Adding Timing-Awareness to AUTOSAR Basic-Software – A Component Based Approach

Dietmar Schreiner[1], Markus Schordan[2], Jens Knoop[1]
[1]Vienna University of Technology, Institute of Computer Languages
[2]University of Applied Sciences Technikum Vienna, Institute of Computer Science
Vienna, Austria

{schreiner,knoop}@complang.tuwien.ac.at, {schordan}@technikum-wien.at

Abstract

AUTOSAR as specified in its current version fosters timing-constraints at application level to support the development of real-time automotive applications. However, the standard's actual specification does not consider any timing information for its Basic Software. In consequence, the over-all timing-behavior of software running at a specific system node can not be calculated and thus validated at development-time. Even worse, any exchange or modification of Basic Software modules induces unpredictable alteration in system timing, and may even lead to a severe miss of execution deadlines. Within this paper we solve this issue by using a component based Basic Software architecture, as much as timing-aware software composition for Basic Software components. We therefore introduce timing contracts as enhancement for existing interface contracts, specify a timing annotation language on a conceptual level, and demonstrate how to capitalize on our approach by calculating timings within composed Basic Software architectures at development time.

1 Motivation

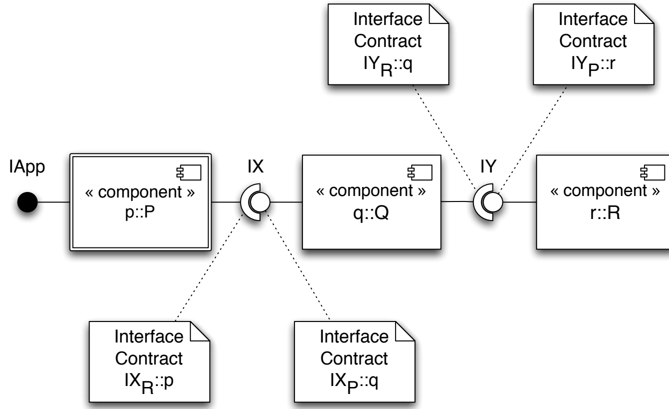
Driven by steadily increasing requirements of innovative applications, automotive electronic systems have reached a level of complexity that requires a technological breakthrough in order to manage them cost efficiently and at high quality. In contrast to other domains, like e.g. avionics, automotive electronic systems are produced in comparatively large quantities (69,1 million vehicles in 2006). Therefore, the price per unit has to be as low as possible, forcing manufacturers to assemble economical, and thus resource constrained electronic building blocks. In consequence, automotive software has to cope with harsh restrictions, especially with limitations of available memory, processing power, and system timing.

The Automotive Open System Architecture (*AUTOSAR*) [1], an upcoming industry standard within the automotive domain, reflects these facts by constituting Component Based Software Engineering (*CBSE*) [10, 2, 11] as development paradigm for automotive applications. CBSE is well accepted within the embedded systems community, as it provides a clear separation of concerns, and hence facilitates extensive software reuse. Therefore, in *AUTOSAR* application concerns are covered by software components, while infrastructural ones are handled within layered component middleware—the *AUTOSAR Basic Software* (*BSW*). This design leads to an increase in application quality, reusability, and maintainability, and consequently to a reduction of costs and time-to-market.

However, one big issue still unsolved within present releases of the *AUTOSAR* standard is that of improved real-time support. While the actual version[1] of *AUTOSAR* incorporates timing-assertions for *Software Components* at application-level, the *AUTOSAR Basic Software* lacks on mechanisms to specify timing guarantees and requirements. *BSW* building blocks—the modules—are standardized in terms of their interfaces—function signatures and datatypes—only, but are completely unaware of execution-times as much as of timing-requirements. In consequence, any exchange of standardized *BSW* modules may lead to severe, unpredictable variances in the system's over-all behaviour.

Within this paper we propose a methodology for timing-aware software-composition for the *AUTOSAR BSW*. We rely on a component based design of the *AUTOSAR BSW* [4, 8], which allows us to specify a timing-aware composition standard. A sound composition fulfills all timing contracts associated with the composed component interfaces; the timing contracts provide vital information on execution times as much as on control-flow dependencies within and in between composed components. As a result, the timing-

[1] At the time of writing this paper, the actual public *AUTOSAR* release is 3.1



(a) Component Architecture

```
application
interface IApp{
   void main();
}

interface IX{
   void f();
   void g();
}

interface IY{
   void u(int x);
   void v(int x);
   void w(int x, int y);
}
```

(b) Interface definitions

Figure 1. Demonstrator application

behavior of a given *BSW* component architecture can be calculated and thus be verified at development time.

2 Component Based Basic Software

In our previous work [8, 4, 9], we demonstrated how to replace *AUTOSAR*'s conventional, layered basic software architecture by a component based design that completely resembles the standard's functionality. Component based *BSW* provides means for software reuse, increased flexibility, and a reduced time-to-market, due to the use of prefabricated building blocks. Moreover, timing-aware software composition becomes available within *AUTOSAR BSW* by capitalizing on some basic principles provided by the component paradigm:

1. **Component Definition:** A component is a trusted unit of execution with well defined points of interaction—the interfaces—and no other external dependencies. As a consequence, timing-information has to be provided at interface-level only. Therefore, our approach introduces timing contracts bound to specific component interfaces.

2. **Composition Standard:** Components are assembled in accordance to a composition standard. The composition standard specifies how interfaces are validly connected, and which constraints have to hold for a sound composition. For the augmented *AUTOSAR* standard, *BSW* components have to be deployed within the same electronic control unit (*ECU*) and thus allow local composition only. Our approach introduces timing constraints, formalized within timing contracts that are associated with the components' interfaces.

3. **Interaction Standard:** Composed components interact at run-time in accordance to a prescribed interaction standard. Within *AUTOSAR Basic Software* components may interact either (i) via function calls or (ii) via shared memory. The proposed timing contracts contain information at function level only as interaction via shared memory does not exist in isolation (it is always contained within a function).

For the purpose of this paper, the proposed approach is demonstrate with a simplistic, artificial component architecture instead of the full-fledged *BSW* stack. This is done for reasons of space and comprehension. However, the described methodology can easily be applied to a real *BSW* stack by the reader without modification.

The demonstrator application is depicted in Fig. 1(a) using a UML 2.0 component diagram: Three components p,q, and r are connected in sequence via two interface types *IX* and *IY*. Each component is one specific implementation of a given component class P,Q, and R. As denoted by UML interface artifacts, component class P provides the *IApp* interface, and requires functionality of interface *IX*. Component class Q provides an implementation of interface *IX* and requires functionality of interface *IY*, which is provided by component class R. The composition is achieved by connecting the associated provided- and required-interface of related components ((P,Q), (Q,R)) in accordance to the underlying composition standard. The interfaces are specified in Fig. 1(b). Interface *IApp* provided by component class P denotes our application's starting-point and hence contains a global *main*-function.

The interface contracts explicitly denoted within Fig. 1(a) contain the precise specification for each interface. Typically interface contracts are assigned at component class level. As part of this paper's contribution, these interface contracts will be extended by timing contracts for each interface of each component implementation. Hence, timing contracts are related to specific implementations instead of component classes, as timing depends on imple-

mentation details. To express this fact, the interface contracts are labeled $Name_{Type}@ComponentInstance$, denoting the specific component implementation they relate to. For example, $IX_R@p$ is the required-interface of type *IX* of component implementation *p*.

3 Timing Contracts

To strengthen the reliability and predictability of component based software architectures, guarantees, and assertions on properties and behaviours of assembled building blocks are formalized in contracts [5, 7, 2, 3]. Contracts typically consist of two obligations, one regarding the client that requires functionality, and a second one regarding the provider of required functionality. To comply with a given contract, both—client and provider—have to satisfy their obligations.

Building upon a component based design for *AUTOSAR Basic Software*, timing-awareness can be added by extending the standardized interface contracts, and by refining the component model's composition standard. Thereto real-time constraints and timing-assertions have to be formalized within timing contracts that further on have to be fulfilled by any sound, timing-aware composition.

To describe timing information as precisely as possible a timing contract has to contain at least the following items that are explained in more detail within the next sections:

1. **Annotated Call graph:** The local call graph of a component's interface denotes the call dependencies between functions implemented by the component.

2. **Timing table:** A timing table contains parametrized execution times for each function of the component. These execution times may either be calculated by analyses or may be measurement based.

3. **Platform specification:** To fully categorize a binary component, the platform specification has to provide information on the chosen target platform, all hardware attributes affecting execution times as much as on used build-tools like e.g. compiler versions. A classification of platform attributes is out of scope of this paper. To demonstrate timing-aware software composition, it is sufficient to keep in mind, that platform specifications of composed components have to match (also see Sect. 3.3).

Although timing contracts contain the same three topics for required- as much as for provided-interfaces, the topics' meaning differs:

Within contracts for provided-interfaces, the timing table contains WCET assertions in form of parametrized execution times. Within contracts of required-interfaces, the timing table contains obligatory upper bounds for execution times, demanded by the component. Therefore, when negotiating execution times during composition time, following simple equation has to hold for each implemented and exposed function:

$$WCET_{provided} \leq WCET_{required}$$

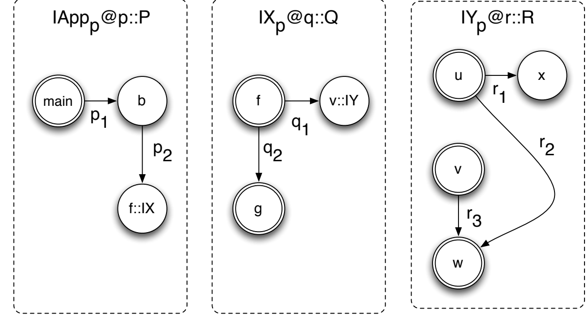


Figure 2. Call graphs for timing contracts

While timing tables are mandatory within contracts for provided-interfaces, they are optional for contracts for required-interfaces.

Call graphs within contracts for provided-interfaces expose implementation specific details of the component, which are used to calculate parametrized execution times. In contracts of required-interfaces, a call graph for required functions explicitly prescribes call dependency within another component. As this connotes a contradiction to the idea of exchangeable black-boxed components, call graphs within timing-contracts of required-interfaces are omitted.

3.1 Annotated Call Graphs

As components are black-boxed architectural entities that are defined by their interfaces, any implementation details are typically completely concealed. Hence, timing information has to be provided on the components' outermost surface—the interfaces. Unfortunately components typically rely on external functionality provided by other (again black-boxed) components. Thus execution times of a component's functions often depend on execution times of functions provided by other components.

Consider following example: Component q implements function f (we will denote this fact as f::q), component p implements function g::p. For this example we assume that f::p depends on g::q; to say so, g::q is called within the implementation of f::p. In consequence f::p's execution time depends on that of g::q. As g::q's execution time is not known at the development time of p, f::p's execution time is also unknown. Even worse, function g could be provided by any other component implementing the proper interface, and thus could be replaced at any time. To handle this issue of unknown external dependencies, we conserve information on this type of program dependencies within call graphs as part of timing contracts.

A call graph of a given program represents call dependencies between the program's functions. The graph's nodes denote functions while directed edges denote calls from one function to another. Call graphs used within timing contracts are local call graphs, which means they express local dependencies within the scope of a single component. Thereby all functions exposed by provided-interfaces are considered to be top-level nodes within the graph. All other nodes are either locally implemented private functions of the component, or are required/called external functions—addressed via a function name and an interface type (e.g. f::IX). Due to the compositional nature of components, an external call may be mapped to any component implementation that provides the specified interface. Thus, all calls to external functions are represented by leaf-nodes and terminate a local call path. To express additional context information like argument intervals, graphs' edges may be annotated by this information or a context identifier that uniquely maps to the information.

To provide a clear picture on the used call graphs, Fig. 2 depicts the call graphs for our application's timing contracts. The top-level nodes—the components' functions accessible via the provided-interfaces—are depicted as double-lined circles (e.g. f::q), while private functions (e.g. x::r) and external functions (e.g. f::IX in b::B) are depicted as single-lined circles. In addition the call graph's edges are annotated by a context identifier (e.g. p_1) that later on is used in Fig. 3 to associate annotations with edges.

3.2 Timing Tables

Timing Annotation Language

For allowing the computation of the WCET of a component's functions we define a parametrized approximation formula (WCET-PAF). The parameters of a WCET-PAF allow to compute a WCET for a function, dependent on the value ranges that are provided for the parameters for each call of that function. The lesser is known about a function's run-time behaviour the less complex is the WCET-PAF, thus, if no structural knowledge can be gathered about a function, a constant execution time, specifying an upper bound of all executions is provided. We have shown in [6] that a context-sensitive flow-sensitive interval analysis provides a good basis for loop bound computations. Here we build on that experience by using intervals as input to the WCET computation of a WCET-PAF.

For the discussion here we restrict the types of variables to integers. The call graph's edges are annotated with the value ranges of each call's actual parameters. In Fig. 3 the annotations of two components' call graphs are shown. The context identifies (e.g. q_1) map to the labels within Fig. 2.

The annotations are attached to edges in the call graph and are used in the computation of the WCET with the timing tables.

$IX_p@q :: Q$	
$f \rightarrow v :: IY$	$q_1 = \{[(0, 120)]\}$
$f \rightarrow g$	$q_2 = \{[]\}$
$IY_p@r :: R$	
$u \rightarrow x$	$r_1 = \{[]\}$
$u \rightarrow w$	$r_2 = \{[(0, 1700), (0, 699)]\}$
$v \rightarrow w$	$r_3 = \{[(0, 12), (15, 25)]\}$

Figure 3. Annotations of edges in Fig. 2.

Function	WCET PAF
$f()$	18+WCET(g)+3*WCET(v::IY)
$g()$	79
$u(x)$	if x=0 then 15
	else 3*WCET(w)+x+11
$v(x)$	47+bound(x)*WCET(w)
$w(x, y)$	34*(bound(x))^2+105*bound(y)
$x()$	14

Figure 4. WCET-PAFs for functions of Fig. 3.

Table Details

A timing table shows how an approximation of the WCET can be computed for each function. This table corresponds to the call graph such that functions that are called are used in the PAF. If there is an edge $f_1 \rightarrow f_2$ in the call graph then there must exist a reference of f_2 in the PAF of f_1. An if-clause is used to express conditional execution, and '+' is used to compute the WCET of sequential execution. Multiplication combined with the WCET of a called function corresponds to the number of times that this function is invoked (i.e. a function call within a loop with known loop bound). In all other cases multiplication corresponds to the execution of statements within (possibly nested) loops or a statement sequence.

Additionally two special operators exist. Operator *bound* computes the length of an interval that is provided for a function and uses that length when evaluating the term. For example, let's assume the interval of x is $[5..10]$, then $(bound(x))^2$ evaluates to 36. The conditional *if-clause* allows a context-dependent selection of distinct execution paths. The condition is evaluated with interval arithmetic of the involved variables. If both branches are possible, then the highest WCET of both branches is used as computed WCET of the *if-clause*. Note that different calls of the same function can be annotated with different intervals of the actual parameters, i.e. edges to the same node with different interval-annotations in the call graph. This allows to compute a context-sensitive WCET for functions, according to the different calling contexts.

The WCET-PAF for each function is used in combination with the parameter intervals as they are annotated in the call graph when computing a timing calculation, as shown in the next section.

3.3 Platform Specification

On the contrary to timing tables, platform specifications are directly related to the binary representation of a component. Hence, the platform specification for timing contracts of provided- and required-interfaces is the same (driven by the component itself). The platform topic for example specifies for which target ECU the software component is compiled for, or which system configuration is required to guarantee the annotated execution times. As these properties exist in a wide variety, we do not provide a formal way to specify them.

3.4 Contract based Timing Calculation

Using WCET-PAFs, the timing for a path through an application is computed by building a term representing a complete computation of the timing, starting at a selected function (e.g. main) and evaluating that term by taking the interval-information for function parameters, as annotated in the call graph, into account.

The WCET-PAF term is built by starting with the PAF of the function of interest, and by iteratively substituting each referenced function in a WCET term with its respective PAF. As described above, edges in the call graph are annotated with intervals of actual parameters. Those intervals are now substituted for the formal parameters in a function's WCET-PAF and the obtained formula is substituted for the function call in the calling function's WCET-PAF.

For example, the WCET-PAF term for function f is built and evaluated as follows:

$$\begin{aligned} f &= 18+WCET(g)+3*WCET(v::IY) \\ &= 18+79+3*WCET(v(0..120)::IY) \\ &= 97+3*(47+bound(0..120)*WCET(w)) \\ &= 97+3*(47+121*WCET(w(0..12,15..25))) \\ &= 97+3*(47+121*(34*(bound(0..12))^2 \\ &\qquad +105*bound(15..25))) \\ &= 97+3*(47+121*(34*(169+105*11))) \\ &= 16341028 \end{aligned}$$

Thus, by using the WCET-PAFs and the call graphs' interval annotations, we can compute a WCET for each function.

4 Conclusion

Within this paper, we proposed a methodology on how to foster timing-awareness within *AUTOSAR Basic Software*. We thereby built upon a component based design for the *Basic Software* and thus capitalized on system properties, enforced by the component paradigm.

Our approach introduces timing contracts as enhancement to standardized interface contracts. These timing contracts are attached to each component implementation within the designed *BSW* architecture. By incorporating the timing contracts into the component model's composition standard, timing-requirements can be calculated and validated at system development time. In addition, any exchange of existing *Basic Software* components can now be validated in terms of altered system timing. The timing contracts contain annotated local call graphs for each implemented interface, and timing tables providing WCETs, or formulas that describe each functions timing behavior.

For demonstration purpose, we designed an artificial application. However, the proposed methodology can easily be applied to the full-fledged *Basic Software* stack.

Within our ongoing research, we aim at an improved extraction of timing tables from source code, using static analysis. Additionally we try to incorporate measurement based WCETs into our approach, to cope with hard-to-analyze functions.

References

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

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

[3] J. Fredriksson, T. Nolte, M. Nolin, and H. Schmidt. Contract-Based Reusable Worst-Case Execution Time Estimate. In *RTCSA*, pages 39–46. IEEE Computer Society, 2007.

[4] 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.

[5] B. Meyer. The Grand Challenge of Trusted Components. In *ICSE*, pages 660–667, 2003.

[6] A. Prantl, M. Schordan, and J. Knoop. Tubound - a conceptually new tool for worst-case execution time analysis. *Post-Workshop Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, 237:141–148, 2008.

[7] 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.

[8] D. Schreiner and K. M. Göschka. 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, pages 635–641. IEEE, 2007.

[9] 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)*, IEEE. IEEE, 2008. to appear.

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

[11] R. C. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, 2000.