Verification of Non-functional Properties of a Composable Architecture with Petrinets

Jan Richling

Louchka Popova-Zeugmann

Matthias Werner
Humboldt-Universität zu Berlin, Institut für Informatik
$\{$*richling* | *popova* | *mwerner*$\}$*@informatik.hu-berlin.de*

Abstract. In this paper, we introduce our concept of composability and present the MSS architecture as an example for a composable architecture. MSS claims to be composable with respect to timing properties. We discuss, how to model and prove properties in such an architecture with time-extended Petrinets. As a result, the first step of a proof of composability is presented as well as a new kind of Petrinet, which is more suitable for modeling architectures like MSS.

Keywords: real time, composability, Petrinets, duration nets, π-PN

1. Introduction

Due to the increasing complexity of todays systems, it gets harder and harder to guarantee desired system's properties. The usual approach to tackle this problem is frequently called "divide and conquer": systems will be *composed* of smaller units that are easier to handle. There exist a vast amount of work dealing with "divide and conquer" for functional aspects, e.g. functional correctness. However, for non-functional properties, like performance, timeliness, configurability, or reliability, only a few approaches exist.

In this paper, we introduce our concept of *composability* that allows to describe "divide and conquer" and also covers non-functional aspects. We show how to proof the composability of an architecture with respect to its timing properties, as it is frequently needed for the construction of real-time systems.

1.1. Composability

Before we are going into formal discussion we want to describe our concept of composability in an informal way. For this purpose, we define the terms *architecture*, *element*, and *system*. Please note, that our definitions may differ from the usage of the terms elsewhere.

An *architecture* describes a set of rules and styles how to construct something. Sometimes, this concept is called *design*[1]. An architecture determines how *elements* can be combined to form new elements. An element that is usable for a certain purpose, i.e., it has some desired properties, is called a *system*. Please note, that the terms *element* and *system* are interchangeable as long the composer's *intension* is not taken into account.

More formally speaking, an architecture A describes a mapping of elements $e \in E$ and *composition operators* $o \in O$:

$$A : E \times E \times O \rightarrow E$$

We define *composability* as a property of the architecture, i.e., of the rule set, rather than a property of the system designed following a rule set. We define the composability of an architecture with respect to at least one element property:

[1]In the every-day use *architecture* sometimes refers to something that is designed following certain rules and styles. Within this paper, we use the term in the defined sense only.

Definition 1.1. (reachability) An architecture A is *reachably composable* with respect to a property P iff A allows the composition of at least one element $e \in E$ with $P(e)$ satisfied.

Definition 1.2. (safety) An architecture A is *safely composable* with respect to a property P iff A allows only the composition of elements $e \in E$ with $P(e)$ satisfied.

Definition 1.3. (usefulness) An architecture A is *usefully composable* with respect to a reachability property P_1 and a safety property P_2 iff A is reachably composable for P_1 and all elements e that may be composed following A and satisfy $P_1(e)$ also satisfy $P_2(e)$.

To illustrate the meaning of the definitions above, consider the following example.

Let assume that there is a set of rules how to connect parts and which parts we may use. This rules are an architecture. If we can build a car (whatever a car is) following this rules then the architecture is reachably composable with respect to a car[2]. If everything what we build (cars and other things) following that architecture can survive smashing it against a wall, the architecture is safely composable with respect to that property. And finally, if we can build cars and every car would survive a crash test, the architecture is usefully composable with respect to cars and crash survivability.

1.2. Related Work

Especially in the times of component-oriented software, there exist a wide range of approaches to define composability for functional properties. For an survey of theoretical approaches for functional composability techniques, see [15]. Several software implementations exist that aim to be composable. The probably best-known examples are middleware layers such as Common Object Request Broker Architecture (CORBA), Distributed Common Object Model (DCOM) or Enterprise Java Beans (EJB). Non-functional properties are frequently added as additional requirements that have to be met. Examples for that approach are secure CORBA (CORBAsec) or Real-Time CORBA. We are not aware of a general definition of composability in the literature, that includes non-functional properties. However, there exist definitions for special domains, for example for the composition in the rights management in secure computer systems (for a survey see [8])

In the area of real-time systems we are targeting for, composability is — although frequently addressed — rarely defined. The probable first definition is by Kopetz [5], who also has developed a composable architecture TTA [4, 5] on the base of the MARS system [3]. Kopetz' and other similar composability definitions rely on *non-interference*. Please note, that while our proposed architecture (see Section 2) also uses a kind of non-interference, our composability definition holds also for general situations.

We claim, that our MSS approach of a composable architecture (see Section 2) is more flexible than, e.g., TTA. It needs less *a priori* knowledge and thus, it makes easier to compose (i.e., MSS allows online composition). However, the cost of this flexibility is a lower utilization. TTA-based systems may allow a much higher utilization of the nodes and the media than MSS does.

The state of the art of Petrinets are discussed separately in Section 3.

The remainder of this paper is organized as followed: Section 2 introduces the Message Scheduled System (MSS) architecture and provides an informal description, Section 3 discusses needed properties of description languages to describe an architecture like MSS, Section 4 provides the specification itself. In Section 5, we are presenting some first proofs and sketch the general proving technique. Section 6 introduces the next step in modeling composable real-time system: Priority Interval Petrinets. We conclude our paper with a description of the work has to be done in the future.

2. Casestudy of an Architecture: Message Schedule System (MSS)

In order to address the problem of composability for embedded real-time systems at a more technical level, we have developed a casestudy. Our approach of composability deals with composability as a property of an architecture. We focus on embedded real-time systems, and the main interest is on the property of correct temporal behavior. Putting this all together we have developed an architecture with

[2]The property of the element is here "being usable as a car".

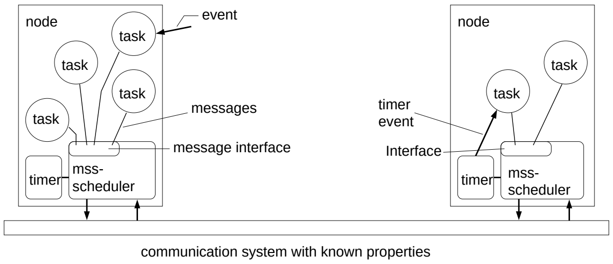


Figure 1: Message Scheduled System

support for composability with respect to temporal behavior. Such kind of architectures could be used in systems that need time-critical services, e.g., control systems or transportation systems.

Temporal behavior is, corresponding to the concepts introduced in section 1, a safety property, meaning that the architecture has to ensure that property. We define the property of correct temporal behavior of a system such that it is fulfilled iff all deadlines [6] inside the system are not violated during system life as long as certain fault assumptions are met.

Our architecture is called *Message Scheduled System* (MSS) [14] and is targeted to the domain of distributed embedded real-time systems. Such systems are widely used in cars, air planes, automation in industry and buildings. They have hard temporal requirements (violations may result in catastrophes) and profit in practice from the advantages of composability. E.g., an extension of an intelligent building or a car's computer system with components whose properties are unknown at design time is a very important issue.

2.1. The Basic Idea of MSS

The idea of MSS is to map all decisions about addition of components onto schedulability tests at different levels. The representation is done in a way that a component (and its interfaces) can be described by the scheduling parameter of all levels, which are:

- Local scheduling of all tasks on a single node
- Global scheduling of
  - real-time communication media between messages of different types
  - messages of the same type but from different senders to the same receiver

The corresponding schedulability decisions are based on known and established scheduling techniques like RMA or EDF [7].

Based on the temporal behavior of a component or a system in MSS which can be described without including all details, it is sufficient to use the information about these three scheduling levels and hide parameters of composed components inside systems parameters such as overall load. It is clear that not each node needs detailed knowledge about the whole system with all its nodes — the summarized parameters such as load are sufficient. The nonfunctional compatibility of two such interfaces — which means the test whether the components or "parts" are composable — can be calculated very easy (three schedulability tests) in polynomial or linear time[3]. The functional compatibility is not an issue because this is well researched over many year (concepts such as object orientation, modularization).

MSS is aware of three different types of components (Figure 1): The smallest one is a *task* which is in MSS an execution unit that produces a set of outgoing messages from a set of incoming messages. Considering such a task as component is useful with respect to composability because in many cases it is sufficient to add a new task to get new functionality.

The next type of components are *nodes* with tasks running at them. A node in MSS is a machine that is able to execute *tasks* and to manage the message traffic of that tasks. From the composability viewpoint a node with its tasks is composed from an empty node and the tasks.

[3] This is only valid under the assumption that the problem mentioned in 2.3 is solved

The third type of components in MSS are *systems* consisting of nodes connected via a network (specified by MSS). Such a system is built by composition from node-components or from other systems or both.

The “glue” between all that are *messages* that are sent and received in a publisher-subscriber semantic by tasks using the node's MSS-scheduler as dispatchers and the communication system as backbone. Each task has a special message (called “wake up message”), and the arrival of a message of that type triggers the execution of the task. Minimal interarrival times of messages of a type are known and part of composability decisions (this restricts the bandwidth usage).

2.2. MSS and Composability

MSS uses scheduling techniques at three different levels to guarantee deadlines. The scheduling parameters are derived from component descriptions (e.g., executions times, minimal interarrival times). Based on a set of component descriptions the calculation of the three schedules makes it possible to decide whether the components are composable or not. The complexity of that calculation is linear with the number of interfaces — that means linear with the number of components because the number of interfaces per component is limited in practice.

Existing knowledge from preceding composability decisions can be re-used because of the possibility to calculate a new schedule based on the old one with less expense. The existing description of a schedule can be summarized into a smaller set of system parameters (e.g., load at nodes) which is a real subset of the system configuration information.

The description of the composed system does also contain information about the structure of composition (e.g., which components are connected and how) so that system-wide properties (that are not known at subsystem level) like end-to-end-times can be calculated in linear time.

Based on the assumption that all the schedules (at three levels) are independent this ensures that the new (composed) system has the property of correct temporal behavior, meaning that all deadlines are met. This implies that MSS is composable with respect to temporal behavior.

2.3. Problems

Like already mentioned, the previous considerations are based on the assumption that the different schedules are independent. In fact, this might not be the case because the execution of a task (task-scheduling at node) is triggered by the arrival of a message which is influenced by the scheduling at communication medium. On the other hand, the order of messages at medium (which influences scheduling there) is driven by sending of messages which depends on task scheduling at different nodes.

The assumption of MSS is that the restriction in minimal interarrival times both for messages and tasks and the distinction of deadlines for task execution and message transfer leads to a independence of the schedules in a way that the output of one schedule can be seen as a valid (with respect to the parameters) input for the next schedule.

This assumption has to be proven. We have developed a formal description for MSS based on time-extended Petrinets and use this description to specify and prove properties.

3. MSS and Time Dependent Petrinets

For a specification of an architecture such as MSS, we have the following requirements:

- R1 The specification should reflect the system and its parts in a “natural” way.
- R2 We need to express concurrency, priorities and time dependence.
- R3 The specification of a concrete MSS system should be “composable” itself, meaning that it can be constructed out of specification pieces according to the components inside the architecture
- R4 It should be possible to generate (“compose”) the specification of a concrete system automatically out of a system description.
- R5 Tool support for specifying and proofing to cope with the complexity of large systems is essential.

We have surved a number of tools and description techniques, among these are UPPAAL [1] and Real Time Logic [2], and Petrinets. Because of existing tools support (R5) and the semantical power, we are focusing here on Petrinets as a base of our description.

In this section, we discuss how to meet the requirements R1 and R2.

A well suitable way to specify concurrent systems are Petrinets. In general, the classical Petrinets describe time as a causality. However, in MSS time dependence is given by the periods and the run-time of the tasks. Each attempt to ignore these, or to consider them as pure causality, leads to senseless simplification. Thus, a time component is needed that provides the ability to model time intervals.

A wide variety of time dependent Petrinets has been already defined during the last decade. Unfortunately, for most of them there exist no analyzing methods but only simulation tools.

The so-called Time Petrinets (IPN), defined by Merlin (comp. [9]), seems to be suitable for our purpose.

The IPN are classical Petrinets in which an additional time interval is assigned to each transition. The interval is relative to the time when the transition was last enabled. When the transition becomes enabled, it can fire only during the interval and it must fire at least at the end of the interval, unless it is disabled by the firing of another transition. Firing a transition takes no time. An efficient method to analyze IPN with rational boundaries of the time intervals has been implemented in the INA tool (comp. [17]) more then 10 years ago. However, IPN are not well suited to model priorities. Within IPN, one *may* describe priorities, but implementing priorities takes time (at least one time unit). One could incooperate additional times to model MSS. However, we would get a time discrepancy between the real system and the model, and we do not consider this as a "natural" way of modeling.

Another time dependent Petrinet is the Timed Petrinet (DPN), described by Ramchandani (comp. [13]) more than 15 years ago. DPN provides the the possibility to describe priorities without time consumption. Also, there exists a method to analyze DPN, and the INA tool is able to handle DPN.

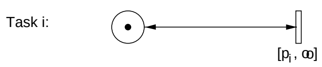
DPN are classical Petrinets with a time duration assigned to each transition. When a transition becomes enabled it starts immediately, and the firing takes exactly as much time as described by the corresponding duration.

3.1. Problems and Solutions

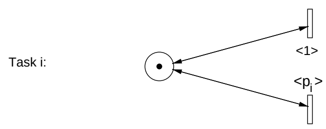
The disadvantage of DPN is that they seem to be inadequat to model time intervals. There is no general algorithm which translates a IPN into DPN (although both Petrinets are equivalent to the Turing machine and consequently equivalent to each other). But there are two reasons why MSS can be modeled adequately by DPN.

Firstly, the time restrictions in the MSS are always exactly defined, with one exception of taks activation times. An activation is possible at every p time units or later, where p is the request period of the task, i.e. a task becomes active within an interval $[p, \infty]$.

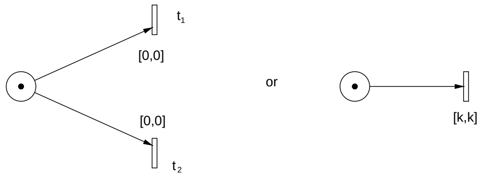
Activating a task could be modeled by IPN:



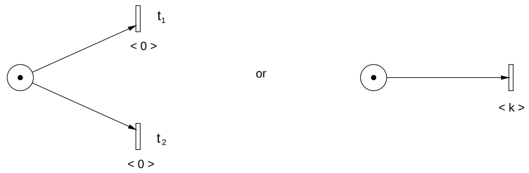
Since the place is not shared, the behavior can be simulated easily with DPN:



The remaining transitions in a IPN model of MSS have intervals $[0, 0]$ or they have a not shared input place:



This can be easily translated into a DPN:



Secondly, IPN has the following property: If a dead state is reachable in the net, then this state can also be reached by firing at integer times only (comp. [10], [12], [11]). Therefore, searching dead states can be executed in finite time if the skeleton of the IPN (i.e., the timeless Petrinet) is bounded, or if paths leading to a dead state have the highest priority.

3.2. Basic Definitions

This section provides the basic definitions we will use in the rest of the paper.

The following notations are used: $\mathbb{N}$ is the set of natural numbers, $\mathbb{N}^+ := \mathbb{N} \setminus \{0\}$. $\mathbb{Q}_0^+$ is the set of nonnegative rational numbers. For each number a the notation $\lceil a \rceil$ denotes the smallest natural namber greater or equal to a.

Definition 3.1. (Petrinet) The structure $N = (P, T, F, V, m_o)$ is called a Petrinet (PN) iff
(1) P, T, F are finite sets with $P \cap T = \emptyset$, $P \cup T \neq \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ and $dom(F) \cup cod(F) = P \cup T$
(2) $V : F \longrightarrow \mathbb{N}^+$ (weight of the arcs)
(3) $m_o : P \longrightarrow \mathbb{N}$ (initial marking)

A marking of a PN is a function $m : P \longrightarrow \mathbb{N}$, such that $m(p)$ denotes the number of tokens at the place p. The pre- and post-sets of a transition t are given by $Ft = \{p \mid p \in P \wedge pFt\}$ and $tF = \{p \mid p \in P \wedge tFp\}$, respectively. Each transition $t \in T$ induces the marking t^- and t^+, defined as follows:

$$t^-(p) = \begin{cases} V(p,t) & \text{iff} \quad (p,t) \in F \\ 0 & \text{iff} \quad (p,t) \notin F \end{cases} \qquad t^+(p) = \begin{cases} V(t,p) & \text{iff} \quad (t,p) \in F \\ 0 & \text{iff} \quad (t,p) \notin F \end{cases}$$

Furthermore, Δt denotes $t^+ - t^-$. A transition $t \in T$ is enabled (may fire) at a marking m iff $t^- \leq m$ (e.g., $t^-(p) \leq m(p)$ for every place $p \in P$). When an enabled transition t at a marking m fires, this yields a new marking m' given by $m'(p) := m'(p) + \Delta t(p)$. The firing is denoted by $m \xrightarrow{t} m'$.

Definition 3.2. (Timed Petrinet) The structure $Z = (N, D, A)$ is called a Timed Petrinet with priorities (DPN) iff:
(1) $S(Z) = N$ is a PN (Skeleton of Z).
(2) $D : T \longrightarrow \mathbb{Q}_0^+$ (duration function).
(3) $A : T \longrightarrow \mathbb{N}$ (priority function).

$D(t)$ is the duration of the firing transition t. It is easy to see, that considering DPNs with $D : T \longrightarrow \mathbb{N}$ will not result in a loss of generality. Therefore, only such time functions D will be considered subsequently.

Within this approach, the definition of a state is fundamental. A state is characterized by a marking together with the momentary local time for enabled transitions, or the sign # for the disabled transitions, respectively.

Definition 3.3. (state) Let $Z = (N, D, A)$ be a DPN with priorities and $d : T \longrightarrow \mathbb{Q}_0^+ \cup \{\#\}$. $z = (m, d)$ is called a state in Z iff:
(1) m is a reachable marking in $S(Z)$.
(2) $\forall t$ ($(t \in T \wedge t^- \leq m) \longrightarrow d(t) \leq D(t)$).
(3) $\forall t$ ($(t \in T \wedge t^- \not\leq m) \longrightarrow d(t) = \#$).

The state $z_0 = (m_0, d_0)$ is called initial state when $d_0(t) = \begin{cases} 0 & \text{iff} \quad t^- \leq m_0 \\ \# & \text{else} \end{cases}$

Definition 3.4. (step) Let $Z = (N, D, A)$ be a DPN with priorities. The set U of transitions is called a step at the state $z = (m, d)$, iff:
(1) $\forall t \ (t \in U \longrightarrow d(t) = 0)$,
(2) if $U = \emptyset$ than $d \neq 0$,
(3) $U^- := \sum_{t \in U} t^- \leq m$,
(4) $\forall t \ (t \in U \longrightarrow A(t) = \max\{ A(t) \mid t^- \leq m\}$),
(5) $\neg \exists U'$ ($U \subset U' \wedge U'$ satisfies $(1), (2), (3), (4)$).

A step is a maximal set of concurrent enabled transitions with highest priority.

Definition 3.5. (firing) Let U be a step at the state (m, d) in the DPN $Z = (N, D, A)$. Firing U at time τ leads to state (m', d') at time $\tau + \alpha$, where $\alpha = \begin{cases} 0 & \text{iff} \quad \exists t \ (t \in U \wedge D(t) = 0) \\ 1 & \text{else} \end{cases}$ and
$m' := m - U^- + \sum_{t \in U, D(t) = \alpha} t^+ + \sum_{d(t) = D(t) - \alpha} t^+$,
and for each $t \in T$

$$d'(t) = \begin{cases} \alpha & \text{iff} \quad t \in U \wedge D(t) > \alpha \\ d(t) + \alpha & \text{iff} \quad t \notin U \wedge 0 < d(t) < D(t) - \alpha \\ \# & \text{iff} \quad t^- \not\leq m' \\ 0 & \text{else} \end{cases}$$

More about time dependent Petrinets can be found in [16].

4. The Specification of MSS Using Timed Petrinets

This section demonstrates how to transform the rather informal description from Section 2 into the more strict Petrinet semantic.

The basic building block in MSS is the task, so we will start our specification using Timed Petrinets with the specification of a MSS task.

4.1. Specification of MSS Tasks

A task in MSS has the following behavior that has to be modeled: The execution is triggered by the arrival of a wake up message (or a message from an external source such as sensors or buttons) that is delivered to the task via the MSS scheduler. This arrival triggers the execution of the task which can be interrupted by the execution of other tasks (preemptive scheduling). The execution itself lasts for the worst case execution time (WCET) of the task, meaning that the task occupies the node's processor for no longer than that time. The time slice of scheduling (shortest interval between two task switches) is the quantum which can be seen as an atomic time interval in the system. Once the execution is finished the task produces an output message that is send by the MSS scheduler via the communication medium (and can trigger the execution of other tasks).

In Section 2 we introduced the concept of three scheduling levels which represent three different points of resource sharing. Here we focus on two of them that are explained above — the processors of nodes[4] and the communication medium. The model of a task must include both resources and must represent the resource sharing between different tasks.

[4]The node's processor is a multiple resource — one processor per node (or even more in case of multiprocessors)

Therefore, we use places for the medium and the nodes, with a capacity of one and one token at the initial marking[5].

Besides tasks and the mentioned resources the specification needs another building block called generator. Generators address the startup problem: Because of MSS's structure a task can only be started if a message arrives, and a message can only arrive if it was sent, and it can only be send if there was a task that needed to be started before. Therefor we introduce generators as a principle to generate messages. In the real world this maps to the mentioned external message sources such as sensors or buttons. More specific, with the generators we want to produce messages with given minimal interarrival time (that is a fundamental requirement of MSS — see also [14]).

The following explanation of the specification refers to Figure 2. This figure shows a system with n tasks running at a single node. As one can easy see, the usage of multiple node places (marked $Node$ in the figure) introduces multiple nodes in a natural way. Here we explain the specification of a task and use the first task in the figure as example; the same explanation applies to each other task.

The *generator* is specified by the place P_1^1 and the transitions t_1^1 and t_2^1 — it is obvious that it generates messages with a interarrival time that is at least p_1 and at most infinite.

Once the generator generated a message, the corresponding token goes into the next part that specifies the execution. The lower part (P_3^1, P_4^1, t_4^1 and t_5^1) is a instrumentation to ensure that violations of deadlines are detected and produce a special state of the Petrinet. Place P_5^1 collects as many tokens as the WCET w_1 is, and transition t_6^1 implements a drop-model, meaning that its firing will last one time unit, simulating the duration of one scheduling quantum. That transition has a priority a_1 equal to the priority of the task in MSS and shares place $Node$ as precondition with other tasks running on the same node. This ensures the mutual exclusion and the fixed priority scheduling. After each quantum a preemption is possible (because of drop-model) and the execution is finished if all tokens are at place P_6^1 (the multiplicity of the arc from that place to the next transition ensures that the next step is not possible before the end of execution). The already mentioned lower part of the net (the instrumentation) has the transition t_4^1 with a duration equal to the task period. This means that the deadline is violated iff tokens are on both P_4^1 and P_5^1, which is checked with the high priority transition t_5^1. The firing of that transition triggers a mechanism (lower part of the figure) to "steal" the tokens from the initial places of all generators in the whole net — the net will reach a dead state (tokens at $t - error$ or $m - error$ indicating deadline violations at medium or node level, respectively).

After finishing execution, a message will be sent. This is done by the MSS scheduler and is specified by the remaining part of the upper part of the net (see Figure 2). The principle (instrumentation and reaching a dead state in case of deadline violation) is similar to the first part. The difference is that the shared resource is the medium (place $Medium$) and that preemption is impossible (a message is atomic and its transmitting lasts two time units in our specification).

4.2. Composition of a System Specification

The specification introduced so far specifies the behavior of a single task that shares resources with others. It is clear that the composition of a system specification is done in a way that all tasks that run on the same node share a node-place, and all tasks of the system share the single medium-place. But, composition of a system needs more, because so far only generated messages can be represented and messages are only send but not received.

The second part of composition combines outputs and inputs, meaning that there may be tasks without a generator, they begin with place P_2^i and have a direct arc from the $medium_j$-transition of another task — this models the arriving of messages. It is clear that MSS's publisher-subscriber semantic can be modeled that way and that the remaining task specification is not influenced by dropping the generator.

This is the point where the third scheduling level (that is not modeled at the moment) of MSS comes into account — it appears if a task gets its input from more than one other task. The scheduling here is very simple — messages are scheduled based on their priority (this can be modeled via transition priorities) and maybe have to wait until the task execution triggered by a previous message is finished.

[5]This allows the specification of a n-way multiprocessor as node in a very natural way — just use n as initial marking and capacity of the corresponding node

The specification so far only considers "wake-up messages" as input messages for tasks, but this is sufficient because only these messages have temporal restrictions that must be fulfilled. Also, the current version of the specification does not model multiple output messages, but this feature can easily be added.

The composition rules allow to build a net specifying a concrete MSS configuration. That net can be analyzed using the tool INA in a very easy way: It is only necessary to search for dead reachable states in the net because those represent missed deadlines. If no such state exists the net will fulfill all temporal requirements.

The composition rules are easy and the interfaces between the different building blocks of the complex specification are simple — only single arcs with no further places or transitions. Therefore such a specification can be build automatically based on an abstract system description following the ideas from [14]. We have developed such a tool, *MGen*, and are able to automatically build and analyze MSS configurations represented by Timed Petrinets. Those, this tool implements a rule-set to specify each configuration of MSS.

5. Techniques and Example for Proof of Properties

The specification and analysis of a concrete system is possible using the techniques described in the section above (specify using *MGen*, analyze using INA). However, in order to solve the problem of proofing the properties of the MSS architecture, this is not sufficient. We have to be able to reason about *all* instances of such a system to meet our composability definition. Thus, we have to detect the architecture's invariants. In other words: Which conditions must be fulfilled by the parameters in order to get a system that does not violate any deadline?

The complete proof for MSS' composability is work in progress. We present here the proof of a lemma for schedulability to demonstrate the method.

5.1. Sufficient Condition for Schedulability in MSS

Using INA, a concrete MSS-instance modeled by a DPN can be model-checked for $t-error$ or $m-error$ (which mark time faults, comp. Section 4.1), but there are certain boundaries for the number of tasks. For example, analyzing a DPN model of MSS with 2 tasks with request periods $p_1 = 4$, $p_2 = 8$ and worst case execution times $w_1 = 2$, $w_2 = 3$, a state space of 6366 states is computed. Increasing the request periods to $p_1 = 8$, $p_2 = 16$ the state space grows to 19261 states and for $p_1 = 16$, $p_2 = 32$ it reaches 81837 states. A system model with 3 tasks ($p_1 = 4, p_2 = 8, p_3 = 16$ and $w_1 = 1, w_2 = 3$ and $w_3 = 1$ leads to a state space of about 2,3 Mio. states.

Thus, a characterization of the non-reachability of m-error and t-error would be very useful. A sufficient condition for them will be proved here.

Figure 2 is used to illustrate the proof.

Theorem 5.1. *Let Z be a DPN model of MSS consisting of one node and n tasks. Z is not dead (i.e. the places m-error and t-error can not be marked, in MSS this means that all deadlines are met), if the following holds:*

$p_1 \geq w_1$ *and*

$$p_i - \sum_{j=1}^{i-1} \lceil \tfrac{p_i}{p_j} \rceil \cdot w_j \geq w_i \;\; \forall i \; (i \in \{2, \cdots, n\}). \tag{1}$$

Outline of the proof: The proof is done by induction on i starting with $i = 2$.

The idea is that the places *t-error* and *m-error* can only be marked, if P_4^i and P_5^i resp. P_8^i and P_9^i can be marked at the same time for at least one $i, i \in \{1, \cdots, n\}$. Hereby, the sequence of firing transitions will be considered, which leads to marking of the places P_4^i and P_5^i. The impossibility of simultaneous marking of the places P_8^i and P_9^i under the given conditions can be verified analogously.

For $i = 2$ the claim can be directly proved.

Because of the priorities it is sufficient for the induction step to consider when the places P_4^n and P_5^n are marked.

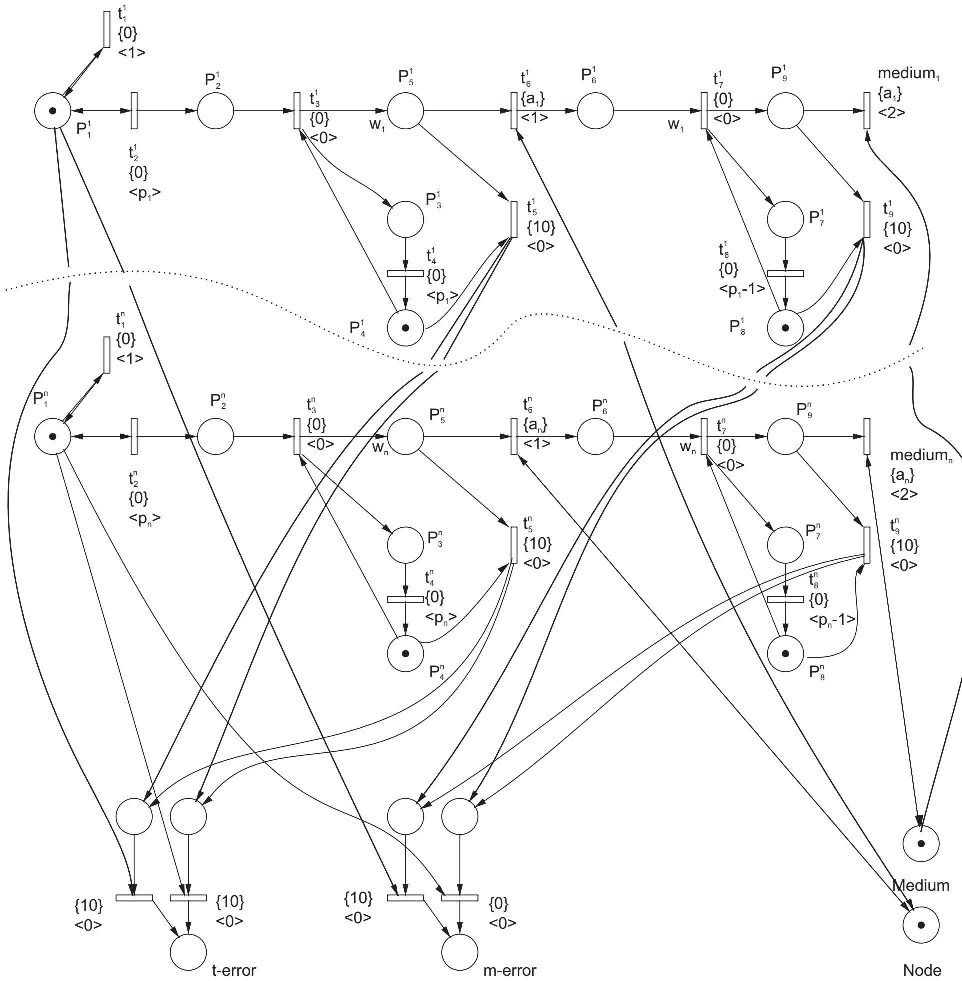


Figure 2: Specification of MSS using DPN.
Legend: *numbers at transitions:* <duration>, {priority}; *numbers at arcs:* multiplicity

The time during which the place P_5^i , $i \in \{1, \cdots, n-1\}$ has been marked since last starting the request period p_n is not greater than $\lceil \frac{p_n}{p_i} \rceil \cdot w_i$. Thus, the maximal time, which the tasks $T_1, \cdots, T_{i-1}$ consume within last duration of the request period p_n is not greater then $\sum_{j=1}^{i-1} \lceil \frac{p_i}{p_j} \rceil \cdot w_j$. Hence, the time remaining for the task T_n is at least $p_i - \sum_{j=1}^{i-1} \lceil \frac{p_i}{p_j} \rceil \cdot w_j$. But, this time is greater or equal than the time w_n and that is why all tokens situated at place P_5^n move to the place P_6^n, before the token at P_3^n moves to the place P_4^n. Following P_4^n and P_5^n can not be marked simultaneously, if condition (1) holds. ■

6. π-PN

Concerning the time component in MSS, IPNs seem to be the most suitable kind of a Petrinet for designing them. But, as already mentioned, describing priorities with IPN takes time and this leads to a time discrepancy between the real protocol and its model. In order to overcome this disadvantage a priority function can be assigned to the IPN. This function associates a priority, a natural number, to each transition. The dynamic behavior of the net is defined similar to the dynamic behavior of the IPNs. Changes from a state into another by elapsing time are done in this new kind of time dependent nets in the same way as in the net without priorities. A transition can fire, when it can fire in the net without priorities and additional, the transition has the highest priorities between all fire-able transitions. We call that new type of time dependent Petrinet with the ability to describe intervals and priorities in a natural way *Priority-Interval-Petrinet* (PI-PN or π-PN).

Definition 6.1. The structure $Z = (N, I, A)$ is called a Priority-Time Petrinet (π-PN) iff:
(1) N is a PN,
(2) (N, I) is a Time Petrinet (IPN),
(2) $A : T \longrightarrow \mathbb{N}$ (priority function).

I is the time function of the net (N, I), indicating the earliest firing time of t ($eft(t)$) and the latest firing time of t ($lft(t)$), respectively. For more about IPN comp. [10].

Definition 6.2. (state) Let $Z = (N, I, A)$ be a π-PN and $c : T \longrightarrow \mathbb{Q}_0^+ \cup \{\#\}$. $z = (m, c)$ is called a state in Z iff:
(1) m is a reachable marking in $S(Z)$.
(2) $\forall t\ (\ (t \in T \wedge t^- \leq m)\ \longrightarrow\ c(t) \leq lft(t))$.
(3) $\forall t\ (\ (t \in T \wedge t^- \not\leq m)\ \longrightarrow\ c(t) = \#)$.

The notion "state" can be interpreted as follows: Within the net, each transition t has a clock. If t is enabled at a marking m, the clock of t shows the time elapsed since t became most recently enabled. If t is disabled at m, the clock stops running (indicated by $\#$).

Now the dynamic aspects of TPNs – changing from one state into another – can be introduced:

The state $z_o := (m_o, c_o)$ with $c_o(t) := \begin{cases} 0 & \text{iff} \quad t \leq m_0 \\ \# & \text{iff} \quad t \not\leq m_0 \end{cases}$ is set as the initial state of the π-PN Z.

Definition 6.3. (state change by time elapsing) Let $Z = (N, I, A)$ be a π-PN. The state $z = (m, c)$ is changed into the state $z' = (m', c')$ by the time elapsing $\tau \in \mathbb{Q}_0^+$, denoted by $z \xrightarrow{\tau} z'$, iff
(1) $m' = m$ and
(2) the time duration τ is possible, i.e.

$$\forall t\ (\ (t \in T \wedge c(t) \neq \#) \longrightarrow c(t) + \tau \leq lft(t) \text{ and } c'(t) := \begin{cases} c(t) + \tau & \text{iff} \quad t \leq m \\ \# & \text{iff} \quad t \not\leq m \end{cases}).$$

Definition 6.4. (state change by firing transition) Let $Z = (N, I, A)$ be a π-PN. A transition $\hat{t}$, which is ready to fire in the state $z = (m, c)$, may fire yielding a new state $z' = (m', c')$, iff
(1) $m' = m + \Delta$,
(2) $eft(\hat{t}) \leq c(\hat{t})$,
(3) $A(\hat{t}) = \{\ A(t)\ \mid t \in T \wedge t^- \leq m \wedge eft(t) \leq c(\hat{t})\}$

$$(4) c'(t) =: \begin{cases} \# & \text{iff} \quad t^- \not\leq m \\ c(t) & \text{iff} \quad t^- \leq m \wedge t^- \leq m' \wedge Ft \cap F\hat{t} = \emptyset \\ 0 & \text{otherwise} \end{cases}.$$

We have got already an implementation of π-PNs for INA. It allows us to model MSS in a more "natural" way. The next step now is to use this possibility and to analyse the resulting model.

7. Conclusions and Future Work

In this paper, we have introduced our concept of composability and have shown, how state-of-the-art modeling and proofing techniques can be used to demonstrate composability properties of an example architecture.

Also, we have discussed limitations and disadvantages of the used modeling techniques. We have proposed a new modeling approach that overcomes some of the drawbacks: π-PN.

Considering the state of our work, the next steps to be done in the future are easy to identify: First, we have to finish the model of the MSS architecture by including the third scheduling level. In parallel, we will port our specification into π-PN. Secondly, we have to identify and to proof further invariants of the architecture description which hopefully allows us the proof of the composability property.

As a further future step, we plan to apply the modeling and proofing methods used within this project to other architectures.

Acknowledgements

This research is supported in parts by the DaimlerChrysler Research and Technology Berlin and in parts by the German Research Foundation (DFG).
We want acknowledge the work of Prof. Starke who has programmed the INA tool.

References

[1] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Yi Wang, and Carsten Weise. New Generation of UPPAAL. In *Int. Workshop on Software Tools for Technology Transfer*, June 1998.

[2] F. Jahanian and A. Mok. Safety Analysis of Timing Properties in Real Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, 1986.

[3] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9(1):25–40, February 1989.

[4] H. Kopetz and G. Grünsteidl. TTP — a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[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] C.M. Krishna and Kang G. Shin. *Real-Time Systems.* The Mc Graw-Hill Companies, Inc., 1997.

[7] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[8] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering.* Wiley & Sons, 1994.

[9] P. Merlin. *A Study of the Recoverability of Communikation Protocols.* PhD thesis, University of California, Computer Science Dept., Irvine, 1974.

[10] L. Popova. On Time Petri Nets. *J. Inform. Process. Cybern. EIK 27(1991)4*, pages 227–244, (1991).

[11] L. Popova and D. Schlatter. Analyzing Path in Time Petri Nets. *Fundamenta Informaticae 37, IOS Press*, pages 311–327, (1999).

[12] L. Popova-Zeugmann. Essential States in Time Petri Nets. *Informatik-Berichte 96*, 1998.

[13] C. Ramchandani. Analysis of Asynchronous Concurrent Systems by Timed Petri Nets. *Project MAC-TR 120, MIT*, Febr. 1974.

[14] Jan Richling. Komponierbare Echtzeitsysteme — Entwurfsmethode und Architekturentwurf. Technical Report Informatik Bericht 127, Institut für Informatik, Humboldt-Universität, Berlin, Germany, Sep 1999.

[15] Mary Shaw. Constructing systems from parts: What students should learn about software architecture. In B. Randell, editor, *Software Architecture and Design*, pages VIII.1 – VIII.25. International Computers Limited and University of Newcastle upon Tyne, 1998.

[16] P. Starke. A Memo on Time Constraints in Petri Nets. *Informatik-Berichte 46*, 1995.

[17] P. H. Starke. *INA - Integrated Net Analyzer.* Berlin, 1997. Manual.