of the analysis. All measurements were made on a machine with an Intel Xeon CPU clocked at 3 GHz, with 4,096 kB of cache and 24 GB of main memory. All timings are CPU time, i.e., the time for the frontend does not include disk I/O time.

The total analysis time is always dominated by the collection of field usages. An algorithm that scales well is therefore essential for this phase of the analysis. Both the type- and the alias-based analysis ensure good scaling properties, with the alias-based analysis taking about 3.5 times as long as the type-based analysis. This factor may seem large, but in absolute numbers, both analyses are still on the order of tens of milliseconds for our example program; further, the almost-linear scaling properties of the points-to analysis ensure that both algorithms will be very responsive even for much larger programs.

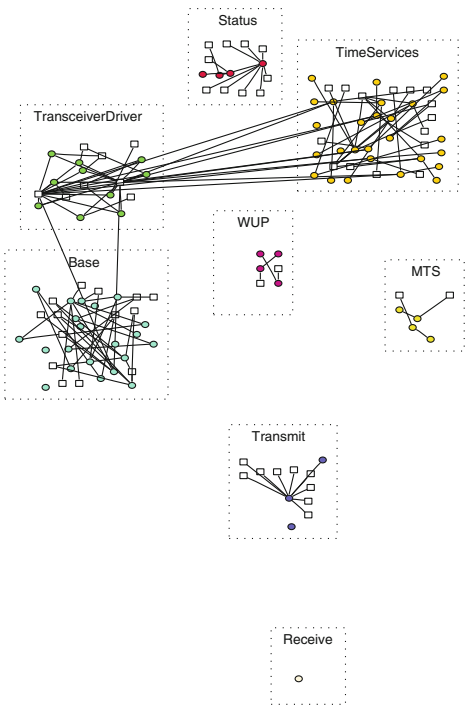
The measurements of the component extraction times reflect effects of filter set size and component graph size: Creation of the component graph, and the reachability calculation for identification of candidate components, take longer for the larger graphs computed using the results of alias-based analysis. The times also vary with the set of structure field filters because a smaller number of filter rules also causes



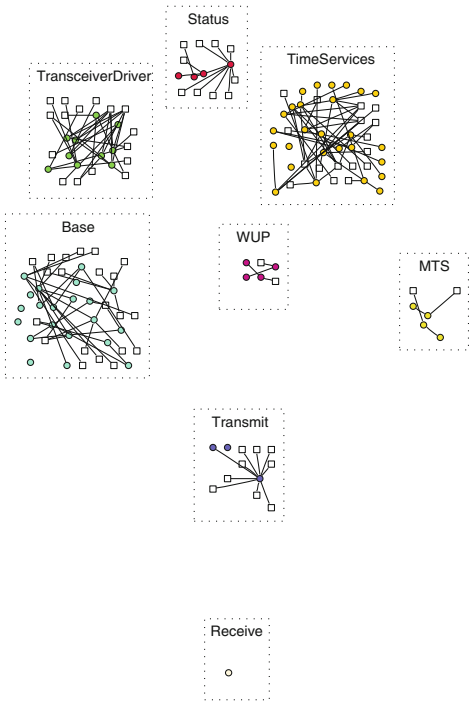
(a) With type-based analysis and the large filter-set. Number of connected components: 24



(b) With alias-based analysis and the large filter-set. Number of connected components: 24



(c) With type-based analysis and the small filter-set. Number of connected components: 14



(d) With alias-based analysis and the small filter-set. Number of connected components: 16

Fig. 4 Results of component recognition analysis for four different configurations: type-based versus alias-based analysis and the small filter set versus the large filter set. Dotted frames denote the manual decomposition, circles represent functions, squares represent structure field instances

Table 6 Memory characteristics of analysis phases

Analysis phase	Memory (MB)
Type-based usage collection	0.5
Alias-based usage collection	0.6
Component extraction	0.2

a larger graph. In contrast, our current implementations of field usage collection are independent of filter rules.

Table 6 lists memory usage characteristics for the phases of the analysis displayed in Table 5, using the same input program. As the numbers show, the analysis is very economical in terms of memory usage. Alias analysis needs slightly more space than type-based field usage analysis; the component graph resulting from alias-based analysis is also larger, but the difference is too small to show in the table. As will be discussed in the next section, the alias-based analysis enables us to perform better component recognition.

3.2 Comparison

Figure 4 shows a comparison of the results of the four different variants of our analysis on the same input program. The results are laid out in a 2×2 matrix corresponding to Table 5 to enable easy matching of analysis measurements with the respective results. Each subgraph shows a division of the program into components represented using dotted frames; these frames correspond to the manual decomposition of the program. Inside each component, its functions are displayed as colored circles, while structure fields are shown as squares. Edges between function nodes represent coupling via function calls, while edges between functions and structure fields represent coupling via data usages. The number of connected components is determined automatically by the algorithm; the total number is stated for each variant. If the number of connected components matches exactly the number of manually determined components we would have an exact result. Our results show that we get close to the (purely) manually determined decomposition as published in [2].

Figure 4a was previously published to demonstrate our type-based analysis using a certain set of function and structure field filters [3]. Starting from this result we varied two parameters of the analysis to obtain four different graphs, as described above: for the graphs in the first row (Fig. 4a, b), the analysis used the original larger set of structure field filters, while for the second row (Fig. 4c, d) it used a smaller subset of filters. The graphs in the first column (Fig. 4a, c) were obtained using type-based data usage analysis, while the two in the second column were computed using aliasing information from our points-to analysis. Thus moving from the first to the second row means a reduction in domain knowledge needed, whereas moving from the left to the right column means a reduction in false connections due to overly conservative analysis.

The main contribution of the paper is the improvement in the direct comparison of Fig. 4a, d, moving from a coarser type-based analysis with a large amount of domain knowledge in the filter set to a more precise alias-based analysis with less domain knowledge needed—the analysis is more precise with less manual intervention. We claim that alias-based type usage analysis enables better component recognition than type-based analysis, at a negligible cost in computational complexity. Alias-based analysis resulted in a decomposition of the example application into the same set of components in both cases. However, due to points-to analysis, we were able to reduce the number of filter rules. As the work of manually specifying which edges to ignore is the largest part of the decomposition effort, this is a very desirable outcome. Further, we did not simply obtain the same decomposition in both cases: the alias-based analysis resulted in a *better* decomposition in which the internal coupling of components is much more visible. This is especially apparent in the *TransceiverDriver* component, which shows tight internal coupling in Fig. 4d; in the type-based result of Fig. 4a, all these structure field accesses had to be filtered away.

Comparing Fig. 4a and Fig. 4c, we can see that a smaller set of filters means that we obtain coupling edges between members of different manually validated components; this smaller filter set combined with purely type-based data usage analysis would thus result in a quite coarse and logically incorrect decomposition into only six rather than eight components.

It turns out, however, that these inter-component edges in Fig. 4c are merely artifacts of using the overly simple type-based data usage analysis: they represent usages that are not present in reality—the functions involved actually access different instances of the same structure type. To keep these components apart, the human domain expert thus had to insert filter rules that might contradict his knowledge of the actual importance of these structure fields in the program. Alias-based data usage analysis removes these false coupling edges; as can be seen in Fig. 4d, alias-based analysis results in the desired decomposition even with the smaller, logically more correct set of filters.

The visible differences between Fig. 4a and b are minor. Although not immediately apparent from the graph, the *Base* component in Fig. 4b contains five more structure fields. This is the result of the alias analysis, which can distinguish between two instances of the same structure type in two different functions in this component. Alias analysis would identify many more structure fields that are not shared, but these results are not visible in Fig. 4b because the large filter set keeps them from being displayed in the graphs. Moving from Fig. 4b to the smaller filter set of Fig. 4d, all of these structure fields become visible.

4 Related work

Lee et al. [13] describe a methodology to recognize components within existing object-oriented source code 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. To keep their analysis effort small, their approach relies on domain knowledge, mainly extracted from UML use-cases and architectural descriptions. In contrary, the component recognition algorithm described within this paper is intended to work on object-oriented but also on non-object-oriented source-codes.[1] It also relies on domain knowledge, which is represented by marks on functions and data structures, to keep the analysis's effort as low as possible.

Emami et al. [14] 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. The component recognition algorithm described within this paper is linear in size of the program, but requires function pointers to be pre-processed (annotated) in case of ambiguity. The integration of points-to information is not covered within this paper, but is subject to ongoing research. Nevertheless, the proposed algorithm's precision is sufficient to calculate the required properties of the input program.

The concept of abstract memory locations described in Sect. 2.1 was successfully applied within the following two publications: Hind, Burki, Carini, and Choi give experimental results of comparing flow-sensitive and flow-insensitive flow analysis algorithms in [15] based on memory locations associated with names. The call graph is constructed while alias analysis is performed in a similar way to the component recognition algorithm. 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 [16]. They use redundant load elimination to demonstrate the effectiveness of the algorithms in terms of opportunities for optimization.

Continuing advances in pointer analysis, which now scales to millions of lines of code even in the flow-sensitive case [17], could be incorporated in our algorithm, giving good decompositions with even fewer filter rules to discriminate objects.

[1] The source code analyzed within this paper is a full-fledged C code.

5 Conclusion and perspectives

Within this article we compared a type-based and an alias-based version of a static cohesion analysis for decomposition of embedded systems software stacks. Both versions offer an almost linear algorithmic complexity; the alias-based version is slightly more expensive in terms of execution time and memory consumption, but is quite more complex w.r.t. its implementation. However, the alias-based version pays the additional implementation effort by cutting down the expenditure of work of domain experts marking relevant functions and types and defining filter rules.

Both analyses are well suited for typical automotive embedded systems codes that do not contain dynamic bindings and type-inconsistent pointer arithmetic. The analyses were benchmarked with an industrial implementation of an automotive communication stack and two variations of manually defined annotations that determine which functions and structure fields have to be considered by the analyses.

Our results show that the alias-based cohesion analysis provides a higher precision by generating a smaller set of solutions that contain more tightly coupled components, compared to a larger set of solutions calculated by the type based analysis. In addition, the alias-based version required fewer filters on irrelevant structure fields to find appropriate solutions, and hence requires less expert knowledge on the domain of the analyzed source code.

We achieve a similar result with our semi-automatic approach as in a previously performed manual decomposition [2], but with significantly less effort required by user. Our evaluation also shows the impact of the precision of the used analyses on the number of filter rules.

The presented semi-automatic recognition of components supports the timing annotation of embedded systems software at the interface level. We consider the annotation at interface level of increasing importance as embedded systems software grows, making a component-oriented design increasingly important.

It is one of the important results of the ALL-TIMES project that the proper combination of manual timing annotations and timing information extracted by tools from existing source code, is crucial for the productive engineering of embedded systems code with certain timing constraints. The timing annotation on multiple abstraction levels is a crucial aspect when using timing tools, and the interface level is one that users can easily relate to. It therefore serves as a good basis for fast comprehension of the timing behavior of an embedded systems code. We also consider the re-engineering and re-factorization of legacy code and its decomposition into components an important aspect that we expect to become increasingly important in future, as existing codes are integrated into larger code bases. This article is a contribution to this endeavor and demonstrates the practicability of the approach

with an existing implementation of an automotive software stack.

Within our ongoing research, we try to improve both versions of the cohesion analysis by introducing weighted edges within the undirected call graphs and the usage graphs. By these means we aim at an increase of the analyses' precision and also at an application to other domains. Also of interest is the application of the algorithm to even larger code bases.

Acknowledgments This work has been partially funded by the research project "Integrating European Timing Analysis Technology" (ALL-TIMES [18]) under contract No. 215068 funded by the 7th EU R&D Framework Programme.

References

1. Hegde, R., Hegde, M., Gurumurthy, K.S.: A paradigm shift from legacy to AUTOSAR architecture in future automotives. In: Das V.V., Thankachan N. (eds.) Computational intelligence and information technology. Communications in Computer and Information Science, vol 250, pp. 548–553. Springer, Berlin (2011). doi:10.1007/978-3-642-25734-6_94
2. Galla, T.M., Schreiner, D., Forster, W., Kutschera, C., Göschka, K.M., Horauer, M.: Refactoring an automotive ebedded software stack using the component-based paradigm. In: Proceedings of the Second IEEE International Symposium on Industrial Embedded Systems (SIES 2007), pp. 200–208. IEEE (2007)
3. Schreiner, D., Schordan, M., Barany, G., Göschka, K.M.: 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), IEEE, pp. 463–468 (2008)
4. SATIrE: http://www.complang.tuwien.ac.at/satire Static Analysis Tool Integration Engine
5. Schreiner, D., Schordan, M., Knoop, J.: Adding timing-awareness to autosar basic-software—a component based approach. In: Proceedings of the 12th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2009), IEEE, pp. 288–292. IEEE Computer Society (2009)
6. Lakhotia, A.: A unified framework for expressing software subsystem classification techniques. J. Syst. Softw. **36**(3), 211–231 (1997)
7. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, pp. 32–41. ACM New York (1996)
8. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (1975)
9. Heinecke, H., Schnelle, K.P., Fennel, H., Bortolazzi, J., Lundh, L., Leflour, J., Mate, J.L., Nishikawa, K., Scharnhorst, T.: 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 and Exposition on Transportation Electronics. SAE-2004-21-0042, Detroit, MI, USA (2004)
10. Edison Design Group, Inc.: EDG web site. http://www.edg.com/
11. Schordan, M., Quinlan, D.: Specifying transformation sequences as computation on program fragments with an abstract attribute grammar. In: Proceedings of SCAM '05, Washington, DC, USA, pp. 97–106. IEEE Computer Society (2005)
12. Schordan, M.: Source-to-source analysis with SATIrE—an example revisited. In: Proceedings of the Dagstuhl Seminar 08161: Scalable Program Analysis, p. 17. Dagstuhl (2008)
13. Lee, J.K., Seung, S.J., Kim, S.D., Hyun, W., Han, D.H.: Component identification method with coupling and cohesion. In: Proceedings of the Eight Asia-Pacific Software Engineering Conference 2001 (APSEC 2001), Washington, DC, USA, pp. 79–86. IEEE Computer Society (Dezember 2001)
14. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. ACM SIGPLAN Notices **29**(6), 242–256 (1994)
15. Hind, M., Burke, M., Carini, P., Choi, J.D.: Interprocedural pointer alias analysis. ACM Trans. Program. Lang. Syst. (TOPLAS) **21**(4), 848–894 (1999)
16. Diwan, A., McKinley, K.S., Moss, J.E.B.: Type-based alias analysis. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI), New York, NY, USA, pp. 106–117. ACM, New York (1998)
17. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: Symposium on Code Generation and Optimization (CGO) (2011)
18. ALL-TIMES Consortium: Project ALL-TIMES: Integrating European Timing Analysis Technology. http://www.all-times.org/