TNO innovation for life

iModulaR 2023

Modeling and analyzing hardware-software interfaces with ComMA

TNO Public › TNO 2024 R10720
15 April 2024

TNO-ESI
www.tno.nl
+31 88 866 50 00
info@tno.nl

TNO 2024 R10720 – 15 April 2024

Modeling and analyzing hardware-software interfaces with ComMA

iModulaR 2023

Author(s)	Jozef Hooman, Wouter Tabingh Suermondt
Classification report	TNO Public
Number of pages	35 (excl. front and back cover)
Number of appendices	4
Sponsor	The research is carried out as part of the iModulaR program under the responsibility of TNO-ESI with Philips as the carrying industrial partner. The iModulaR research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

Summary

Technical diversity of a complex system can be reduced using modularity and interface management. At the multi-disciplinary system level, this can be modelled using MBSE (Model-Based System Engineering) where typically the focus is on the system structure with the components and their interfaces. At the mono-disciplinary level this is further refined where each discipline uses its own tools and techniques to include more behavioural aspects. Frequently this leads to problems during the integration phase, especially for interfaces where multiple disciples are involved.

In this report, we address the integration problems of hardware-software interfaces. We investigate whether the ComMA methodology helps to clarify hardware-software interfaces, to remove ambiguities, and to detect issues with such interfaces early during development. As a case study, ComMA is applied to the interfaces of a component of a Philips MR scanner. For reasons of confidentiality, the interfaces of the component are described in an abstract way as a client interface that needs sensor data and a sensor interface to obtain data. The implementation of the component has been instrumented to obtain logs of component execution during a few experiments. The main focus is on specifying and analysing two aspects: (1) the timing behaviour of the interfaces, and (2) the relation between the two interfaces, expressing that the client receives recent data. We show how the monitoring features of ComMA provide insight in the implementation of the component and lead to questions about the design. The experiments also revealed useful information about recently added ComMA features and wishes for future improvements. Moreover, we briefly address the relation between MBSE and ComMA.

Contents

1 Introduction

The iModulaR project addresses the reduction of technical diversity by improving modularity, with a focus on interface management. The project started in October 2021 and is a cooperation between TNO-ESI and Philips MR. In 2022, the project investigated improvements at the system level by means of the reference architecture and interface management. At this level, the emphasis was on structural improvements. The results have been published in [1].

When proceeding from the system level to a more detailed level where multiple disciplines realize the system components, often many integration problems are observed. Typically, issues occur concerning the behaviour of the components, an aspect that has not been addressed at the system level where the focus is on structure. Issues most frequently occur at interfaces that bridge different engineering silos, such as software engineering and electrical engineering. Since this is often due to unclear and unspecified behaviour such as the allowed sequences of events and assumptions about timing behavior. The question is whether the ComMA approach [2] can help to avoid these integration problems.

With ComMA (Component Modeling and Analysis), digital interfaces of components can be specified including a state machine of the interface protocol, constraints on timing and data, and constraints on the relations between interfaces. Based on a trace of observed component behaviour, the ComMA tooling can monitor whether this trace conforms to the specification. For timing constraints, the observed data is recorded and presented in a graphical form. More information about ComMA, related work, and industrial applications can be found in scientific publications [3, 4, 5] and other articles [6, 7].

As a case study, the modelling of a hardware-software interface has been studied. For reasons of confidentiality, we describe the application in a very generic way and call it a sensor server. This server provides a basic interface to client applications using the sensor and hides the details of the specific server used. It this way it is possible to change the physical sensor to other versions or devices of other vendors without affecting client applications using the sensor. Besides state machine behaviour, timing aspects are very relevant for this interface. Moreover, it is important that a client application receives recent sensor data.

First the relevant components and their interfaces specified using Model-Based System Engineering (MBSE) techniques where we investigate the possibility to express more detailed interface behaviour. Next a manual transformation to ComMA has been made.

To enable ComMA-based monitoring, the server implementation has been instrumented with logging statements to obtain traces of a few experiments with the system. Analysis of these traces using the monitoring functionality of ComMA led to interesting insights and questions about the timing behaviour of the server and the data obtained by a client application. In a meeting with a number of hardware and software engineers it was confirmed that it is important to clarify these questions early in the development phases to avoid integration problems later. The potential of ComMA to prevent these issues was recognized.

This report is structured as follows:

- Chapter 2 presents an MBSE model of the sensor server case study.
- Chapter 3 describes the ComMA models of the case study.

- Chapter 4 shows the possibilities to simulate ComMA models and generate documentation.
- Chapter 5 describes the used setup and two of the experiments that have been performed.
- Chapter 6 summarizes the results of monitoring the traces obtained in both experiments, with a focus on timing constraints and component constraints.
- Chapter 7 illustrates the use of monitoring to detect violations of the specified state machine behaviour.
- Chapter 8 contains concluding remarks.
- Details can be found in the appendices.

2 System modelling

For the sensor server case study we used the SysML language [8] to construct an MBSE model. In this case study, there is a SensorClient component that needs certain censor data. The SensorServer component provides an abstraction of the particular sensor used. In this way, changes of the sensor are transparent for the client. A block diagram of this solution is shown in Figure 2.1.

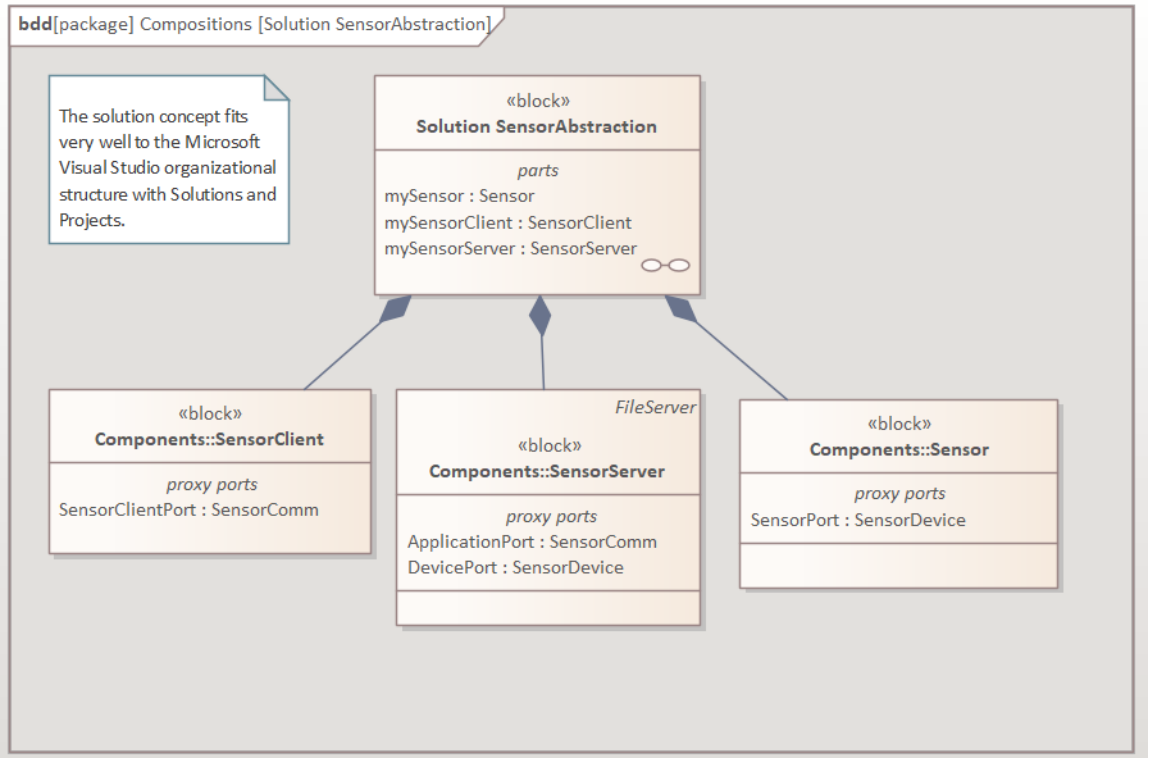

Figure 2.1: Block definition diagram of the sensor abstraction solution

Figure 2.2 depicts how instances of the components are connected.

Observe that the components have ports with a name and an interface type, here called SensorComm and SensorDevice.

The commands of these interfaces are defined as shown in Figure 2.3.

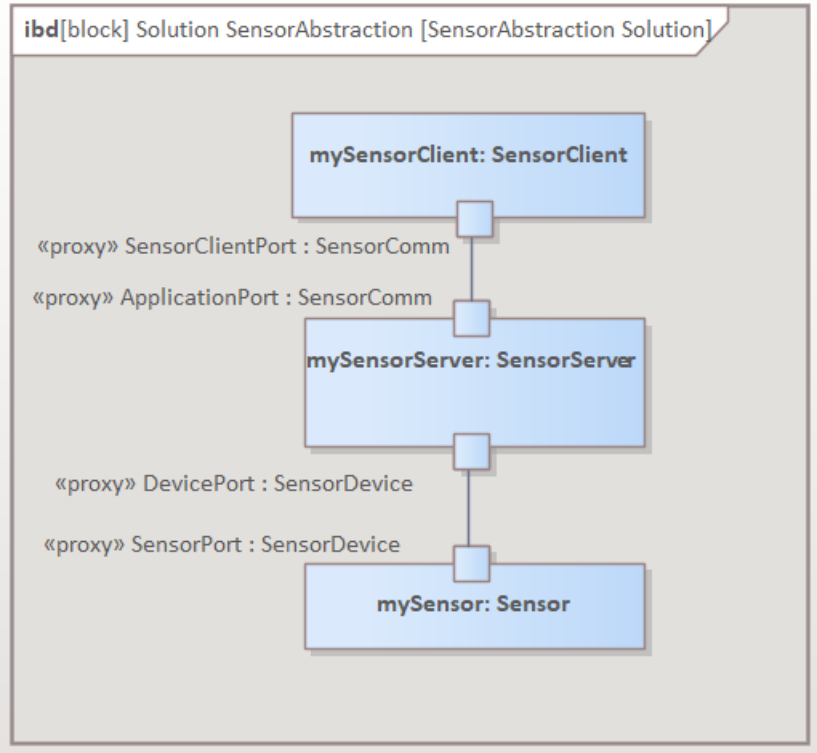

Figure 2.2: Instances of the components and their connection

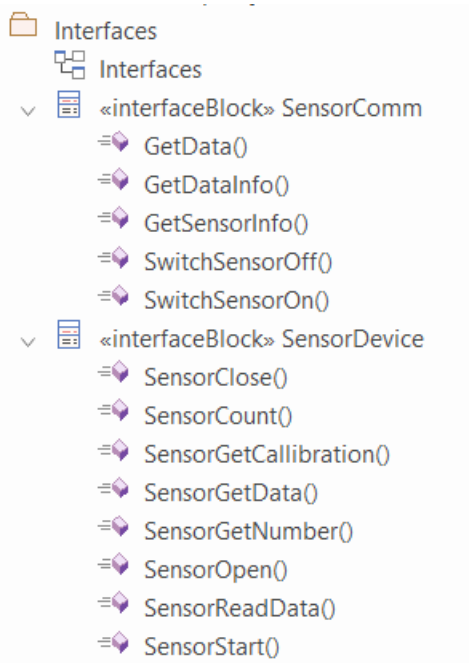

Figure 2.3: Definition of interfaces

3 ComMA models

In this chapter, the definition of the interfaces is refined using the ComMA approach. Interfaces SensorCom and SensorDevice are modelled in Sections 3.1 and 3.2, respectively. Section 3.3 contains the model of the SensorServer component.

3.1 ComMA model SensorComm interface

The signature of the SensorComm interface is defined in file SensorComm.signature, see Figure 3.1. The allowed order of the commands is specified in file SensorComm.interface by means of a state machine as shown in Figure 3.2.

```
signature SensorComm

commands
void GetSensorInfo
void GetDataInfo
real GetData
void SwitchSensorOn
void SwitchSensorOff
```

Figure 3.1: The signature of interface SensorComm

```
interface SensorComm version "0.1"

machine SensorComStateMachine {
    in all states {
        transition trigger: GetSensorInfo
            do:
            reply
        transition trigger: GetDataInfo
            do:
            reply
    }

    initial state SensorOff {
        transition trigger: SwitchSensorOn
            do:
            reply
            next state: SensorOn
    }

    state SensorOn {
        transition trigger: GetData
            do:
            reply(*)
            next state: SensorOn

        transition trigger: SwitchSensorOff
            do:
            reply
            next state: SensorOff
    }
}
```

Figure 3.2: The state machine of interface SensorComm

The state machine describes the interface from the viewpoint of the server, that is, it specifies the response of the server to calls of the client. These client calls are the triggers of the transitions. Statement *Reply(*)* denotes that the return value is not specified by the state machine.

Also two time constraints, called GetData_reply and GetData_GetData, are specified as shown in Figure 3.3, expressing that after a GetData command we expect a reply to this command within 100 ms and a next call within 100 ms.

```
timing constraints
GetData_reply
command GetData - [ .. 100.0 ms ] -> reply to command GetData

GetData_GetData
command GetData - [ .. 100.0 ms ] -> command GetData
```

Figure 3.3: Time constraints of interface SensorComm

3.2 ComMA model SensorDevice interface

In the signature of the SensorDevice interface we ignored the parameters and only model the return value of the abstract command SensorGetData. This command is an abstraction of three commands as explained in Chapter 5. The signature is shown in Figure 3.4.

```
signature SensorDevice

commands
void SensorCount
void SensorOpen
void SensorStart
void SensorGetCallibration
void SensorGetNumber
void SensorReadData
real SensorGetData
void SensorClose
```

Figure 3.4: The signature of interface SensorDevice

The state machine of this interface can be found in Appendix A. It is based on the events that were observed during the experiments described in Chapter 5. Moreover, two time constraints with names SensorGetData_reply and SensorGetData_SensorGetData on command SensorGetData have been defined, as shown in Figure 3.5.

```
timing constraints
SensorGetData_reply
command SensorGetData - [ .. 100.0 ms ] -> reply to command SensorGetData

SensorGetData_SensorGetData
command SensorGetData - [ .. 100.0 ms ] -> command SensorGetData
```

Figure 3.5: Time constraints of interface SensorDevice

3.3 ComMA model SensorServer Component

The ComMA specification of the SensorServer component, as shown in Figure 3.6, expresses that it provides interface SensorCom and requires interface SensorDevice.

```
component SensorServer

provided port SensorComm ApplicationPort
required port SensorDevice DevicePort
```

Figure 3.6: ComMA definition of the SensorServer component

For the SensorServer component, time constraints can be defined, e.g., between events of the two interfaces. For the moment, the focus is on formulating a data constraint called LastData, shown in Figure 3.7, which expresses that GetData uses the data that was last obtained by SensorGetData. This constraint describes a pattern of events and then requires certain reply values to be equal.

```
data constraints
variables
real t1
real t2

LastData
DevicePort::reply(t1) to command SensorGetData;
no [DevicePort::reply to command SensorGetData]
until ApplicationPort::reply(t2) to command GetData where t1 == t2
observe diff := t1-t2  // when positive older time stamp is used
```

Figure 3.7: Constraint LastData of the SensorServer component

The “observe” part declares an expression; the values of this expression will be recorded during the check of the property to obtain a graphical representation of the differences between the values.

4 Simulation and document generation

In this chapter we describe the simulation of a ComMA model (Section 4.1) and document generation (Section 4.2).

4.1 Simulation

Based on a ComMA model, a simulation can be generated, given a so-called parameters file which describes the values of input parameters for provided interfaces and the values of replies for required interfaces. For the SensorServer, this leads to the user interface of the simulation shown in Figure 4.1. The left part shows the possible actions in the current state. The simulation steps are shown on the right.

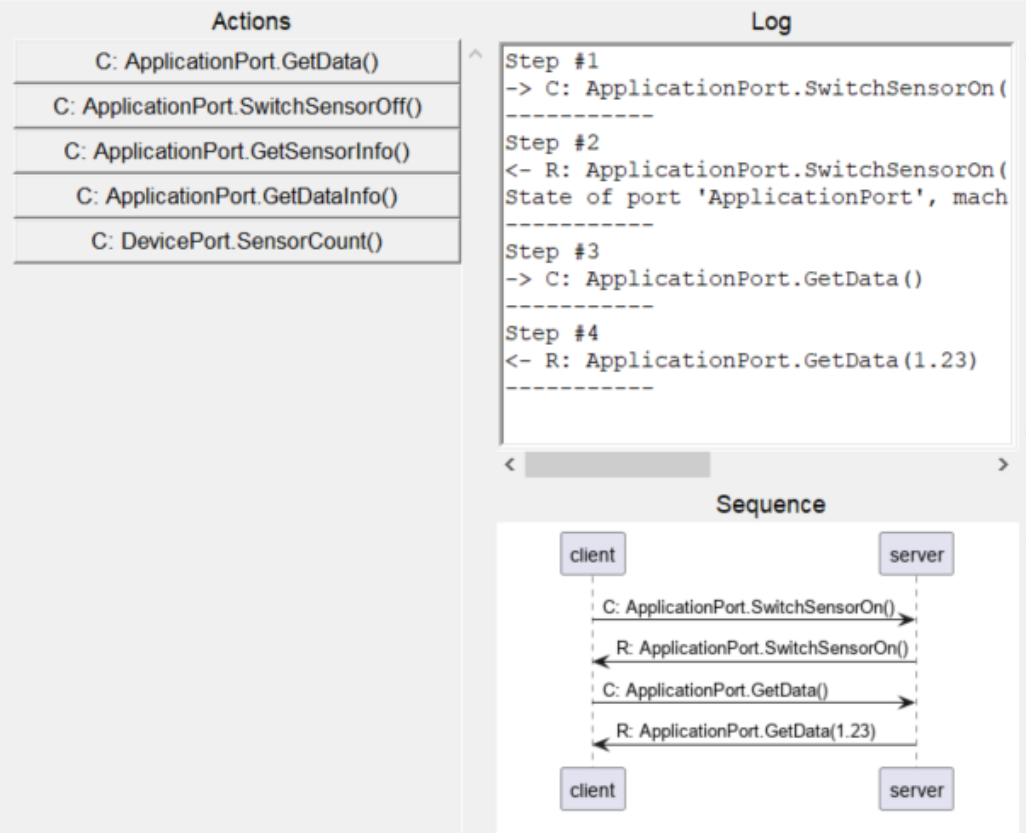

Figure 4.1: Simulation of the SensorServer component

Observe that in this simulation we get a reply to command GetData without having done any call on the DevicePort to the SensorDevice interface. To avoid this scenario, we require that a reply to GetData can only occur after a reply to command SensorGetData has been

obtained. This is expressed by the functional constraint of the SensorServer component shown in Figure 4.2.

```
FirstGetData {
    use events
    command ApplicationPort::GetData
    ApplicationPort::reply to command GetData
    command DevicePort::SensorGetData
    DevicePort::reply to command SensorGetData

    initial state NoData {

        transition
            do: DevicePort::SensorGetData
            next state: NoData

        transition trigger: DevicePort::reply(real v) to command SensorGetData
            next state: DataAvailable

        transition trigger: ApplicationPort::GetData
            next state: NoData
    }

    state DataAvailable {

        transition
            do: DevicePort::SensorGetData
            next state: DataAvailable

        transition trigger: DevicePort::reply(real v) to command SensorGetData
            next state: DataAvailable

        transition trigger: ApplicationPort::GetData
            next state: DataAvailable

        transition
            do: ApplicationPort::reply(*) to command GetData
            next state: DataAvailable
    }
}
```

Figure 4.2: Functional constraint of the SensorServer component to obtain sensor data first

Then the simulation shows the expected behaviour.

4.2 Document Generation

For each ComMA interface a Word document can be generated based on a template and a few input parameters. The specification of the generation task for interface SensorDevice is shown in Figure 4.3.

This leads to Word document; the first part of the title page is shown in Figure 4.4.

The document contains all details of the model such as the list of commands and the state machine which is represented in tabular form and as a figure, see the snapshot in Figure 4.5.

Timing constraints are represented as sequence diagrams, see Figure 4.6.

```
Generate Documentations {
    documentation for interface SensorDevice {
        template = "Template.docx"
        targetFile = "SensorDeviceInterface.docx"
        author = "My Name"
        role = "R&D: Designer"
    }
}
```

Figure 4.3: Specification of document generation

Interface Design Specification

SensorDevice

Author: My Name
R&D: Designer

Figure 4.4: First part of the title page of the generated document

SensorReadData()		ObtainedSerialNum	reply
SensorGetData()		ObtainedSerialNum	reply
State ObtainedCallibration			
Event	**Guard**	**Target State**	**Actions**
SensorGetNumber()		ObtainedSerialNum	reply
State ObtainedSerialNum			
Event	**Guard**	**Target State**	**Actions**
SensorReadData()		ObtainedSerialNum	reply
SensorGetData()		ObtainedSerialNum	reply
SensorClose()		PoweredOn	reply

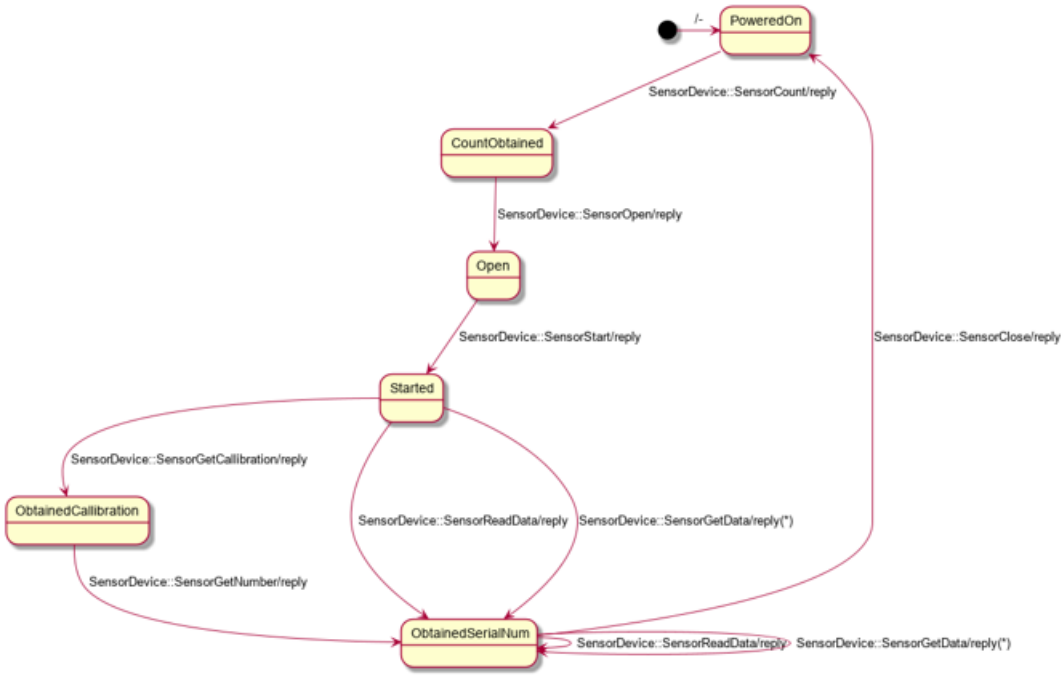

Figure 4.5: State machine representations in the generated document

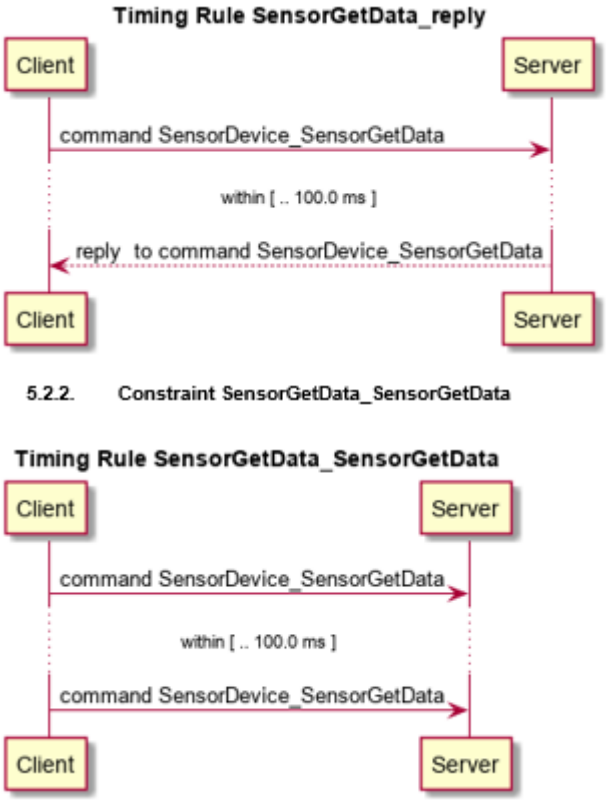

Figure 4.6: Sequence diagram representation of timing constraints in the generated document

5 Experiments performed

A number of experiments have been done where a client performs calls to interface SensorComm of the SensorServer and the SensorServer performs calls to the SensorDevice interface. We present the results of 2 runs: run2 and run4.

- Run2: start with lowest possible load, starting and stopping a client application a few times. With this client application off there is 5% CPU usage, with the application on the CPU usage is 26%. The load was measured by the Microsoft tool Resource Monitor which is standard available on any MS Windows computer
- Run 4: first all applications are off, next all applications are on. Additional CPU stress is created by applying the tool CPUSTRESS, see Figure 5.1, to be downloaded from the Microsoft Sysinternals. We created such an amount that the cpuload is almost 100%.

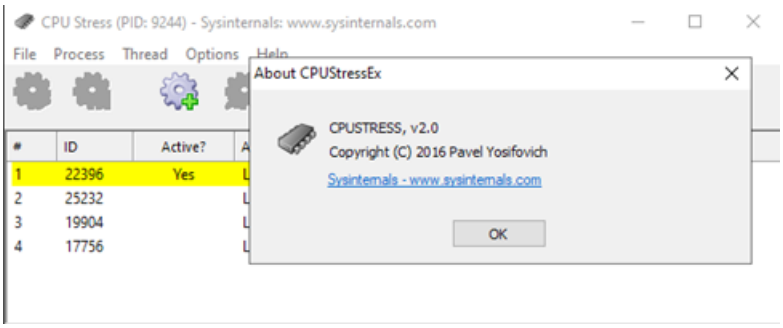

Figure 5.1: Screenshot of the Microsoft CPUSTRESS tool

We instrumented the source code of the SensorServer to log all calls and replies of the two SensorServer interfaces. For each interface, the information is written to a separate output file. After the run, the output files are merged into one file where all timestamps are placed in chronological order.

We have abstracted from the details of getting data from the device by combining a number of calls to the device into a single ComMA command called GetSensorData. This command represents three sensor calls to obtain subsamples and construct a combined sensor data sample.

For both runs a trace has been obtained in the ComMA JSON format. These traces are monitored with ComMA, which includes

- Checking if the traces conform to the interface models, i.e. the state machine and the timing constraints
- Checking the component constraints

For both runs there are no issues with the state machines, so the description of the monitoring results in the next two chapters concentrates on the constraints. The reporting of violations of state machine behaviour is explained in Chapter 7.

6 Monitoring results

In this section we briefly summarize the results of monitoring the traces of run2 and run4 in Sections 6.1 and 6.2, respectively. For each trace, we list the results of the ComMA monitoring for the two interfaces and the component constraint.

All details can be found in Appendices B and C.

6.1 Results of Run2

Run2 is the initial experiment to set a baseline for further referencing. We executed the setup with minimal CPU load. We started the SensorServer and afterwards the client application. We noticed that the reported data rate is between 6.6 and 7.1 data samples per second. The full run was completely error free.

6.1.1 Interface SensorComm

Constraint GetData_reply is always satisfied. There are 6 warning that constraint GetData_GetData is violated; observe that the graph of Figure B.3 shows a few very large values. Zooming in shows an alternation of high and low values, see Figure B.4. The large variation in the timing of the last constraint is also visible in the generated statistics, see Figure 6.1.

	GetData_reply	GetData_GetData
Times invoked	7400	7399
Mean value	4.913243243243243	160.61548857953778
Variance	2.3889597516435996	533666.0520428385
Std deviation	1.5456260063946905	730.5245047517835
Min	1	8
Lower bound	-	-
Median	5	236
Max	18	50502
Higher bound	100	100

Figure 6.1: Run2: statistics time constraints SensorComm interface

6.1.2 Interface SensorDevice

Property SensorGetData_reply is always satisfied; there are a few violations of property SensorGetData_SensorGetData. The statistics can be found in Figure 6.2. Observe that the number of SensorGetData calls is much higher than the number of GetData calls.

	SensorGetData_reply	SensorGetData_SensorGetData
Times invoked	36426	36425
Mean value	0.03343765442266513	33.37597803706246
Variance	0.0323195776893784	43.28647853898991
Std deviation	0.1797764658941164	6.579246046393911
Min	0	0
Lower bound	-	-
Median	0	32
Max	1	363
Higher bound	100	100

Figure 6.2: Run2: statistics time constraints SensorDevice interface

6.1.3 Component constraint

Component constraint LastData is violated 32 times. Looking at the violations, this is due to a SensorGetData call and reply between the GetData call and its reply:

command SensorGetData

reply(v1) to command SensorGetData

command GetData

command SensorGetData

reply(v2) to command SensorGetData

reply(v1) to command GetData

In this case, the trace shows that the last value before the call is returned and not the last value before the reply. In Appendix D we will discuss alternative formulations of the constraint.

6.2 Results of Run4

The experiment of Run4 is similar to Run2, except that it is under full load conditions; all client applications are enabled and CPUSTRESS adds about 82% of additional load. The reported data rate is now recorded to less than 1.5 data samples per second. Furthermore we noticed timeout errors on the USB interface (libusb). The SensorServer also reports “device not available” warnings, but was always able to reconnect.

6.2.1 Interface SensorComm

Constraint GetData_reply is always satisfied, but shows a few higher peaks compared to Run2. Constraint GetData_GetData is violated very often; the graph of Figure 6.3 shows a period with very large values (probably the period where the CPU stress has been added).

Also here zooming in shows an alternation of high and low values, see Figure 6.4.

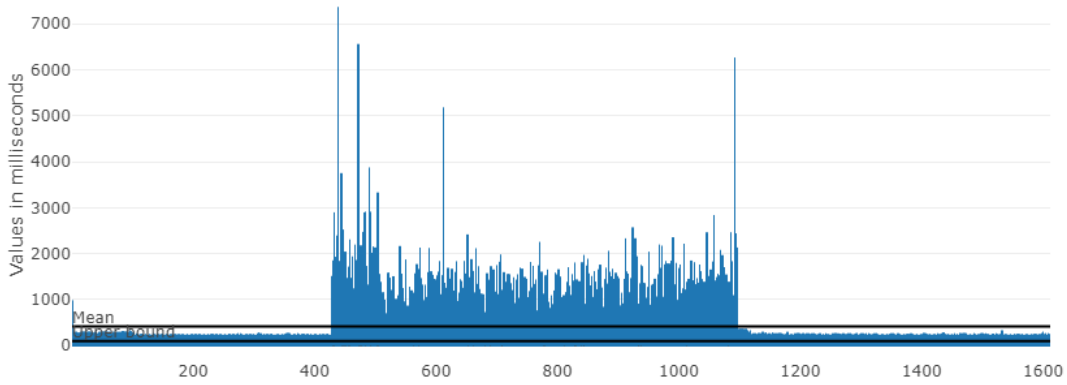

Figure 6.3: Run4: time between two consecutive GetData calls

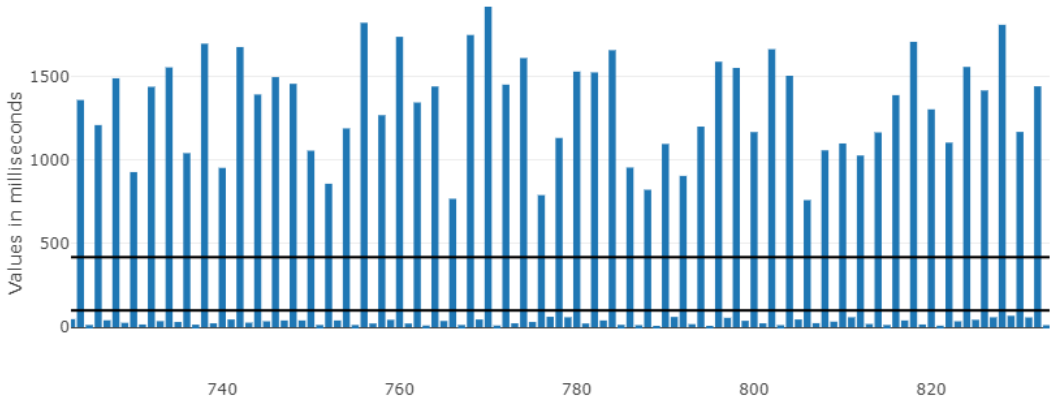

Figure 6.4: Run4: detailed view of time between consecutive GetData calls

The statistics are shown in Figure 6.5. Note that the mean value of GetData_GetData (approximately 418) is much higher than the value for Run2 (approximately 160).

	GetData_reply	GetData_GetData
Times invoked	1612	1611
Mean value	5.2673697270471465	418.6635630043451
Variance	21.020945190845477	458868.4976108936
Std deviation	4.584860433082503	677.398330091604
Min	0	6
Lower bound	-	-
Median	5	236
Max	71	7369
Higher bound	100	100

Figure 6.5: Run4: statistics time constraints SensorComm interface

6.2.2 Interface SensorDevice

There are a number of violations of constraint SensorGetData_reply and also of property SensorGetData_SensorGetData.

Figure 6.6 shows the statistics. Note again that there many more calls of SensorGetData than GetData calls.

	SensorGetData_reply	SensorGetData_SensorGetData
Times invoked	20721	20720
Mean value	0.6756430674195261	34.322586872586875
Variance	2450.8833066497386	3101.8248952386193
Std deviation	49.50639662356511	55.694029260223395
Min	0	0
Lower bound	-	-
Median	0	32
Max	6482	6482
Higher bound	100	100

Figure 6.6: Run4: statistics time constraints SensorDevice interface

6.2.2.1 Component constraint

Component constraint LastData is violated 88 times. Similar to Run2, we observe SensorGetData calls and replies between the GetData call and its reply.

7 Detecting violations of state machine behaviour

The state machines described in Chapter 3 have been constructed such that they match the observed behaviour in the traces. To illustrate the detection of state machine violations, we change the state machine of the SensorDevice interface (see Figure A.1) by commenting out a transition as shown in Figure 7.1.

```
    state ObtainedSerialNum {
        transition trigger: SensorReadData
            do:
            reply
            next state: ObtainedSerialNum
//      transition trigger: SensorGetData
//          do:
//          reply(*)
//          next state: ObtainedSerialNum
        transition trigger: SensorClose
            do:
            reply
            next state: PoweredOn
    }
```

Figure 7.1: Changing the SensorDevice interface

Monitoring the trace of run2 then shows an interface monitoring error since the trace contains a SensorGetData event in state ObtainedSerialNum which is not allowed by the modified state machine. In the monitoring dashboard, the error is shown by a sequence diagram, as depicted in Figure 7.2.

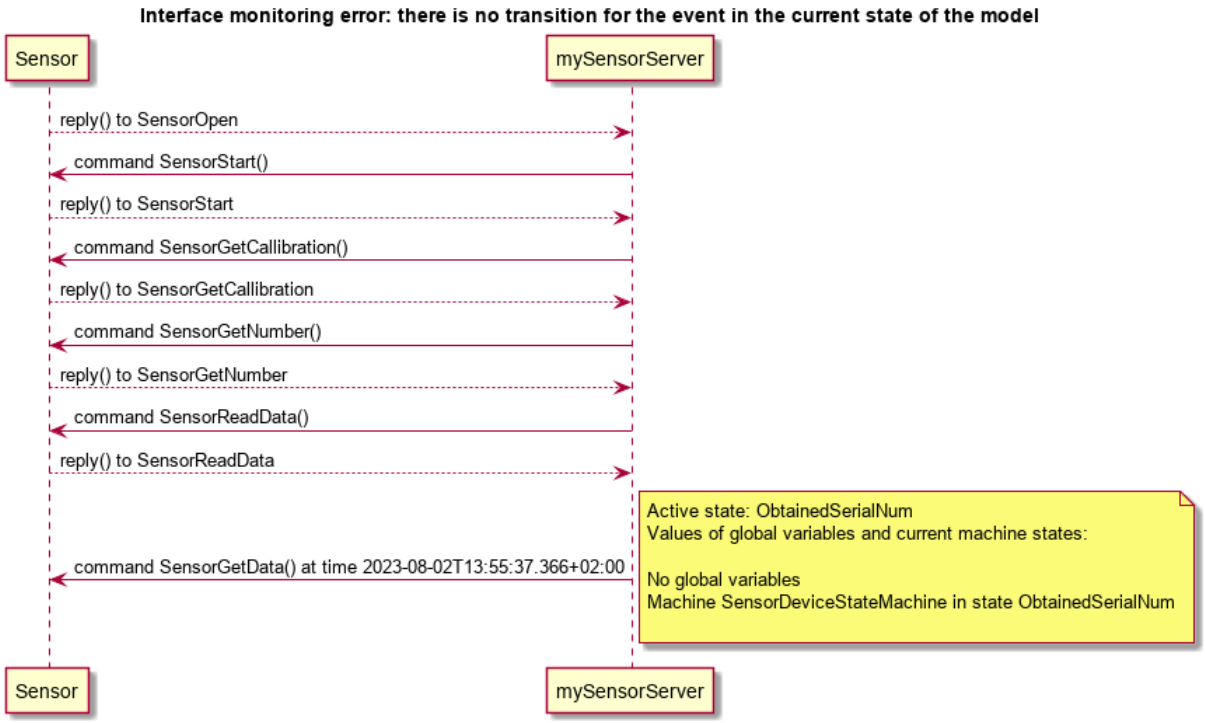

Figure 7.2: Monitoring detects an event in the trace that does conform to the state machine

8 Concluding remarks

Summarizing the iModulaR research, we observe the value of a three layer approach to relate software, electrical and mechanical component modelling to the MBSE system level models. This layering is depicted in Figure 8.1.

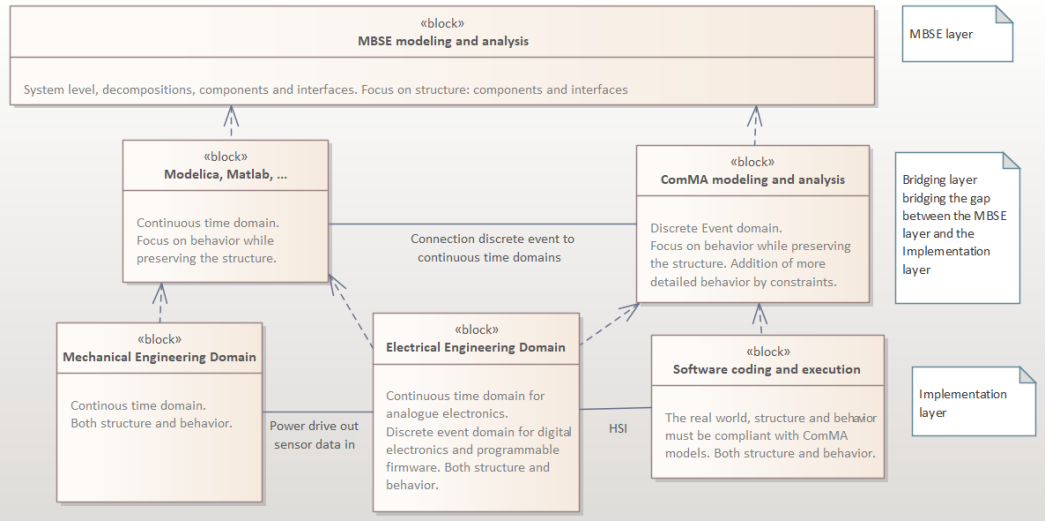

Figure 8.1: Three layer approach in component specification

The lowest, implementation, layer always exists and can act as a reference. The implementation contains structure while the behaviour can be observed when executing it. The implementation layer is connected to the MBSE layer by means of a bridging layer.

We distinguish two major domains: the continuous time domain and the discrete event domain. The continuous time domain is mostly analogue while the discrete event domain is digital. Both domains have their own formalisms and their own methods and tools. ComMA is an example from the discrete event domain while Modelica is an example from the continuous time domain.

In general, we see that both the MBSE as well as the bridging layer share the capabilities to model exactly the same structure. Hence, it is possible to generate the one from the other and vice versa. With respect to the behaviour, however, it is unclear how the bridging and MBSE layers relate.

For example, ComMA has a rich language to express many behavioural details. We have seen similar capabilities with Modelica. Currently, it is not clear how this connects to the modelling of behaviour in SysML. In practice, system models are often incomplete and focus mostly on the structural part because their purpose is often to support managerial and organization aspects of the system reasoning. On the other hand, the modelling methods in the bridging layer can be tightly connected to the implementation due to their rich modelling expressiveness and analysis of behaviour. For example ComMA can do detailed behavioural analysis by monitoring traces of code execution. We expect that Modelica can do similar analysis in the continuous time domains. However, further research on these topics is required.

We extensively studied the monitoring capabilities of ComMA in the sensor server case study. It raised a number of questions about the implementation of the component. A few examples:

- The number of calls to the device to retrieve data is much higher than the number of GetData calls (a factor 5 in Run2 and a factor 12 in Run4). This implies that most retrieved data is not used and, given the long time between GetData calls (especially in Run4), the question is whether this can be avoided.
- Why is there an alternation in the time between GetData calls, see for instance Figure 6.4.
- Is possible to specify more precisely which sensor data should be used for a GetData call? How recent should it be?
- Observing run4, it seems the GetData calls are much more delayed than calls to the device. Can this be explained? Would also be interesting to understand the reason for the connection loss in run4.

The case study was a good test of new ComMA features such as the improved notation for component constraints and the possibilities to simulate a component. Logging and monitoring long traces in JSON format showed a few issues with the ISO 8601 format for time stamps. The use of generic component constraints to express the relation between values and time stamps of events of different interfaces revealed a bug in monitoring such constraints. We also observed that the formulation of such constraints may have a large impact on time needed for monitoring.

The case study lead to a few wishes for improvements of the ComMA tooling. This includes the generation of a graph for generic constraints, the improvement of the sequence diagram for component simulation, and the generation of a diagram for components.

The relation between MBSE and more detailed techniques like ComMA requires further study. The main question is how to transition gradually from a structural descriptions at system level to more detailed models that include behaviour. Since ComMA is suitable for digital interfaces only, the study of continuous interfaces is also a relevant topic for future work.

References

[1] A. Vasenev, W. Tabingh Suermondt, A. Behl, and J. Lukkien. “Step-Wise MBSE Introduction into a Company: An Interface-Centric Case Study”. In: *2023 18th Annual System of Systems Engineering Conference (SoSe)*. 2023, pp. 1–6.

[2] *Eclipse CommaSuite*. https://eclipse.dev/comma. 2023.

[3] I. Kurtev, M. Schuts, J. Hooman, and D.-J. Swagerman. “Integrating Interface Modeling and Analysis in an Industrial Setting”. In: *Proc. 5th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD 2017)*. 2017, pp. 345–352.

[4] I. Kurtev and J. Hooman. “Runtime Verification of Compound Components with ComMA”. In: *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*. Ed. by N. Jansen, M. Stoelinga, and P. van den Bos. Vol. 13560. Lecture Notes in Computer Science. Springer, 2022, pp. 382–402.

[5] I. Kurtev, J. Hooman, M. Schuts, and D. v. d. Munnik. “Model based component development and analysis with ComMA”. In: *Science of Computer Programming, special issue on Success Stories in Model-Driven Engineering* 233 (2024), p. 103067.

[6] M. Schuts, D.-J. Swagerman, I. Kurtev, and J. Hooman. “Improving Interface Specifications with ComMA”. In: *Bits & Chips* (14 September 2017).

[7] N. Roos. “ComMA interfaces open the door to reliable high-tech systems”. In: *Bits & Chips* (8 September 2020).

[8] *SysML*. https://www.omgsysml.org/. 2023.

Appendix A
State machine of interface SensorDevice

The state machine which specifies the behaviour of interface SensorDevice is shown in Figure A.1.

```
machine SensorDeviceStateMachine {

    initial state PoweredOn {
        transition trigger: SensorCount
            do:
            reply
            next state: CountObtained
    }
    state CountObtained {
        transition trigger: SensorOpen
            do:
            reply
            next state: Open
    }
    state Open {
        transition trigger: SensorStart
            do:
            reply
            next state: Started
    }
    state Started {
        transition trigger: SensorGetCallibration
            do:
            reply
            next state: ObtainedCallibration
        transition trigger: SensorReadData
            do:
            reply
            next state: ObtainedSerialNum
        transition trigger: SensorGetData
            do:
            reply(*)
            next state: ObtainedSerialNum
    }
```

```
state ObtainedCallibration {

    transition trigger: SensorGetNumber
        do:
        reply
        next state: ObtainedSerialNum
}
state ObtainedSerialNum {
    transition trigger: SensorReadData
        do:
        reply
        next state: ObtainedSerialNum
    transition trigger: SensorGetData
        do:
        reply(*)
        next state: ObtainedSerialNum
    transition trigger: SensorClose
        do:
        reply
        next state: PoweredOn
}
```

Figure A.1: The state machine of interface SensorDevice

Appendix B

Results of Run2

B.1 Interface SensorComm

B.1.1 Timing constraint GetData_reply

Figure B.1 shows that Constraint GetData_reply is always satisfied.

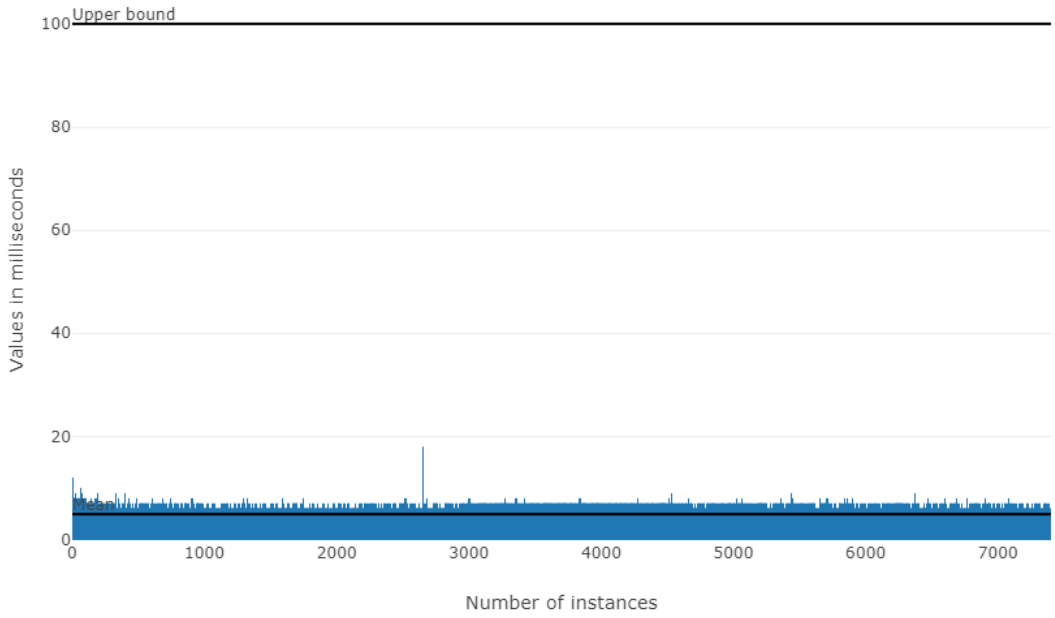

Figure B.1: Run2: time between a GetData call and its reply

Zooming in shows small values, see Figure B.2.

B.1.2 Timing constraint GetData_GetData

There are 6 warning that constraint GetData_GetData is violated; observe that the graph of Figure B.3 shows a few very large values. Zooming in shows an alternation of high and low values, see Figure B.4.

B.2 Interface SensorDevice

B.2.1 Timing constraint SensorGetData_reply

The graph of the SensorGetData_reply property shows small values, see Figure B.5.

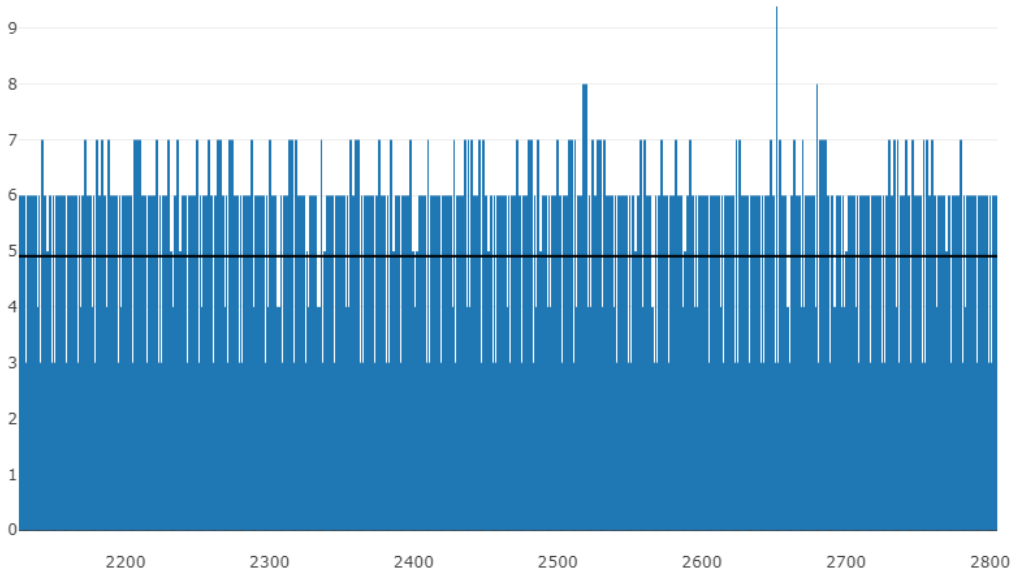

Figure B.2: Run2: detailed view of time between GetData call and its reply

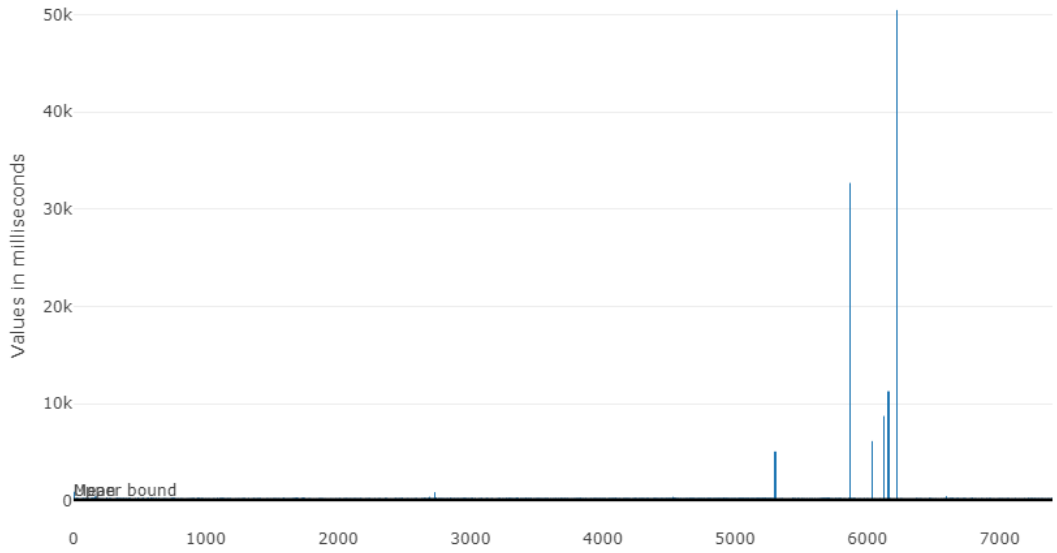

Figure B.3: Run2: time between two consecutive GetData calls

B.2.2 Timing constraint SensorGetData_SensorGetData

The graph of property SensorGetData_SensorGetData in Figure B.6 shows a few violations.

B.3 Component constraint

Figure B.7 of observed differences shows that component constraint LastData is violated 32 times. Note that a positive value indicates that an older message is used.

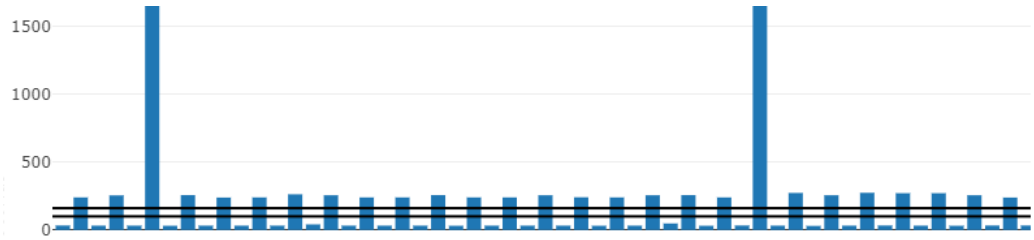

Figure B.4: Run2: detailed view of time between consecutive GetData calls

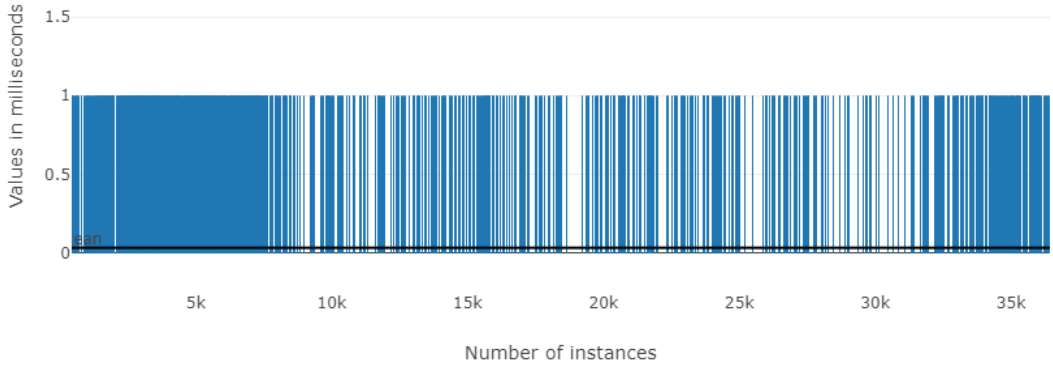

Figure B.5: Run2: time between a SensorGetData call and its reply

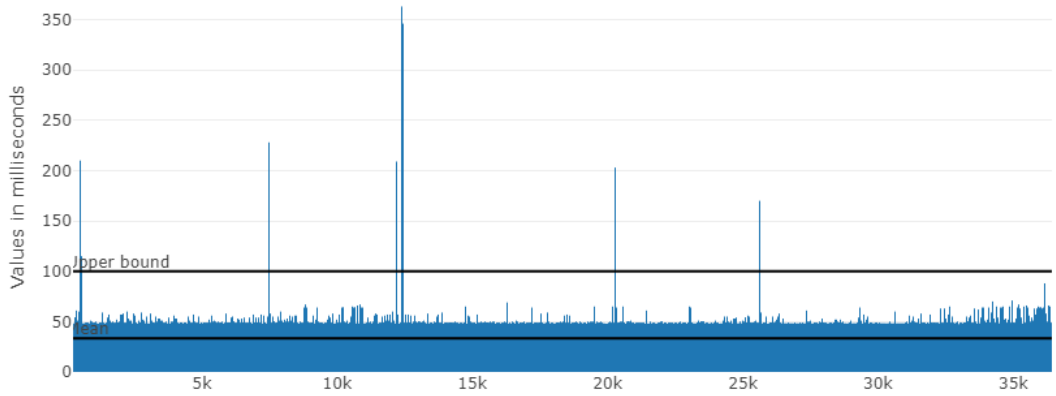

Figure B.6: Run2: time between two consecutive SensorGetData calls

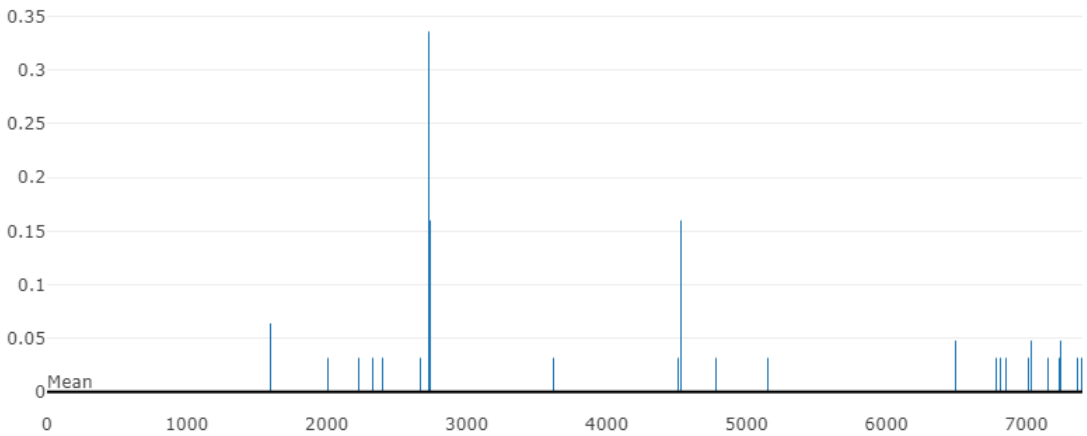

Figure B.7: Run2: time difference between reply of SensorGetData and corresponding reply of GetData

Appendix C

Results of Run4

The experiment of Run4 is similar to Run2, except that it is under full load conditions; all client applications are enabled and CPUSTRESS adds about 82% of additional load.

In de next sections we describe for Run4 the results of the ComMA monitoring for the two interfaces and the component constraint.

C.1 Interface SensorComm

C.1.1 Timing constraint GetData_reply

Constraint GetData_reply is always satisfied, but shows a few higher peaks compared to Run2, as shown in Figure C.1.

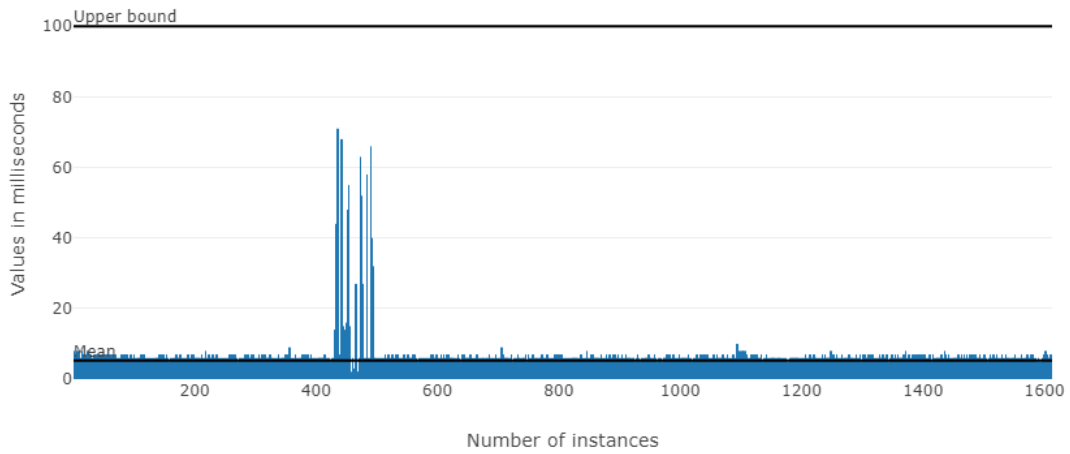

Figure C.1: Run4: time between a GetData call and its reply

C.1.2 Timing constraint GetData_GetData

See Section 6.2 for the graphs of constraint GetData_GetData.

C.2 Interface SensorDevice

C.2.1 Timing constraint SensorGetData_reply

There are a number of violations of constraint SensorGetData_reply, see Figure C.2.

C.2.2 Timing constraint SensorGetData_SensorGetData

Figure C.3 shows a number of violations of property SensorGetData_SensorGetData.

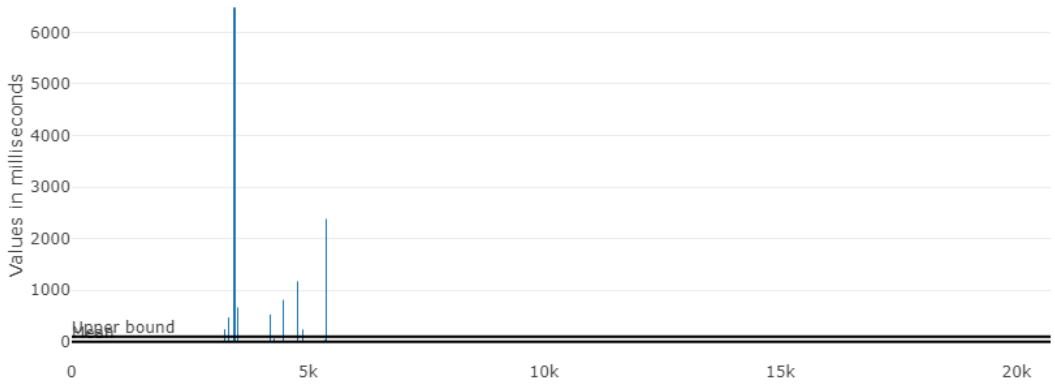

Figure C.2: Run4: time between a SensorGetData call and its reply

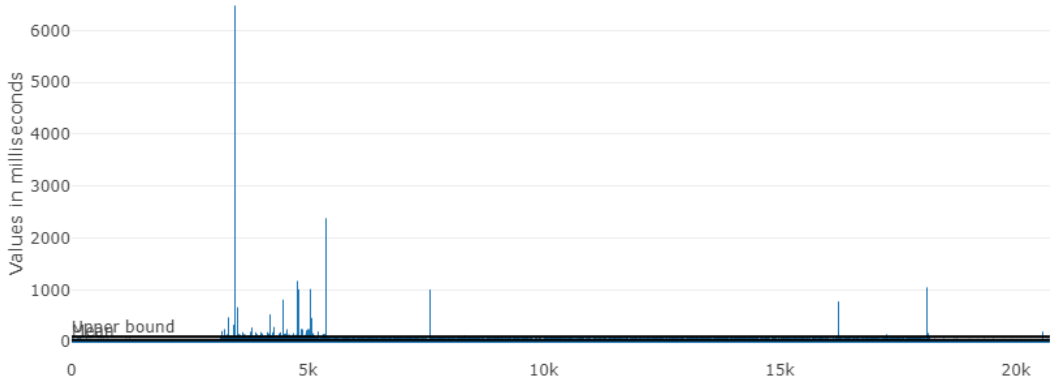

Figure C.3: Run4: time between two consecutive SensorGetData calls

Zooming in shows high and low values, see Figure C.4.

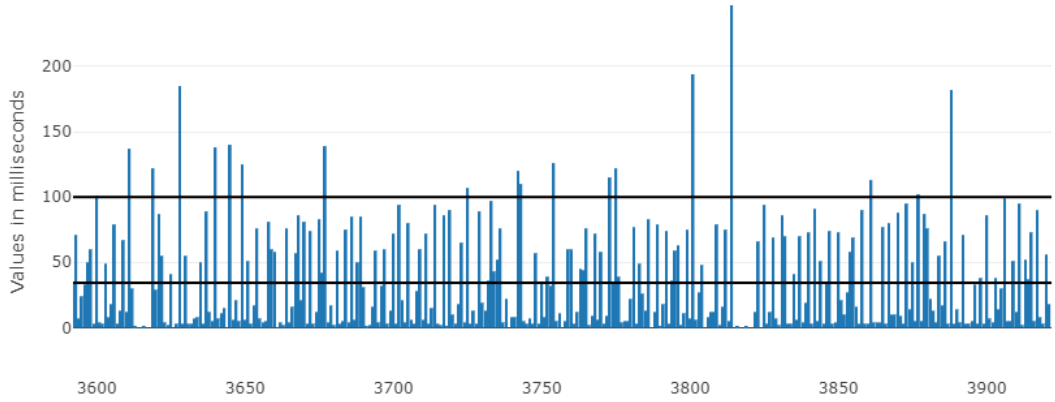

Figure C.4: Run4: detailed view of time between two consecutive SensorGetData calls

C.3 Component constraint

Component constraint LastData is violated 88 times, see the graph of Figure C.5.

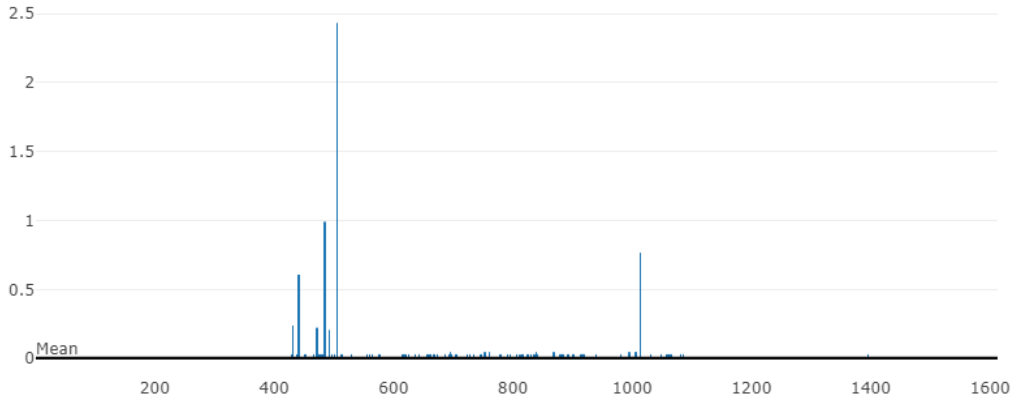

Figure C.5: Run4: time difference between reply of SensorGetData and corresponding reply of GetData

Appendix D
Variations of the component constraint

Given the violations of component constraint LastData, we experimented with a constraint which expresses that the value of the last SensorGetData reply before a GetData call should be used. This is expressed in Figure D.1,

```
LastDataBeforeGetData
DevicePort::reply(t1) to command SensorGetData;
no [DevicePort::reply to command SensorGetData];
command ApplicationPort::GetData;
no [command ApplicationPort::GetData]
until ApplicationPort::reply(t2) to command GetData where t1 == t2
observe diff := t1-t2  // when positive older time stamp is used
```

Figure D.1: Constraint LastDataBeforeGetData of the SensorServer component

This property holds for Run2, but for Run4 we see a violation as shown in Figure D.2.

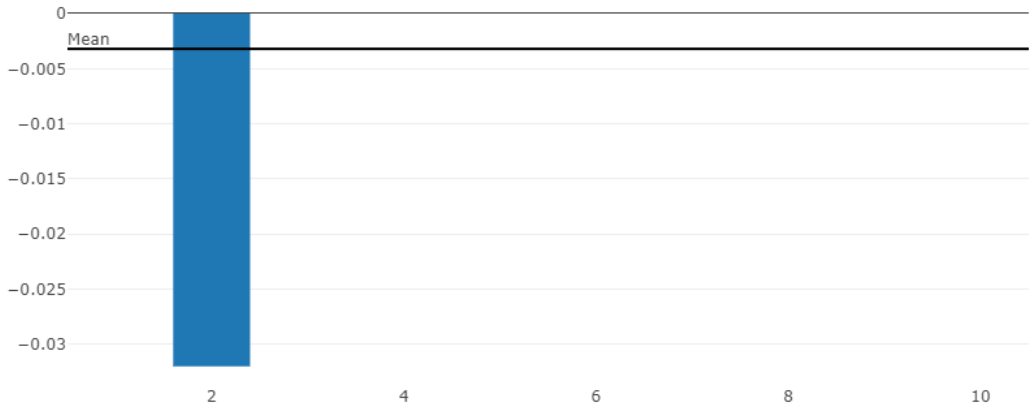

Figure D.2: Run4: time difference corresponding to property LastDataBeforeGetData

Note that the difference is negative, because a value is used that is more recent than the specified one. In this case not the value of the reply to SensorGetData before the GetData is used, but a value after GetData (and before the reply to GetData).

So we might want a combination of the constraints LastData and LastDataBeforeGetData. To express this we experimented with the addition of a functional constraint called RecentData to the component, which can be expressed as a state machine as shown in Figure D.3. In such a constraint we first express the events that are used in the state machine and next defined the allowed transitions. Note that in state AwaitReplyGetData we specify two possible reply values to command GetData.

```
RecentData {
    use events
    command ApplicationPort::GetData
    ApplicationPort::reply to command GetData
    command DevicePort::SensorGetData
    DevicePort::reply to command SensorGetData

    variables
    real d1
    real d2

    initial state NoData {
        transition
            do: DevicePort::SensorGetData
            next state: AwaitData
    }
    state AwaitData {
        transition trigger: DevicePort::reply(real v) to command SensorGetData
            do: d1 := v
            next state: DataAvailable
    }
    state DataAvailable {
        transition
            do: DevicePort::SensorGetData
            next state: DataAvailable
        transition trigger: DevicePort::reply(real v) to command SensorGetData
            do: d1 := v
            next state: DataAvailable
        transition trigger: ApplicationPort::GetData
            next state: AwaitReplyGetData
    }
    state AwaitReplyGetData {
        transition
            do: DevicePort::SensorGetData
            next state: AwaitReplyGetData
        transition trigger: DevicePort::reply(real v) to command SensorGetData
            do: d2 := v
            next state: AwaitReplyGetData
        transition
            do:
            ApplicationPort::reply(d1) to command GetData
            next state: DataAvailable
        transition
            do:
            ApplicationPort::reply(d2) to command GetData
            next state: DataAvailable
    }
}
```

Figure D.3: Functional constraint of the SensorServer component to express that recent data is returned

Run4 satisfies this constraint, but a violation is observed when monitoring Run2. The ComMA tooling provides the information shown in Figure D.4 about the error.

Observe that in this case the reply to GetData uses the return value of another SensorGetData call between the GetData call and its reply.

Finally, we experimented with the generic component constraint shown in Figure D.5 which expresses that if we observe a reply to command SensorGetData at time t1 with value v1 and next, after an arbitrary number of other events, we observe a reply to command GetData at time t2 with value v2 where v2 is greater than v1, then either v1 and v2 are different or t2 – t1 is less than 100. This expresses that command GetData returns a recently obtained value. Note that we assume that the return values are increasing to obtain an efficient check.

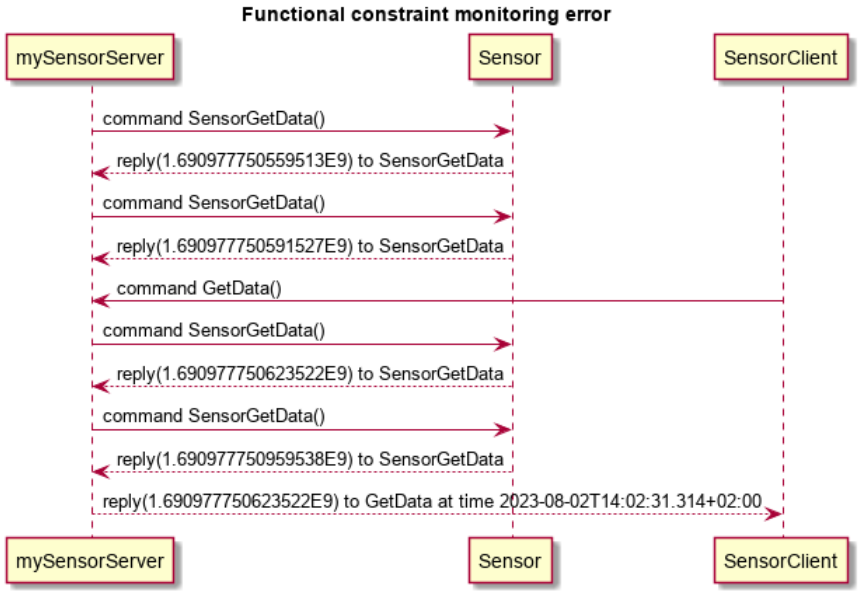

Figure D.4: Sequence diagram showing violation of the RecentData property in Run2

```
generic constraints
variables
real t
real t1
real t2
real v1
real v2
MaxDistance
[ t1, DevicePort::reply(v1) to command SensorGetData ];
[ t, any ApplicationPort::event ] | [ t, any DevicePort::event ]
until [ t2, ApplicationPort::reply(v2) to command GetData, v1 <= v2 ]
where v1 != v2 or t2 - t1 < 100.0
```

Figure D.5: Generic constraint of the SensorServer component to the maximum distance in time between equal reply values

Monitoring detects 18 violations of this constraint for Run2 and 24 violations for Run4. Note that this rule does not check a second reply to GetData with the same value. It might be interesting to check that all replies to GetData have different values.

TNO-ESI

High Tech Campus 25
5656 AE Eindhoven
www.tno.nl