Composing Timed Petrinet Specifications of a Composable System

Jan Richling   Matthias Werner
Louchka Popova-Zeugmann
Humboldt-Universität zu Berlin, Institut für Informatik*
{richling | mwerner | popova}@informatik.hu-berlin.de

Abstract

Ideally, a system is designed starting with a formal model. However, in the real world, many systems are designed without a formal model in mind. For these systems, it is hard to show that a formal model meets the informal design.

Within this paper, we demonstrate on the example of the composable MSS architecture how to bridge the gap between informal description and the formal model. We discuss, how modeling can be done in a "natural" way so that the mapping between system and model components and the composition of model components is rather obvious.

1 Introduction

It is an old idea to handle the complexity of systems by composing them out of smaller components. We are interested in composing systems with *non-functional* properties. Non-functional properties such as reliability, temporal behavior or performance are much harder to deal with than functional properties. We define *composability* as a property of an *architecture*, rather than as a property of an actual system, see [9].

The Message Scheduling System (MSS) architecture [11] is claimed to be a safely composable architecture with respect to temporal behavior, i.e., real time. It allows the useful composition of control systems, as used for plant automation or traffic control. MSS components are described by parameters that are close to implementation and rather informal, see Section 2. Currently, we prove the composability [9] of the MSS architecture using Timed 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.

Please note, that we have to deal with two kinds of composition: At the system level, we have a composable architecture that allow to decide *at runtime* whether or not a new component can be introduced to the system. On the other hand, we want to compose the model. That allows us to bridge the gap between the informal but "obvious" description and the rather complex formal model.

In this paper, we discuss the requirements R1, R4, and R5. For a discussion of R2 and R3, see [9].

1.1 Modeling

To prove properties of a system one need to describe the system in a formal way. However, there is always a gap between the formal model and the real world. The usually suggested way (e.g., [14])to overcome this gap is to start the system's design with the formal description. Thus, both can be archived, the formal reasoning within the formal description, and (if there are the suitable tools) the derivation of an implementation.

Unfortunately, most systems (so the MSS architecture) are designed without a formal specification in mind. In this case, deriving a formal specification from the system's specification (modeling) is rather hard, since there are no means to prove the correctness.

The proper way here is to make the model as close as possible to the actual system description. Additional, the model should reflect the system in a "natural" way.

*This research is supported in parts by DaimlerChrysler Research and Technology Berlin and in parts by the German Research Foundation (DFG).

Overall, the translation from the system's actual description to the model should be "obvious".

Even the easiest translation rules may lead to a complex model, when the system is quite complex itself. To keep the model "obvious", we have to model small "obvious" parts, and compose these parts in an "obvious" way.

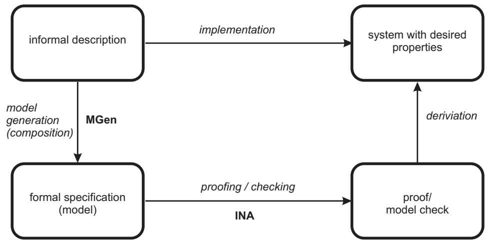

Figure 1: Formal derivation of system properties

There is no general way to derive a formal model (FM) from an informal description (ID) and show its obviousness. This is especially true for modeling of non-functional properties. However, on the base of the experiences in with modeling of functional properties, we can identify some "rules of thumb":

C1 Entities in the ID should also be separately identifiable in the FM

C2 Properties of entities in the ID should be directly mapped to attributes of entities in the FM.

C3 Composition in the FM should reflect composition in the ID and vice versa.

1.2 Related Work

Almost every aspect of "real life" has been already modeled, and a big range of models are based on Petrinets. But still, creating FM from ID seems to be rather an art than a well-understood process, and often lots of loops and refinements are required during the process of model creation.

However, there are some approaches to allow an automatic model generation. On approach is the use of semantically complete implementation languages like SDL or ALGOL. E.g., an automatic transformation from SDL'92 into a certain class of Petri Nets is studied in [3]. In fact, this transformation does not take account of all features of the SDL'92 and therefore the results of the analysis have to be proved on the original (SDL) system.

There are lots of approaches in the object-oriented area. Almost all of them focus on functional properties. An overview about object-oriented modeling one can find, e.g., in [12].

Frequently, the opposite way is gone: A implementation is derived from a FM. Currently, workflow approaches are quite popular. In the area of design of asynchronous circuits and systems exist tools, which synthesize self-timed digital circuit specified in terms of PN (comp. [1]). These tools can be consider as a generator: PN $\rightarrow$ real system.

An interesting approach by modeling multi agent systems with Reference Nets is shown in [5]. Reference Nets are timeless with nets as tokens (comp. [2]). Further similar studies are referred in [5].

The approach that is probably the closest to our is by [4]. The method proposed their is possible language independent. The paper introduced among other things a Petri Net generator which derives a (Timed) Petri Net from a program automatically. The net involves the time consumption of any action and the data dependencies among conflict decisions. Using Petrinet analysis these problems are studied.

The remainder of the paper is organized as followed: Section 2 describes the composable MSS architecture. Section 3 discusses, how MSS components can be modeled by Timed Petrinets and how this components can be composed in such a way, that that it allows a easy derivation from the informal description. In Section 4, we describe the *MGen* tool. It automatically generates TPN models of systems within the MSS architecture. Finally, in Section 5 we introduce a new class of Petrinets that allow describe systems such as MSS systems in an even more "natural" way, following requirement R1.

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

Our proposed architecture is called *Message Scheduled System* (MSS) [11] 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.

Here we only present few details on MSS, its description can be found in [10, 11, 9].

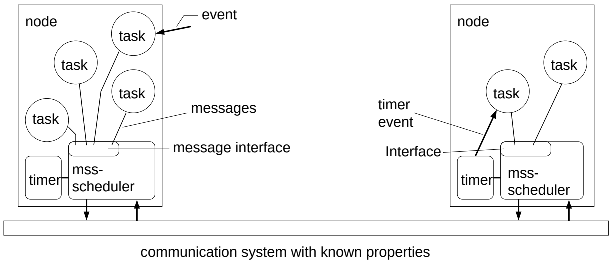

Figure 2: Message Scheduled System

2.1 The Basic Idea of MSS

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 RMA or EDF [6].

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 of the distributed system needs detailed knowledge about the whole system with all its nodes — the summarized parameters such as load are sufficient. The nonfunctional compatibility of two components — which means the test whether the components or "parts" are composable — can be calculated very easy (three schedulability tests) in polynomial or linear time. The functional compatibility is not an issue because this is well researched over many year (concepts such as object orientation, modularization, component systems such as CORBA [7]).

MSS is aware of three different types of components (Figure 2): 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.

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 Component Descriptions

From the composability viewpoint a small set of parameters describes the (functional and nonfunctional) behavior of a component. These parameters include (for a task component) the types of messages the component uses, the minimal interarrival times for them, the execution time for local execution, the minimal interarrival time for local execution and the priorities both for messages and execution.

It is also possible to express such a parameter set in a language similar to IDL [7] as introduced in [10], or in an XML-based approach.

According to Section 1 this (parameter set, language representation) introduces an rather informal description of a component, or, if we look at a composed system, an informal description of the system.

What we need now is a technique to specify the behavior of MSS formally, and a method to compose complex system descriptions out of that informal parameter set in a "natural" way because modeling becomes very complex if system size grows.

3 The Specification of MSS Using Timed Petrinets

We have decided to specify our architecture using Timed Petrinets (TPN) [8], because they allow us to specify time durations and priorities which both are needed for MSS.

For more information on that kind of Petrinets and considerations about other techniques applicable to MSS refer to [9].

Figure 3 shows the specification introduced in [9]. Here we concentrate on the composability issues of that specification and explain only few introducing details of the specification itself — for the whole explanation refer again to [9].

3.1 Specification of MSS Tasks

The basic building block in MSS is the task, so we start specifying a tasks behavior:

The execution of a task 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 and the communication medium. The model of a task must include both resources and must represent the resource sharing between different tasks.

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

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 [10]).

The following explanation of the specification refers to Figure 3. 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.

The *generators* are modeled by the part of the net shaded middle grey in the upper left of the figure. This generates token with a given minimal interarrival time and therefor specifies a behavior described above.

The box right from the generator models a task that shares the places "node" and "medium" (light-grey in the bottom) with other task. One of the main ideas behind that specification is that tokens are generated if one of the temporal requirements is not fulfilled. These tokens trigger the dark-grey shaded part of the net (call "instrumentation") that brings the whole net into a dead state (by "stealing" token from the initial generator places). That mechanism allows to check the specification for "dead reachable states" to prove the correctness of the specified system (if the system is correct, no dead state is ever reached). Therefor this specification is not only a specification, but allows also the analysis of behavior.

3.2 Composition of a System Specification

The specification parts introduced so far are the basic building blocks of each MSS specification. Generator, task, the places for medium and node and the instrumentation are simple combined to specify a single task running at a single node (this becomes another kind of building block). It is clear that the further 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. Such a composed system specification may look like the whole net shown by Figure 3. But, composi-

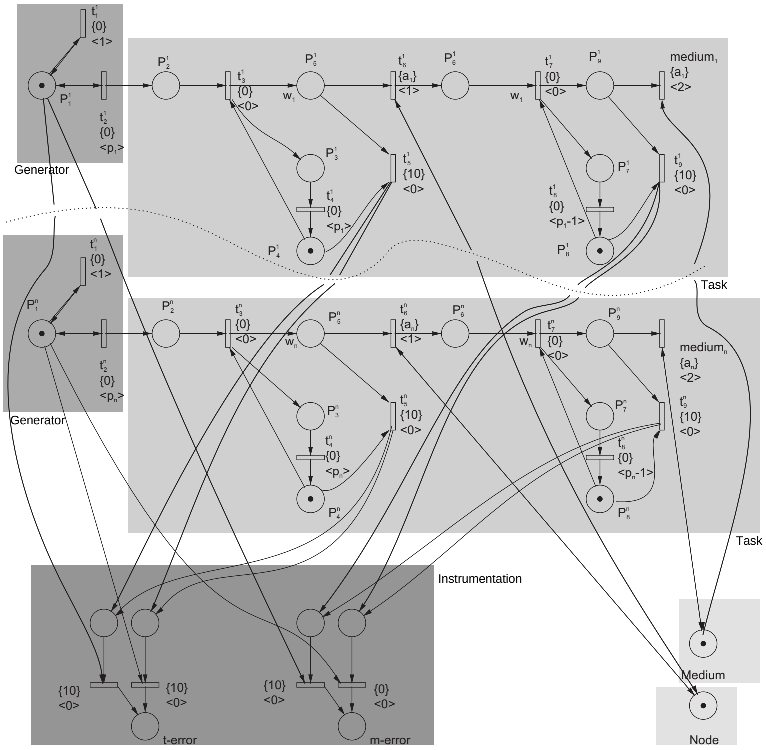

Figure 3: TPN description of a single-node MSS

tion 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.

4 Tool Support

The composition rules allow to build a net specifying a concrete MSS configuration. This is possible because of the composable structure of the model — the "parts" are clear distinguished in function and have simple interfaces (shared places or arcs between one parts transition and others part place or vice versa).

The resulting net (specifying a large MSS instance) can be analyzed using the tool INA [13] in a very easy way: It is only necessary to search for dead reachable states in the net because those represent missed deadlines (see "instrumentation" above). If no such state exists the system specified by that net will fulfill all temporal requirements.

The composition rules for a Timed Petrinet model of a concrete MSS-instance are simple 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 [10] (introduced in Section 2). We have developed such a tool, *MGen*, and are able to automatically build MSS models represented by Timed Petrinets. Those, this tool implements a rule-set to specify each possible configuration of MSS. With other words, we compose the system specification out of basic building blocks using the informal (or parametric) description as composition control. This is possible because of the "natural" modeling and its composable structure — according to Figure 1 we compose a model.

Moreover, we have expanded this technique into an automatic model checker for MSS: *MGen* composes the net specification out of the (informal) description and a version of INA slightly modified by us is used to analyze this net for dead reachable states. Thus, our technique allows us to have an automation not only for the modeling, but also for model checking based on an parametric and rather informal description.

5 Next Steps

We have shown, how to compose a formal Timed Petrinet model from an informal description in a "natural" way. However, some components like the generator could be modeled even more "naturally" if we could use time intervals instead of durations.

There is a class of Petrinets that utilize time interval: Time Petrinet (also called interval Petrinets, IPNs). Time Petrinets have no notion of priorities, which is needed to model state-of-the art real-time systems. One can model priorities with IPNs, but describing priorities with Time Petrinets consumes time and this leads to a time discrepancy between the real protocol and its model.

In order to overcome this disadvantage, we have defined a new class of Petrinet. It is a Time Petrinet where priority functions can be assigned. 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 Time Petrinet. 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 this new kind of Petrinet *Priority Interval Petrinets*, or π-PNs for short.

We have got already an implementation of π-PNs for INA. It allows us to model MSS in a even more "natural" way. Currently, we adapt the *MGen* tool to generate models on the base of π-PNs.

6 Conclusions

Within this paper, we have discussed, how to derive a formal Timed Petrinet model from a rather informal description. On the example of the MSS architecture we have shown, that a proper architecture design supports the automatic generation of a formal model, even if the architecture is not developed with a formal model in mind.

We have introduced our tool *MGen* which allows an automatic model creation that can be used as an input for a checker such as INA. Using *MGen*, we closed the tool chain that allow us to reason about implementation properties from an informal description.

In addition, we have suggested π-PNs, that allow us an even more "natural" way of modeling real-time systems.

Although we have demonstrated how a formal model can be derived from an informal description by an "obvious" mapping, the concept of obviousness in proofs is rather weak. If possible, the preferable way is to start with the formal model.

References

[1] S. Bulach, A. Brauchle, H.-J. Pfeiderer, and Z. Kucerovsky. Perti net based design and implementation methodology for discrete event control systems. In *LNCS 2075*, pages 81 – 100. 2001.

[2] R Falk. Petri nets as token objects: An introduction to elementary object nets. In *LNCS 1420*, pages 1 – 25. 1998.

[3] F. Fischer, E. Dimitrov, and U. Taubert. Analysis and formal verification of SDL'92. Informatik-Bericht 43, Humboldt University of Berlin, Berlin, 1995.

[4] M. Heiner. Petri net based software validation. Technical Report TR-92-022, ICSI Berkeley, California, March 1992.

[5] M. Köhler, D. Moldt, and H. Rölke. "modelling the structure and behaviour of perti nets agents. In *LNCS 2075*, pages 224 – 241. 2001.

[6] 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.

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

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

[9] J. Richling, L. Popova-Zeugmann, and M. Werner. Verification of non-functional properties of a composable architecture with petrinets. In *Proceedings of the CS&P'2001 Workshop*, 2001. (to appear).

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

[11] 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.

[12] 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.

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

[14] K. J. Turner, editor. *Using formal description techniques.* John Wiley & sons, Chichester, UK, 1993.