

# Composability Concept for Dependable Embedded Systems

## – *Extended Abstract* –

Matthias Werner\*    Jan Richling<sup>‡</sup>    Nikola Milanovic<sup>‡</sup>    Vladimir Stantchev<sup>‡</sup>

## 1 Introduction

*Composability* is a sought after property. However, whereas there is usually an intuitive understanding of this concept, a clear definition has been frequently missing.

In this paper, we refine our concept of *composability* given in [12]. We see *composability* as a property of an *architecture* not of systems. In addition, we do not want to reduce composability to the functional specification; rather, we are focusing on how composability effects the *non-functional* properties, especially the composed system’s real-time behavior, its reliability, its security, etc.

We see the advantage of our approach in having a wide application domain. It works for functional and non-functional properties, it may consider both, hardware and software, and it allows for a description of composition anomalies [1, 2].

In this paper, we show how to apply our concept to an embedded hardware-software co-designed architecture and to composition and decomposition of dependable services in embedded environments.

### 1.1 Related Work

Already in 1996, a lack of a consistent definition for a composability concept was stated [6]. In fact, there was a slight improvement since then.

The term composability appears mainly in two (partly intersecting) areas:

- Software engineering in general, with a focus in Object-Oriented Programming (OOP) and Aspect-Oriented Programming (AOP)
- Actual architectures and systems, e.g., plug-and-play

In the first context, Bergmans gives a typical definition of the term composability [3]:

Composability allows for the modular specification of modules with multiple independent concerns.

Although this definition meets the intuitive understanding, it is not very helpful for our purposes. E.g., it leaves the open question concerning *what* composability is an attribute of.

More formal definitions, e.g., with the  $\pi$ -calculus, usually reduce composability whether two systems are (functional) *compatible* or not but are not open for things like *emergent properties* (cp. Section 2.2). Additionally, many approaches neglect non-functional properties.

Definitions of the second area are mostly not applicable outside its very specific domain. One (of a few) exception is a more general definition given by Kopetz [5]:

An architecture is said to be composable with respect to a specified property if the system integration will not invalidate this property, once the property has been established at the subsystem level. Examples of such properties are timeliness or testability. In a composable architecture, the system properties follow from the subsystem properties.

Similarly to Kopetz’s definition, we define composability as an attribute of an architecture. However, we do not limit it to the preservation of certain properties of system components, but distinguish different kinds of composability regarding properties of the composed systems.

We anticipate our approach to be advantageous and applicable in the different areas.

The rest of this extended abstract is organized as follows: Section 2 introduces our concept of composability and discusses its effect to composed systems’ properties. Sections 3 and 4 provide two examples for the application of the composability concept: the composable Message Scheduled System Architecture and the composable service architectures, e.g., web services. Finally, the paper concludes with a short summary in Section 5.

## 2 Basic Definitions

### 2.1 Architecture and Elements

For brevity we skip in this extended abstract formal definitions and derivations. The main idea of the architecture-centered composability concept may be found in [12].

Composability means dealing with systems and its components.

\*TU Berlin, [mwerner@cs.tu-berlin.de](mailto:mwerner@cs.tu-berlin.de)

<sup>‡</sup>Humboldt University,  
{[richling](mailto:richling@informatik.hu-berlin.de)|[milanovi](mailto:milanovi@informatik.hu-berlin.de)|[vstantch](mailto:vstantch@informatik.hu-berlin.de)}@informatik.hu-berlin.de

Components are the building blocks of a system. However, since composed systems can serve as components for other systems, we omit a distinction between systems and components and call both *elements*.

An element is a (discrete or continuous) state machine. I.e., it is represented by a tuple of variables  $(v_0, v_1, \dots, v_n, f_0, \dots, f_m)$ , where  $v_i$  denotes a state variable and  $f_i$  denotes a function variable.

The state variables represent the state of the element, whereas the functions describe the element's dynamics. If the element is static, the function variables are skipped. Within a certain element, certain variables may be bounded or bound to a single value. Other variables may remain free. Also, state variable may change their value with the time, regarding the function variables.

A core term in our concept is the *architecture*. An architecture is a set of rules and styles how to create systems.<sup>1</sup> More precisely, an architecture  $A$  is a tuple  $(E, O)$  where  $E$  is a set of *initial* (or *atomic*) elements and  $O$  is a set of *composition functions* (or *composition operators*).

We say that an element  $e$  belongs to (or is a valid element of) an architecture  $A$  if  $e$  is either one of  $A$ 's initial elements, or there is a *composition path* that describes a composition process<sup>2</sup> to get  $e$  by  $A$ 's initial elements and composition operators only.

A composition operator  $o \in O$  is a function that maps two elements to a third element. Please note, that  $o$  is a higher order function, if the elements contain function variables.

## 2.2 Composition and Properties

We are interested in properties of elements, and what happens to them during a composition process. The (observable) properties of an element  $e$  can be derived from the component variables. A property  $p$  of a element  $e$  is one of its variables, or it is the result of a function of one or more of the variables,  $p(e) = f_p(v_1, \dots, v_n, f_1, \dots, f_m)$ .

Most approaches to composability assume that any component in a system can be identified after a composition process, and that this elements keep its properties. Our approach is not limited in this way.

If an operator  $o$  combines two elements  $e_1$  and  $e_2$  to form an element (i.e., a system)  $e_S$ , it can do one of the following with the element variables:

- A variable remains unchanged. The variable still belongs to  $e_1$  or  $e_2$ , respectively, inside  $e_S$ . (value invariant component property)

<sup>1</sup>In the English language, architecture may have another meaning: the result of a construction process. We refer to the first meaning only.

<sup>2</sup>Sometimes we refer to the composition process only as "composition".

- A variable is bound to a(nother) value. Again, the variable belongs still to one of the elements. (invariant component property but changeable value)
- A variable is bound to another variable. Now the variable belongs to the system. (transferred property)
- New variables can be generated. These belong to the system. (emergent property)

Obviously, when composing, properties may change. Especially, there may be properties defined for a composed element which are not defined for the elements to compose. (emergent properties).

As an example consider the runtime of a program. The runtime is only observable if the program is composed with a concrete processor. For the pure code or the processor alone, the runtime property can not be defined.

Please note that some state variables may remain free, even after composing. These variables may serve as interfaces, thus an element can interact with its environment.

## 2.3 Composability

In our concept, composability is a property of the architecture with respect to a property of the architecture's elements. We distinguish between *reachability* properties and *safety* properties.

An architecture  $A$  is called *reachable composable* with respect to a property  $p$  iff there is a valid element  $e$  of  $A$  with  $p(e)$ .

An architecture  $A$  is called *safe composable* with respect to a property  $p$  iff for any element  $e$  of  $A$  is  $p(e)$ . In this case,  $p$  is an invariant of  $A$ .

Our composability approach allows to deal with different architectures as well as to subsume other definitions of composability. E.g., the definition from [5] given in Section 1.1 can be expressed in terms of reachability property for certain element properties.

Please note, that composability is architecture-dependent: If two architectures own the same set of initial elements, one may be (safe or reachable) composable regarding a certain property, while the other may not.

## 3 Composing an Embedded Real-Time System

In this section we introduce an example of a composable architecture: Message Scheduled System (MSS). We present here only the details which are relevant to the composability approach. More details regarding technical aspects may be found in [10, 9].

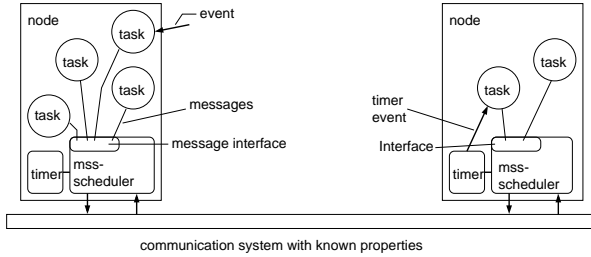


Figure 1: Message Scheduled System

### 3.1 The basic ideas of MSS

MSS is an architecture for distributed embedded real-time systems allowing decisions about composition, and composing *at run-time*.

The MSS architecture allows three different types of atomic elements: (Figure 1): The smallest element is a *task* ( $E_t$ ). In MSS this is a unit of execution that is able to produce a set of outgoing messages based on a set of incoming messages. Regarding composability it makes sense to consider such a *task* as an element because in many cases it is sufficient to add a *task* to a system to get new functionality.

The next type of elements are *nodes* ( $E_n$ ). A *node* in MSS is a machine capable to execute and schedule *tasks* and to send/receive the messages needed and produced by these *tasks*. Other than a *task*, a *node* is not software, it is a combination of hardware (the processor, the memory, etc.) and software (operating system, MSS-scheduler). Considering such a “mixed” unit as an element allows unique view of the system without considering a border between hardware and software.

The third type of atomic elements is a *communication system* ( $E_b$ ) that is able to transfer messages between nodes. Each MSS system has exactly one communication system.

Composing atomic elements creates a *system*<sup>3</sup>. They consist of one or more *nodes* executing *tasks* which communicate via the *communication system* connecting the *nodes*.

The “glue” between the elements are *messages* which are sent and received by the *tasks* using the *node’s* MSS-scheduler and the *communication system*. The semantic of this communication is publisher/subscriber. Each *task* has a special message (*wake up message*, *WUM*). The arrival of this message triggers the execution of the *task*. The *minimal inter-arrival times* (MIT) of messages of each type are known and are part of the composition decisions (scheduling at the level of communication medium).

<sup>3</sup>Such a system consists both of software and hardware.

### 3.2 MSS and Composability

The difference between MSS and the most of the other composable architectures (such as TTA [4]) is that the decision about composition (i.e., the decision whether a composition is possible without violating safety properties) is made during run-time.

The basic idea of composability in MSS is to map all decisions about addition or removal of elements onto scheduling decisions at three different levels to guarantee deadlines both for task execution and message transfer. The adherence of all deadlines is a safety property of MSS. These scheduling decisions can be taken based on well known algorithms such as RMA or EDF [7].

Based on this mapping the temporal behavior of an element can be described without including all details. It is sufficient to have information regarding scheduling at all three levels. Parameters of already composed elements are hidden using summarizing parameters such as load. This implies it is not necessary to have global detailed knowledge about the system at each node.

A composition is performed via addition of an atomic element or a system to an existing system. The decision whether this is possible without violating safety properties or not is made just in time at the moment of composition request. The data needed for such a composition decision can be deduced from elements descriptions for *tasks* (execution times and MITs), for *nodes* (computation capabilities) and for *communication medium* (bandwidth).

The complexity of this calculation is linear to the number of interfaces of elements, i.e., in most cases linear to the number of elements. If the composition decision is positive the composing takes place and is successful. Otherwise the original system remains unchanged.

Using the framework introduced in Section 2 it is possible to describe MSS with formulas reflecting the informal descriptions from Section 3.1. This is shown in [12], here we will only focus on properties concerning composability.

Therefore it is necessary to define desired properties. MSS guarantees adherence of all task deadlines in a composed system. This property now is called  $P_T$ . Furthermore, MSS guarantees end-to-end message delays (property  $P_M$ ).

Please note that  $P_T$  is an invariant property (or a set of invariant properties) while  $P_M$  is an emergent property – at level of *node* it makes no sense to consider end-to-end message delays.

Related to  $P_T$  and  $P_M$  it holds for MSS:

$$\text{safe\_composable}(MSS, P_T) \wedge \text{safe\_composable}(MSS, P_M) \quad (1)$$

Actually, (1) is an assertion that we prove formally using timed petri nets. The approach for that is described in [8, 11].

## 4 Service Composability

In this section we show how the proposed model can be used for composability of trusted and dependable services (Web services, JINI services...). This is important issue because that way we can model dependable collaboration of embedded systems. We first give an informal definition of service, and then proceed with describing service providers and service methods as elements and properties of the architecture. Then we explain service composition and decomposition with respect to various properties.

We consider a service provider to be service implementation coupled with the hardware it is executing on. Therefore, service provider has functional, as well as non-functional properties. Every service provider exposes one or more service methods that clients can invoke. Each method has a contract specifying constraints of invocation for that method. We will show later how contracts map into restrictions of composition operators.

If we have service architecture  $A_s$ , we define service provider as an element of that architecture:  $provider(stateVariables, messages, transferFunctions)$ . State variables are qualities of the service provider, both functional and non-functional. Transfer functions change values of state variables responding to messages. Messages are free state variables, used for interfacing service providers with clients and other providers. Each message consists of a tuple:  $(methodName, parameters, returnValue)$  where  $parameters$  and  $returnValue$  are sequences.

For each variable (property) we define a contract that binds the value of that property to some domain, or a set of values. That way we restrict composition with respect to that property.

A service provider is therefore called trusted if it has a contract for all state variables, which can be used to restrict composition and decomposition. In that way we ensure dependability and reuse.

### 4.1 Service Composition

The goal of service composition is to derive new service providers from existing ones.

Formally, if we have two services,  $X(v_{x1} \dots v_{xn}, msg_{x1} \dots msg_{xm}, \delta_{x1} \dots \delta_{xp})$  and  $Y(v_{y1} \dots v_{yq}, msg_{y1} \dots msg_{yr}, \delta_{y1} \dots \delta_{ys})$  we compute function  $A(P, Q, \circ)$  by checking functional compatibility first. If it returns true, we have a new functional state variables and messages, and proceed to deriving contracts and non-functional properties.

At the end, we have a new architecture element  $Z(v_{z1} \dots v_{zt}, msg_{z1} \dots msg_{zu}, \delta_{z1} \dots \delta_{zw})$ . Now we will take a look at an example.

Suppose we have a printer service provider and a camera service provider. We specify the printer service provider  $P(queue, status, document, resolution, colorMode, price, print, configure, \delta_{p1} \dots \delta_{pn})$  and the camera service provider  $C(queue, status, resolution, colorMode, getJpeg, getMpeg, \delta_{c1} \dots \delta_{cm})$ . Messages  $print$  and  $getJpeg$  are defined as  $(print, resolution, colorMode, status)$  and  $(getJpeg, resolution, colorMode, image)$ .

We define several types of contracts: *domain contracts* (bound the value of a variable to certain domain), *preconditions* (conditions that must be satisfied before the change of variable value), *postconditions* (conditions that must be satisfied after the change of variable value), *invariants* (conditions that must be satisfied after the change of each variable).

For example, domain contract for provider  $P$  on variable  $document$  can be:  $document \in \{image, text\}$ , which means it is impossible to print audio file on a printer. Preconditions for the same provider are  $resolution > 0 \wedge resolution \in \{P.resolution\} \wedge colorMode \in \{P.colorMode\}$ . Invariant contract is the existence of a printing queue, because we require each service provider to have a request queue. We can define similar constraints for the other provider.

The idea of composing is to provide new behavior (messages), new qualities (state variables), or both. If we try to compose service providers  $P$  and  $C$  in order to derive new behavior, we obtain the following:  $PC(queue, resolution, colorMode, price, printImageFromCamera, new transfer functions)$ .

Some state variables have vanished ( $document$ ), some are transferred ( $resolution, colorMode, queue$ ), and some remain unchanged ( $price$ ). We also have an emerging property ( $printImageFromCamera$ ). Tuple of variables for  $printImageFromCamera$  message is:  $(printImageFromCamera, resolution, colorMode, status)$ . It is derived from tuples for  $print$  and  $getJpeg$  messages. We also have a new contract. It is computed from the two composed contracts and gives new boundaries for variables of  $PC$  element:  $resolution > 0 \wedge resolution \in \{C.resolution \cap P.resolution\} \wedge colorMode \in \{P.colorMode \cap C.colorMode\}$ . Transfer functions are obtained by merging two state machines,  $P$  and  $C$ .

What we achieved here is composition of two service providers, Camera ( $C$ ) and Printer ( $P$ ), to obtain new provider, PrinterAndCamera ( $PC$ ). Properties of the composition results ( $PC$ ) are derived from the properties of the two composition operands ( $P$  and  $C$ ).

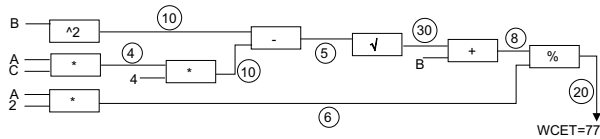


Figure 2: Decomposition with respect to non-functional property (worst case execution time)

## 4.2 Service Decomposition

Sometimes we want to take one 'big' service provider (element) and decompose it into a number of 'smaller' providers (elements) that provide the equivalent functionality. This is generally the case when a single element does not exist that matches the requirements or specification of a problem. Formally we can denote it as follows:

$$S(v_s, msg_s, \delta_s) = S_1(v_{s1}, msg_{s1}, \delta_{s1}) \circ \dots \circ S_p(v_{p1}, msg_{p1}, \delta_{p1})$$

For example, we can take a look at the formula for obtaining roots of the quadratic equation with coefficients A, B and C. Suppose that we have a service architecture with providers that expose basic mathematical operations: addition, subtraction, division, multiplication, power and square root. We could then decompose formula into a graph, where vertices are existing service providers (mathematical operations), and edges are compositions.

When we decompose a service and construct a decomposition graph, we can add further restrictions using contracts (Figure 2). Suppose that we want to obtain a result with certain decimal precision, or with some worst case execution time. Then we would have to compose again with respect to specified non-functional properties and see whether the graph satisfy new requirements.

We showed how we can use the proposed model for composability to perform composition and decomposition of services. In order to build a viable service architecture, other rules and elements must be added to architecture: synchronous and asynchronous calls, backward error recovery (transactions), forward error recovery (exception handling), and other types of contracts (event, rendering).

## 5 Conclusion

We have introduced a concept of composability and presented two application examples. We have shown that our approach is feasible to describe hardware-software co-design compositions as well as composed services.

We believe that we have provided a framework with a wide range of applicability. Currently, we

are applying our concept to a large-scale example: We are formally proving the composability property of the MSS architecture using timed petri nets. Other applications, e.g., a dependable service architecture as a generalization of the example in Section 4, will follow.

## References

- [1] Mehmet Aksit, Jan Bosch, William van der Sterren, and Lodewijk Bergmans. Real-time specification inheritance anomalies and real-time filters. *Lecture Notes in Computer Science*, 821:386–389, 1994.
- [2] L. Bergmans, B. Tekinerdogan, M. Glandrup, and M. Aksit. On composing separated concerns, composability and composition anomalies. In *ACM OOPSLA'2000 workshop on Advanced Separation of Concerns*, Minneapolis, October 2000.
- [3] Lodewijk M.J. Bergmans. Composability: Why, what, and how? In *Workshop on Composability Issues in Object-Oriented — Tenth European Conference on Object-Oriented Programming, July 8-12, 1996, Linz - Austria*.
- [4] H. Kopetz, M. Braun, C. Ebner, A. Krueger, D. Millinger, R. Nossal, and A. Schedl. The design of large real-time systems: The time-triggered approach. In *IEEE Real-Time Systems Symposium*, pages 182–189, Vienna, Austria, 1995.
- [5] Hermann Kopetz. *Real-Time Systems — Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061, 1997.
- [6] Carine Lucas, Patrick Steyaert, and Kim Mens. Research topics in composability. In *Workshop on Composability Issues in Object-Oriented — Tenth European Conference on Object-Oriented Programming, July 8-12, 1996, Linz - Austria*, 1996.
- [7] C. L. Lui and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, 20(1):46–61, January 1973.
- [8] L. Popova-Zeugmann, M. Werner, and J. Richling. Using state-equation to prove non-reachability in timed petrinets. *Fundamenta Informaticae*, 2003.
- [9] J. Richling. Message Scheduled System - A Composable Architecture for Embedded Real-Time-Systems. In *Proceedings of 2000 Int. Conference on Parallel and Distributed Processing techniques and Applications (PDPTA 2000)*, volume 4, pages 2143–2150, June 2000.
- [10] J. Richling, M. Werner, and L. Popova-Zeugmann. Automatic composition of timed petrinet specifications for a real-time architecture. In *Proceedings of 2002 IEEE International Conference on Robotics and Automation*, Washington DC, May 2002.
- [11] J. Richling, L. Popova-Zeugmann, and M. Werner. Verification of non-functional properties of a composable architecture with petrinets. *Fundamenta Informaticae*, 51:185–200, 2002.
- [12] M. Werner and J. Richling. Komponierbarkeit nicht-funktionaler Eigenschaften - Versuch einer Definition (engl: Composability of non-functional properties — an attempt of a definition). In *GI Fachtagung Betriebssysteme*, Berlin, 2002.