GEFÖRDERT VOM

Deliverable D3.3.B
Extended Analysis based on the SPES Architecture Meta-Model

Eike Thaden, OFFIS

Tuesday 25th January, 2011

Projektbezeichnung	SPES2020	
Verantwortlich	Eike Thaden, OFFIS	
QS-Verantwortlich	UPB	
Erstellt am	10.12.2010	
Zuletzt geändert	17.12.2010	
Freigabestatus		Vertraulich
		Projektöffentlich
	✓	öffentlich
Bearbeitungszustand		in Bearbeitung
		vorgelegt
	✓	fertig gestellt

Weitere Produktinformationen

Erzeugung	Eike Thaden (ET)
Mitwirkend	

Änderungsverzeichnis

Änderung			Geänderte Kapitel	Beschreibung der Änderung	Autor	Zustand
Nr.	Datum	Version				
1	10.12.2010	V0.1		Initiale Version	ET	in Arbeit
1	17.12.2010	V0.2		Internal Version	ET	in Arbeit
1	25.01.2011	V0.3		Internal Review	ET	in Arbeit
1	25.01.2011	V0.3		Review-Kommentare eingepflegt	ET	final

Abstract

This document describes the necessary steps to create a SPESMM model using our SPESMM UML profile which can be analyzed with the current version of our realtime scheduling analysis tool OrcaRT. To demonstrate the usage of our concepts we use an example named ”WheelBrakingSystem” which was modeled with IBM Rational Rhapsody.

Contents

1. Introduction **1**
1.1. Scheduling Problems . . . 1
1.2. Scheduling Analysis Backend . . . 1
1.2.1. Extensions . . . 2
1.3. System Model . . . 2
1.3.1. Task Network . . . 2
1.3.2. Hardware Architecture . . . 2
1.3.3. Allocation . . . 2

2. Modeling for Scheduling Analysis **4**
2.1. Preparation . . . 4
2.2. SystemModel, AbstractionLevel and Perspective . . . 4
2.3. Aspects: Realtime, Safety . . . 5
2.4. The Logical Perspective . . . 5
2.4.1. Logical Perspective: Required Behavior Specification . . . 7
2.4.2. The Logical Perspective: Parts inside of the Top Level Component . . . 9
2.5. Technical Perspective . . . 9
2.5.1. Electronic Control Unit (ECU) . . . 11
2.5.2. CAN Bus . . . 15
2.6. Allocation . . . 15
2.6.1. Computing Component to Task Allocation . . . 17
2.6.2. Communication Component to Signal Allocation . . . 17
2.7. Open Issues . . . 19
2.8. Conclusion . . . 24

A. Preliminaries **25**

1. Introduction

This document describes an extended analysis of the realtime aspects where ”extended” means that an integration into the SPES Architecture Meta-Model allows the seamless integration with all supported SPES Architecture Modeling frontends. As of writing this report, the tool chain of ZP-AP-3 comprises as frontend tool only IBM Rational Rhapsody™, but thanks to the common meta-model compatibility of the analysis tool with different frontends which might be added in the future is given. The main part of this document shows how embedded systems have to be specified with the SPES Architecture Meta-model to enable analysis with the OFFIS realtime analysis tool suite OrcaRT, using IBM Rational Rhapsody™. OFFIS provides an adapter capable of translating Rhapsody models with applied SPES Architecture UML Profile to their Eclipse UML pendants. It is integrated into the OFFIS Open System Platform which was written to ease the handling of models and tools. After the successful translation of a Rhapsody model scheduling analysis can be performed if the model complies to the rules described in this document. A plugin for scheduling analysis is integrated into the Open System Platform. The plugin translates the Eclipse UML model into the format required by OrcaRT, the OFFIS realtime scheduling tool suite.

Throughout this document we make use of the SPES Meta-model available for our project partners as UML 2.0 profile from the SPES Wiki [3]. Realtime properties are formalized using the Requirement Specification Language (RSL), which is described in [2, annex 6.1].

1.1. Scheduling Problems

The goal of scheduling analysis is to verify that tasks involving (distributed) computation possibly combined with communication via data buses does not violate defined (hard) deadlines, e.g. in a car the driver airbag has to be activated within 1 ms after the crash sensors detect a frontal crash depending on the current speed of the car. Let’s imagine that the required computation is performed by two tasks on two different processors, the first one connected to the crash sensor, the second one connected the airbag. The processors are connected by a data bus which is possibly used by other processors for communication, too. First issue is that the execution times of the tasks are not always constant but depend on the internal processor and memory states. Further difficulties arise if there are other tasks on one or both processors. In this case a scheduler has to arbitrate access to the processors. The crash event might occur in a situation where the responsible tasks is not currently running but might be suspended.

Scheduling analysis is performed by identifying the worst case scenario and verify that no deadline is ever exceeded. From a realtime perspective the system is than considered safe.

1.2. Scheduling Analysis Backend

In the context of SPES2020 OFFIS provides a custom prototype backend for scheduling analysis. This backend basically implements well-known techniques for holistic scheduling analysis

described/invented e.g. in [4]. The used techniques are *holistic* meaning that the result of the analysis is a system-wide scheduling fix-point or a failure to find such a fix-point.

In simple cases it can be decided one by one for each ECU whether the respective task set on that ECU is schedulable or not. Unfortunately if tasks in the system have varying execution times and other tasks in the system are activated by receiving signals from them, we observe the effect of jitter propagation. Those receiver tasks are then not activated absolutely periodically anymore, but with a jitter inherited by their sender task. This jitter has to be considered while performing scheduling analysis for any lower-priority task. Through this effect local changes on one ECU may have negative consequences for the schedulability of the whole system. Holistic scheduling tackles this problem by searching for a global fix-point.

1.2.1. Extensions

For systems using a mix of fix-priority preemptive scheduling and time-triggered scheduling, the classical construction of the critical instance (the one with the worst case behavior) is sometimes too conservative. This implies worse over-approximation than necessary. In this case systems which might be very well schedulable are declared to be non-schedulable. More detailed information can be found in [1, chap. 4].

1.3. System Model

The system model consists of a task network, a hardware architecture and an allocation between both of them.

1.3.1. Task Network

A task network consists of set of tasks and a set of signals, where each task may send one or more signals and may receive one signal. Each signal is send by exactly one task, but may be received by multiple tasks. Each task is characterized by an activation period (periodic task) or a minimal interarival time (sporadic task), a deadline and best and worst case execution times for some or all ECU types. Each signal is characterized by a number of bytes to transmit and a deadline.

1.3.2. Hardware Architecture

Basically the hardware architecture consists of Electronic Control Units (ECUs) and data buses. Each ECU may be physically connected to one or more data buses. ECUs are characterized by a scheduler type (either fixed-priority preemptive or time-triggered), time needed for OS-specific tasks (time needed by the scheduler, etc.) and an ECU (processor) type. Each bus is characterized by a transmission speed, the overhead for frames in bytes and a bus type which determines whether access to the bus is arbitrated either priority-based (CAN) or time-sliced-based (FlexRay in static configuration, etc.)

1.3.3. Allocation

Each task has to be allocated to exactly one ECU. Signals however are required to be allocated to one data bus if at least one receiver is on another ECU as the sender. Otherwise they may be on the bus, too. Currently we assume that each signal can be transmitted using one data

bus only, excluding hardware architectures where there is no direct bus connection between a sender task's ECU and any receiver task's ECU.

2. Modeling for Scheduling Analysis

In this chapter we describe how to use IBM Rational Rhapsody adequately to model input systems for our scheduling analysis.

2.1. Preparation

Start by creating a new Rhapsody Project using Rhapsody's *New* wizard and save the project. We recommend copying the file SPES Architecture UML Profile (*spesmm.sbs*) into your newly created project directory *<project name>_rpy*. The dependency of your Rhapsody project on that file will then automatically be saved with relative path. Of course in general the profile file could be anywhere on your file system.

Next step is to add the SPESMM profile to the new project (*File→Add Profile to Model..*), choose the file *spesmm.sbs* in the file dialog. Your project is now prepared to be used with the SPES Architecture Meta-Model.

2.2. SystemModel, AbstractionLevel and Perspective

There are some structural elements in the SPES Architecture Methodology (for more details please have a look at [2]). Each of the concepts *SystemModel*, *AbstractionLevel* and *Perspective* is implemented in the SPES Architecture Rhapsody Profile as UML stereotype for the Meta class *Package*.

First we create one package and apply the stereotype *SystemModel*. All other packages have to be inside of this top level package. For the scheduling analysis we just need one package with the stereotype *AbstractionLevel*. Of course you can create arbitrary many other abstraction levels, e.g. for representing the system under development with lower granularity. In the properties of the system model package under *Tags*, set the *topLevel* tag to the abstraction level with the lowest granularity (top level abstraction level). If you have more than one abstraction level set the tag *successor* of each abstraction level (except the lowest level) to the next fine-granular abstraction level to establish an order of granularity between abstraction levels. Make sure to avoid circular dependencies.

The lowest instantiation of the *abstraction level* concept is used for modeling the realtime scheduling properties of our system. For the rest of this this document we call this particular abstraction level *scheduling abstraction level*. Inside of our scheduling abstraction level we need at least two packages with the stereotype *Perspective*: first the *LogicalPerspective* and second the *TechnicalPerspective*. While names for abstraction levels can be arbitrary, please follow the naming conventions in SPES2020 of the perspectives and use the before-mentioned names. Inside of each abstraction level package, each perspective has to be unique, e.g. there may not be more than one technical perspective. For each perspective set the tag *kind* accordingly, e.g. to *logical*.

The result should look similar to Figure 2.1.

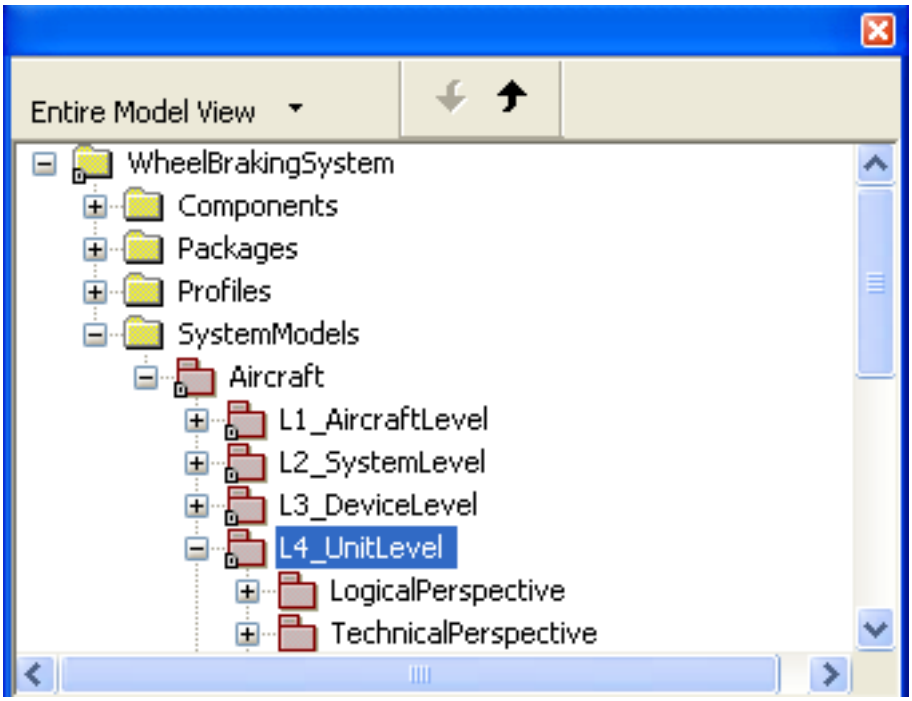

Figure 2.1.: SystemModel, AbstractionLevel and Perspective

2.3. Aspects: Realtime, Safety

Directly below the *SystemModel* package ”WheelBrakingSystem” there is a package ”Aspects”, as shown in Figure 2.2. This package defines some *Aspect* elements which are used to later to categorize SystemRequirements. The elements in our example are just chosen arbitrarily except for the ”Realtime” *Aspect* which is used throughout this document.

2.4. The Logical Perspective

In general on the logical perspective one describes the logical components of the system without information about processing units, communication media, and so on. A comprehensive definition of the term ”logical component” can be found in [2]. In the logical perspective of the abstraction level dealing with scheduling analysis we follow this directive by keeping all hardware related elements out of this perspective (all of them are located in the technical perspective). However we make explicit, that two logical components which are implemented as operating system software tasks (running on a processor with a specific realtime operating system) or on dedicated (but different) hardware need to communicate through hardware communication media, e.g. bus systems.

The rational for introducing communication components is simple: So far (on higher abstraction levels) we assumed that communication paradigm over connectors is either instantaneous (simple connectors) or further defined by a complex connector type. For scheduling analysis we have to have a detailed characterization of all the properties related to communication, e.g. its duration. We also must have a way to allocate communication to communication media in the technical perspective.

More about this this follows later when describing the part concept on the scheduling abstraction level. For now we start with the overview of the logical perspective in our example as

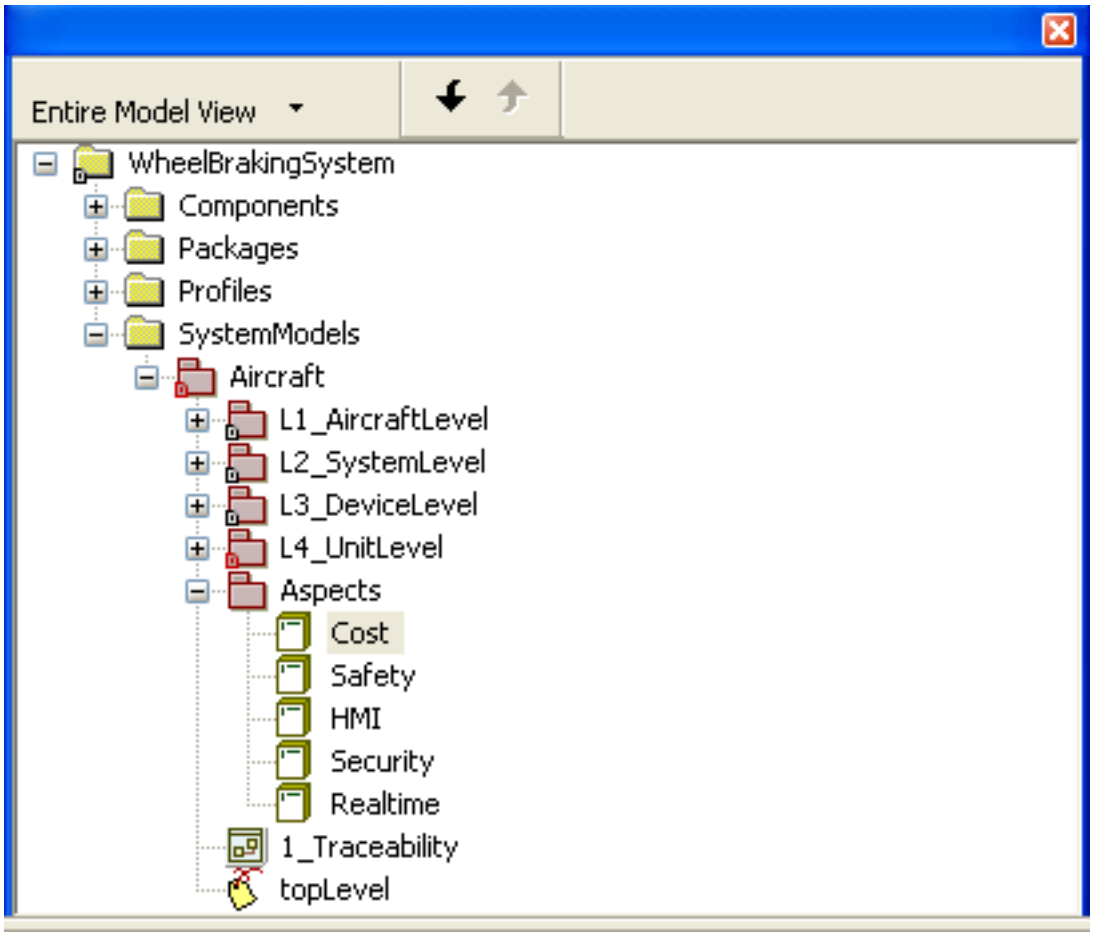

Figure 2.2.: Aspects

given in Figure 2.3.

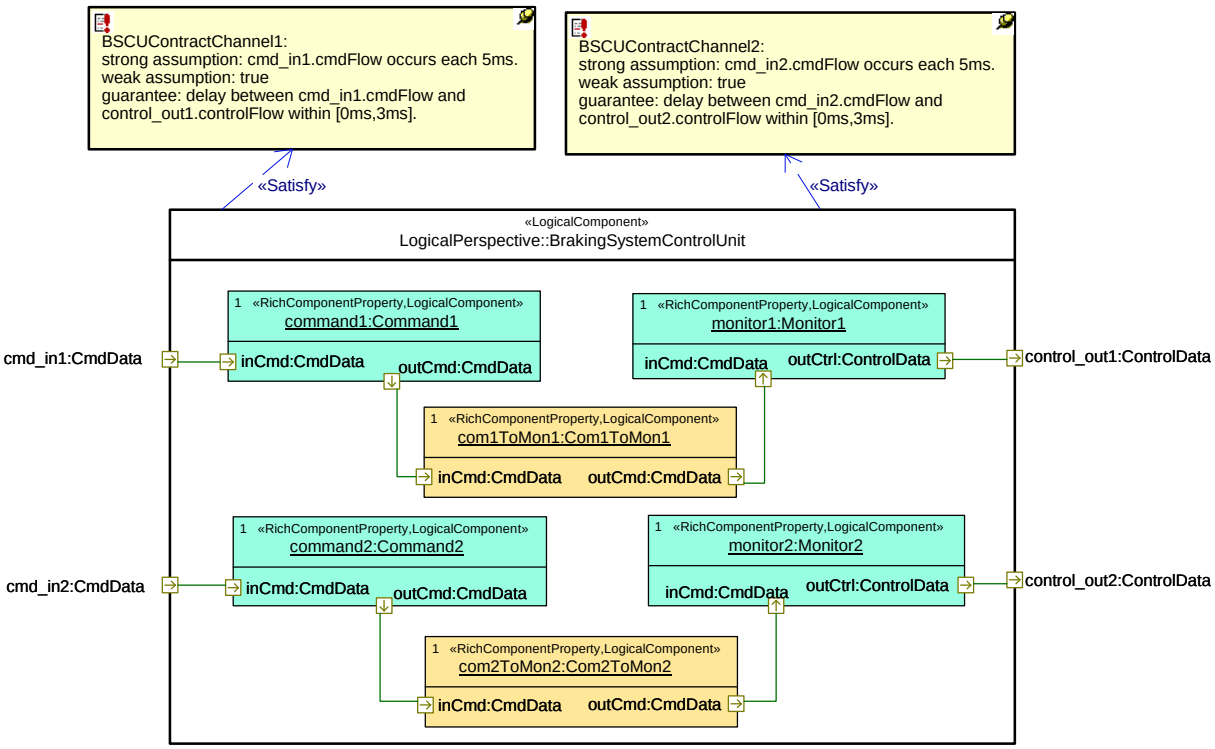

Figure 2.3.: Logical Perspective: Top Level Component with Parts. For more detailed explanation see section 2.4.1

In the SPES Architecture Meta-Model each Perspective has exactly one top level component. For each Perspective there is a tag *root* which has to be set to the top level *RichComponent* inside of the package where top level means that this component is the root of the decomposition hierarchy inside of the perspective. In our example the top level component is called ”BrakingSystemControlUnit”. It has some ports which indicate its system boundaries, two incoming ports ”cmd_in1” and ”cmd_in2” on the left side and two outgoing ports ”control_out1” and ”control_out2” on the right side.

2.4.1. Logical Perspective: Required Behavior Specification

The required behavior of our top level component is specified by using formal contracts in the form of a stereotype *System Requirements* containing RSL expressions, ”BSCUContractChannel1” and ”BSCUContractChannel2”. In Figure 2.3 you can only see those names and the content of the comment field, not to be mixed up with the actual tags of the contracts. We can further see that there are ”Satisfy” dependencies between the top level component and each System Requirement. In Figure 2.4 there are two example for RSL expressions. First the *strongAssumption* tag contains an expression which means that on port *cmd_in1* an event *cmdFlow* is expected to occur (because the expression is in an assumption) periodically each 5ms. Second the *guarantee* tag contains an expression which ensures (because the expression is part of a guarantee) the occurrence of an event *controlFlow* on port *control_out1* 0-3ms after the aforementioned *cmdFlow* event has been received. We do not have a weak assumption in this case. The difference between strong and weak assumption basically is that strong assumptions are checked during design time. A violation of a strong assumption of a component means, that the

component was integrated into an improper environment, e.g. a hair dryer may not be operated underwater. In contrast to strong assumptions, weak assumptions are not checked during design time. They allow distinguishing different runtime contexts of a component, e.g. a hair dryer component may be used with a power supply of either 110 V or 240 V respectively. Changing the voltage during operation presumably is not permitted (and not likely to happen) in this case and could destroy the component!

In the Tags tab of the features dialog of the System Requirement you can see several tags. First of all this requirement belongs to the aspect ”Realtime” as it specifies required realtime behavior. The classification of requirements is not always that clear. Therefore one System-Requirement may belong to multiple aspects. The requirement has a strong assumption which characterizes the required behavior of the environment of the component. The requirement in example is that an event ”cmdFlow” occurs exactly every 5ms on port ”cmd_in1”. If the environment would generate ”cmdFlow” e.g. each 3ms, the contract is violated and the component fulfilling this requirements has not been deployed correctly. A virtual integration test can be used to detect those kinds of integration problems.

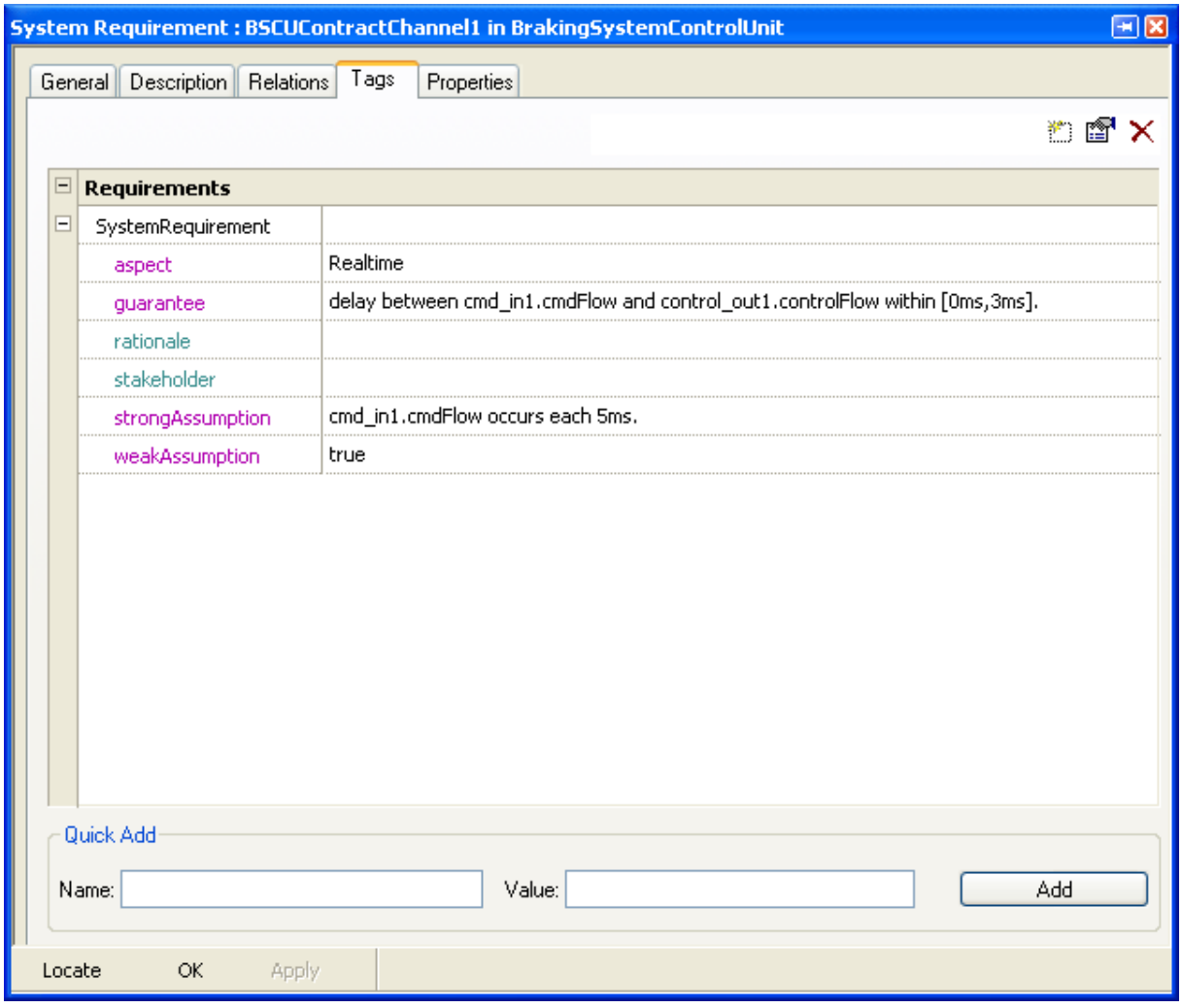

Figure 2.4.: Logical Perspective: Tags of a System Requirement

Let’s assume that the component fulfilling requirement ”BSCUContractChannel1” has been deployed in a compatible environment. Then the component guarantees a specific behavior, given in the requirements ”guarantee” tag. In this case the component guarantees to generate an

event ”controlFlow” on port ”control_out1” in the time interval of $[0ms, 3ms]$ every time after an event ”cmdFlow” has been received on port ”cmd_in1”.

2.4.2. The Logical Perspective: Parts inside of the Top Level Component

On the scheduling abstraction level, whenever logical components representing functionality are communicating, dedicated communication components have to be inserted between the communication partners. These ”special” communication components can be seen as concretion of connectors between ports of logical components on the next higher abstraction level. In Figure 2.3 there are ”yellow” sub components (parts) of the logical perspective's top level component named ”Com...”. In this example the logical components ”Command1” and ”Monitor1” are connected with each other using communication component ”Com1ToMon1”. This communication component has been introduced because on the next higher abstraction level, there are the pendants of ”Command1” and ”Monitor1” and a direct connector between them.

There are six parts inside of the top level component, each part typed by the stereotype *RichComponent*. All incoming ports on the border of the top level component are delegating their communication to one part; all outgoing ports are forwarding communication from parts to the outside world.

For the adapter to work properly it is important that the parts connected to top level component's delegation ports are always allocated to *software task components* (simply called *software tasks* further on) in the technical perspective. We call those parts *computing parts* (and their types are called *computing components*, respectively.

Furthermore the current implementation expects that between computing parts there always is one part (type by a communication component with stereotype *RichComponent*) which is allocated to a *Message* in the technical perspective. We call such a part a *communication part* and its type a *communication component*. Note that there currently is no explicit stereotype for computing and communication components in the SPES Architecture UML Profile. All components share the same stereotype: *RichComponent*. Only the allocation relation between elements of the logical and technical perspectives are the place to specify the role of a component inside of the logical perspective. This might be changed later. Section 2.6 describes in detail for our example how to interpret the allocation relation and how it is established.

There are some special constraints due to currently unimplemented features of the SPESMM-to-Orca converter. Please have a look at section 2.7 for details.

Parts of the logical component are described using System Requirements, too. Figure 2.5 shows that in our example each part of the top level component on the logical perspective satisfies exactly one System Requirement (but this is not required in general).

Realtime contracts currently always have to be in the form as shown in Figures 2.5 and 2.4. That means that there as to be an strong assumption on the activation behavior of one event on the input port of component. There also has to be a guarantee which guarantees a reaction on that event in a certain time interval. If such a contract is missing for a computing component which gets its input from the environment the scheduling analysis cannot be performed (”Missing Trigger”). The maximal delay between activating event on reaction is interpreted as deadline for the response time of a software task (technical perspective).

2.5. Technical Perspective

The technical perspective has one top level component, too. Figure 2.6 shows this top level component which represents the whole technical computer and communication system.

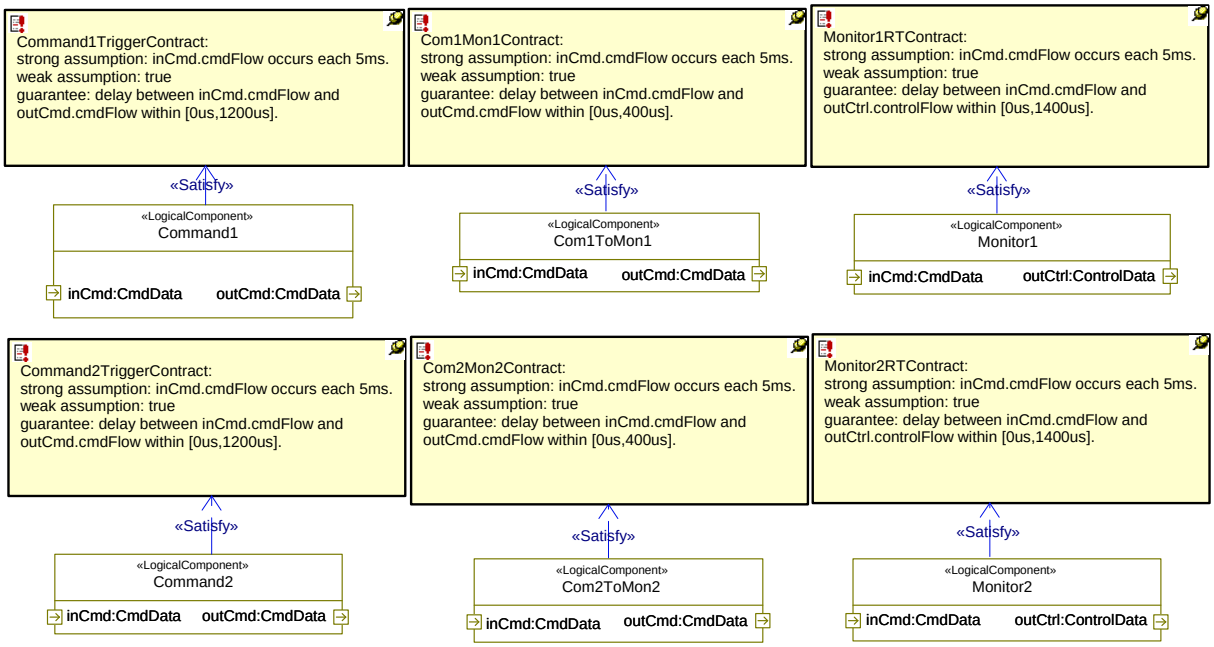

Figure 2.5.: Logical Perspective: Each part of the top level component satisfies one contract specified with stereotype *System Requirement*

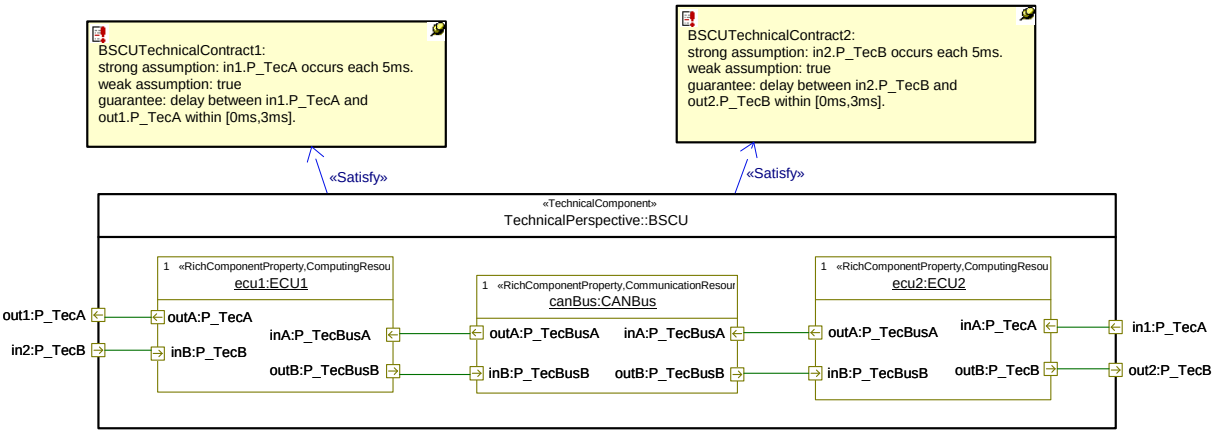

Figure 2.6.: Technical Perspective: Top Level Component

The top level component contains three parts: Two ECUs and and CanBus connecting them. In the technical perspective ports are abstractions of physical jacks. The top level component in our example is called ”BSCU”. It has two contracts attached which are currently not evaluate by the realtime analysis. This may be changed later.

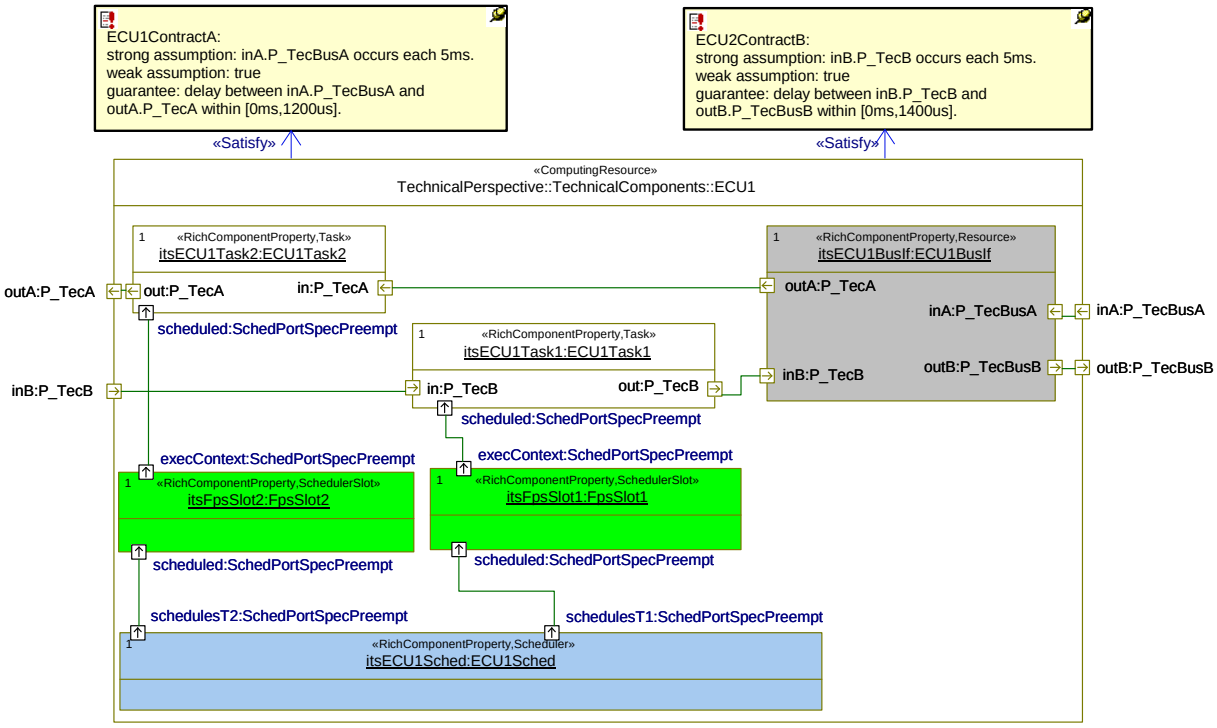

Figure 2.7.: Technical Perspective: ECU1

2.5.1. Electronic Control Unit (ECU)

In Figure 2.7 one can see the inside of ”ECU1” (the type of part ”ecu1” of the top level component). The RichComponent ”ECU1” represents a physical electronic control unit (ECU) of the system under development. It has the stereotype ”ComputingResource” applied. The ”ComputingResource” element itself has no information about the actual hardware used. Instead this information is available by specifying the processor type of the Computing Resource. Figure 2.8 shows that ”ECU1” has type ”ARM7”. ”ARM7” itself is an element with stereotype *HWProcessor* (Figure 2.9). Note that ”ARM7” is only used as example here. The user is free to use any ECU type name in his models as long as it complies to the UML naming conventions for classes. The tags of *HWProcessor* will be used for further extensions.

There are two parts whose Components have ”Task” in its names. These parts represent operating system software tasks deployed to this ECU. The types of each task have the stereotype *Task* applied. They specify all information required to deploy them to an ECU, currently the execution time depending on the ECU’s processor type. This is done by adding attributes like it has been done in Figure 2.10. Both attributes have stereotype *ExecutionTimeSpec* applied. In Figure 2.11 three tags are used to specify the characteristics of tasks.: ”scope” specifies the name of the HWProcessor for which the values are valid. Tags ”lower” and ”upper” specify the minimal respectively maximal execution time on the HWProcessor.

The tasks are connected to the outside of the ECU, e.g. to actuators and sensors. They are also connected to one ”FpsSlot”. Each slot itself is connected to one scheduler. Slots have an unique

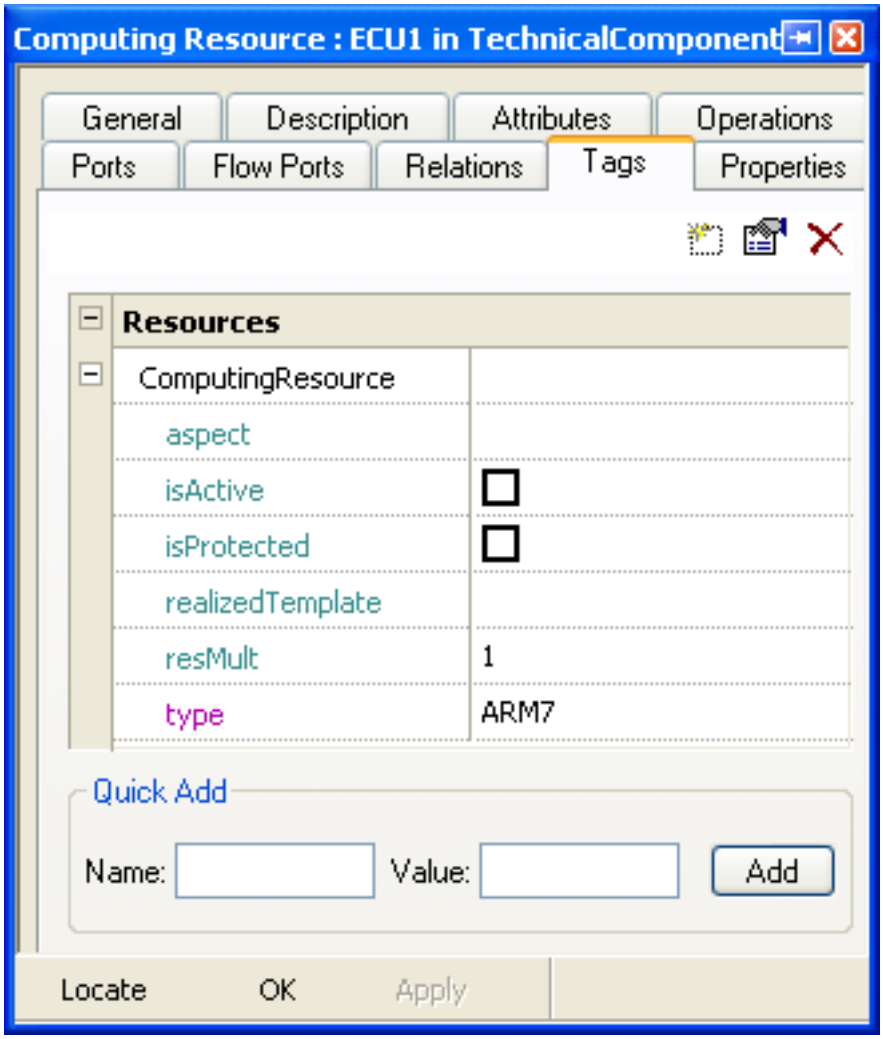

Figure 2.8.: Technical Perspective: Tags of SchedulingResource ”ECU1”

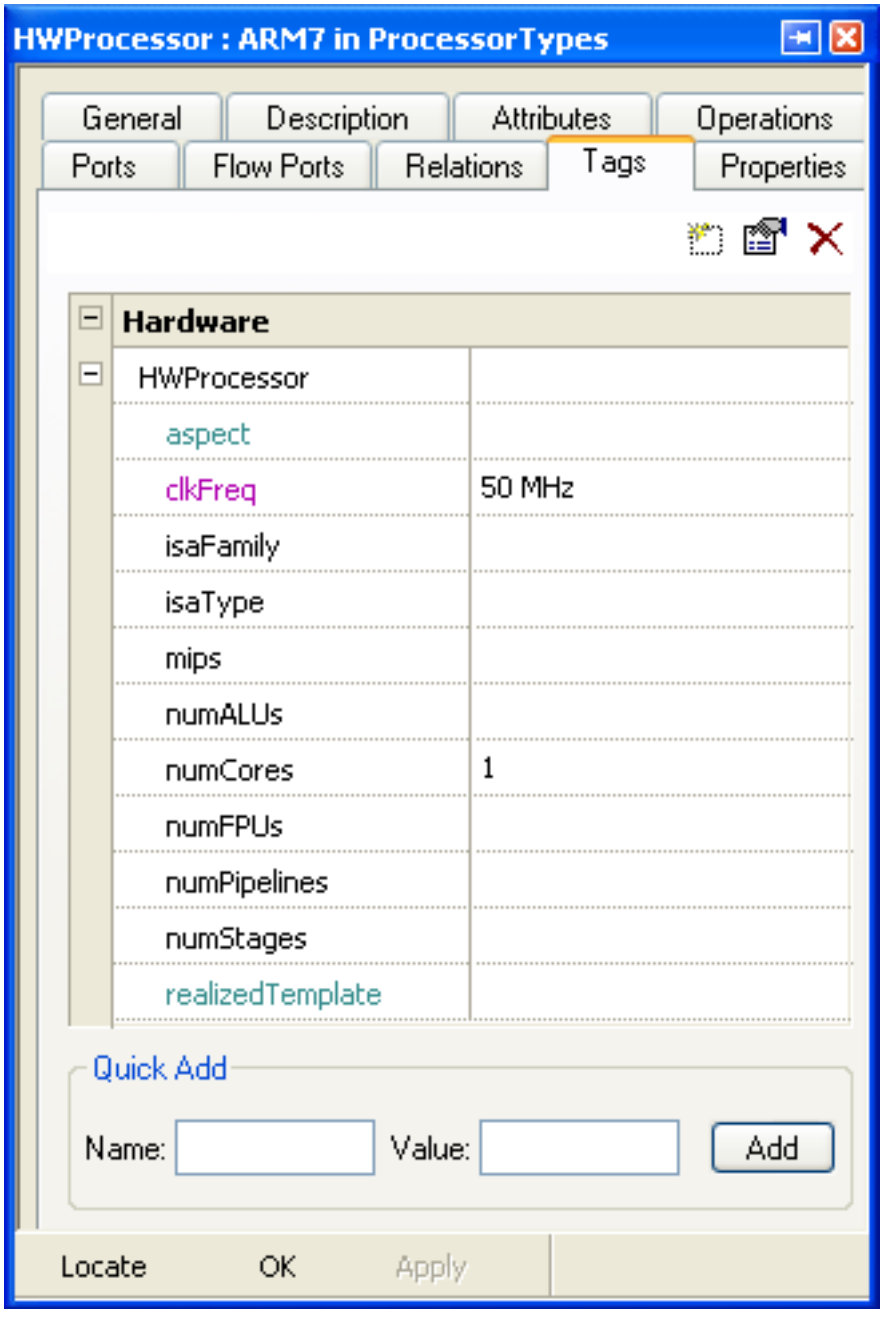

Figure 2.9.: Technical Perspective: *HWProcessor* "ARM7"

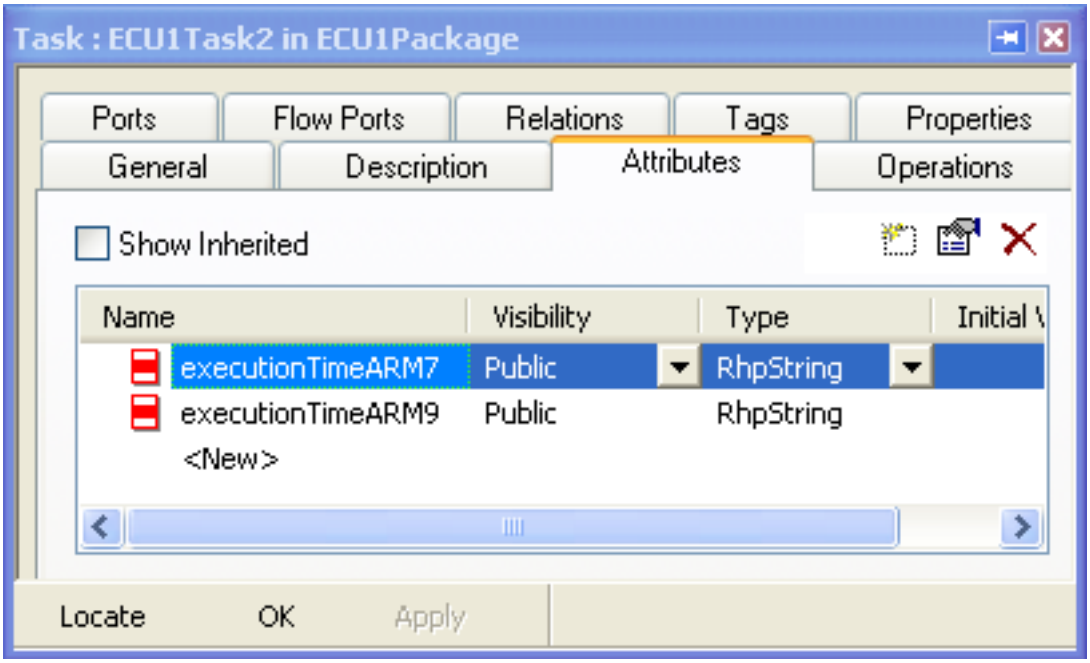

Figure 2.10.: Technical Perspective: Attributes to specify Execution Times via stereotype ”ExecutionTimeSpec”

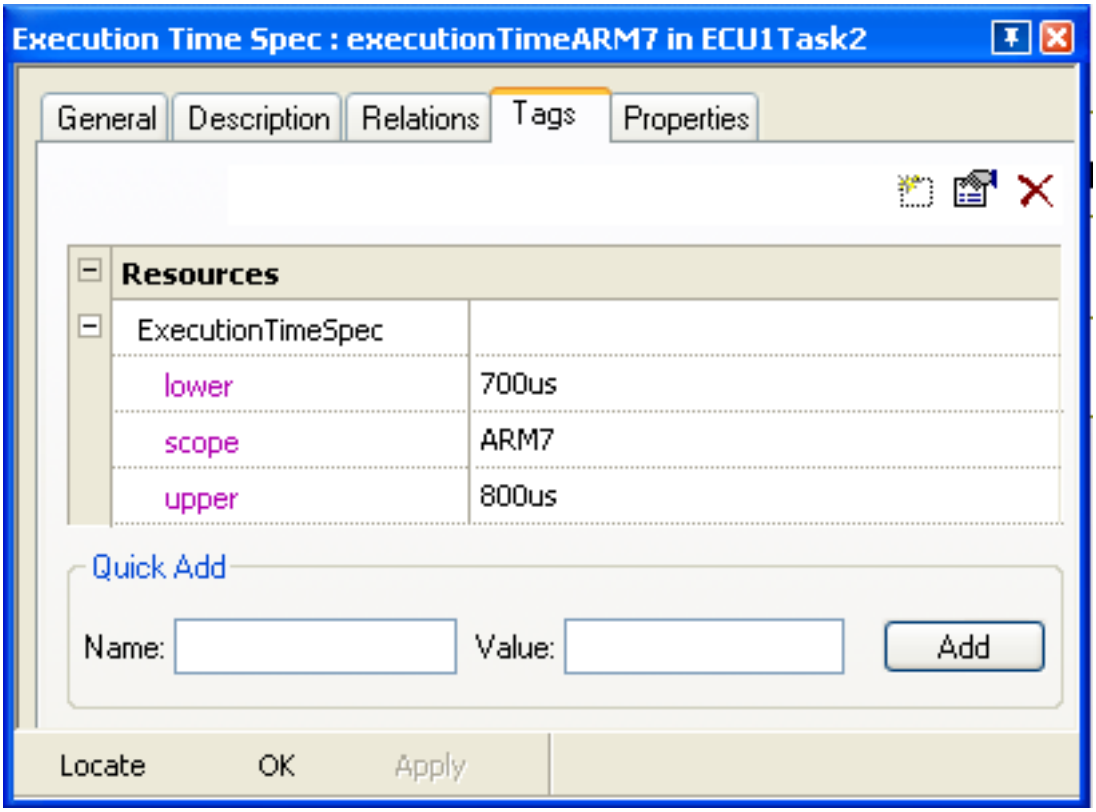

Figure 2.11.: Technical Perspective: *ExecutionTimeSpec* for attribute ”executionTimeARM7”

priority assigned (Figure 2.12). The remaining element in Figure 2.7 – ”ECU1BusIf” – is not currently used during the scheduling analysis. It represents properties of the communication stack which will be used later to make the specification of HW Elements more flexible.

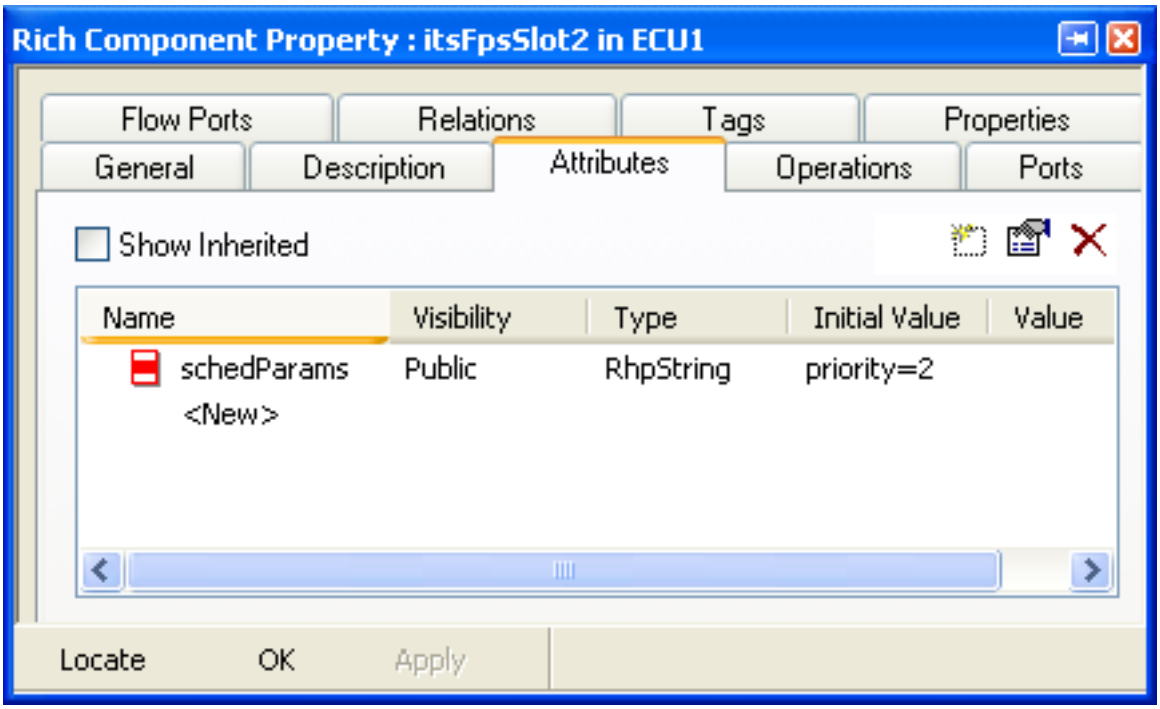

Figure 2.12.: Technical Perspective: SchedulingSlot Parameters

The Scheduler in the ”ECU1” component has some tags, too. The only currently important tag is called ”policy_SchedulingPolicy_policy” which is set to ”FixedPriority” in our example, indicating that ”ECU1” uses a fixed-priority preemptive scheduler for arbitrating its tasks.

2.5.2. CAN Bus

In Figure 2.14 it can be seen that the content of the *CommunicationResource* component ”CAN-Bus” uses similar concepts as the ECU. Both *Scheduler* and *SchedulerSlot* are the same stereotypes as described above. A new concept is that of *Trigger*. A *Trigger* represents the fact that sending a message over a bus is initiated by arrival of a triggering event (e.g. ECU wants to send data frame) and has certain properties like required transmission duration. These properties are intermediate results from our analysis and cannot be specified by the user. As for the ECUs RSL contracts on buses are currently not evaluated but shown in example for demonstration purposes only. How to allocate communication components in the logical perspective to bus slots in a bus on the technical perspective is described in detail in section 2.6. Note that this doesn’t mean bus communication is currently unsupported. Bus communication is a highly important part of the holistic scheduling analysis. However the required bus properties are specified directly as tagged values on the bus components, instead of (mis-) using the RSL for this purpose.

2.6. Allocation

It is necessary to specify the relation between elements of the logical and the technical perspective to be able to perform scheduling analysis. Elements of the logical perspective are allocated

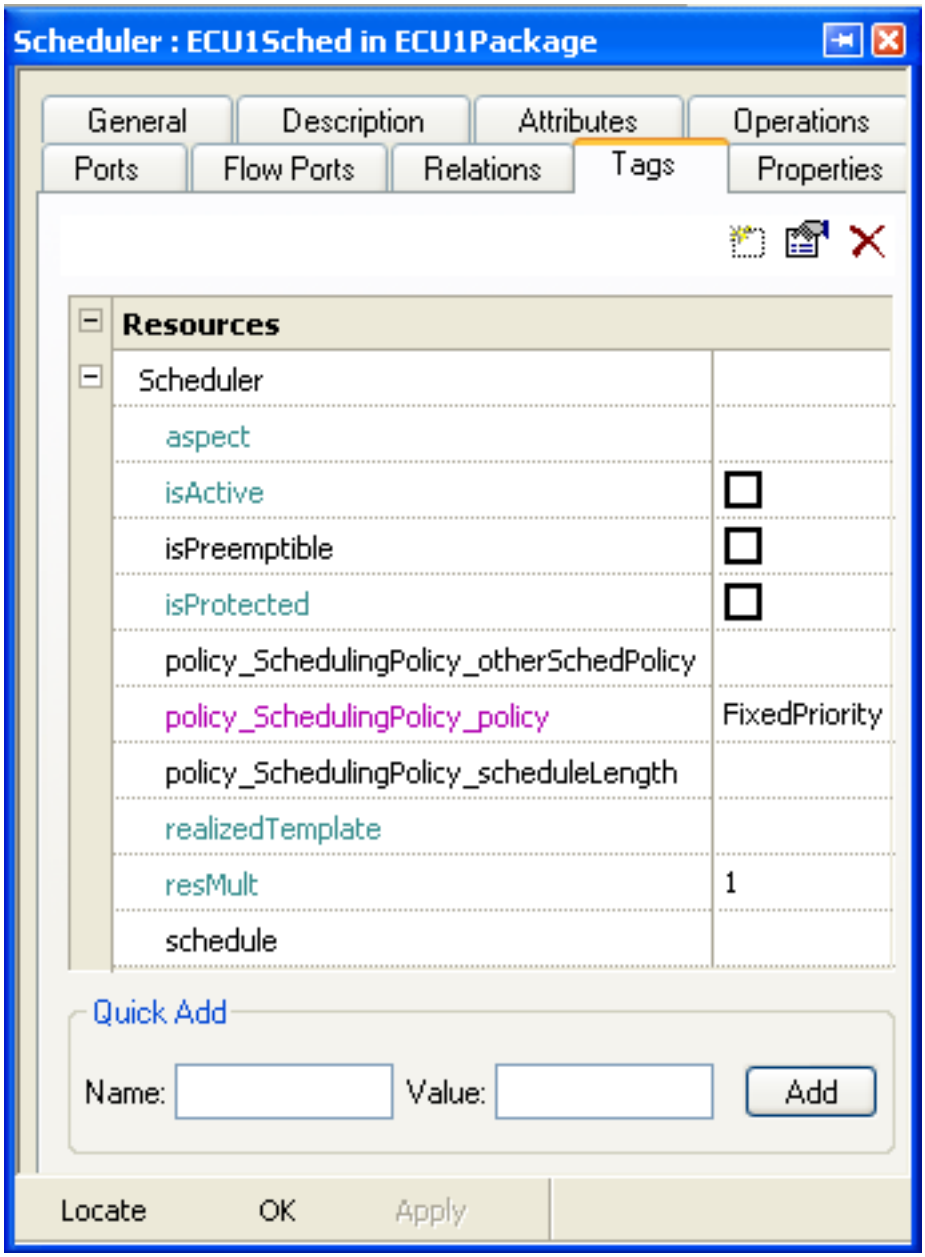

Figure 2.13.: Technical Perspective: Scheduler of ECU1

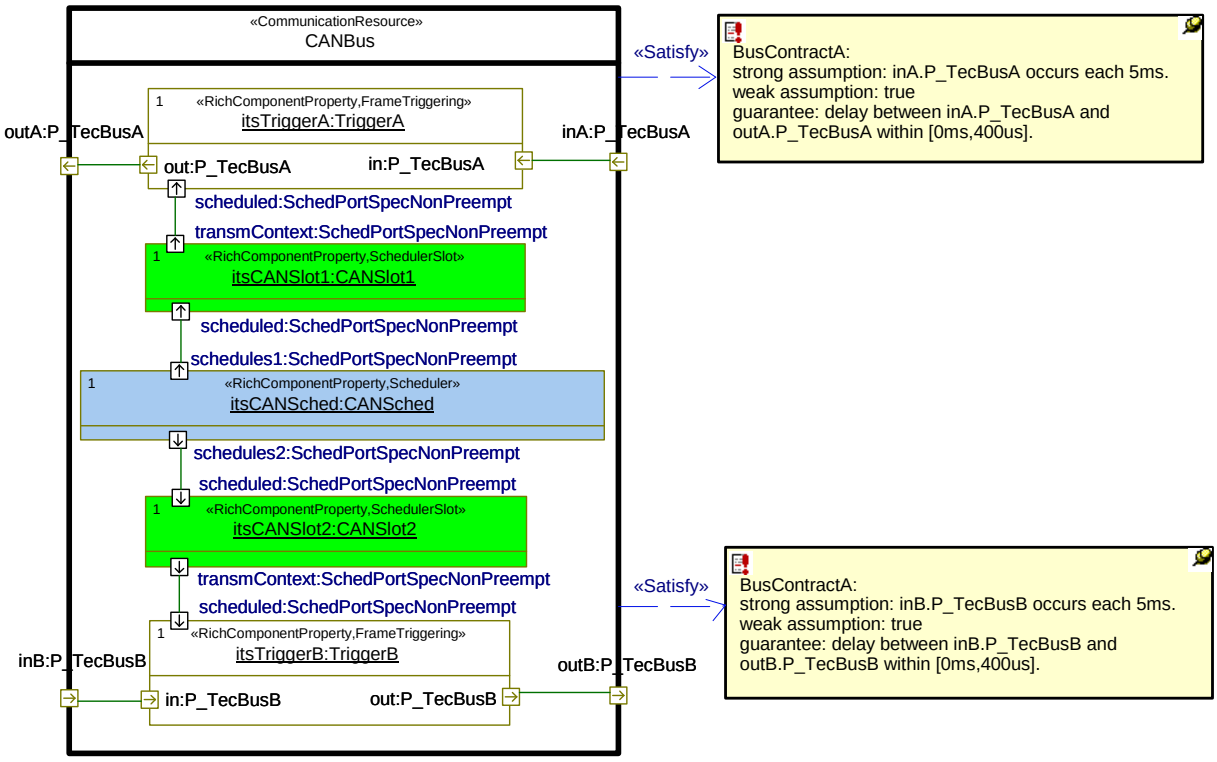

Figure 2.14.: Technical Perspective: CAN Bus

on elements of the technical perspective. This involves mapping of behavior, too, and implies that the mapping relation is formally verified. Verification of the allocation relation is NOT tackled in this document.

Allocation is described in to steps: First the allocation of computing components in the logical perspective to tasks in the technical perspective is described in section 2.6.1. Second the allocation of communication components to signals, which are itself allocated to message, which are allocated to frames is described in section 2.6.2.

2.6.1. Computing Component to Task Allocation

Each computing component part (the part found in the top level component) inside of the logical perspective has to be allocated to exactly one task part (the part used in one *ComputingResource*). Note that the allocation relations as visualized in Figure 2.15 are always between parts, never between types. In addition to the part allocation you may specify allocation dependencies between ports of the part's types. Currently this is not evaluated by the OrcaRT adapter.

2.6.2. Communication Component to Signal Allocation

Communication component of the logical perspective are allocated to elements of type *Signal* in the technical perspective. *Signal* is just a helper class which allows one to group data of several communication components into one *Message*. A *Message* is grouped together with other Messages into one Frame. There are currently some limitations, have a look at section 2.7 for details.

Note that the arrows and hierarchy of Signals, Messages and Frames as seen in Figure 2.16 are only used for better illustration. The real allocation is done using tags.

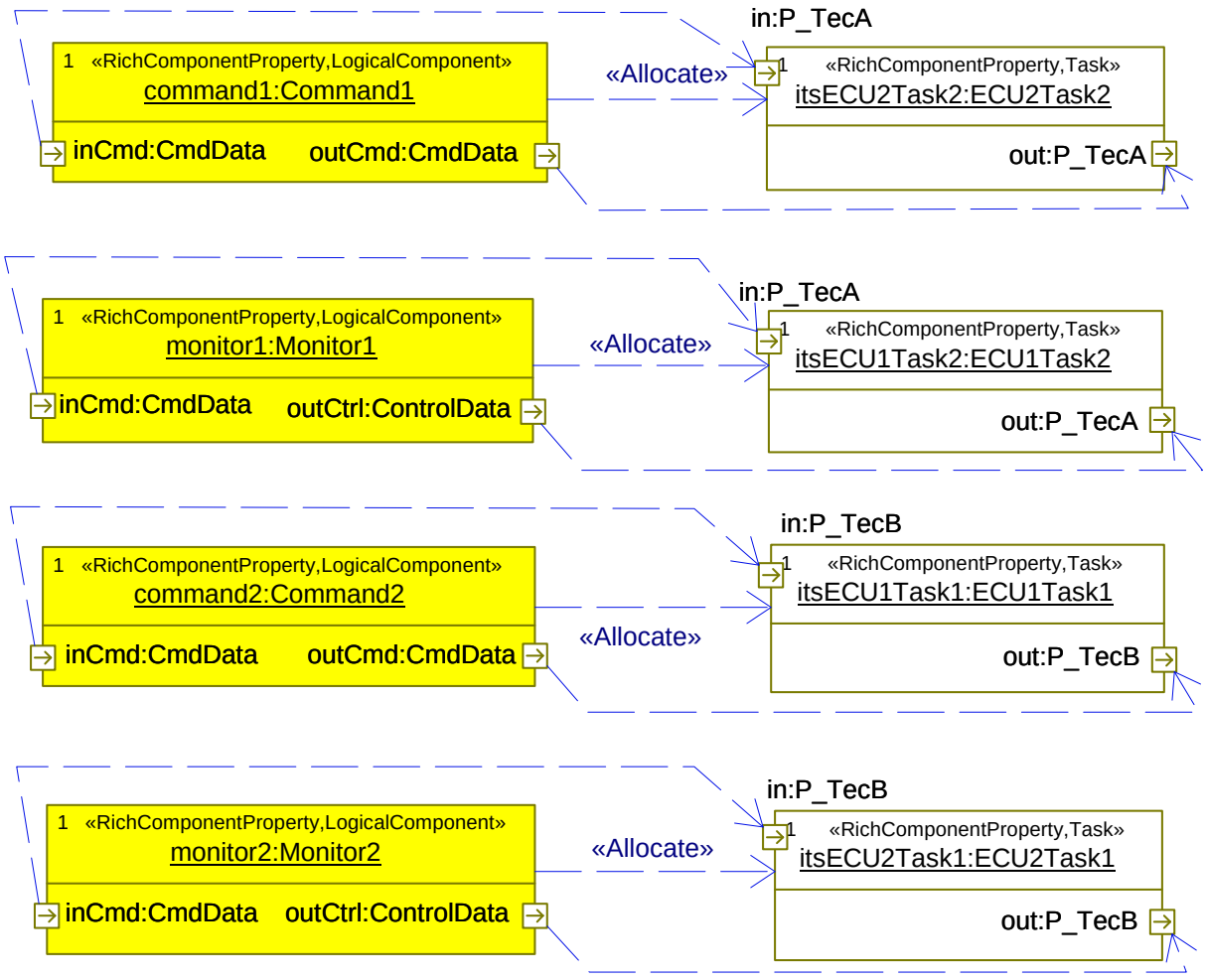

Figure 2.15.: Technical Perspective: Allocation of Computing Components (logical perspective) to Tasks (technical perspective)

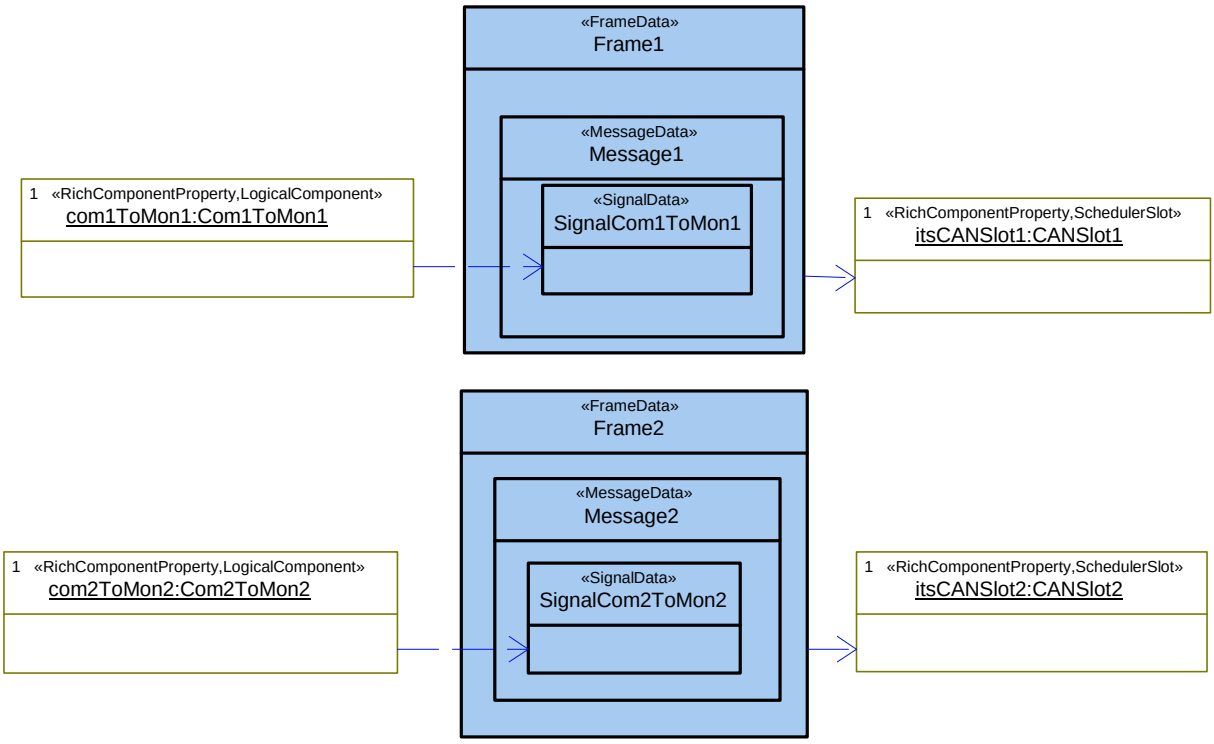

Figure 2.16.: Technical Perspective: Signals, Messages and Frames

In Figure 2.17 you can see that tag ”commuicationComponent_Signal_compComp_part” points to the original communication component in the logical perspective. You further see that a length tag has been specified which holds the number of bits inside of this Signal.

Figure 2.18 shows how Messages specify their Signals using a *SignalToMessageMapping* element. The tags of one of these Mapping elements can be seen in Figure 2.19. The tag ”signal” is set to the Signal which belongs to this Message. All other tags are currently ignored.

Figure 2.20 shows how to map message ”Message1” via the *MessageToFrameMapping* to frame ”Frame1”. All other tags are currently ignored.

Finally Figure 2.21 shows how a Frame is allocated to an actual Bus Slot, in our example to ”itsCanSlot1”. All other tags are currently ignored.

2.7. Open Issues

There are some limitations in the current implementation of the adapter (SPESMM to OrcaRT). For special scheduling properties we adopt an case study-driven approach where additional implementation effort is spend according to the requirements of the specific case studies. The following limitations are present in the current implementation:

- Having more levels of hierarchy in the logical perspective than in the example (two) is not supported
  - The top level component directly contains sub components (parts) which may not be further decomposed.
- The bus type is limited to CAN. However profile elements for supporting e.g. TDMA-based buses are already in place but are (completely) translated to OrcaRT currently.

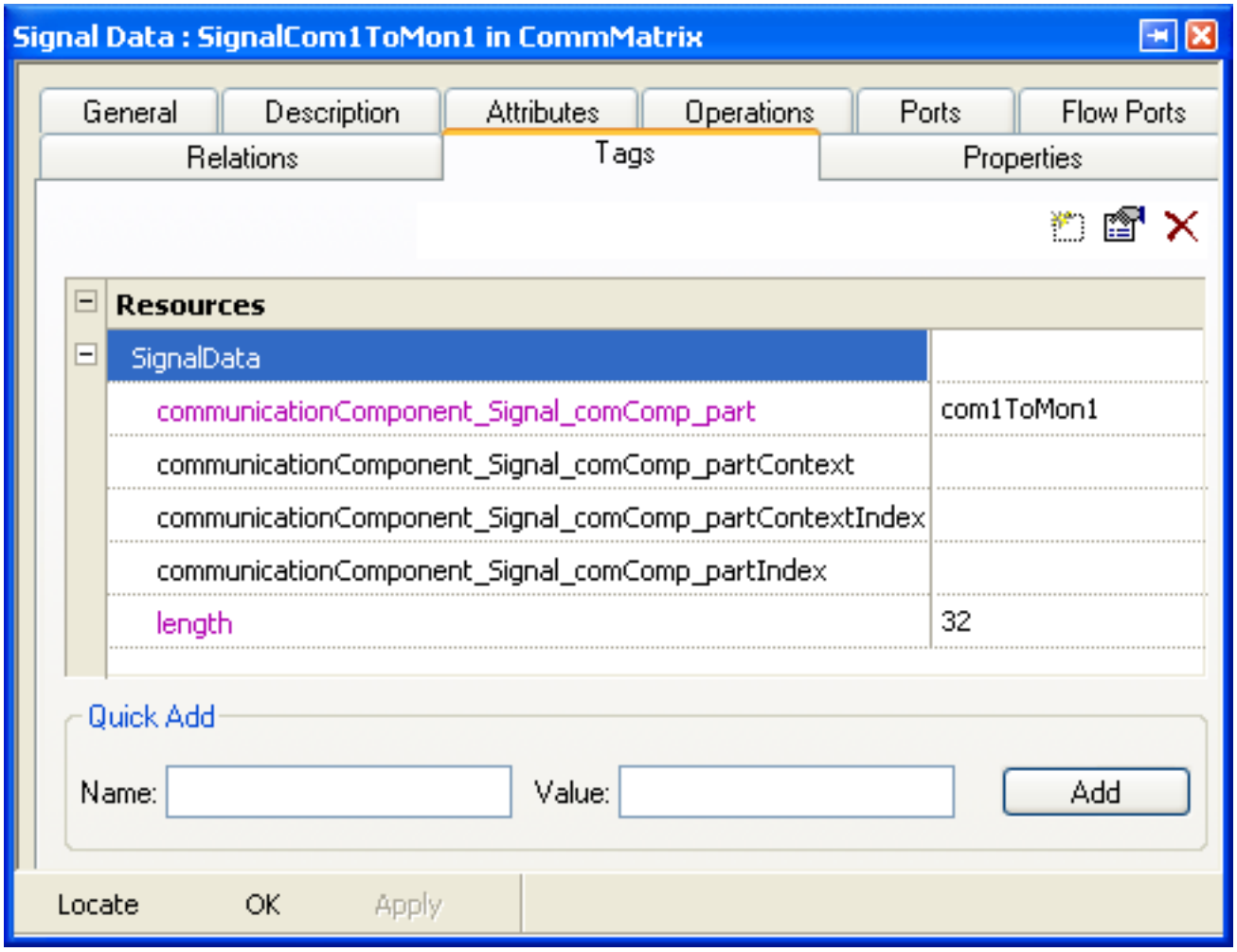

Figure 2.17.: Technical Perspective: Tags of one Signal

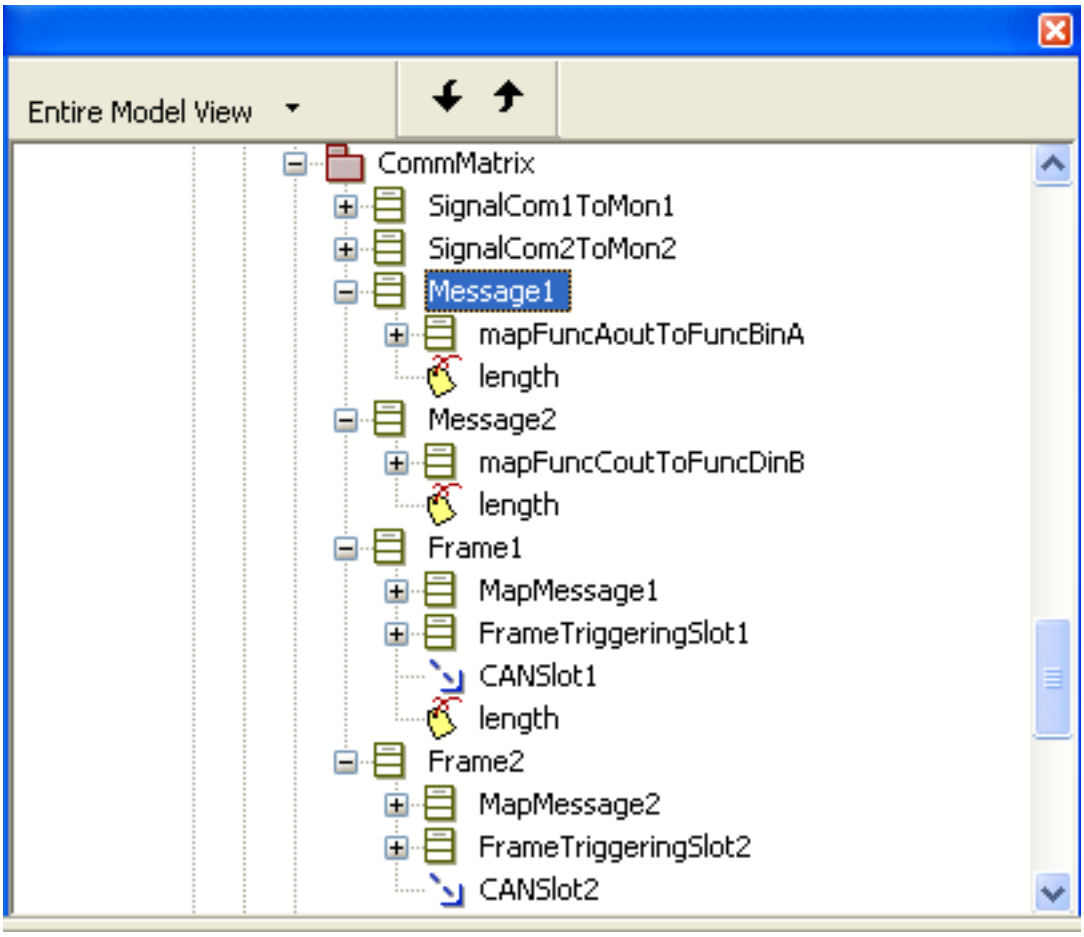

Figure 2.18.: Technical Perspective: Messages to Frames and Frames to SchedulingSlots

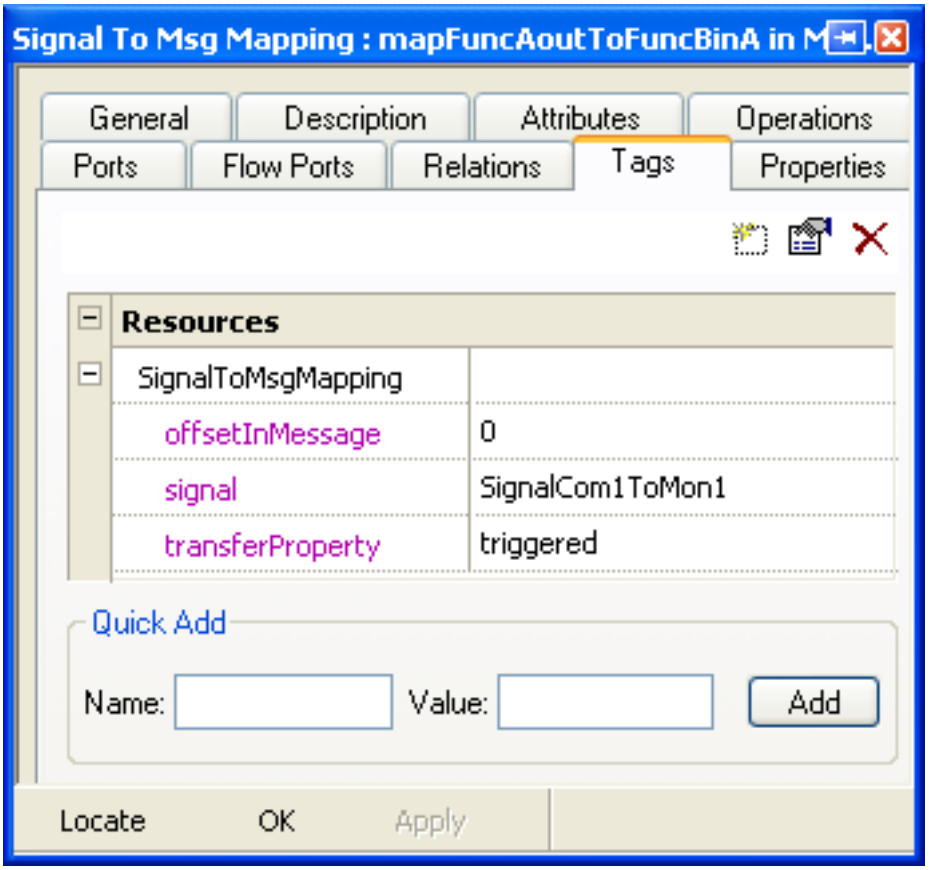

Figure 2.19.: Technical Perspective: Signal to Message Mapping

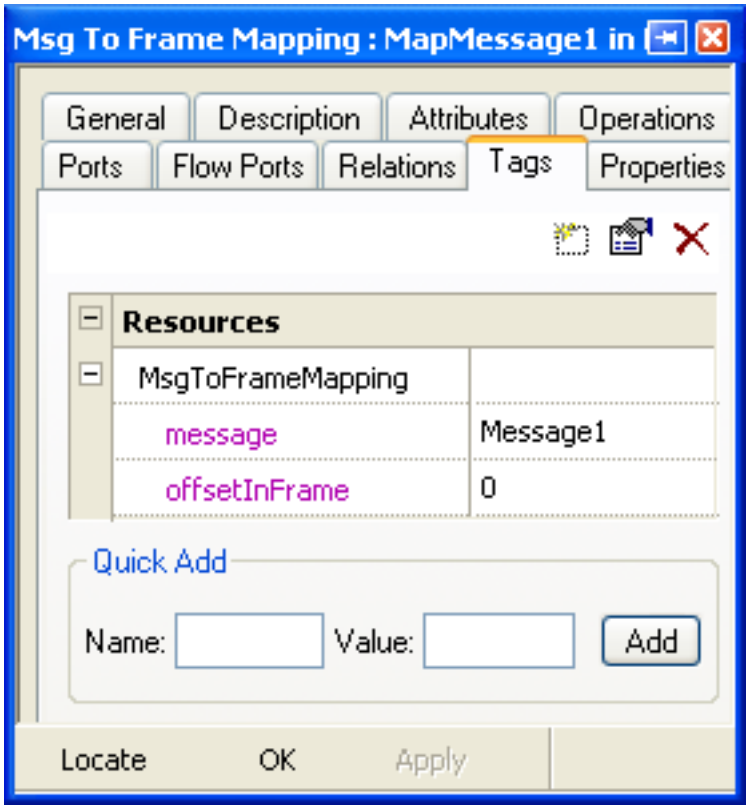

Figure 2.20.: Technical Perspective: Message to Frame Mapping

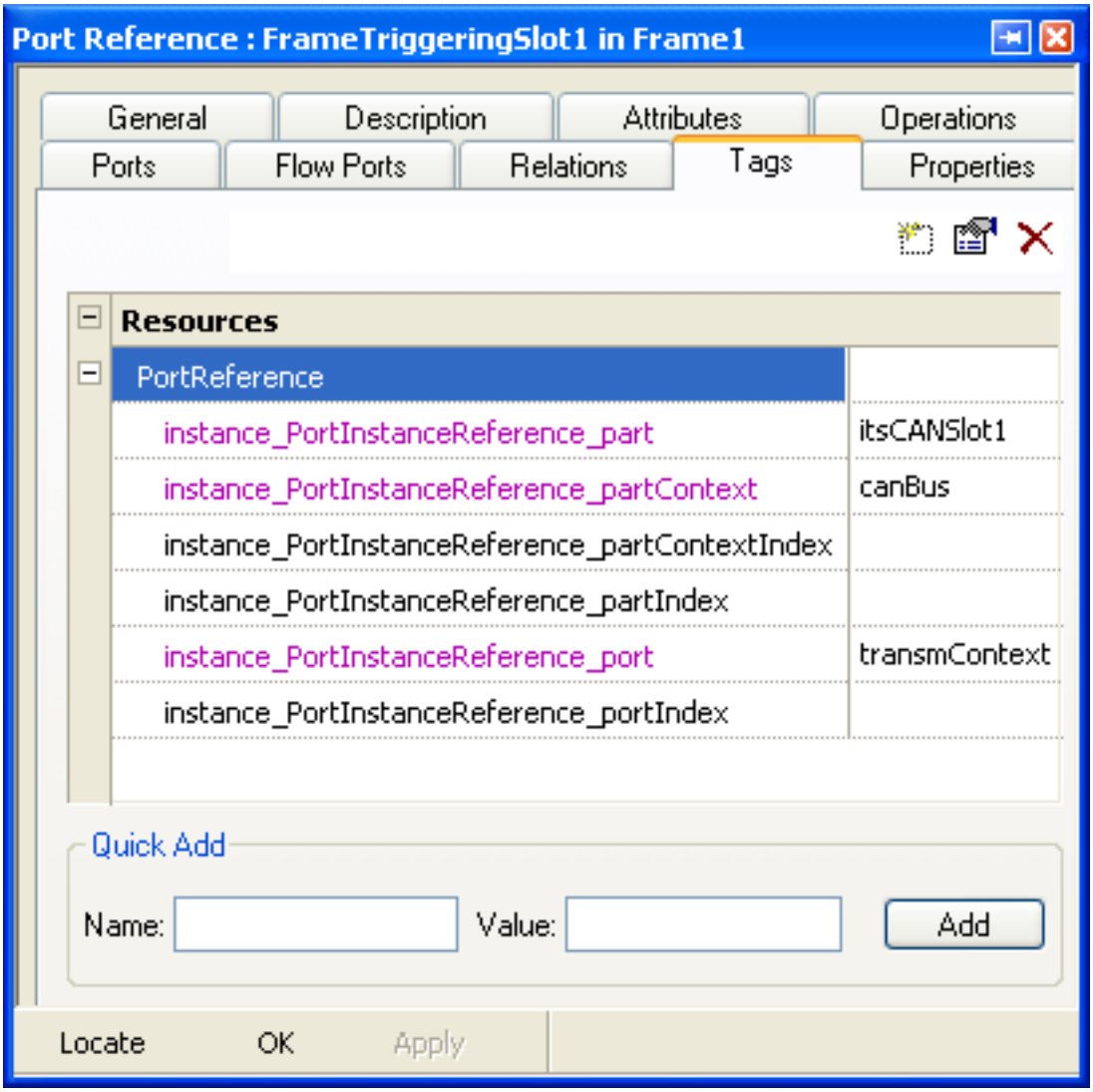

Figure 2.21.: Technical Perspective: Frame to Bus Slot Mapping

- The generation of end-to-end-deadlines (time available starting from activation of first task in a task chain until the last task has to be finished) is not automated yet. This will be changed in the next release.
- each logical component except for the top level component has to have exactly one in port.
  - This limitation stems from the restrictions inside of the current version of OrcaRT. It might by loosened later for newer versions of OrcaRT.
- For communication elements, currently only one-to-one allocation is allowed: One communication component is allocated to exactly one Message which is allocated to exactly one Frame.
  - On the one hand there are some restrictions in OrcaRT, but mainly the converter to OrcaRT has to be slightly extended.
- The current allocation and grouping mechanisms for communication elements have to be reassessed and possibly changed to be more compliant to the concepts of contract-based design. This is presumably out-of-scope of SPES 2020 and will be scheduled to later projects.

There are some conceptual decisions in the current version of our software:

- We assume a simple allocation relations between elements in the logical perspective to elements in the technical perspective.
  - Allocation relations in general can be quite complex (particularly as complex as required by the application). On the scheduling abstraction level however we propose to limit the allocation complexity to keep the models understandable and compatible to our OrcaRT concepts.

2.8. Conclusion

This document has shown the required concepts to model an embedded systems in a way that it can be translated to and analyzed with the current version of our realtime analysis tool OrcaRT. The underlying meta-model is subject to changes therefore in later releases of our tools, this document might be partially outdated.

A. Preliminaries

Before you can start to model with the SPES Architecture UML Profile you need the right tool environment. Currently the system requirements are the following:

- Operating System: Windows XP (Vista, or Windows 7 might work as well)
- Modeling Tool: IBM Rational Rhapsody (tested version: 7.5.2), including SysML support (e.g. installation mode *IBM Rational Rhapsody Developer*)
- SPES Architecture UML Profile
  `http://spes2020.offis.de/profile/spesmm-1.0.0.zip`
- Sun Java Runtime Environment or Java Developers Kit (32 Bit) (latest version. as of writing this is 1.6.0, Update 23)
  `http://www.java.com`
  `http://java.sun.com`
- Eclipse Helios, 32 Bit (v3.6.1)
  `http://www.eclipse.org`
- Open System Platform from OFFIS Eclipse Update Site
  `http://spes2020.offis.de/eclipse/3.6/`
- OrcaRT Tool Suite
  `http://spes2020.offis.de/software`

Bibliography

[1] Alexander Metzner. *Effizienter Entwurf verteilter eingebetteter Echtzeit-Systeme*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2007. Available online at `http://oops.uni-oldenburg.de/volltexte/2007/64/`.

[2] OFFIS. SPES2020 Architecture Modeling. Technical report, OFFIS, 2010.

[3] SPES Partners. SPES 2020 Wiki, page of ZP-AP-3. `https://spes.informatik.tu-muenchen.de/index.php/ZP-AP-3`.

[4] Kenneth William Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, 1996.