A Formally-Proven Composable Architecture for Real-Time Systems

Jan Richling
Computer Architecture Group
Institute for Computer Science
Humboldt University of Berlin
richling@informatik.hu-berlin.de

Matthias Werner
Communication and Operating
System Group
Institute for Telecommunication Systems
Technical University Berlin
mwerner@cs.tu-berlin.de

Louchka Popova-Zeugmann
Group “Logic in Computer Science”
Institute for Computer Science
Humboldt University of Berlin
popova@informatik.hu-berlin.de

***Abstract*— The Message Scheduled System (MSS) is a composable real-time architecture that allows the extension of systems at runtime without compromising timing guarantees. In this paper, we introduce the MSS architecture and discuss its guarantees as well as the way of proving the holding of these guarantees for any system that follows the MSS architecture.**

I. Introduction

Embedded real-time systems are widely used in cars, air planes, automation in industry and buildings. They have frequently hard temporal requirements, i.e., violations of deadlines may result in major inconvenience, cost or catastrophes.

In this paper, we discuss the formal guarantees of a real-time architecture, called *Message Scheduled System* (MSS) [7] and sketch the formal proof. MSS architecture is *composable* at runtime, i.e., it is possible to add new components to extend a system at runtime without invalidating certain properties. The decision *whether* a component can be added or not is also done at runtime, what distinguishes MSS from other systems such as TTA [2]. We are able to proof that systems of the MSS architecture are safe composable with respect to their timing behavior, i.e., following the composability decision it is not possible to construct a MSS instance that compromises timing guarantees. As a formalism for our proof we use *Prioritized Timed Petrinets* (PTPN) as introduced in [8].

The remainder of this paper is organized as follows: Section II introduces shortly our understanding of “architecture”. Section III describes the actual MSS architecture. In Section IV we describe the formal guarantees of MSS. Section V gives a short overview about the correctness proof. Finally, Section VI concludes our paper and suggests future research.

II. Architecture and Properties

In our notion, an architecture is a set of rules how to build systems, rather than a system built regarding certain rules (organization). One has to distinguish between properties of the architecture and properties of a system that is constructed within a certain architecture.

Then, composability is a property of the architecture, not of the system. We distinguish between safe composability and reachable composability, both with respect to a system property P. An architecture A is safe composable with respect to P iff all systems built in A have the property P. An architecture A is reachable composable with respect to P iff A allows to build at least one system that has the property P. By combination of safe and reachable composability, one can derive sub-architectures that include all desired properties. A more in-depth discussion of this topic one may find in [9].

Following this approach, MSS is a safe composable architecture with respect to timeliness properties as formulated in Section IV-B.

III. Message Scheduled System Architecture

Message Scheduled System (MSS) [7] is a composable architecture for distributed embedded real-time systems.

Please note, that we focus on the nonfunctional timing behavior. Composition of functional behavior is a well-known area. Successful architectures that allow composition with respect to functional behavior are, e.g., CORBA [5].

The idea of MSS is to map all decisions about addition of components (composability decisions) onto schedulability tests at different levels. The corresponding schedulability decisions are based on known and established scheduling techniques like *rate monotonic algorithm* (RMA) and *earliest deadline first* (EDF) [4]. These scheduling techniques condense scheduling information in more abstract metrics, as e.g., *load*. Using such metrics, temporal behavior of a system in MSS can be described without including all details: The parameters of three different scheduling levels are sufficient to allow a decision about addition of further components. The test whether the components are composable can be calculated easily (applying three schedulability tests) in linear time.

MSS is aware of three different types of components (Figure 1):
The first one is a *task* (Q^i) which is in MSS an execution unit that produces a set of outgoing messages from a set of incoming messages or events[1]. Considering such a task as component is useful with respect to composability because in many relevant cases it is sufficient to add a new task to

[1] Within the MSS context, a task is a rather small execution unit. Thus, the restriction that it only has to deal with messages at begin and end of execution is justified. If communication is needed in between, a task may be split into several parts.

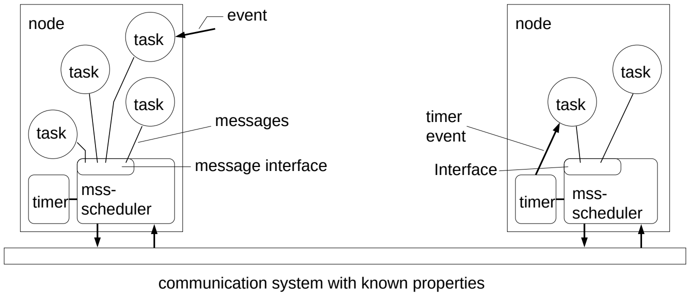


Fig. 1. Message Scheduled System

get new functionality. A task is characterized by its messages (input and output) and by its temporal behavior (worst case execution time if it runs on a certain node, and its periodicity. Please note, that if it is not yet assigned to (i.e., composed with) a node, a task has no runtime.

The next type of components are *nodes* (J^j). A node in MSS is a machine capable of executing and scheduling tasks and to send/receive the messages needed and produced by these tasks. Other than a *task*, a *node* is not pure 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 to the system without distinguishing between hardware and software. Actually, this reflects the reality of an embedded real-time system: Components in such a system often are devices that are delivered together with their operating software without the option of separation or modification.

The third type of components is the *communication system* (CS) that is able to transfer messages between nodes. Each MSS system has exactly one communication system. The communication system has real-time behavior and offers priorities, i.e., a message is delivered within a fixed time[2], and if more than one message is tried to be sent in parallel, only the message with the highest priority will be sent without delay. E.g., the CAN bus [1] may serve as an example for a communication systems that incorporates both requirements.

IV. Behavior and Guarantees of MSS

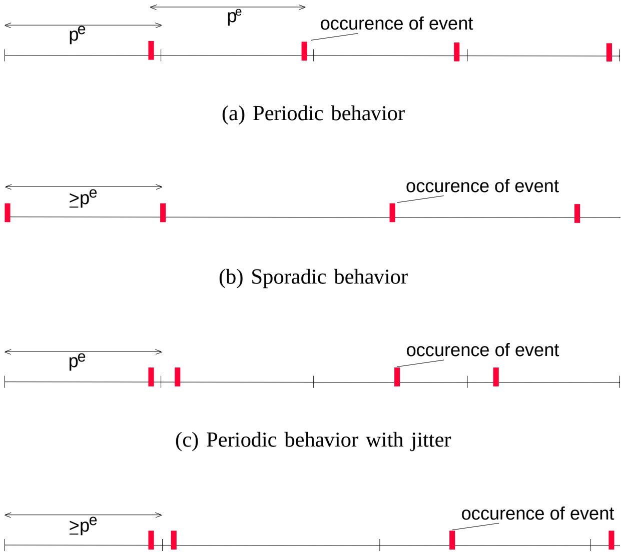
A. Dynamic Behavior

During runtime, a task may receive two kinds of events: external events (e.g., sensor inputs) and messages as a result of a task's computation (output messages of other tasks).

Messages are transmitted by the communication system. They are handled in a publisher-subscriber semantic by tasks using the node's MSS-scheduler as a dispatcher and the communication system as a backbone. Each task has a set of special events (called “wake-up set”, WUS) that can contain both messages or external events and is not allowed to be empty.

At runtime, a MSS system behaves as follows: A task Q^i running at node J^j receives an event that is element of the wake-up set. This triggers task's execution which requires w^i time units of CPU cycles[3] of node J^j. Messages that are not element of the WUS can be read during execution in their latest version — for these messages MSS does not give temporal guarantees. Once the execution is finished, the task sends all messages from its output set. These again may be received by other tasks and so on.

As for most real-time architectures, we assume a certain behavior for system stimuli, i.e., for external events. MSS knows four different types of temporal behaviors that will be assumed for external events and guaranteed for internal events (e.g., messages), respectively. Given an event e, we assign a time parameter p^e that describes its periodicity in the following ways:



(a) Periodic behavior

(b) Sporadic behavior

(c) Periodic behavior with jitter

(d) Sporadic behavior with jitter

Fig. 2. Possible timing behaviors

- **Periodic behavior** An event e that appears at time t will reappear at time $t + p^e$ — see Figure 2(a).
- **Sporadic behavior** An event e that appears at time t may reappear, but not before time $t + p^e$ (Figure 2(b)).
- **Periodic behavior with jitter** Within all intervals $I = (t, t + kp^e]$, $k = 1, 2, \ldots$, at least $k - 1$ instances, and at most $k + 1$ instances of event e appear. (Figure 2(c))
- **Sporadic behavior with jitter** Within all intervals $I = (t, t+kp^e]$, $k = 1, 2, \ldots$, at most $k+1$ instances of event e appear. (Figure 2(d))

Please note, that there is a half-order of the behaviors in the way that periodic behaviors are stricter than sporadic behaviors, and jitter-free behaviors are stricter than jittered behaviors.

[2] This implies an unique message size, which is frequently the case in the target domain of control systems.

[3] These CPU cycles are subject of scheduling at the node level because other tasks may also need CPU cycles.

MSS is able to deal with sporadic behavior with jitter which implies that all the other behaviors from Figure 2 (which are often used in control systems) are also supported.

B. Model and Guarantees

In addition to the introduced notation for different MSS components, we denote the following definitions:

- $J^j(Q^i) = 1$ if task Q^i runs at node J^j, and 0 else.[4]
- $Y_{i,j} = 1$ if task Q^i broadcasts a message that is element of the wake-up set of task Q^j, and 0 else[5].
- $H_i = 1$ if task Q^i gets input from an external source and 0 else.
- Y is the set of messages, Y^k is a message
- H is the set of external events, H^k is an external event

Also, in order to perform the proof of MSS' correctness the following assumptions are made:

- Transmitting a message using the communication system costs one time unit.
- All nodes are identical.[6]
- For all tasks Q^i the worst-case execution time w^i is known.
- External events have at least sporadic behavior with jitter, as described in Section IV-A, and for each event e, p^e is known.

To a running MSS instance, new task and nodes may be added at runtime. However, such a composition may only take place, if all of the following preconditions hold for the resulting system:

$$\sum_{\forall i: J^j(Q^i)=1} \frac{w^i}{p^i} \leq ln(2) \tag{C.1}$$

$$\sum_{\forall k} \frac{1}{p^{Y^k}} \leq ln(2) \tag{C.2}$$

$$\forall i, j, e_i \in WUS(Q^j), \exists t_i \in [0, p^{e_i}] \Rightarrow td(t_i) \leq t_i \tag{C.3}$$

$$\text{with } td_i(t) = 2 \cdot p^{Q^j} + \sum_{k=1}^{i-1} p^{Q^j} \cdot \left\lceil \frac{t}{p^{e_k}} \right\rceil$$

These preconditions can be tested without really composing the new system. The calculations are simple since it is possible to use results from previous composition decisions as basis and to distribute the calculation among the nodes (each node calculates only its part of the conditions). If preconditions (C.1 - C.3) are fulfilled, MSS guarantees the following temporal behavior after composition:

G.1 Each task Q^i will start execution after its node receives the corresponding wake-up event Y^i at time t and will finish before $t + 2p^{Y^i} + p^{Q^i}$.

G.2 For each chain of tasks Q^1,$Q^2, \ldots, Q^n$ with Y^i is WUE of Q^i and $Y_{i,i+1} = 1$ for all $n = 1, 2, \ldots, n-1$, the following is true: If Y^1 is delivered at time t, all messages by Q^n will be delivered at time $t + 2\sum_i(p^{Y^i})$

These guarantees are invariant during follow-up compositions or decompositions, i.e., they remain unchanged as long as all participating elements are present in the system.

Based on G.1 and G.2 all end-to-end times in a MSS instance are predictable.

V. Proving the Correctness of MSS

Although the conditions C.1 - C.3 are based on established results of scheduling theory (cf., e.g., [3]), it is necessary to prove the correctness of the MSS architecture: The classical results hold only if certain preconditions (e.g., independence of tasks) are fulfilled. In MSS, not all of these preconditions hold (e.g., the execution of a task triggers a message that in turn triggers the execution of another task), and thus it is unclear if the scheduling conditions are valid.

For the sake of brevity, we provide only the outline of our correctness proof. The full proof is part of the dissertation of Jan Richling, which is yet unpublished.

In order to prove that MSS is composable with respect to temporal behavior we have to show that *each* instance of MSS, i.e., each system that can be composed using the components and rules of MSS, fulfills the guarantees G.1 and G.2, as long as the conditions C.1-C.3 are met.

For a single instance (actual system S) this can be done using the following steps:

- Create a formal specification of S
- Map violations of G.1 and G.2 onto error condition E within the specification
- Show that for each behavior of S error condition E cannot occur

MSS as an architecture allows infinite number of concrete systems so this approach cannot be used for all possible instances. Instead, we have to use the concept of composability not only for the architecture, but also for its correctness proof in a way that we can specify the MSS architecture, not only concrete instances.

We use Prioritized Timed Petrinets (PTPN) [8] for specifying MSS components and systems in a way that we create a mapping between a MSS component (e.g., a task) and the PTPN that specifies the temporal behavior of this component. Furthermore, we define rules how to compose these component specifications according to a composition in MSS.

Using this approach, we are able to construct (or compose) the formal specification of each MSS instance out of the specification of components. For this purpose, we wrote a program that automatically generates the overall specification out of specification blocks specifying components of MSS, following a simple composition language [6]. Figure 4 shows an example of a resulting (i.e., composed) PTPN model of a single MSS instance.

[4] Please note, that several tasks can run on one node, but a task can run only on exactly one node.

[5] Please note, that we do not consider messages here that are not element of WUSs because these messages have no impact onto the temporal behavior of the task.

[6] This is for sake of shortness. In reality, MSS can deal with different types of nodes, which especially leads to different runtimes at different nodes.

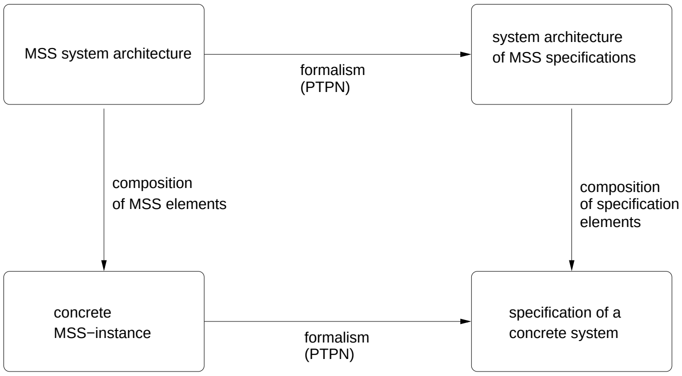


Fig. 3. Composable specification of MSS

Applying the ideas from Section II, we now have an architecture for specification (consisting of elements and rules for composition) where each element specifies a concrete MSS instance. This architecture of PTPN specifications can be seen now as the specification of the MSS architecture. Figure 3 shows the relation between these two architectures and their elements.

The architecture of MSS specifications defines a subclass of all possible PTPNs. In order to prove the composability of MSS we now have to show that within this architecture an error condition E cannot occur. In our PTPN specification, E means that states representing violations of guarantees G.1 or G.2) are not reachable within the subclass of PTPN defined by our architecture. Currently, we use instrumentation within the PTPN (cf. place labeled “error” in Figure 4) to express these undesired states. We hope we can eliminate this construct in future and prove within the “pure” specification.

In order to prove this, we use the technique presented in [8], facilitating state equation and firing inequalities for PTPN.

VI. Conclusion and Future Work

In this paper we introduced the MSS architecture that is composable with respect to real-time properties. We presented the formal conditions and guarantees of the MSS architecture. The derived formulae are similar to well known equations from *Time Demand Analysis* (TDA) [3]. However, since the requirements of TDA do not meet MSS requirements (especially independence condition), a formal proof of MSS correctness is needed. A sketch of this proof has been presented as well.

In future we plan to remove some instrumentation from our PTPN model to gain a more easier proof of MSS' composability property.

Also, we would like to apply our concept of proving composability using PTPN to other composable architectures such as TTA [2] or composable service architectures [9].

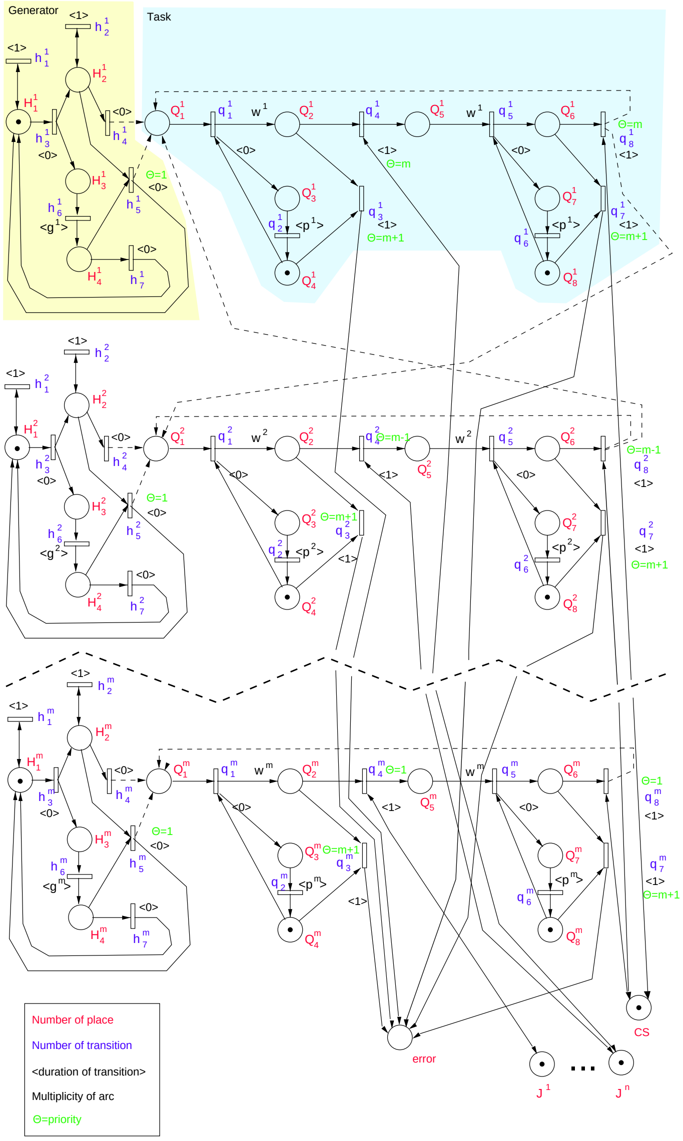


Fig. 4. PTPN model of a MSS instance

References

[1] *CAN Specification Version 2.0*. Robert Bosch GmbH, Stuttgart, 1991.

[2] 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.

[3] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *10th IEEE Real-Time Systems Symposium*, pages 66–171, 1989.

[4] C. L. Lui and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time enviroment. *JACM*, 20(1):46–61, January 1973.

[5] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Inc., Framingham, MA, USA, 1995.

[6] 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.

[7] Jan Richling. Message Scheduled System - A Composable Architecture for Embedded Real-Time-Systems. In *Prooceedings of 2000 Int. Conference on Parallel and Distributed Processing techniques and Applications (PDPTA 2000)*, volume 4, pages 2143–2150, Jun 2000.

[8] M. Werner, L. Popova-Zeugmann, and J. Richling. A method to prove non-reachability in prioritized duration petrinets. *Fundamentae Informatica*, 61(3,4):351–368, 2004.

[9] M. Werner, J. Richling, N. Milanovic, and V. Stantchev. Applying composability to dependable embedded systems. In *Proceedings of the International Workshop on Dependable Embedded Systems at the 22nd Symposium on Reliable Distributed Systems (SRDS 2003)*, Oct 2003.