

JOHANNES WIENKE

FRAMEWORK-LEVEL RESOURCE AWARENESS  
IN ROBOTICS AND INTELLIGENT SYSTEMS



JOHANNES WIENKE  
FRAMEWORK-LEVEL RESOURCE AWARENESS  
IN ROBOTICS AND INTELLIGENT SYSTEMS

Improving dependability by exploiting knowledge about system resources

A doctoral thesis presented for the degree of  
Doctor of Engineering (Dr.-Ing.) at

CoR-Lab / CITEC  
Faculty of Technology  
Bielefeld University  
Inspiration 1  
33619 Bielefeld  
Germany

REVIEWERS

Dr.-Ing. Sebastian Wrede  
Prof. Dr. Davide Brugali

BOARD

Prof. Dr. Philipp Cimiano  
Dr. rer. nat. Malte Schilling

DEFENDED AND APPROVED

September 13, 2018

Printed on permanent paper as per ISO 9706.

## ABSTRACT

---

Modern robots have evolved to complex hardware and software systems. As such, their construction and maintenance have become more challenging and the potential for **failures** has increased. These failures and the resulting reduction of **dependability** have a considerable effect on the acceptance and usefulness of robotics systems in their intended applications. Even though different software engineering techniques have been developed to control dependability-critical aspects of such complex systems, the state of the art for experimental robotics and intelligent systems is that – if at all – functional properties are systematically controlled through techniques such as unit testing and simulation runs. Yet, system dependability can also be impaired if nonfunctional properties behave unexpectedly. This thesis focuses on the utilization of **system resources** such as **CPU**, memory, or network bandwidth as an important nonfunctional aspect, which has not received much systematic treatment in robotics and intelligent systems so far. Unexpected utilizations of system resources can have effects ranging from merely wasting energy and reducing a robot’s operational time to a degradation in its function due to processing delays. Even safety-critical situations can arise, for instance, if a motion planner or obstacle avoidance **component** cannot react before a collision. Therefore, the systematic analysis of a system’s **resource utilization**, a guidance of developers regarding these aspects, and testing and **fault detection** for unexpected resource utilization patterns are an effective contribution of this thesis towards more reliable robots.

In this work I describe a concept for integrating **resource awareness** into component-based robotics and intelligent systems. This concept specifically addresses the often loosely controlled development process predominant in experimental research. As such, the presented methods have to be applicable without a high overhead or large changes to the evolved development methods and system structures. Within this concept, which I termed *framework-level resource awareness*, I have explored methods in two directions: On the one hand, a set of tools helps developers to understand and systematically control the resource utilization while developing and testing systems. On the other hand, I have applied machine learning techniques to enable autonomous reactions at runtime based on predictions about the resource utilization of system components. With the two views, this work explores novel directions for implementing resource awareness in research systems and the conducted evaluations underline the suitability of the framework-level resource awareness concept.



## ACKNOWLEDGMENTS

---

Pursuing a PhD is a huge and long endeavor and many people influenced and supported me on the way to this thesis. First of all, I have to thank my parents for giving me the opportunity to study and for their constant support and questions about the completion of this thesis. Of course, I also have to thank Vanessa for supporting me every day, even though regular conference trips and occasional long days at the university had a noticeable impact on the time we could spend together.

Special thanks are directed to Sebastian for supervising this thesis, his constant support, and many lively and fruitful discussions. Moreover, I have to thank Prof. Brugali for accepting my invitation to review this thesis. My apologies to both of you for the amount of pages you have to work through.

Many decisions regarding the work included in this thesis were the result of interesting discussions with my colleagues. I have to thank all of them for their willingness to collaborate and to discuss. Their feedback helped to shape many solutions presented here and I could always find someone to debug the most obscure problems. They all contributed to a wonderful and fun work environment.

Finally, I have to thank everyone who agreed to proof-read this thesis. Thank you, Dennis, Hendrik, Jan, Jochen, Leon, Michael, Norman, Torben, and Vanessa.





# CONTENTS

---

	<b>I RESEARCH TOPIC</b>	<b>1</b>
1	INTRODUCTION	3
2	FUNDAMENTAL CONCEPTS AND TERMINOLOGY	7
	2.1 Resources and related concepts . . . . .	7
	2.1.1 Resource categorization schemes . . . . .	8
	2.1.2 Metrics, KPIs, and performance counters . . . . .	10
	2.1.3 A conceptual model of system resources . . . . .	11
	2.2 Dependable computing and FD* . . . . .	12
	2.2.1 Dependability . . . . .	13
	2.2.2 Threads to dependability . . . . .	14
	2.2.3 Means of dependability . . . . .	15
	2.2.4 Dependability and performance . . . . .	17
3	A SURVEY ON BUGS IN ROBOTICS SYSTEMS	21
	3.1 Tool usage . . . . .	22
	3.2 Bugs and their origins . . . . .	24
	3.3 Performance bugs . . . . .	25
	3.4 Bug examples . . . . .	27
	3.5 Summary . . . . .	28
	3.6 Threats to validity . . . . .	29
4	A CONCEPT OF RESOURCE AWARENESS	31
	4.1 Resource awareness in computing systems . . . . .	32
	4.1.1 Server infrastructure operation . . . . .	32
	4.1.2 Cloud computing . . . . .	33
	4.1.3 Model-based performance prediction . . . . .	33
	4.2 Resource awareness in robotics . . . . .	35
	4.2.1 Space robotics . . . . .	35
	4.2.2 Cloud robotics . . . . .	35
	4.2.3 Resource-aware algorithms . . . . .	36
	4.2.4 Resource-aware planning and execution . . . . .	36
	4.2.5 Infrastructure monitoring of robotics systems . . . . .	37
	4.2.6 Model-driven approaches . . . . .	38
	4.3 Summary . . . . .	39
	<b>II TECHNOLOGICAL FOUNDATION</b>	<b>41</b>
5	COMPONENT-BASED ROBOTICS SYSTEMS	43
	5.1 Component-based software engineering . . . . .	43
	5.2 CBSE and distributed systems . . . . .	45
	5.3 CBSE in robotics . . . . .	45
	5.4 Patterns in component-based robotics systems . . . . .	49
	5.5 Summary . . . . .	50

6	MIDDLEWARE FOUNDATION: RSB	51
6.1	Architecture	54
6.1.1	Event model	55
6.1.2	Naming model	56
6.1.3	Notification model	58
6.1.4	Time model	60
6.1.5	Observation model	61
6.1.6	Extension points	61
6.2	Introspection	61
6.3	Domain data types: RST	63
6.4	Tool support	65
6.5	Interoperability with other middlewares	66
6.6	Applications	67
6.7	Summary	68
7	A HOLISTIC DATASET CREATION PROCESS	71
7.1	Challenges in creating datasets	72
7.2	Description of the holistic process	73
7.3	Realization based on RSB	75
7.3.1	Data sources	75
7.3.2	Calibration	76
7.3.3	Unification	76
7.3.4	View generation and annotation	77
7.4	Summary	77
8	SYSTEM METRIC COLLECTION	79
8.1	Available system metric sources	80
8.2	Resource acquisition tools	83
8.3	Implementation	84
8.3.1	Host collection	85
8.3.2	Processes collection	86
8.3.3	Subprocess handling	87
8.3.4	Data representation	88
8.3.5	System integration	88
8.4	Summary	89
	<b>III DEVELOPER PERSPECTIVE</b>	<b>91</b>
9	RUNTIME RESOURCE INTROSPECTION	93
9.1	Available tools	93
9.2	Resource utilization dashboard implementation	95
9.2.1	Time series database adapter	95
9.3	Dashboard design	97
9.4	Evaluation	98
9.4.1	Qualitative evidences	99
9.4.2	Quantitative evaluation	101
9.5	Summary	103
10	SYSTEMATIC RESOURCE UTILIZATION TESTING	105
10.1	Related work	105

10.2	Performance testing framework concept . . . . .	107
10.3	Realization . . . . .	109
10.3.1	Load generation . . . . .	109
10.3.2	Environment setup . . . . .	114
10.3.3	Test execution . . . . .	115
10.3.4	Test analysis . . . . .	116
10.3.5	Automation . . . . .	119
10.4	Evaluation . . . . .	120
10.5	Summary . . . . .	122
11	<b>MODEL-BASED PERFORMANCE TESTING</b>	125
11.1	Related work . . . . .	126
11.2	Language design . . . . .	127
11.2.1	Metamodel . . . . .	129
11.2.2	Editors . . . . .	130
11.2.3	Code generation . . . . .	132
11.3	Notable language features . . . . .	132
11.3.1	Inline data generation . . . . .	132
11.3.2	Type safety for embedded custom code . . . . .	133
11.3.3	Expressive custom code via embedding . . . . .	134
11.4	Evaluation . . . . .	135
11.5	Summary . . . . .	136
	<b>IV AUTONOMY PERSPECTIVE</b>	139
12	<b>A DATASET FOR PERFORMANCE BUG RESEARCH</b>	141
12.1	Recording method . . . . .	142
12.2	Included performance bugs . . . . .	144
12.2.1	Algorithms & logic . . . . .	144
12.2.2	Resource leaks . . . . .	144
12.2.3	Skippable computation . . . . .	144
12.2.4	Configuration . . . . .	145
12.2.5	Threading . . . . .	145
12.2.6	Inter-process communication . . . . .	145
12.3	Automatic fault scheduling . . . . .	145
12.4	Summary . . . . .	147
13	<b>RUNTIME RESOURCE UTILIZATION PREDICTION</b>	149
13.1	Feature generation . . . . .	151
13.1.1	Accumulated event window features . . . . .	152
13.1.2	Adding previous system metrics . . . . .	155
13.1.3	Baseline: system metrics . . . . .	156
13.1.4	Preprocessing . . . . .	156
13.2	Model learning . . . . .	157
13.3	Evaluation . . . . .	158
13.3.1	Results on the ToBi dataset . . . . .	158
13.3.2	Influences of the component behavior . . . . .	164
13.4	Learning from performance tests . . . . .	167
13.4.1	Evaluation . . . . .	169

13.4.2	Influences of the test structure . . . . .	170
13.5	Related work . . . . .	172
13.6	Summary . . . . .	173
14	RUNTIME PERFORMANCE DEGRADATION DETECTION	175
14.1	Related approaches . . . . .	176
14.2	Residual-based performance degradation detection . . .	177
14.3	Evaluation . . . . .	179
14.3.1	Results on the ToBi dataset . . . . .	179
14.3.2	Influence of component behavior . . . . .	183
14.4	Summary . . . . .	184
	<b>V PERSPECTIVES</b>	187
15	CONCLUSION	189
16	OUTLOOK	193
	<b>VI APPENDIX</b>	195
A	SURVEY: FAILURES IN ROBOTICS SYSTEMS	197
B	FAILURE SURVEY RESULTS	207
C	SURVEY: DASHBOARD EVALUATION	231
D	DASHBOARD SURVEY RESULTS	235
E	TOBI DATASET DETAILS	237
	ACRONYMS	239
	GLOSSARY	245
	BIBLIOGRAPHY	253

## LIST OF FIGURES

---

Figure 2.1	Linux kernel system metric discretization example	10
Figure 2.2	Conceptual model of system resources . . . . .	11
Figure 2.3	System failure terms and performance counterparts	18
Figure 3.1	Participant development time . . . . .	22
Figure 3.2	Monitoring tools usage frequencies . . . . .	22
Figure 3.3	Debugging tools usage frequencies . . . . .	23
Figure 3.4	Observed mean time between failures . . . . .	24
Figure 3.5	Frequency of system failure reasons . . . . .	25
Figure 3.6	Frequency of bug effect on system resources . . . . .	26
Figure 3.7	Frequency of reasons for performance bugs . . . . .	26
Figure 5.1	Prevalence of keywords in robotics publications . .	46
Figure 5.2	Citation counts for robotics frameworks . . . . .	47
Figure 6.1	Dependency graphs of robotics middlewares . . . .	53
Figure 6.2	Dependency graph of RSB C++ . . . . .	54
Figure 6.3	Architecture concept of RSB . . . . .	55
Figure 6.4	RSB concepts and relations . . . . .	57
Figure 6.5	Exemplary RSB scopes . . . . .	58
Figure 6.6	RSB introspection mechanism structure . . . . .	63
Figure 6.7	Evolution of the RST data type count . . . . .	65
Figure 7.1	HUMAVIPS vernissage recording setup . . . . .	72
Figure 7.2	Holistic dataset creation process schema . . . . .	74
Figure 7.3	ELAN export of a dataset trial . . . . .	77
Figure 8.1	Structure of the host collector . . . . .	85
Figure 8.2	Structure of the process collector . . . . .	86
Figure 8.3	System integration of collection daemons . . . . .	89
Figure 9.1	Implementation scheme of dashboards . . . . .	94
Figure 9.2	Dashboard data pipeline . . . . .	95
Figure 9.3	Time series database adapter architecture . . . . .	96
Figure 9.4	Time series database adapter processing steps . . .	97
Figure 9.5	Generic resource dashboards . . . . .	98
Figure 9.6	TTS memory leak in the ToBi system . . . . .	100
Figure 9.7	BonSAI RSB participant leak . . . . .	101
Figure 9.8	Usage frequency of the dashboards . . . . .	102
Figure 9.9	Improved understanding of system resources . . .	102
Figure 10.1	Performance testing concept . . . . .	109
Figure 10.2	Structure of the testing API . . . . .	110
Figure 10.3	Exemplary performance test structure . . . . .	114
Figure 10.4	Testing framework execution steps . . . . .	115
Figure 10.5	Exemplary performance testing time series . . . . .	116
Figure 10.6	Testing framework application in Jenkins CI . . . .	119

Figure 10.7	Test execution number influence on detection . . . .	123
Figure 11.1	Modularization of the performance testing DSL . .	128
Figure 11.2	Metamodel of the performance testing DSL . . . . .	129
Figure 11.3	Test phase with Protobuf data . . . . .	131
Figure 11.4	Test suite example . . . . .	131
Figure 11.5	Integration of custom Java code . . . . .	133
Figure 11.6	DSL and AST for a test case . . . . .	134
Figure 12.1	ToBi dataset recording scene . . . . .	143
Figure 12.2	Scheduling of performance bugs . . . . .	146
Figure 13.1	Feature generation and synchronization . . . . .	153
Figure 13.2	Examples for file descriptors metrics . . . . .	160
Figure 13.3	Memory metric for the objectbuilder component .	160
Figure 13.4	Prediction error per metric on ToBi dataset . . . . .	162
Figure 13.5	System metric prediction examples . . . . .	163
Figure 13.6	Mock component prediction errors . . . . .	165
Figure 13.7	Mock component prediction examples . . . . .	166
Figure 13.8	Prediction error from performance tests . . . . .	168
Figure 13.9	Mock component prediction examples from tests .	169
Figure 13.10	Prediction errors depending on test configs . . . . .	172
Figure 14.1	Fault detection scheme . . . . .	178
Figure 14.2	Detection scores per component . . . . .	181
Figure 14.3	Detection scores per fault . . . . .	182
Figure 14.4	Fault detection scores for a mock component . . . .	184

## LIST OF TABLES

---

Table 3.1	Origin differences of performance and general bugs	27
Table 6.1	Extension points in Robotics Service Bus (RSB)	62
Table 8.1	Comparison of Linux system metric sources	81
Table 10.1	Identified performance test actions	111
Table 10.2	Available evaluation data	121
Table 10.3	ROC AUC scores for regression detection	122
Table 11.1	Comparison of Java testing framework and DSL	135
Table 13.1	AEW feature dimensions for ToBi components	155
Table 13.2	Regression results on AEW features	158
Table 13.3	Comparison of KR-FA and baseline Mean	159
Table 13.4	Regression results without degraded metrics	161
Table 14.1	Mean ToBi fault detection results	180

## LIST OF CODE LISTINGS

---

Listing 6.1	RST data type for produced speech utterances . . .	64
Listing 10.1	Action interface and exemplary implementation . .	112
Listing 10.2	Protocol Buffers data generation API . . . . .	113
Listing 10.3	ParameterProduct instantiation . . . . .	113



## NOTATION

---

### MARGIN NOTES

- ⊙ Key point
- ◆ Definition

### STATISTICAL SIGNIFICANCE

- \*  $p \leq 0.05$
- \*\*  $p \leq 0.01$
- \*\*\*  $p \leq 0.001$
- \*\*\*\*  $p \leq 0.0001$

### ATTRIBUTION OF AUTHORSHIP

I will speak of myself using *I* in case of work originally done by myself alone. In case the results of a collaboration with others are presented, I will use *we*. The respective collaborators are indicated by the co-authors of the publication the results are based on.



## Part I

### RESEARCH TOPIC

The first part of this thesis will introduce the general scope of work. I will motivate the necessity for research, explain the relation to other disciplines, and will formulate the research questions addressed in this thesis. Moreover, general terms and concepts will be defined.



## INTRODUCTION

---

Since Czech writer Karel Čapek first coined the word *robot* in his 1920 play *R.U.R* [Čap14], robots have evolved from a distant futuristic fantasy to real machines that are able to fulfill meaningful purposes. Even though completely autonomous and reliable multi-purpose robots are not yet available, research has made serious progress within the last decades. Starting with the first commercially available pick-and-place robot in the 1950s [Dhi91], technology has constantly evolved and new application areas for robots have been opened up. Ranging from robotic vacuum cleaners and lawn mowers to physical [human–robot interaction \(HRI\)](#) in industrial settings as well as space exploration missions, these applications require complex robotic systems, often both, in terms of hardware and their control software, which inevitably increases the potential for a diverse set of issues or complete failures. Such failures have negative effects for the users of robotic systems as well as the manufacturers. For robots intended to be used by end users, failures first result in a degraded user experience [BBY16] and eventually mean a decrease in business revenue for the manufacturer, in case a product cannot be sold as expected. For robots that are physically harmful, failures can also result in severe injuries, which need to be avoided at all cost. To avoid these effects, and also to cope with increasing regulations for industrial robotics applications [Laz16], developers need to systematically analyze and verify the [dependability](#) of their systems and take measures to avoid catastrophic failures in case of potential bugs. Such measures must cover the complete systems ranging from their hardware design and safety to the controlling software system.

In the context of this thesis, I am concerned with the software aspects of robots and other intelligent systems such as virtual agents or smart homes, which are constructed using comparable principles. The software industry has developed a set of established methods to control the dependability of their systems. Despite many advanced techniques, the robotics research community has primarily adopted functional testing, for instance, through unit tests and simulation runs. These techniques mainly deal with [functional requirements](#) of the systems and their constituting software artifacts, which means whether intended functionality is provided at all, or not. However, dependability also encompasses further nonfunctional aspects. This thesis particularly focuses on the utilization of [system resources](#) as one of these nonfunctional aspects, which has not received much systematic treatment in robotics so far. Unexpected utilization of system

resources can have effects ranging from merely wasting energy and reducing a robot's operational time to a degradation in its operation due to processing delays. Even safety-critical situation can arise, for instance, if a motion planner or obstacle avoidance **component** cannot react timely before a collision. Therefore, the contributions of this work towards more reliable robotics systems are a systematic analysis of **resource utilization**, a guidance of developers regarding these aspects, and testing and **fault detection** for unexpected resource utilization patterns.

It is hard to subsume all kinds of research and development activities regarding robotics systems under the single umbrella *robotics* without stumbling across differences. Robotics has become an increasingly broad discipline with a multitude of different systems and applications. Among others, common directions and application areas include:

- Industrial manipulators in closed production cells without human interaction.
- Research prototypes of manipulators in close physical HRI.
- Robotic vacuum cleaners or lawn mowers.
- Medical robots.
- Wheeled service robots with multimodal HRI capabilities.
- Humanoid robots.
- Bio-inspired robots.

Apart from the breadth of types of robots and their intended applications, also their origins differ and with that their development activities. While mature technologies such as industrial manipulators are usually constructed and programmed in the context of established companies, more cutting-edge technology is often developed in an academic context or in start-ups originating from this area.<sup>1</sup> The demand for dependability and techniques to achieve this property is most obvious for systems that have reached the product level. However, achieving a sufficient level of dependability is required and beneficial in research settings, too. Despite less strict regulatory environments, also in research robots can impose security hazards and their system design has to ensure that no scientist or laboratory visitor is harmed. Moreover, scientific tasks have become increasingly dependent on the proper functionality of integrated systems, for instance, for analyzing HRI with real robots. The success of such experiments depends on a reliable robot system. Finally, the close collaboration of robotics research institutions and industrial partners has established

---

<sup>1</sup> For instance, the Franka Emika robot.

a short transfer chain from research into industrial applications and if research prototypes already possess the required level of dependability, the time to market is reduced. Yet, achieving dependability in research settings is a challenge. Software developed in academic contexts is often of lower quality because software and system development is carried out under different conditions than in an industrial setting [Wreo8, pp. 31 sqq.; HLN10]. Scientific development is usually embedded into research projects with developers being either PhD students primarily interested in finishing their thesis or constantly changing student assistants [HLN10]. The development process cannot be as rigorous as it would need to be to embed all available methods that ensure a proper system dependability. Due to the lack of time, expertise, or incentives, many available techniques are not applied, partially due to the overhead their application creates.

The solutions developed in this thesis target exactly such a scientific research environment. Instead of providing techniques that require rigor, time, and a complete shift of development practices, the focus in this work was to establish methods that are easily applicable in the existing ecosystems and processes. By reducing the amount of effort and change these methods cause to existing and established workflows, the hypothesis is that the resistance to use them is minimized. Thus, the developed methods are applicable to distributed component-based systems as they are commonly seen in robotics and intelligent systems research. The idea is to improve their dependability by incorporating [resource awareness](#) through generally applicable methods. This idea also encompasses that methods should follow the natural separation of concerns created by the component-based paradigm often used in these domains. Individual developers should be able to decide which methods to apply and the generated results and outputs should be assignable to individual components so that responsibilities become evident. I will later define this approach as [framework-level resource awareness](#). Generally, the explicit tradeoff taken here is that applicability is favored for groundbreaking results. Yet, the developed methods have proved their usefulness in the evaluations presented in this work and are therefore effective means to improve the dependability of robotics and intelligent systems.





Before explaining the approach of this work, I will first introduce fundamental concepts and related terms to establish a common ground for the following explanations. Due to varying influences and directions, diverging understandings and definitions for common computing terms related to the general idea of [resource awareness](#) have evolved. I will summarize these definitions for the relevant sub-domains and establish the terminology used throughout this work.

### 2.1 RESOURCES AND RELATED CONCEPTS

Establishing resource awareness inevitably encompasses understanding the behavior of *resources* of the involved hardware. Thus, a precise understanding of the term *resource* as well as knowledge about the connection of resources to other parts of the system is required. Generally, Merriam-Webster defines a resource as “a source of supply or support” [MwRes] or “an available means” [MwRes]. In the computing domain, a resource is therefore – in the broadest sense – a means available to perform computations. Stemming from different directions, such means are usually classified as being of abstract or concrete nature. Abstract resources are unrelated to physical execution environments with the aim to describe the utilization of available computing means in a generic and comparable fashion. Originating from the area of complexity analysis of algorithms (e.g., using Big-O-notation), abstract resources are commonly called *computational resources* [Wik16a] and examples are space and time. On the other hand, actual physical execution environments (such as a server or desktop computer) have concrete resources of limited availability that are utilized to enable the computation. These resources are either direct or indirect results of the physical properties of the host system hardware and can therefore be called *system resources* [IHE15; Wik16e]. Such system resources are the *central processing unit (CPU)*, working memory, file descriptors, disk space, network bandwidth, etc. An implementation of an algorithm with an abstract complexity in space and time will eventually utilize a certain amount of system resources of a host system when being executed on input data. The basic distinction between computational and system resources has been a useful tool also in other areas of software engineering, for instance, in model-driven engineering [BKRo9].

☛ *computational resource*

☛ *system resource*

This thesis focuses on analyzing the concrete utilization of system resources. Therefore, if not explicitly stated otherwise, any use of the

word resource automatically refers to system resources for the sake of brevity. For robotics one can think of further resources regarding the physical system such as battery or fuel. In case such resources are of critical interest, for instance, in planetary exploration missions, these types of resources are usually well controlled within the software of appropriate robots (cf. Section 4.2 on page 35). Therefore, I will ignore such resources as a topic of this thesis and focus on system resources, which have not received a systematic treatment in robotics so far.

Most system resources are of limited availability. This availability is also termed a resource's *capacity* [IHE15; Mol15] or *quantity* [Kub17], which is then *utilized* [IHE15; Mol15] by applications running on a computer. As all common computer systems nowadays are multi-tasking systems where processes are executed in parallel, *resource contention* [Koz10; Wik16c] constantly happens on such systems, because the parallel processes compete for the available resources. Depending on the ratio of capacity and utilization, different effects and effect strengths can be observed. These range from no noticeable implications for the running processes to different slow downs and ultimately execution failures in case required resources cannot be provided by the system at all. The latter situation is called *resource starvation* [Wik16d]. The way contention affects the running applications on a system depends on the type of resources that are affected. Therefore, a categorization of resources is helpful.

### 2.1.1 Resource categorization schemes

Muskens and Chaudron [MCo4] propose a generic two-dimensional classification of system resources along the dimensions *processing – non-processing* and *pre-emptive – non-pre-emptive*. *Processing* resources are those that process (and therefore discard) something. For instance, a CPU processes instructions and a network link processes packets. Other resources such as memory do not discard the elements they operate on and are therefore *non-processing*. For the second dimension, *pre-emptive* resources are those where a desired request for quantity can be rejected, reduced, interrupted, or postponed by a resource scheduling system without being fatal to the requesting process. This means that the resource can be reused for serving other processes in parallel despite still being requested. This, for instance, includes the CPU, where a process can be interrupted at any time or receive only a fraction of the available CPU cores at a given time to handle contention. On the other hand, *non-pre-emptive* resources such as memory or file handles can be used only by one requester at any given moment and they become available again once they are released by the requester. Not provisioning the requested resources to the requester is usually an exception that needs to be specifically handled in the requesting process' code. The second dimension (pre-emptiveness)

matches the classification proposed in *The Kubernetes resource model* [Kub17], where the matching properties are called *compressible* and *non-compressible*. The first dimension is primarily based on the physical nature of the resources whereas the second dimension addresses usage and provisioning properties. Another example for a comparable classification approach is presented in Seneviratne et al. [SLB13] with the aim to predict the resource utilization in grid computing environments. The terms they introduce are *time-shared* (pre-emptive) and *space-shared* (non-pre-emptive).

Becker et al. [BKR09] have proposed to classify system resources as *processing* or *passive* resources, with a special subclass of processing resources being used to model network connections. Although this definition seems to match the *processing – non-processing* dimension of Muskens and Chaudron [MC04], Becker et al. [BKR09] specifically mention memory as a *processing resource*. In the sense that working memory has a limited speed at which read and write operations can be *processed*, this definition makes sense, but it disregards the available memory size, which is of passive nature.

In the robotics domain, Volpe et al. [Vol+00] propose four distinct resource categories. These are *depletable*, *non-depeletable*, *atomic* and *concurrent* resources. A resource is *depletable* if its resource capacity decreases when being utilized, for instance, the battery. Conversely, the capacity of a *non-depletable* system resource is permanent. *Concurrent* resources can be utilized by multiple consumers (e.g., the CPU) in parallel whereas *atomic* resources can be used by only one consumer at a time, e.g. a sound device. Within their model, these four categories are mutually exclusive (i.e., every system resource belongs to exactly one of these categories) despite addressing distinct properties. If these categories were meant as pairwise attributes instead, the dimension *depletable – non-depletable* would be a new dimension unrelated to aforementioned schemes whereas *concurrent – atomic* would relate to the *pre-emptive – non-pre-emptive* dimension of Muskens and Chaudron [MC04].

Finally, Shimizu et al. [Shi+09] distinguish between *computation*, *communication*, and *storage* resources. This scheme is based primarily on the physical nature of the available system resources. However, beyond this categorization – and in line with the varying views on memory between Muskens and Chaudron [MC04] and Becker et al. [BKR09] – Shimizu et al. [Shi+09] show that different parameters related to system resources exist. These parameters can be categorized individually. Therefore, a more fine-grained model is required to depict the actual situation of what constitutes system resources in terms of physical properties, measurable metrics, and their impact on the execution of software applications. I will devise this model in [Section 2.1.3](#) on page 11 after explaining the remaining concepts.

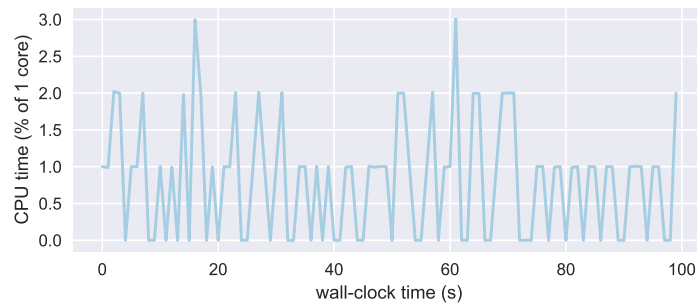


Figure 2.1: Example for discretization artifacts created by the Linux kernel for a system metric representing CPU utilization.

The presented resource categorization schemes differ in the number of categories or dimensions they provide. However, a general trend is visible: Resources are categorized either based on their physical properties (e.g., *depletable* – *non-depletable* or *computation*, *communication*, *storage*) or based on the way contention is commonly addressed by the operating system (e.g., *time-shared* – *space-shared*).

### 2.1.1.2 Metrics, KPIs, and performance counters

So far, resources have been described primarily based on their physical availability and the way they operate. However, from the perspective of processes being executed on a computer and a developer or software supervising the execution of these processes and the computer system in total, such properties are not directly measurable.

- system metric* ◆ Instead, the operating system provides a set of *system metrics* [IHE15; Syd11, p. 9], which express the utilization of system resources. In the area of *application performance monitoring (APM)*, such system
- key performance indicator (KPI)* ◆ metrics are also called (system-level or efficiency-oriented) *key performance indicators (KPIs)* [IHE15; Mol15, p. 2]. The relation of these metrics to the actual utilization of the system resources is often not straightforward, potentially ambiguous, and operating system specific. In some situations, low-level metrics exist that are closely coupled to the physical hardware where the relation is much clearer. For instance, *hardware performance counters* [Wik16b] allow monitoring the internal details of a CPUs computation. As these detailed metrics often require access via special tools or instrumentation (e.g., *Processor Counter Monitor (PCM)* [iPCM] for Intel CPUs) and their interpretation requires special knowledge, most developers do not use them.
- system metric source* ◆ The operating system provides different *system metric sources*, which are the technical interfaces that allow reading out the system metrics. For instance, common system metric sources in Linux are the proc file system [MSO01, p. 147; Procl7] or the taskstats interface [Lino6] (cf. Section 8.1 on page 80 for a systematic review).

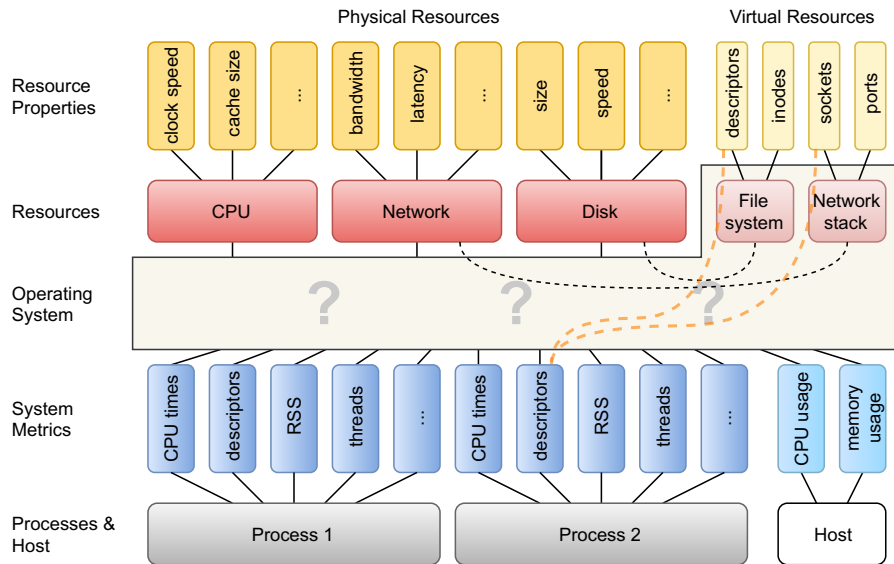


Figure 2.2: A conceptual model of system resources. The operating system maps resource and their properties to measurable system metrics for individual processes and the host system itself. The exact mapping is usually an implementation detail of the operating system and its concrete version and thus a black box.

When collecting system metrics over time, one acquires *time series* data of the resource utilization of processes and the host system. These time series reflect the dynamics of the resource utilization as exposed by the operating system. Unfortunately, the real dynamics are often hidden behind several layers of abstraction and discretization created by the operating systems and their system metric sources. These distortions have different reasons, ranging from pure implementation details to optimizations, which make acquiring the metrics less expensive. For instance, when measuring the CPU utilization of a process using the Linux `proc` file system, the metric is not updated continuously (e.g., with a fixed rate) but instead the update depends on factors such as changes of the monitored process' state [Sta15] and a minimum required value increment. Hence, a continuously small utilization of CPU time is usually visible as spiking, which is visualized in Figure 2.1 on the preceding page.

### 2.1.1.3 A conceptual model of system resources

To visualize the relation of the aforementioned aspects of system resources, Figure 2.2 exemplifies a conceptual model of how the different aspects relate to each other. At the top of the figure the hardware resources of a computer system are depicted, which have a set of resource properties describing their capabilities grounded in their physical structure. These resources are connected to the operating system where usually a kernel with appropriate driver code is responsible of

*virtual resource* making these resources usable for computational tasks. The operating system itself can provide further *virtual resources* with own properties. These virtual resources usually reflect a software abstraction of physical system resources. The gray dashed lines exemplify how the file system and the network stack are virtual resources, which are derived from the hard disks and the networking hardware of a host system.

In the direction of applications and system users, the operating system exposes system metrics for individual processes and the host system in total. These system metrics allow measuring the resource utilizations, and they often have different abstractions than the real physical properties of the system resources. Thus, no strict 1:1 mapping exists among the physical resources with their properties and the system metrics. For instance, as depicted with the orange dashed lines, Linux provides a system metric to count open file descriptors currently handled by a process. This metric includes real files relating to the file system but also network sockets are represented as file descriptors. Therefore, the measurable system metrics often represent an interpretation of the physical world, with no direct mapping to physical properties. Finally, the developer or system maintainer making use of the resource information usually has a simplified mental model of the provided information that does not necessarily match the technical representation provided through system metrics.

## 2.2 DEPENDABLE COMPUTING AND FD\*

*dependable computing* Whenever one talks about the reliability, safety, or fault tolerance of computer-based systems, two distinct areas can be seen as origins for research and terminology in this direction. One of these research areas is *dependable computing*. This term and related concepts have primarily been coined in the 1980s as a community effort to unify the terminology used to describe aspects related to the *dependability* of technical systems [Avi+04]. The results of this effort have been collected by Laprie in his seminal publication “Dependable Computing and Fault Tolerance: Concepts and Terminology” [Lap95, reprint of a 1985 paper]. An updated version of the definitions found in this paper has been presented with Avižienis et al. [Avi+04]. As a second discipline, the control engineering community has also tried to standardize terminology. Isermann and Ballé [IB97] presents the most notable result of these efforts with updated definitions of terminology being described in Isermann [Ise06].

In the following paragraphs I will summarize the most important terms and definitions from both areas to establish a common ground for the remainder of this document.

### 2.2.1 Dependability

According to Avižienis et al. [Avi+04], *dependability* is defined as “the ability to avoid service failures that are more frequent and more severe than is acceptable” [Avi+04]. The “*service* delivered by a system (in its role as a provider) is its behavior as it is perceived by its user(s)” [Avi+04] and “a *user* is another system that receives service from the provider” [Avi+04]. Therefore, dependability is achieved in case the rate of failures of a system is within an acceptable range for its application. Isermann [Ise06, p. 24] has collected two further definitions of dependability, where the first subscribes to the aforementioned functional perspective based on the delivered service of a system, whereas the second definition focuses on the ability to rely on a dependable system. This second definition is, however, comparable to the original definition of dependability from Laprie [Lap95]: “computer system dependability is the quality of the delivered service such that reliance can justifiably be placed on this service” [Lap95].

Dependability is usually understood as a general concept that encompasses different attributes, which can be assessed individually. From the list of attributes presented in Avižienis et al. [Avi+04] I consider most important the following ones:

**AVAILABILITY** “[R]eadiness for correct service” [Avi+04]. A better conceivable definition is provided by Isermann [Ise06] as the “probability that a system or equipment will operate satisfactorily and effectively at any period of time” [Ise06, p. 23]. Others agree on this view [III10; Mol15].

**RELIABILITY** “[C]ontinuity of correct service” [Avi+04] or “a measure of the continuous service accomplishment” [Lap95]. This can also be described as the “ability to perform a required function for a certain period of time” [Ise06, p. 21]. While *availability* describes the chance that a system can potentially be used at all, *reliability* focuses on the chance that the system, once used, correctly delivers its service.

**SAFETY** “[A]bsence of catastrophic consequences on the user(s) and the environment” [Avi+04] or in other words the “ability of a system not to cause danger to persons or equipment or the environment” [Ise06, p. 23].

**INTEGRITY** “[A]bsence of improper system alterations” [Avi+04]. Isermann [Ise06] provides a conflicting definition of *integrity* that instead focuses on the detection of faults: “integrity of a system is the ability to detect faults in its own operation and to inform a human operator” [Ise06, p. 24].

**CONFIDENTIALITY** “[T]he absence of unauthorized disclosure of information” [Avi+04].

I have excluded *maintainability*, which Avižienis et al. [Avi+04] define as the “ability to undergo modifications and repairs” because, in contrast to the aforementioned attributes, maintainability expresses a property of a system that is not of immediate relevance to the user of the system. A system with poor maintainability will eventually degrade in the aforementioned attributes and therefore becomes less dependable, however, this relation is indirect.

*robustness* ♦ Another frequently used term related to dependability is *robustness*. The IEEE standard 24765 [III10] defines robustness as “the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions”. As such, robustness quantifies a system’s ability to cope with unexpected conditions imposed on the system by its environment, for example, the user or other software components. Avižienis et al. [Avi+04] describe robustness as a secondary attribute, which refines or specializes the aforementioned primary ones. Unfortunately, no indication is given which attributes are addressed exactly. However, robustness is commonly seen as an extension of *reliability* and in this view robustness describes the dependability of a system in cases it has to operate under conditions that go beyond the intended design [Jen14]. Reliability is therefore restricted to delivering the service under nominal conditions.

### 2.2.2 Faults, failures and errors: threats to dependability

Dependability is the ability of a system to deliver its service. However, technical systems fail and therefore the times in which a system does not perform its service according to its “*service specification* [which] is an agreed description of the expected service” [AL86] are called *service outages* [Avi+04]. Regarding such outages, several terms are commonly used to describe the causes: *failure*, *error*, *fault*, and *bug*. In the dependable computing domain, the first three of them have proper definitions and their most recent forms from Avižienis et al. [Avi+04] are partially compatible with the ones found in the control engineering discipline. According to Avižienis et al. [Avi+04], the “event that occurs when the delivered service deviates from correct service” is called a system or service *failure* [Lap95; Avi+04]. Isermann and Ballé [IB97] provide a compatible definition of a failure as a “permanent interruption of a system’s ability to perform a required function under specified operating conditions” and Isermann [Ise06] later clarifies that “a failure is an event” [Ise06, p. 20].


*failure* ♦


Avižienis et al. [Avi+04] define an *error* as follows:

Since a service is a sequence of the system’s external states, a service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an error.



This means that an error is the externally observable deviation of the delivered service from the agreed upon service specification. Finally, the “adjudged or hypothesized cause of an error is called a *fault*” [Avi+04].

In contrast, the control engineering community uses the term *fault*  *fault* comparably to *error* in the dependable computing sense: “an unpermitted deviation of at least one characteristic property (feature) of the system from the acceptable, usual, standard condition” [Iseo6, p. 20]. Furthermore, the term *error* is used to describe a measurable effect of a *fault* (in the control engineering sense): a “deviation between a measured or computed value (of an output variable) and the true, specified or theoretically correct value” [Iseo6, p. 413]. That means both control engineering versions are related to the dependable computing definition of *error* but must not be confused with the term *fault* as the cause of an observable deviation from the service specification.

In contrast to the aforementioned terms, neither dependable computing nor control engineering formally define the term *bug*, despite  *bug* the fact that this term is commonly used among users and software developers. However, Avižienis et al. [ALRo1] indicate that the dependable computing community treats a bug as a special class of faults (in the dependable computing sense) and therefore a bug is a software or hardware defect that ultimately impairs a system’s dependability.

### 2.2.3 Means of dependability

To increase the dependability of technical systems a vast amount of techniques has been developed. Within the dependable computing domain, these techniques are grouped into four different categories [Avi+04] (using the dependable computing terminology):

**FAULT PREVENTION** aims at preventing the occurrence of faults in the first place, for instance by using appropriate engineering methods [Avi+04; Gol13].

**FAULT TOLERANCE** aims at avoiding “failures in the presence of faults” [Avi+04], for instance by automatically detecting these faults and recovering the service.

**FAULT REMOVAL** aims at reducing the number of faults in a technical system, for example, through manual verification [Gol13].

**FAULT FORECASTING** aims at estimating the number of future faults and their impact on the system to justify the dependability of a system [Avi+04].

One common source for fault tolerance techniques is the control engineering discipline, which has established definitions for common

tasks focusing on the *runtime* detection of system faults to reconfigure the system before a severe service outage can take place. Common tasks in this area are defined in Isermann and Ballé [IB97] as:

- fault detection* ♦
- FAULT DETECTION Determination of faults present in a system and the time of the detection.
  - FAULT ISOLATION Determination of the kind, location and time of detection of a fault. Follows fault detection.
  - FAULT IDENTIFICATION Determination of the size and time-variant behaviour of a fault. Follows fault isolation.
  - FAULT DIAGNOSIS Determination of the kind, size, location and time of detection of a fault. Follows fault detection. Includes fault isolation and identification.<sup>1</sup>

Another important task is *system reconfiguration*, which follows the aforementioned tasks with the aim to counteract the detected faults by adapting the system configuration [WF13]. Combinations of these tasks are commonly referred to as *fault detection and isolation (FDI)*, *fault detection and diagnosis (FDD)* or *fault detection, isolation, and recovery (FDIR)*, where FDI refers to *fault detection* and *fault isolation*, FDD additionally includes the identification (which results in *fault diagnosis*), and FDIR additionally includes *system reconfiguration* [WF13].

Regarding the implementation of fault detection techniques, different classes of methods have evolved, ranging from plausibility checks to complex model-based methods.<sup>2</sup> However, the common ground for all these methods is that they observe a running system to decide whether it delivers the correct service or not. As such, the term *fault* for the related tasks has to be understood in the phenomenological sense of the definition from the control engineering domain, because such methods are not directly capable to detect the underlying hardware or software defects.

*monitor* ♦ as *monitors*. A monitor, is “a software tool or hardware device that operates concurrently with a system or component and supervises, records, analyzes, or verifies the operation of the system or component” [III10].

### 2.2.3.1 Unified terminology

As seen, dependable computing and control engineering have established partially conflicting definitions for common terms used when discussing concepts related to the dependability of technical systems.

<sup>1</sup> It is important to note that *fault diagnosis* is a more general term for the combination of *fault isolation* and *identification*.

<sup>2</sup> Refer, for instance, to Ding [Dino8] for an overview of fault detection methods.

Because dependable computing provides a concise set of definitions for common attributes of dependability (cf. [Section 2.2.1](#) on page 13) I will use these terms throughout the course of this work. However, for naming threats to dependability I will use the terms from the control engineering community. This decision is based on the idea that fault detection is a well-established technique. A constant tension regarding the term *fault* would arise if the dependable computing definitions were used, because – from an external viewpoint – fault detection techniques can detect only a visible effect (definition of fault from control engineering) and not the underlying software or hardware defect (dependable computing). As the control engineering definitions lack a term to describe the root cause of a failure, I will use the term *bug* for this purpose. To summarize, the used definitions are:

**FAILURE** “An event that occurs when the delivered service deviates from correct service” [[Avi+04](#)]. ⊙ *unified terms for describing system problems*

**FAULT** “An unpermitted deviation of at least one characteristic property (feature) of the system from the acceptable, usual, standard condition” [[Iseo6](#), p. 20], which can potentially be observed through measurable *errors*.

**BUG** A software or hardware defect that potentially causes a failure.

Bugs can be introduced into systems in different ways. For example, when adding new functionality, the new implementation might contain an unknown bug. Conversely, an important case in the context of this work is if a modification impairs existing functionality of a piece of software. In this case, the bug is a *software regression* [[NTYo8](#)] and one major aspect of software testing is *regression testing* to immediately detect such software regressions. ◆ *software regression*

#### 2.2.4 Dependability and performance

A term closely related to the concept of dependability is *performance*. This term is as often used as imprecise or implicit definitions exist for it. The IEEE standard 24765 [[III10](#)] defines *performance* as “the degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage” [[III10](#)]. In this view, performance is an overarching property quantifying the mission success of a system given a set of measurable constraints such as *KPIs*. This view is supported by [Ibidunmoye et al. \[\[IHE15\]\(#\)\]](#). However, the question remains, how to measure performance. As performance addresses the service delivered by a system, a general definition is hard to establish due to the different application domains. Therefore, [Molyneaux \[\[Mol15\]\(#\)\]](#) explains that performance should always be defined from a user acceptance perspective ◆ *performance*

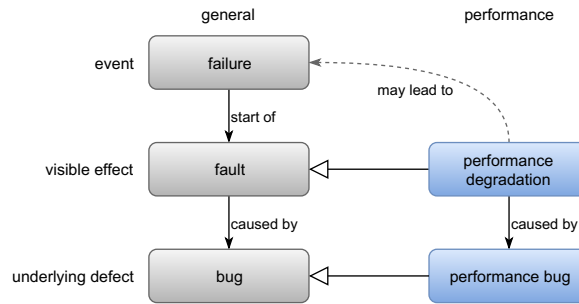


Figure 2.3: Relation of general terms used to describe system failures and performance-related counterparts. Arrows with a hollow triangle as the tip represent inheritance as in UML class diagrams.

for the system at hand. Nevertheless, a set of commonly used metrics has evolved, which often includes *response times*, *latency*, *availability*, *throughput*, and *utilization* [Mol15; Syd11; IHE15]. While the first three of them have a clear relation to the perception of a system from a user’s perspective, the last two represent internal workings of the software system. Molyneaux [Mol15] therefore differentiates between *service-oriented* and *efficiency-oriented KPIs* [Mol15, p. 2]. System metrics are either directly used to represent efficiency-oriented KPIs, or they are the basis for derived ones. Therefore, measuring the performance of a system involves both, system-related and user-related metrics. When a quantification and a potential agreement on the performance of a system is intended, performance is often replaced with a defined set of measurable properties that is then termed *quality of service (QoS)*. The agreement on a required level for these properties is furthermore called *service level agreement (SLA)*. Originating from the telecommunications domain [ITU08], these terms have been adopted by the software engineering and system operation communities [Dobo4; Wik17] where the meaning was extended to nonfunctional properties of software systems, thereby including performance.

*quality of service (QoS)* ◆

Performance is not a binary property. Hence, one usually cannot simply say that a system does not perform at all because the service, from a functional point of view, is still delivered. Instead, an impairment of a system’s performance is often called a *performance degradation* [e.g. She+09; Mol15; Koz10; IHE15; Avi+04; Foo+15]. The reasons for such performance degradations in the sense of software or hardware defects are called *performance bugs* [Rey+06; She+09]. It is important to note that a performance bug usually does not lead to a complete service outage. Given a performance bug, the system still delivers its functional service, but a performance degradation is the visible effect of such a bug. This distinction of origin and visible phenomena resembles the distinction between the general terms bug and fault shown earlier. Given these general terms, performance bugs can be seen as special cases of bugs and a performance degradation is a special type of fault. However, for performance bugs no direct coun-

*performance degradation* ◆

*performance bug* ◆

terpart to failures exists. As performance bugs usually do not lead to a service outage, no single event exists that describes the start of a service outage. Nevertheless, performance bugs can (usually gradually) lead to complete system failures [IHE15], for instance, if a memory leak results in processes being killed, or an increased delay leads to a timeout in a client subsystem. In these cases, the general terms apply again. Figure 2.3 on the facing page summarizes the relation of these concepts and the terms used throughout this thesis.

Similar to the general bugs, a change to a piece of software can also affect the performance of the (existing) software without the intention to do so. This is often called a *performance regression* [KBT05; MHH13; Ngu+12; Sha+15; ZAH12; JH15]: “Performance regressions are regressions [...] caused by the degradation of system performance compared to prior releases” [JH15].

performance regression

The effects of performance bugs on deployed software products differ from those of other types of bugs. For instance, Jin et al. [Jin+12] note that performance bugs are more costly to detect and hide longer before being noticed. However, performance bugs have a higher potential for successful recovery at system runtime without being noticed by end users because their visible effect is often only a performance degradation without complete service outage.

In the context of this thesis, performance is mainly analyzed from the technical perspective through system metrics as representatives of efficiency-oriented *KPIs*. Therefore, whenever I use the term *performance*, I will refer to its efficiency perspective except otherwise noted.



To better understand the state of [dependability](#) in current robotics systems, I have carried out an online survey.<sup>1</sup> The aim of this survey was to collect the impressions of robotics developers on the [reliability](#) of systems, reasons for [failure](#), and tools used to ensure successful operation and to debug in case of failures. Such an analysis is necessary to verify that the developed solutions address relevant problems [Ste13]. The survey focuses on software issues and software engineering aspects. Apart from general [bugs](#), [performance bugs](#) have been specifically addressed to understand their nature and their effects on systems. A considerable amount of work in this direction has been done in other computing domains such as high-performance computing or for cloud services [e.g. Gun+14; Jin+12; ZAH12]. However, in robotics such work is missing. To my knowledge, only Steinbauer [Ste13] presents a systematic study on general [faults](#) in robotics systems, but without specifically analyzing [performance](#) aspects.

I implemented the survey as an online questionnaire (following method advice from Gonzalez-Bañales and Adam [GA07]) and distributed it among robotics researchers using the well-known mailing lists euRobotics (euron-dist)<sup>2</sup> and robotics-worldwide<sup>3</sup> as well as more focused mailing lists. The detailed structure of the survey can be found in [Appendix A](#) on page 197. Please refer to this appendix for details on the phrasing of questions and permitted answers. Results presented in the following sections are linked to the respective questions of the survey.

In total, 61 complete submissions and 141 incomplete ones<sup>4</sup> were collected. 86 % of the participants were researchers or PhD candidates at universities, 7 % regular students and 7 % from an industrial context ([A.12.1](#)). Participants had an average of 5.8 years of experience in robotics (sd: 3.3, [A.12.2](#)). On average, participants spend their active development time primarily with software architecture and integration as well as [component](#) development, despite individual differences visible in the broad range of answers (cf. [Figure 3.1](#) on the following page, [A.12.3](#)). Other activities such as hardware or driver development are pursued only for a limited amount of time.

---

<sup>1</sup> An initial discussion of survey results was published in Wienke et al. [WMW16]. Parts of this chapter are based on this publication.

<sup>2</sup> <https://lists.iais.fraunhofer.de/sympa/info/euron-dist>

<sup>3</sup> <http://duerer.usc.edu/mailman/listinfo.cgi/robotics-worldwide>

<sup>4</sup> Incomplete submissions also include visitors who only opened the welcome page and then left.

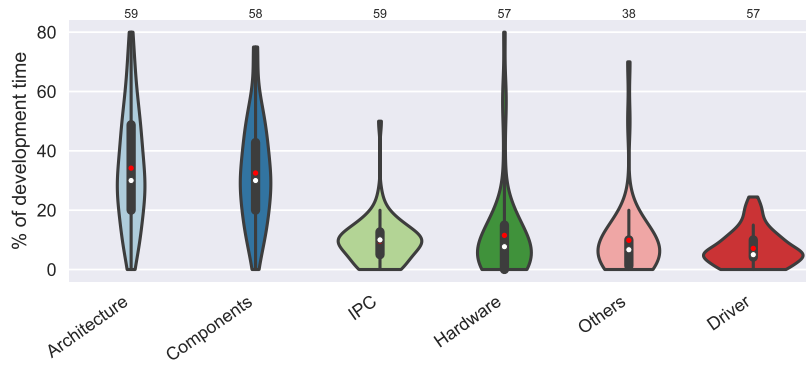


Figure 3.1: Development time spent by survey participants on different aspects. Individual answers have been normalized to sum up to 100%. Inside the violins, a box plot is shown with the white dot representing the median and the red dot the mean value. Numbers above the plot express the sample size, which differs as answers were optional.

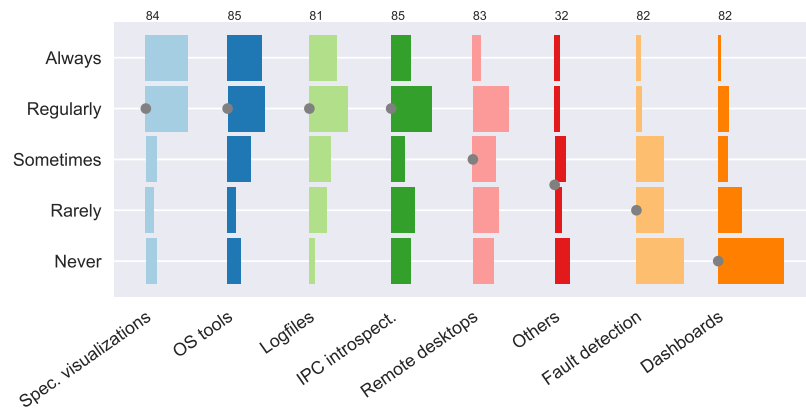


Figure 3.2: Usage frequencies for different categories of monitoring tools. For each category the answer counts are displayed as a histogram and the gray point marks the median value. Categories are ordered by median and, if equal, mean values. Numbers above the histograms express the sample size.

### 3.1 TOOL USAGE

A first set of questions tried to assess which software tools are used to monitor and debug robotics systems in general. For different types of tools, participants could rate on a 5 point scale from 0 (Never) to 4 (Always), how often the respective type of tool is used during usual development and operation of their systems. For general monitoring tools (A.2.1) the answers are depicted in Figure 3.2. According to the developers' opinion, special purpose visualization tools such as *RViz* [RViz] or debug windows for image processing operations are most frequently used to monitor systems. These are followed by low-level operating system level tools such as *ps* or *htop* and log files. Tools related to distributed systems such as utilities of the inter-process com-



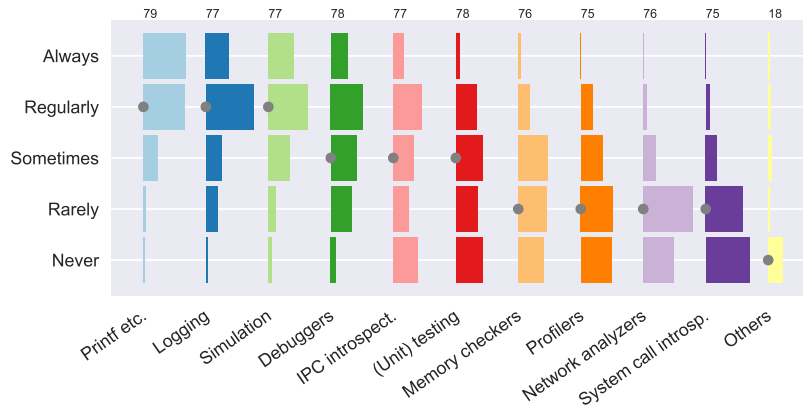


Figure 3.3: Usage frequencies for different debugging tools and methods.

munication (IPC) mechanism form the final category of tools that is regularly used. Remote desktop connections are used only sometimes. In contrast, autonomous fault detection methods and special dashboards for visualizing system metrics are rarely used, although such tools are well-established for operating large-scale systems with high dependability requirements.

A second question regarding monitoring tools asked for the exact names of tools that are used (A.2.2). The answers to this question are summarized in Appendix B.1 on page 207. The most frequently mentioned category of tools matched the previous question (visualization tools, most notably RViz [RViz]). These are followed by middleware-related tools, most notably the ROS command line tools and *rqt*, as well as operating system tools with *htop* and *ps* being the most frequently mentioned examples. Finally, manual log reading, remote access tools, custom mission-specific tools, and generic network monitoring tools such as *Wireshark* [WiSha] are used. Additionally, one participant also explicitly mentioned hardware indicators such as LEDs for this purpose.

Regarding tools used to debug robotics systems (A.3.1), participants often use basic methods such as `printf` or log files as well as simulation (cf. Figure 3.3). General-purpose and readily available debuggers are less often used than these basic methods. Unit testing is sometimes practiced and accepted in the robotics and intelligent systems community.

The actual tools being used have been summarized in Appendix B.2 on page 208 based on question A.3.2. Debuggers represent the most frequently mentioned category of tools with *GDB* [GDB] leading this category. Another frequently used debugging tool is *Valgrind* [Valg] for checking memory accesses. Besides `printf` debugging, other categories of used tools are middleware utilities, simulation and visualization (with *gazebo* being mentioned most often), and unit testing.

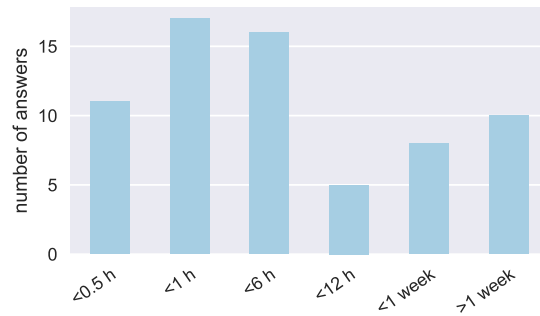


Figure 3.4: Participant answers for the observed [MTBF](#) in their systems.

### 3.2 BUGS AND THEIR ORIGINS

In a second set of questions I have addressed the reasons for and effects of bugs in robotics systems. Due to the limited availability of failure rates in robotics systems, one question asked participants for the [mean time between failures \(MTBF\)](#) they have observed in systems they are working with ([A.4.1](#)). As visible in [Figure 3.4](#), the answers form a bimodal distribution where one part rates the [MTBF](#) of their systems to be within the range of minutes to a few hours, whereas others indicate [MTBF](#) rates in the range of days to weeks. I have thought about different explanations for this discrepancy:

- The participants' systems differ in maturity.
- Answers with a higher [MTBF](#) include the system's idle time, despite an explicit note in the explanation of the question that the *operation* time is the basis for this number.
- Differences can be explained by the way people use debugging or monitoring tools in their systems. For instance, systems of survey participants who frequently use more advanced tools could be more reliable with higher [MTBF](#) rates. However, no significant relations exist in the data.

As for the first two hypotheses no data are available to validate them and the third one cannot be proofed using the survey results, the effective reasons for the bimodal distribution are unknown.

To generally understand why systems fail, participants were asked to rate how often bugs from a set of categories were the root cause of [system failures](#) ([A.4.2](#)). The categories have been selected based on related survey work from robotics and other domains [[Ste13](#); [Gun+14](#); [Jin+12](#); [McCo4](#)]. [Figure 3.5](#) on the facing page displays the results for this question. Hardware bugs represent the most frequent category followed by a set of categories representing high-level issues (configuration, coordination, deployment, IPC) as well as general logic bugs. Most of the high-level issues seem to be technical problems and not specification problems because specification issues only rarely cause failures.

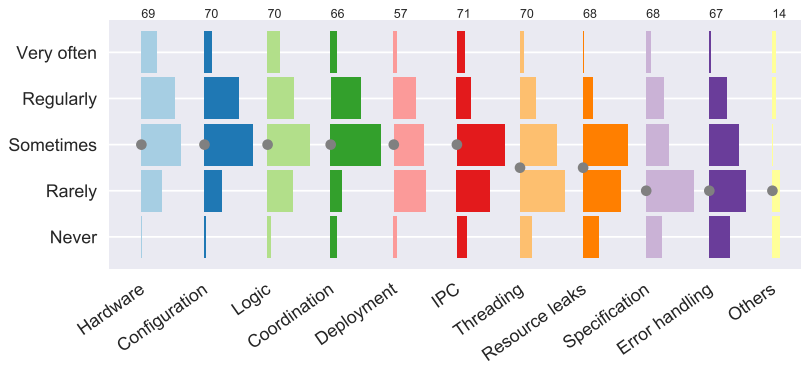


Figure 3.5: Frequencies of different bug categories being the reason for system failures.

Apart from the aforementioned categories, participants could describe further causes in text form (A.4.3). After removing items that relate to categories already presented in the previous question, answers can be summarized as a) environment complexity/changes (8 mentions) b) low-level driver and operating system failures (3 mentions) c) hardware configuration management (1 mention) and d) hardware limitations (1 mention). Appendix B.3 on page 209 shows the answers in detail as well as how categories have been assigned. In the survey, I explicitly excluded the (physical) environment as an origin of system failures because it does not represent a real defect in any component of the system. However, the results still show how important the discrepancy between intended usage scenarios and capabilities of systems in their real application areas is in robotics and intelligent systems.

### 3.3 PERFORMANCE BUGS

To understand performance bugs in robotics and intelligent systems, a dedicated set of questions was added to the survey. First, participants were asked for the percentage of bugs that affected **resource utilization** (A.5.1). On average, 24 % (sd: 17 %) of all bugs affected **system resources**. Participants also had to rate how often different system resources were affected by performance bugs (A.6.1). These results are visualized in Figure 3.6 on the following page. Memory, CPU, and network bandwidth are the most frequently affected **resources**. Network bandwidth can be explained by the distributed nature of many of the current robotics systems. These resources are followed by disk space. Countable resources such as processes or network connections are rarely affected. A question for further types of affected resources (A.6.2) yielded IPC-related **virtual resources** such as event queues and IO bandwidth in addition to the previous categories (cf. Appendix B.4 on page 210).

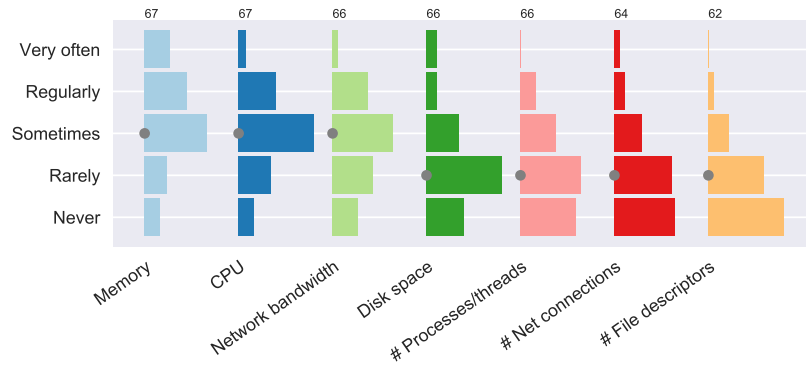


Figure 3.6: Frequency of bug effects on system resources.

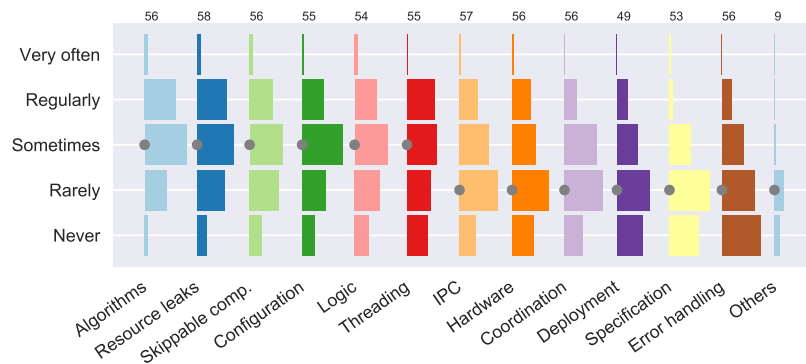


Figure 3.7: Frequency of reasons for performance bugs.

To get an impression of common causes for performance issues in robotics and intelligent systems, a question asked participants to rate how frequently different categories of root causes were the origin of performance bugs in their systems (A.7.1). The categories are the ones of the previous general questions on bug origins (A.7.1) extended with two items specifically targeting performance bugs: skipable computation, that is, unnecessary computation that does not affect the functional outcomes (based on the results in Gunawi et al. [Gun+14]) and algorithmic choices. Figure 3.7 depicts the results for this question. The most frequent reason for performance bugs is the choice of inappropriate algorithms followed by resource leaks and unnecessary computations. To my surprise, configuration issues are also among the frequent causes for performance bugs. When comparing answers to this question with the answers for origins of general bugs (A.4.2, comparison in Table 3.1 on the next page), most categories are less likely origins for performance bugs than for general bugs apart from resource leaks. Interestingly, no significant difference exists in the way threading issues affect performance bugs compared to general bugs.

CATEGORY	CHANGE
Communication	-0.28
Configuration	-0.51**
Coordination	-0.79****
Deployment	-0.71****
Error handling	-0.44*
Hardware	-0.98****
Resource leaks	0.47**
Logic	-0.44*
Others	-0.58
Specification	-0.46*
Threading	0.04

Table 3.1: Changes to the mean ratings for different categories being the origins of failures when comparing performance bugs to general bugs. A change of 1 would indicate a shift from one answer category to the next higher one. Significances have been computed using a Mann-Whitney U test.

### 3.4 BUG EXAMPLES

Finally, participants were asked to describe the bugs they had observed in their systems in detail. Two questions in this direction were asked with four sub-answers explicitly requesting a) the visible effects on the system, b) the underlying defect causing the bug, c) the steps performed to debug the problem, and d) the affected system resources. These questions were added to the survey to get an impression of the actual problems current robotics developers are facing in their systems and how they are addressed.

The first of these questions asked for a description of any type of bug participants remembered from their systems that is particularly representative for the kind of bugs frequently observed (A.9). In total, 21 answers were submitted for this question with a complete listing of the answers available in [Appendix B.5](#) on page 211. Most notably, 10 of the answers (48%) were related to basic programming issues such as segmentation faults or memory leaks, for instance caused by C/C++ peculiarities. 8 answers (38%) described an issue that can be classified as a performance bug. Issues related to the IPC usage or infrastructure were mentioned by 4 answers (19%). Also, 4 answers indicated bugs related to the coordination of the system (e.g., loops in the controlling state machines) of which 2 answers were related to unexpected environment situations. Also, 2 answers were related to timing aspects and 2 answers indicated that a bug was never or only accidentally understood and solved. Please refer to the tagging in [Appendix B.5](#) on page 211 for details on how answers were counted.

A second question asked participants to describe the most interesting bug they remembered in the same format. This was done to get an impression of which extreme types of bugs are possible in robotics systems. 14 participants answered this question and their answers are listed in [Appendix B.6](#) on page 219. In line with the previous question, programming problems related to low-level issues also represent the most frequently mentioned type of bugs with 6 answers (43%). Furthermore, 3 answers (21%) described bugs caused by driver or operating system problems.

Answers to both questions indicate that debugging of memory-management-related programming issues is often performed using established tools such as *GDB* [GDB] or *Valgrind* [Valg] – however – with varying success. One answer specifically mentioned that these tools are often not helpful for distributed systems.

### 3.5 SUMMARY

The presented results show that a great potential for improvements in the dependability of robotics systems still exists. With [MTBF](#) rates in the range of hours, a large part of the surveyed systems is far from being reliable enough for longer-term operations and work in this direction is needed, even if the majority of developers reached with this survey is working on research systems, which rarely end up in production use cases. Nevertheless, an appropriate level of dependability is required also in this domain to allow efficient experimentation and reliable studies. Still, monitoring tools that are specifically geared towards operating systems with high availability and reliability such as fault detection or dashboards for a quick manual inspection of systems states are only rarely applied in robotics. The survey does not offer answers why this is the situation. Reasons could include the overhead of deploying such approaches, which might not be feasible in smaller, short-lived systems; or the lack of knowledge about such approaches, especially as many robotics researchers do not have a background related to maintaining large-scale systems. Therefore, improving approaches and making them more easily usable is one promising direction to increase their adoption.

Regarding system failures and their origins, the quantitative results from this survey indicate that hardware issues are among the most frequent causes for failure. This contradicts the findings from Steinbauer [Ste13], which might be caused by the wider range of applications covered in this survey. Generally, system failures seem to originate more frequently from bugs occurring in high levels of abstraction such as coordination, deployment, or configuration and less often from component-internal issues such as crashes. Still, a majority of the requested bug descriptions for representative bugs dealt with such component-internal issues. One reason for this might be

that, although frequently being observed, such component-related issues are often noticed immediately and therefore are perceived as part of the development work and not as system failures. In any case, these issues are strikingly often caused by basic programming issues, often related to the manual memory management and syntax idiosyncrasies of C/C++. A general shift in robotics towards less error-prone languages with automatic memory management, clearer syntax, and better debugging abilities has the potential to avoid a large amount of bugs currently found during development and operations.

Generally, systems are often debugged using log files and `printf` instructions specifically placed for debugging. Participants have indicated that debuggers and memory checkers such as *Valgrind* [Valg] are used less often. The detailed bug reports show that these tools are applied frequently only to debug programming issues on the component level. Participants have also indicated that these tools cannot be easily used for other problems related to the distributed systems nature of current robots. Further work on debugging infrastructure respecting this fact might improve the situation. Finally, simulation seems to be an important tool for debugging robotics systems and explicit support for simulation-based testing and debugging might provide one future direction for more dependable robotics systems.

Regarding performance aspects, one quarter of the bugs found in current (research) systems can be classified as performance bugs. In the descriptions of representative bugs even more than one third of the answers was performance-related. Therefore, specifically addressing such issues is not only a niche but instead provides the potential to avoid a large amount of failures in the future. The survey has indicated that performance bugs are significantly less often caused by high-level aspects such as coordination or deployment and also by hardware issues. Therefore, addressing them on the level of system components as intended in this thesis is a valid approach that will be able to capture many of the issues present in current systems.

⊙ *performance bugs can be addressed at the component level*

### 3.6 THREATS TO VALIDITY

The survey results represent the opinions and memorized impressions of interviewed developers, not objective measurements of the real effects. As such, results may be biased. However, general tendencies derived from the results should still be valid because a complete misassessment is unlikely across all participants.

Due to the distribution of the survey via primarily research-related mailing lists, results are representative only for systems developed in this context and cannot be generalized towards industrial, production-ready systems.

The categories used in questions regarding the frequencies of bug origins may have partially been hard to distinguish from each other.

Sometimes, ratings might thus be blurred between categories. When possible, the conclusions drawn from the survey have been based on a grouping of multiple categories to mitigate this effect.



Chapter 2 has introduced **dependability** as a global aim of technical systems, how dependability can be achieved in such systems, and how **performance** issues can affect it. Moreover, the survey presented in the previous chapter has shown that **performance bugs** are a common problem in robotics system. **Resources** form a major aspect of performance and as visible in the survey results, an unexpected **utilization** can severely impact dependability. Therefore, *awareness* of availability and utilization of **system resources** is required for improving the dependability of technical computing systems. Awareness, in this context, is required along the whole lifecycle of technical systems – including planing, development and operation [SS11] – because performance issues can originate or appear in any of these stages [Mol15, p. 7]. As dependability is a requirement for any technical system, a multitude of methods and approaches have been developed that enable such an awareness at different stages of the lifecycle of computing systems. Systems that incorporate such techniques are often termed to be *resource-aware* or to incorporate *resource awareness*. However, no accepted definition exists for this term and usage varies. Often, *resource awareness* is used to indicate that a piece of software or an architecture adapts dynamically to the available resources [e.g. Krö17, p. 6; PKK12; Zha+15; Men+12]. This definition is limiting, as more aspects of a system and the surrounding development and operations require knowledge about resources. Therefore, I will use the term *resource awareness* to describe an overarching concept that comprises the software itself as well as the software development process and life cycle. Resource awareness, in this broader understanding, is the idea of uncovering the resource utilization of a system and making this information usable, either for developers during software development, or for supervision at operation time by tailored software or human operators. In most existing cases, resource awareness is a result of intrinsic properties of pieces of software, which – for instance – were constructed by following a resource-aware programming paradigm. Such an approach results in an obligation for programmers to explicitly incorporate the necessary techniques – which they need to know and understand in detail – into their programs. I will call this *implementation-level resource awareness*. On the other hand, resource awareness can also be achieved through generic means of the software infrastructure and testing landscape without requiring modifications to the actual domain-specific program implementations. In the following, methods subscribing to such an approach will be termed to

resource awareness

implementation-level  
resource awareness

*framework-level  
resource awareness*

implement *framework-level resource awareness* and this work focuses on such methods. Although framework-level methods are restricted in the effects they can achieve compared to implementation-level ones, their application is much easier, which follows the essential requirement of this work to introduce methods that can be applied in existing systems with the least amount of overhead. This overhead comprises the necessary time to install and maintain solutions as well as their own demands for system resources [Kha+15; SW05]. The reduction of effort tries to avoid potential arguments against using a solution for resource awareness in the first place.

In the following sections I will summarize existing work related to the concept of resource awareness. I will give an overview of general resource awareness methods and highlight approaches that implement framework-level resource awareness.

#### 4.1 RESOURCE AWARENESS IN COMPUTING SYSTEMS

Before going into the details on how resource awareness is currently realized in robotics, I will first present related approaches in other software engineering domains.

##### 4.1.1 Server infrastructure operation

Generally, the operation of server infrastructure for frequently used applications imposes high demands on dependability and therefore a variety of resource awareness methods originates from this domain. As applications are constantly changing, tools for this purpose need to be easily applicable and general-purpose. Many framework-level monitoring and alerting systems have been developed that respect these requirements and are readily available. Common implementations of such systems (for instance *Prometheus* [Prom]) continually acquire KPIs of the system and running applications, collect them for later retrieval, visualize them using dashboards, and alert in case collected metrics violate specified checks [Tur16]. As such, monitoring systems contribute to the resource awareness principles by making software operators aware of their systems' resource utilizations. A variety of commercial and open-source implementations of such tools exists. Aceto et al. [Ace+13] and Fatema et al. [Fat+14] provide extensive surveys on the available tools from the perspective of operating cloud computing systems. Depending on the scale of the system and the intended level of detail for monitoring, such monitoring systems may have a noticeable resource utilization. Some systems address this by sub-sampling the monitoring data, for instance, Sigelman et al. [Sig+10], whereas Meng et al. [Men+12] propose to also make the monitoring system itself resource-aware.

Monitoring approaches are also available for the underlying network and data distribution systems of complex [distributed systems](#). For example, the PIP framework [[Rey+06](#)] allows expressing expectations regarding different aspects of performance, including resources, using a declarative language. These expectations are checked at system runtime and violations are reported. PIP has been evaluated against different data dissemination and multicast systems.

Another common technique is *load balancing*, which aims to distribute usage load – for instance, from requests on a web server – across multiple parallel instances to prevent overloading of individual instances [[Bou01](#)]. This technique tries to mitigate [resource starvation](#) on the individual nodes. Many off-the-shelf load balancers for diverse applications are available and therefore contribute to framework-level resource awareness. Examples include solutions for web servers such as *Apache* [[Apache](#)] or *nginx* [[nginx](#)], the standalone *HAProxy* [[HAProxy](#)], or even on the network level in routers [[Cis15](#)].

#### 4.1.2 *Cloud computing*

Apart from generic infrastructure operation tools, cloud computing as a prominent discipline has developed methods that are more specific for the [platform as a service \(PaaS\)](#) and [infrastructure as a service \(IaaS\)](#) idea with virtualized hardware resources that can be acquired or released on demand. For instance, Malik et al. [[MHC12](#)] describe a framework to automatically select appropriate cloud providers for executing applications based on resource and [QoS](#) properties with the aim to maximize the performance of the application. Johnsen et al. [[JST12](#)] addresses the comparable problem of deciding when to acquire and release resources by means of a model-based approach. After modeling applications and cloud providers using the ABS modeling language, developers can execute the models to simulate different scenarios before the actual development starts. This idea matches with the general aims explained by Hähnle and Johnsen [[HJ15](#)] who argue that resource awareness for cloud computing should already be addressed during the design phase because this will lead to applications that incorporate resource awareness on the implementation level.

#### 4.1.3 *Model-based performance prediction*

Along these lines of Hähnle and Johnsen [[HJ15](#)], model-based approach for forecasting the performance and resource utilization of systems form one important method to incorporate resource awareness in the design phase. A prominent example targeting component-based systems is the Palladio Component Model (PCM) [[BKR09](#)]. In PCM [components](#) are described in terms of their behavior and as-

sociated resource utilizations; their assembly to systems; deployment targets and available resources; as well as request-based system usage scenarios specified as user-role specific models. PCM provides different mechanisms to simulate and validate the actual resource utilization and performance of systems modeled this way. It has also been extended to event-based systems [Rat+14] with explicit model elements to specify event-based component connections including queuing behavior. This extension makes PCM an interesting candidate to model event-based robotics systems in case a model-driven approach is intended. Another solution for predicting the resource utilization of a system is presented in Jonge et al. [JMC03] and Muskens and Chaudron [MCo4]. Here, the Robocomp **component model** describes component-based systems using multiple task-specific models. Operations are annotated with the expected resource utilization and a method is presented to derive the effective demands for chained service calls.

Others have presented resource forecasting methods based on standardized modeling languages. For instance, Garousi et al. [GBLo9] show an approach to predict the resource utilization in distributed real-time systems based on **Unified Modeling Language (UML)** sequence diagrams with custom annotations. Comparably, Tribastone and Gilmore [TG08] use the standardized **Modeling and Analysis of Real-Time Embedded Systems (MARTE)** [OMG11] profile for UML to annotate resource information. These UML models are then transformed into PEPA [GH94] models for analysis. The same authors also proposed another method that uses Layered Queuing Networks as the target formalism instead [TMW10].

All the aforementioned methods require system models that contain the necessary resource utilization and capacity information. Creating such models and keeping them up to date with the evolving system requires a considerable amount of work, which might prohibit the application of these methods. To address these issues, Kounev et al. [Kou+10] describe an approach how the aforementioned PCM models can be maintained at runtime after their initial construction. Based on these models, a prediction of a system's performance at runtime is intended. Caban and Walkowiak [CW15], in contrast, describe how existing choreography or orchestration descriptions of web services can be used for simulation-based prediction.

For a systematic survey on further model-based performance forecasting systems, please refer to Koziol [Koz10]. Generally, model-based techniques can be classified as framework-level resource awareness implementations, because the same methods can be applied to different models and all parts of a system. However, in case appropriate models do not exist, their construction and maintenance usually conflicts with the aim to limit the required effort for applying a resource awareness method.

## 4.2 RESOURCE AWARENESS IN ROBOTICS

Despite being sometimes neglected, resource awareness has also been addressed in robotics. Some approaches used here have been adopted from other disciplines but robotics has also developed own solutions. In the following, I will give an overview of those.

### 4.2.1 *Space robotics*

Probably the first systematic architectural approach to specify and exploit knowledge about the utilization of system resources in robotics has been formulated in the *CLARAty* architecture [Vol+00; Vol+01], which has been used for multiple rovers for planetary exploration missions [Nes07a]. Inside *CLARAty*, each functional component has the duty to provide predictions about its resource utilization for a given task on request from the planning layer, which can then incorporate the resource utilization in generated plans. Prediction is performed inside the functional components to locate such knowledge close to the actual implementation. Resource predictions can be requested with varying levels of accuracy to control the required time and processing power for generating the prediction. During plan execution, predictions are compared with actual utilization values and repair actions are take if necessary [Vol+00]. The *CLARAty* approach – despite being enforced by the architectural framework – requires implementation-level changes in each component to provide the required resource predictions. As planetary exploration exceptionally demands for well-managed resources, further works for resource awareness exist here, where awareness for system resources is often a side effect of maintaining physical resources such as power. For instance, Castano et al. [Cas+06] present another architecture for a Mars rover that explicitly include resource predictions in the planning process. Ai-Chang et al. [AiC+04] describe the *MAPGEN* mixed-initiative planner for the Mars rover that is used to plan the execution of scientific experiments, initiated either through the rover software itself or by a remote operator. The planning tool *APGEN* allows precomputing CPU usage profiles before sending a command to the rover.

### 4.2.2 *Cloud robotics*

Another robotics research area where resource awareness is tackled is cloud robotics. Here, the idea is – among others – to offload computations onto cloud computing services to overcome the resource restrictions of the robot platforms [Wan+16]. In this domain, resource demands need to be known to allocate the required cloud resources and also to decide whether to upload a task to the cloud or to process it locally because offloading requires network bandwidth and

might lead to communication latencies and errors. This trade-off is termed the *resource allocation and scheduling problem* [Wan+16]. Hu et al. [HTW12] propose an optimization approach to solve this problem, which is embedded in their software architecture.

#### 4.2.3 Resource-aware algorithms

Resource awareness can also be realized on an algorithmic level. A common implementation-level approach, which implicitly enables resource awareness inside individual computations, is the use of *anytime algorithms* [DB88; Zil96]. “Anytime algorithms are algorithms whose quality of results improves gradually as computation time increases” [Zil96]. The iterative processing allows interrupting such an algorithm at any given moment at which the latest approximate results are returned [Zil96]. This way, the planing layer of a system can decide on the desired time or resources a computation should utilize. Anytime algorithms have been used for many applications in robotics.<sup>1</sup> Among others, these include localization and *simultaneous localization and mapping (SLAM)* [Fox+99; NR11], mobile robot path planning [Thr+00; Lik+05; Kar+11], computer vision machine learning tasks [Mör+10; ZVo8; Uen+06], execution planning problems [TLI11; Yu+15], and modular robot reconfiguration [Dut+14]. Furthermore, many general algorithms used as the basis for robotics applications can be formulated as anytime algorithms [e.g. PGT03; Pai+15; MY13].

A different and more integrated approach that couples algorithms to the available resources is *invasive programming* [Pau+14]. With the aim to efficiently use many-core systems for robotics, the operating system level in invasive programming differs from traditional PC-based systems. Resource management is shifted from the operating system to the applications, which have to explicitly request resources and handle situations in which less resources are granted than requested. This brings awareness about such restrictions closer to the actual implementation of algorithms and programmers have to explicitly handle such situations. Paul et al. [Pau+14] and Kröhnert [Krö17] show how this programming model can be used to implement resource-aware computer vision algorithms.

#### 4.2.4 Resource-aware planning and execution

Another robotics area where resource awareness has been addressed before is planing and execution. Here, the idea is that while planing actions and executing them, a robot should consider the available resources and their expected utilization to prevent overloading the system and degrading the *QoS*. For this purpose, a planner can make

<sup>1</sup> Sometimes, approaches realizing anytime algorithms do not even explicitly mention that they implement this idea.

use of the resource awareness already implemented in the system's components. In addition to the space robotics work already discussed in [Section 4.2.1](#) on page 35, further more general resource-aware planning and execution approaches exist.

Park et al. [PKK12] present an architectural framework-level solution to the planing problem. In their system, first a task-based plan for a given goal of a robot is generated. The software architecture provides one or multiple sub-architectures and realizing components for each of the tasks. The effective architecture of the system is then rearranged by selecting the sub-architectures and components that best fulfill the task requirements while utilizing the least amount of resources, in their examples CPU and memory. Additionally, optimization is used to decide which components need to be active at which time to further reduce resource utilizations. For this purpose, a static model of resource utilization is assumed (a constant amount of resources is utilized only if a component is active), which has been generated beforehand through simulation.

Another framework-level approach in the planing and execution area is described in Kröhnert et al. [Krö+14]. Here, the state-chart-based execution of the robot as well as its perceived state of the world are exposed to a prediction component, which aims to predict further actions of the robot including their resource requirements. For this purpose, Markov chains and static resource utilization values per state are used (CPU and memory). The Markov chains are trained by observing executions of the robot whereas the resource values have been hand-crafted beforehand based on experience and insights into the algorithms realizing the states. Results of the prediction are not used for generating or adapting plans.

The aforementioned approach has later been integrated into the concept of *speculative resource management* [Krö17] where the idea is to manage resources based on a prediction starting from the current state of the robot system. Although Kröhnert [Krö17] also presents different resources-aware algorithms (cf. [Section 4.2.3](#) on the facing page) the integration between the prediction and such algorithms as building blocks for resource-aware systems is left for future work.

#### 4.2.5 *Infrastructure monitoring of robotics systems*

Robotics systems are usually larger systems of interconnected, potentially distributed, components. To effectively monitor such systems at runtime, special monitoring software is required, which has been introduced for general systems in [Section 4.1.1](#) on page 32. Interestingly, not many of the concepts from server operations and infrastructure monitoring made their way into robotics and are actively used there, although solutions commonly realize framework-level resource awareness.

In the ROS [Qui+09] ecosystem, the Advanced ROS Network Introspection (ARNI) [BW14; BHW16] provides a system to introspect a running ROS system and acquire runtime information about it. Besides its main purpose to collect and visualize information about the message flow inside the distributed system, ARNI also acquires information about system resources such as CPU, memory, and the GPU for hosts and individual ROS nodes. It further allows visualizing them using an interactive dashboard. Additionally, basic constraints such as range checks on the measures properties can be defined and violations will be reported by ARNI.

Another example of a monitoring approach for ROS-based robots can be found in Monajjemi et al. [MWV14]. Here, the *Drums* framework provides low-level system resource monitoring on the host and component level. The basic concept of the framework is that a framework-specific adapter exposes the system-level resources such as PIDs or sockets of components such as ROS nodes to allow the low-level monitoring. Based on this information, *Drums* instruments the running system with monitoring processes, which acquire resource utilization information using means of the operating system. The acquired **system metrics** are then processed and exposed to a human operator using standard solutions available from the infrastructure monitoring domain (*Graphite* [Graph] in this case).

#### 4.2.6 Model-driven approaches

Also, for robotics few model-based approaches to resource awareness exist. In case models are available, these model-driven methods are representatives of framework-level resource awareness.

Lotz et al. [LSS11] describe a generic pattern how to integrate the aforementioned monitoring capabilities into component-based robotics systems constructed using **model-driven software development (MDSD)** principles. They propose to instrument each component with a *black box* that realizes several software sensors. These software sensors provide monitoring data described by profiles. The black box optionally exposes an interface to the user-code inside the component to acquire processing details. Sensor data are exposed through an additional diagnosis port in each component and can be used by service requesters. The code generation of the model-driven approach is responsible of generating the black box instrumentation into each component. In the provided examples, component states and communication information are reported using this technique. However, the presented approach would also allow generating an additional software sensor for reporting resource utilization information into the black box. As no interface has to be exposed to the user-code, this could realize framework-level resource awareness.



In Steck and Schlegel [SS11], the same authors further describe an extension to their *SmartSoft* MDSD framework for developing robotics applications. The extension enables users to attach desired QoS properties for communication patterns and realtime requirements to the model. During model-transformation this information and additional knowledge about the capabilities of the target platform can be used to verify the required constraints and to perform schedulability analysis.

#### 4.3 SUMMARY

The review of related resource awareness approaches has shown that the topic is generally accepted and that methods exist for different aspects of resource awareness. However, in robotics, these methods are either on the implementation level, which requires that developers actively realize or apply them (e.g., resource-aware algorithms, planning), or the methods that follow the idea of framework-level resource awareness are limited to basic functions such as monitoring and make strong assumptions (e.g., constant resource utilization). Model-driven methods could be an option. However, only few of them have been proposed and in case model do not exist, the effort resulting from their construction only for the purpose of achieving resource awareness usually prohibits the application. Thus, a need for further approaches with broader scopes becomes visible that enable resource awareness without requiring deep changes to the systems, their components, and the assorted development workflow.



## Part II

### TECHNOLOGICAL FOUNDATION

This part introduces the technological foundation the developed methods build upon. Apart from an introduction to state of the art development methods for current robotics and intelligent systems, the set of technologies that was used in this thesis will be presented. Finally, this part also explains how information about the current resource utilization is acquired as a prerequisite for the work in further parts of this thesis.



Nowadays, any robotics system that does not merely serve as limited proof of concept is usually a complex software system with a multitude of different functionalities and technologies constituting the intended system behavior [Nes07b]. Therefore, robotics is a multi-disciplinary problem and most systems combine software developed by different authors, each being experts in their discipline. Thus, robotics faces similar software engineering challenges as other domains with comparable complexity and diversity [BS09]. Additionally, due to the experimental nature of systems in many robotics areas and their research background, systems and their individual parts are constantly changing to answer new research questions. This implies that systems have to be easily adaptable while still maintaining a sufficient level of isolation for the different domains experts. Reuse of software parts is an essential property needed to fulfill these requirements and *component-based software engineering (CBSE)* [HC01; CL02; Szy03] is a common approach to realize this. CBSE has been widely adopted and accepted in the robotics community.

◆ *component-based software engineering (CBSE)*

One of the premises of the work pursued in this thesis was to create solutions that are applicable to a wide range of systems without high overheads that would prevent their application. As such, component-based robotics systems are a natural target for this work. Therefore, I will give a short introduction of CBSE and related terms and concepts in the following paragraphs. Then, I will briefly present existing component-based frameworks from the robotics domain to show the importance of CBSE for current robotics development. The remaining chapters in this part of the thesis will then introduce the specific frameworks and techniques used on which the developed *framework-level resource awareness* methods build.

### 5.1 COMPONENT-BASED SOFTWARE ENGINEERING

CBSE represents the idea to split up a large software engineering problem into smaller, well-separated parts, which – when composed – create the solution to the problem. These parts are formed following separation of concerns and one intended effect is their reusability in other contexts. Moreover, the individual parts can be developed independently and implementations can be changed to support maintenance and development of component-based systems.

These *parts* of software are called *components* and due to the wide adoption of the CBSE idea, many definitions of CBSE and its consti-

◆ *component*

tuting concepts exist. A good review of the different definitions of the term *component* can be found in Crnkovic et al. [Crn+02] where a minimum consensus is derived. It states that “a component is a unit of composition, and it must be specified in such a way that it is possible to compose it with other components and integrate it into systems in a predictable way” [Crn+02]. Components are connected to other components via specified and documented interfaces and executable code implements these interfaces [Crn+02]. Apart from the basic definitions as a composable unit, other definitions put more emphasis on the functionality a component represents. For instance, IEEE 24765 [III10] defines a component (among other potential definitions) as a “set of functional services in the software, which, when implemented, represents a well-defined set of functions” [III10]. On the other hand, some authors put more emphasis on the ability to deploy and update components at runtime, for instance Crnkovic et al. [Crn+02]: “components can be composed at run time without the need for compilation” or Lewis and Fowler [LF14]: “a component is a unit of software that is independently replaceable and upgradeable”. In any case, component reuse largely depends on a specification of nonfunctional properties and the documentation of design and implementation aspects [Crn+02].

*component interface* ♦ Third parties access a component’s functionality via *component interfaces*. These are “access points [...] [which] allow clients of a component, usually components themselves, to access the services provided by the component” [Szy03, p. 42]. The component interface usually “names a collection of operations and provides only the descriptions and the protocols of these operations” [Crn+02].

*component model* ♦ To achieve executable and reusable components, a stricter and more technical specification of components, their interfaces, and their execution context is required. Such a specification is called *component model*. It “defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution, and deployment” [WSo1]. Actual frameworks that realize a component model and which allow executing components specified for the

*component platform* ♦ respective component model are called *component platform* [Szy03, p. 44] or *component model implementations* [WSo1]. Different examples for established component models exist, for instance within the Common Object Request Broker Architecture (CORBA) [II12], the Microsoft Component Object Model (COM) [Mic17], or JavaBeans [Sun97].

Besides a suitable specification and documentation of components, reuse is to a large extent determined by how accepted a component model is [WSo1]. As interoperability between different component models is usually not supported, components can be reused only inside the model they have been created for and therefore the market share of a component model determines whether it is worth to develop components for a certain model [WSo1].

## 5.2 CBSE AND DISTRIBUTED SYSTEMS

Even though CBSE provides a general principle how applications can be combined from reusable parts, the granularity at which the separation into components takes place can differ and even multiple levels of granularity may be used in a single application. On the lower granularity levels, CBSE can be used to compose single executable applications based on multiple in-process components. On the other end of the scale, large-scale *distributed systems* communicating via network links can be constructed following the CBSE approach. Individual components of such an architecture might internally use CBSE principles on finer levels of granularity. For the course of this thesis, I will focus on the latter case of distributed systems constructed using CBSE principles, which are common in robotics nowadays (cf. [Section 5.3](#)). As distributed systems need to communicate via network links, appropriate transport mechanisms, protocols, and data formats are required to enable the communication between components potentially written in different programming languages. Therefore, component platforms commonly include a *middleware* layer, which provides an abstraction of the heterogeneity of the underlying technology in distributed systems [Cou+12], especially regarding the message passing between distributed components through hiding the distribution [Kra09, p. 1–4]. Middlewares in distributed systems can be seen as “the software layer that lies between the operating system and the applications on each site of the system” [Kra09] with the aim to make application development easier [Kra09].

## 5.3 CBSE IN ROBOTICS

As initially stated, CBSE has been widely adopted in the robotics community within recent years. A first idea for the acceptance can be established by analyzing the number of publications regarding this topic. [Figure 5.1](#) on the next page shows the percentage of publications per year that contain the keyword *robotics* as well as each of the other keywords shown in the plot. I have acquired these numbers by scraping the scientific search engine Google Scholar<sup>1</sup> using *academic-keyword-occurrence* [ACO]. The estimated number of publications per year returned by Google Scholar for the compound query `+"robotics" +"<keyword>"` has been divided by the numbers of publications in that year for only the term *robotics* (query `+"robotics"`) to normalize for the general growth of the robotics domain. Although these figures are far from perfect (Google Scholar only estimates the total count and the search includes fuzziness), they show that – in relation to other terms and techniques – CBSE and the less specific but related term *middleware* are active topics in robotics research,

<sup>1</sup> <https://scholar.google.com>

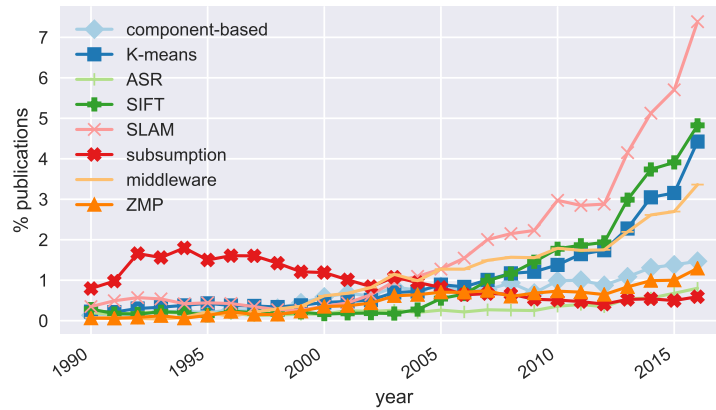


Figure 5.1: Percentage of robotics publications per year containing different keywords. The dataset was generated by scraping Google Scholar using *academic-keyword-occurrence* [ACO].

which are still gaining attention. Despite being less used in publications than long-established navigation- and vision-related terms, or classical algorithms (*SLAM*, *SIFT*, and *K-means*), these software engineering aspects are more frequently mentioned than common terms from speech recognition and motion research areas (*ASR* and *ZMP*). Hence, they are mid-table and far from being irrelevant to robotics research.

Based on the CBSE concepts, various robotics frameworks have been developed in the last 20 years [Bru+13]. In this context, the terms *framework* and *middleware* are often used interchangeably. To understand the relevance of different robotics frameworks, I have taken the list of component-based ones from Bruyninckx et al. [Bru+13] and acquired citation counts from Google Scholar for the initial publications introducing each framework (cf. Figure 5.2 on the facing page). These initial publications are in detail: Quigley et al. [Qui+09] (ROS), Bruyninckx [Bru01] (OROCOS), Ando et al. [And+05] (OpenRTM), Metta et al. [MFNo6] (YARP), Fleury et al. [FHC97] (GenoM), and Jang et al. [Jan+10] (OPRoS). I have excluded Proteus [MP08], which was a research consortium that did not result in an actual framework realizing middleware functionality, and SmartSoft [SW99] because it has been cited less than 100 times. I have added the citation counts for other common robotics software products such as the PCL 3D perception library [RC11] and the Gazebo simulator [KH04] to the plot to provide reference points.

The plot visualizes that the ROS [Qui+09] middleware dominates CBSE in robotics with by far the highest number of citations. ROS is an open-source framework for distributed robotics systems with support for multiple programming languages and operating systems. In ROS, systems are constructed as a graph of *nodes* (components), which are usually executed as individual operating system processes. They communicate via a custom network protocol in a peer-to-peer

relevance of different  
robotics middlewares



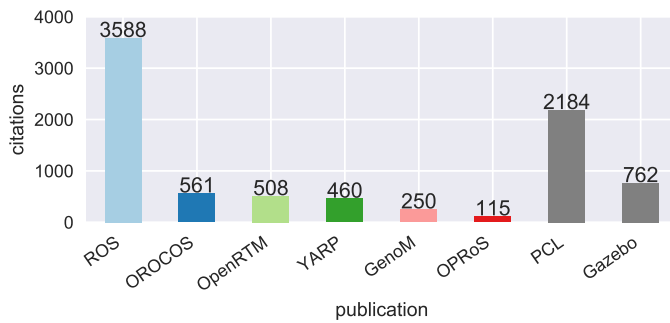


Figure 5.2: Citation counts for initial publications of component-based robotics frameworks at August 7, 2017.

fashion using defined communication patterns. ROS implements a publish-subscribe pattern where components distribute messages to multiple interested receivers via named *topics*. Moreover, a [remote procedure call \(RPC\)](#) pattern implementation exists. The data types used for communication are described in a custom [interface description language \(IDL\)](#).

A considerable ecosystem has evolved around the ROS core, including device drivers for different robotic platforms; common robotics software components for simulation, motion planning, navigation, interaction, or visualization; and tools to introspect and debug the system at runtime or offline. It is reported that ROS is used by at least 186 different institutions [ROS14] and others have postulated an exponential growth of the ROS ecosystem [Cou11]. ROS is therefore a good example for the component market effects described in [Section 5.1](#) on page 43 and its market share seems to be important enough that other frameworks started to interface with ROS [Cou11].

The OROCOS framework [Bruo1] is the second-most cited component-based robotics framework and particularly suitable for real-time motion control applications. Within the OROCOS project, the *Real-Time Toolkit (rtt)* [RTT] provides a component model and a component platform to execute components in a real-time context. rtt components are implemented in C++ and use data flow ports as the primary communication mechanism. The data types are realized as plugins called *typekits*, which can be created from ROS IDL files or from C++ headers. OROCOS components can be co-located inside real-time processes as individual threads. IPC between different real-time processes is realized by different transport plugins, including a CORBA and a custom message queue transport. A limited set of OROCOS components is distributed directly with the base libraries. Additionally, motion planning libraries are tightly integrated with the ecosystem. OROCOS can interface with ROS using the *rtt\_ros\_integration* [RRI] packages.

Another framework for component-based robotics engineering is *OpenRTM-aist* [And+05; ASKo8], a Japanese effort to provide a common ground for robotics development. OpenRTM-aist is based on CORBA and therefore reuses a well-defined component model. Components communicate using data flow ports (publish-subscribe) and service calls following the RPC pattern. Furthermore, components have configuration ports that allow controlling parameters at runtime. Ports and their data types are declared using the CORBA IDL.

A component-based framework comparable to OpenRTM-aist is *OPRoS* [Jan+10]. OPRoS provides a well-defined component model and an execution engine as a component platform. Communication between components is network-based. In contrast to OpenRTM-aist, transport mechanisms are flexible and besides CORBA, others have been implemented. Components with their ports and data types are declared using an IDL. OPRoS places emphasis on providing an integrated development environment for creating and maintaining components through Eclipse-based tools.

The *YARP* framework [MFNo6] connects multiple operating system processes (acting as components) to a distributed robotics system. Such processes use *ports* to exchange data. Ports implement a publish-subscribe pattern which is realized through different transport implementations respecting varying requirements. For instance, a [transmission control protocol \(TCP\)](#) based transport enables reliable communication while a multicast transport is more efficient at distributing the same messages to multiple receivers. Ports are named and can be connected programmatically or at runtime using a command line tool. In contrast to the aforementioned frameworks, YARP does not use an IDL-based approach for data types. Instead, *bottles*, which are dynamically allocated arrays of flexible size and included data types, are exchanged between processes. Comparable to ORO-COS, YARP can also interface with ROS.

*GenoM* [FHC97; FHM12] is a framework for building real-time robotics systems as a graph of modules. These modules are generated from user-defined executable C code for the implemented algorithms and a declaration of the properties of each module. A generator creates a compilable project where each module is executed as a separate operating system process. Modules offer *services* that are the callable actions following an RPC pattern. Additionally, a publish-subscribe pattern is available through *posters*. Each module provides such a memory that is only writable by itself but world-readable. Instead of other frameworks, communication between components is realized through shared memory and not via the network. GenoM seems unmaintained since several years.

For an overview of further robotics middlewares with a more detailed comparison, please refer to [Iñi+12].

## 5.4 PATTERNS IN COMPONENT-BASED ROBOTICS SYSTEMS

For the presented robotics middlewares that are still actively used and developed,<sup>2</sup> a common architectural style can be observed: components usually form individual operating system processes and communicate using the network. Only in case of strong performance requirements or real-time functionality, components are combined inside a single process (for example in OROCOS). Data types used for the network communication are usually defined in an IDL (except in YARP) and the frameworks provide tools to introspect and manipulate systems at runtime for logging and debugging purposes.<sup>3</sup> Additionally, the aforementioned robotics middlewares usually provide a set of predefined *communication patterns* that guide and restrict the design space for the component interface [Scho06].

Regarding the types of components that exist in robotics systems, Brugali and Scandurra [BS09] have identified three distinct categories based on the position on the components inside the system architecture. *Horizontal components* “provide functionality to a variety of applications that may implement totally different use cases” [BS09]. Examples include libraries for mathematical computations, device drivers, or simulators. *Vertical components* “capture [...] know-how in specific functional areas such as kinematics, motion planning, deliberative control, and address the requirements of target application domains such as service robotics, space robotics, or humanoid robotics” [BS09]. Most of the reuse in robotics is achieved on this level [BS09]. Finally, *application components* are at the top of the software architecture and orchestrate the functionality provided by the rest of the system such that the system’s mission is accomplished. So, they are tightly coupled to the target scenario and hardly reusable.

The operating system processes in frameworks such as ROS or YARP usually execute individual vertical components or application components. Architectures following such a style, where applications are developed “as a suite of small services, each running in its own process and communicating with lightweight mechanisms” [LF14] are also called *microservice architectures* [LF14]. Processes in such an architectural style are “built around business capabilities and independently deployable” [LF14], which resembles separation of concerns, and the processes “may be written in different programming languages” [LF14]. Such *microservices* are components that are called out of process by their clients [LF14] using IPC mechanisms. Overall, microservice architectures “are [...] about being able to very rapidly iterate” [Bri16]. Such an architectural style meets the requirements of robotics research, where experts from different domains use different

◆ *horizontal component*

◆ *vertical component*

◆ *application component*

◆ *microservice architecture*

◆ *microservice*

<sup>2</sup> This means all presented ones except GenoM.

<sup>3</sup> Examples for data logging: *rosviz* [ROSb], *yarpdatadumper* [YarpDD]. Examples for runtime introspection: *OROCOS TaskBrowser* [OTB], *rtshell* [RTsh] (OpenRTM-aist), *rostopic* [ROSt].

programming technologies and a high level of flexibility is required to fulfill experimentation on different system levels at a fast pace. Although the term *microservice architecture* has not yet received wide attention in the robotics community (I have found only one source that explicitly classifies ROS systems as microservice systems [WC17, p. 296]), many systems in research robotics are structured this way.

## 5.5 SUMMARY

The previous analysis has shown that many research systems in the robotics and intelligent systems domains are constructed based on comparable principles that can be subsumed under the microservice architecture term. The framework-level resource awareness concept presented in this work therefore targets this architectural style and builds upon it. Apart from the predominance of the style and therefore the wide applicability of the developed methods, microservices also provides a set of natural extension points that enable an efficient implementation of framework-level resource awareness. The separation of the system in microservices or components (from now on used interchangeably) results in graspable units with clear boundaries (the component interface) and developer responsibilities. Therefore, methods developed in this thesis will follow this separation. Methods will be applicable for individual components and generated results will be related to these components so that the decision to use the methods and the responsibility to handle results are assigned to individual developers. As the network communication between the components can be introspected at runtime with generic means, a clear path to acquire knowledge about the component's processing without manual instrumentation exists. Finally, the focus on components also allows achieving a level of resilience against the ongoing system changes (also for the [resource awareness](#) methods), because the behavior of several components is often unaffected from ongoing modifications to other parts of the system. To summarize, microservice-based systems in robotics and intelligent systems are the targets of the work in this thesis.

*targeted system types* ⊙

*focus on individual components* ⊙

The work in this thesis addresses systems constructed using the ideas of [CBSE](#) and [microservice architectures](#). In the robotics context this usually implies a fixed [middleware](#) for a system. For my work, I have used system built around our own middleware called [Robotics Service Bus \(RSB\)](#), which I am going to explain here. [RSB](#) is in constant use in different robotics and intelligent systems scenarios since our initial publication about the middleware [[WW11](#)] and has since then been maintained and improved by many people. Parts of the description in this chapter are based on the original publication, but updated to the current state. Development of [RSB](#) was initiated to fulfill requirements of different research projects that could not be addressed with existing middlewares at that time. These are in detail:

◆ [Robotics Service Bus \(RSB\)](#)

◎ [middleware requirements](#)

**INTEGRATION OF HETEROGENEOUS PARTS** To enable a rapid construction of new research systems, the middleware needs to enable the integration of diverse types of [components](#) and robotics platforms. Components are often reused across research projects, hardware, and software platforms. Therefore, integration of technologically diverse software artifacts needs to be enabled by the middleware to speed up the system construction and modification process; and therefore also the research tasks. Moreover, architecture and computational power of intended target platforms vary and [RSB](#) needs to be usable on platforms ranging from embedded devices to well-equipped mobile robots or smart-home systems with multiple desktop computers or servers for processing.

**INTEROPERABILITY** For being able to reuse components from the component market of other middleware, [RSB](#) needs to interface with these systems, ideally without requiring modifications to components from both ecosystems. Therefore, explicit interoperability is required to enable component reuse as envisioned by the [CBSE](#) idea (cf. [Chapter 5](#) on page 43).

**FACILITATION OF RESEARCH TASKS** To fulfill the scientific requirements, the middleware has to support common tasks related to the experimental work robotics systems. These especially include recoding datasets and [introspecting](#) running systems to analyze their structure.

Existing middlewares, such as the ones presented in [Section 5.3](#) on page 45, failed to meet at least one of the requirements. One common issue we have observed is that dependency footprint of these

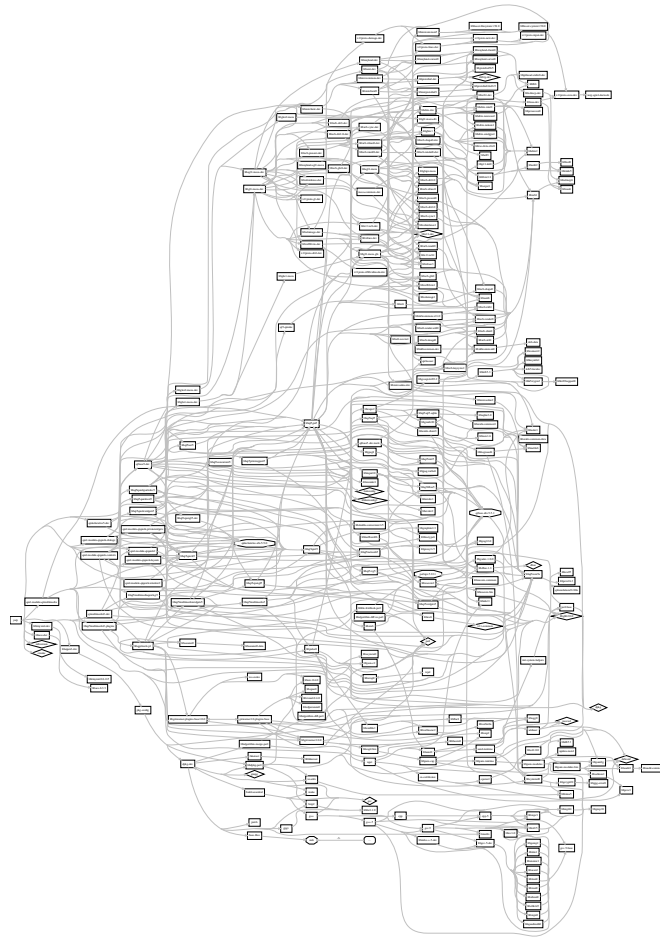
frameworks is large. Such a large dependency graph results in a considerable amount of work when using the software on new hardware platforms because developers have to port all dependencies to the new system. Additionally, large dependency footprints at runtime result in higher memory requirements, which small embedded devices cannot fulfill. To visualize this issue, the dependency graphs of the YARP and ROS C++ implementation on an Ubuntu Xenial system are depicted in [Figure 6.1](#) on the facing page. The implications of large graphs such as the presented ones contradict the first one of the defined requirements.

Another issue with existing middlewares is that these often result in a high level of [vendor lock-in](#) by providing a complete application development framework for component developers, for instance with build systems, tools, or tailored support libraries. Despite being convenient in the first place, this couples the component implementation to the middleware. As a consequence, transitions to other frameworks are costly because large parts of the component and build system code rely on the framework and therefore need to be rewritten for the new framework. For instance, ROS provides *catkin* as its own build system, which prevents reusing components in other frameworks without creating a new build system for these cases.

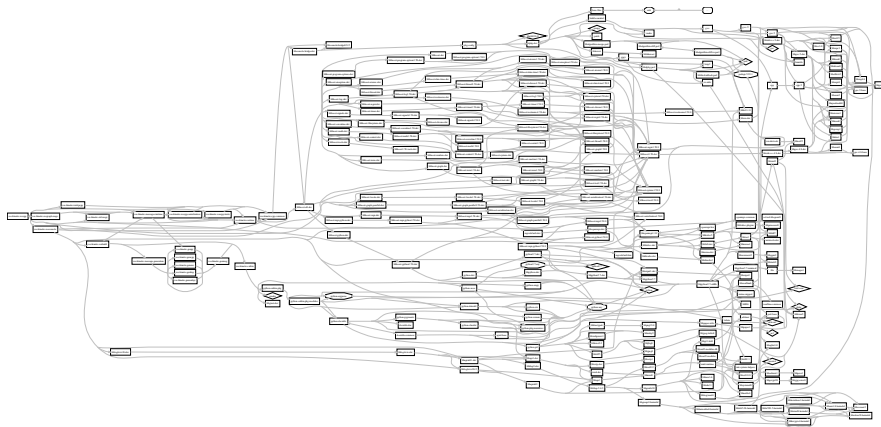
Regarding the technical realization, existing robotics frameworks often require a central server to establish communication channels, for example, the YARP nameserver or the ROS master. Such a central service has to be started before client components and forms a single point of failure.

Finally, existing middlewares do not possess principled approaches for interoperability with other frameworks. Although some work has been done to enable interoperability between selected ecosystems (cf. [Section 5.3](#) on page 45), the created solutions are often ad hoc and tailored to the specific interoperability case, or they do not provide complete transparency because artifacts ripple through the architecture and need to be addressed manually in the components. For example, when using YARP to communicate with ROS systems, special data types need to be generated and used inside the YARP components [YRT], which prevents reuse without modification.

*technological decisions* ☉ To avoid such issues and to fulfill the defined requirements, we decided to realize [RSB](#) as a lightweight library. We explicitly focused on the communication of components across the network and did not provide further application development functionality. Apart from reducing vendor lock-in, the library approach and the lack of an explicitly enforced [component model](#) also allows integrating with libraries or frameworks that themselves impose a lifecycle. One example where [RSB](#) could be used due to this decision is the multimedia framework *GStreamer* [Gst], which requests such a lifecycle for its plugins.



(a) YARP 2.3.70



(b) roscpp Kinetic

Figure 6.1: Dependency graphs of C++ implementation of other robotics middlewares. Boxes represent system packages up to the level of C/C++ base libraries. Generated using *debtree* [DebT] on Ubuntu Xenial.

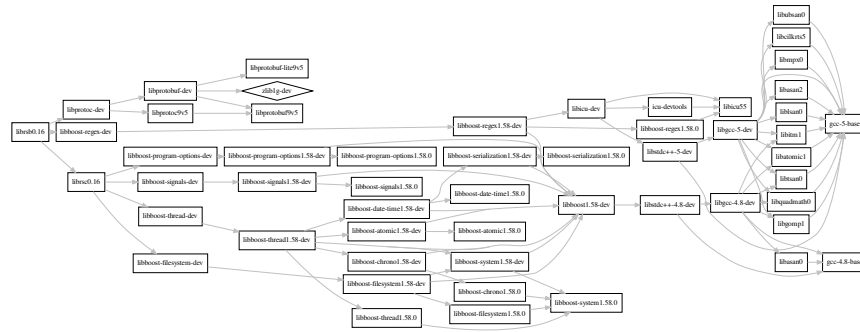


Figure 6.2: Dependency graph of the **RSB** C++ implementation.

Regarding the implementation of **RSB**, we decided to reduce the dependency graph to a minimal size as long as the reimplementing of larger parts of common functionality can be avoided. The dependencies that are included must be well-known with good package coverage in Linux distributions and support for different operating systems and hardware architectures. Finally, **RSB** can be used with different programming languages (C++, Java, Python, and Common LISP) and for each language a different implementation of the middleware exists. In contrast to an approach with a main C library and bindings for other languages, this approach prevents unconventional programming idioms and C library dependencies in target languages, allows using established build systems in each language, and results in an immediate verification of the defined communication protocols. To visualize the consequence of these decisions, [Figure 6.2](#) depicts the dependency graph of the **RSB** C++ implementation, which is much smaller compared to the graphs shown in [Figure 6.1](#) on the previous page. Additionally, **RSB** does not possess a central nameserver to prevent the aforementioned issues. Instead, the naming and discovery tasks are realized using decentralized protocols.

## 6.1 ARCHITECTURE

In essence, **RSB** can be described as a message-oriented, event-driven middleware based on a logically unified bus. As a result, message senders are decoupled from receivers and the native communication semantic is  $m : n$  broadcast. [Figure 6.3](#) on the facing page visualizes the core architectural concepts of **RSB**. Processes use participants to exchange **events** using the unified bus. On the bus, the events are encoded as **notifications**. *Participants* form the primary interface between the user code and the middleware and *informer* participants can be used to send events whereas receiving is performed through *listeners*. We adopted this naming scheme from Barrett et al. [[Bar+96](#)].

Besides the broadcast semantics, other communication patterns are constructed based on this fundamental mechanism. For instance, an

*participant* ◆  
*informer* ◆  
*listener* ◆



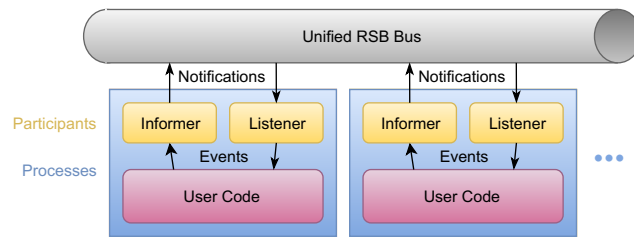


Figure 6.3: Architecture concept of **RSB**. Blue processes are connected to the **RSB** bus using participants (yellow).

asynchronous **RPC** pattern is contained in the **RSB** core. Patterns are also represented as special participants, which internally use the base participant types. Therefore, participants form a hierarchy.

In the following sections, I will describe the architecture of **RSB** in detail along selected modeling dimensions for event-based systems introduced by Rosenblum and Wolf [RW97] but with the interpretations from Cugola et al. [CDF01]. First, I will describe the event model, which provides “a definition of event types with a detailed description of the nature and structure for the events” [RSS07]. Second, the naming model describes how generated events can be addressed. Afterwards, the notification model is presented with the mechanisms used to transport events between participants. Third, the temporal dynamics of sending and receiving events are explained in the time model. Finally, the observation model is introduced, which specifies how participants indicate the relevant events they want to receive. Figure 6.4 on page 57 presents the involved concepts and entities and their relations and supplements the following explanations.

### 6.1.1 Event model

Regarding the nature of an event we have adopted the definitions of Faison [Faio6], who defines an *event* as “a detectable condition that can trigger a notification” [Faio6, p. 71]. A *notification*, in turn, is “an event-triggered signal sent to a runtime defined recipient” [Faio6, p. 71]. Hence, all information required to fully specify and trace the condition an event represents need to be present in its framework representation. Therefore, we have decided to equip events in **RSB** with a rich set of information that allows tracing each event and its relation to others in detail. An event contains the following items:

**PAYLOAD** The payload of an event is a user-defined object of the respective programming language that contains the primary information specifying the condition the event represents. It is of an arbitrary domain type to reduce the framework lock-in through an early transition from framework types to domain objects (cf. Section 6.1.3 on page 58 for the technical realization).

**ID** Each event has a unique ID to allow a global identification. The ID is based on the ID of the sending participant (represented as a [universally unique identifier \(UUID\)](#)) and a sequence number. It can be represented as UUID, too.

**METADATA** Each event is supplemented by a set of metadata to provide further details regarding the origin and processing of the event. Metadata originate from the [RSB](#) framework as well as from the user code that creates the event. The framework provides detailed timing information regarding the creation, processing, and delivery of an event and its associated notification, as, for instance, proposed by Luckham [[Luc10](#), p. 96]. User code can provide further timestamps and string entries in a key-value store for further information.

**METHOD** An event can be tagged with a method that indicates its role in a communication or the kind of action this event represents, comparable to methods in the [Hypertext Transfer Protocol \(HTTP\)](#) protocol [[IET14](#)].

**CAUSAL VECTOR** As an event is often the result of processing caused by other events in the system, [RSB](#) events contain a causal vector as proposed by Luckham [[Luc10](#), p. 97]. This vector contains IDs of other events that caused the respective event to be created. It provides information for automatic system analysis and debugging.

**DESTINATION SCOPE** Specifies the channel of the logically unified bus an event is visible on as described in [Section 6.1.2](#).

In contrast to other robotics middlewares, this event model provides more metadata directly at the framework level to improve the runtime traceability and automatic analysis of systems constructed with [RSB](#). Moreover, we have reduced the vendor lock-in by allowing any kind of programming language object to form the payload of an event. This approach prevents that client code is coupled to special framework data types and instead, the actual domain data types can be used directly when interfacing with the middleware. In combination with appropriate extension points and configuration possibilities, this approach is one method to meet the aforementioned interoperability requirement (cf. [Section 6.5](#) on page 66).

### 6.1.2 Naming model

An interested participant has to be able to express which set of events it intends to receive. The naming model in [RSB](#) is based on what Faison [[Faio6](#), p. 81] calls a *channel-based subscription model*. In its basic form, clients subscribe to a specific channel of the bus system (green

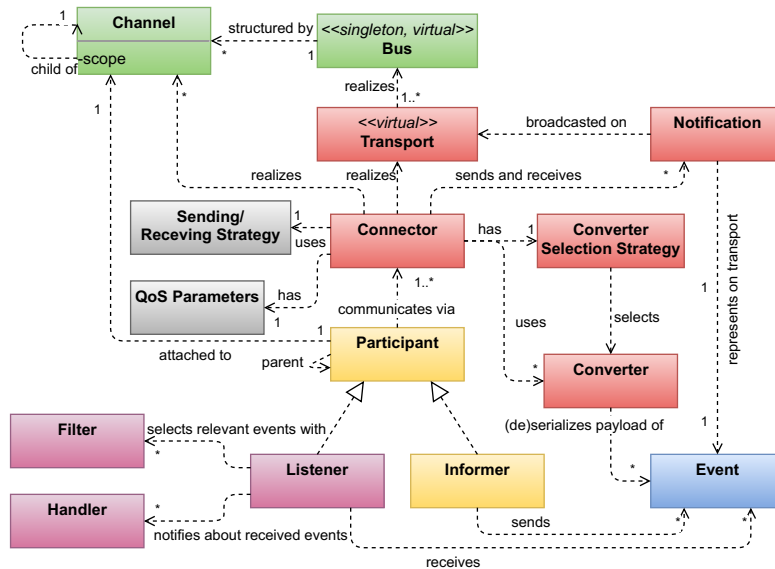


Figure 6.4: Overview of RSB concepts and their relations. Aspects of the event model are colored blue, while elements of the naming model are colored in green. Notification model parts are filled in yellow and red to distinguish between client level and back-end realization respectively. Entities of the observation model are depicted in purple and aspects related to the timing model in gray. virtual annotations express that the respective entity is not an actual artifact represented in code, but instead is implicitly formed.

boxes in Figure 6.4) and receive all events sent on the respective channel, comparable to TV programs. RSB extends this concept through a channel hierarchy. All channels form a tree structure, starting from a global root channel, and an event sent on a channel is thus visible on this single channel and on all parent channels including the root one. To express this channel hierarchy, a notation comparable to UNIX file systems, compatible with requirements for path segments in Uniform Resource Identifiers (URIs) [IET98], is used. A channel represented this way is called a *scope*. For instance, the root scope is written as / and a direct sub-scope could be /sub/. Further levels are added by extra segments separated by forward slashes.

Figure 6.5b on the following page shows an exemplary list of used scopes from Scenario ToBi on page 142 with the resulting hierarchy depicted on the left in Figure 6.5a on the following page. An event sent to the scope /armserver/openGripper can be received by listeners on the channels /armserver/openGripper, /armserver, and on the root scope /. Another event sent to the scope /speech/tts/mary/-server will be visible on scopes /speech/tts/mary/server, /speech/tts/mary, /speech/tts, /speech, and /. However, it will not be visible to listeners on scope /speech/tts/mary/server/say\_raw or on scope /monitoring/father.



(a) Hierarchical representation

(b) Scope representation

Figure 6.5: Different representations of RSB scopes found in Scenario ToBi.

The chosen hierarchical channel layout provides benefits for data logging and recording purposes because it allows addressing parts of or the whole system communication using a single scope declaration. As the hierarchical structure provides a means to structure the data space, for example with sub-scopes for different services or subsystems, logic segments of system communication can be formed. Therefore, the hierarchical scoping theme facilitates dataset creation for research tasks, and runtime introspection and debugging.

### 6.1.3 Notification model

As introduced before, RSB forms a logically unified bus. Each participant is associated to one channel, but multiple participants can participate at the same channel, which results in  $m : n$  semantics. To realize the notification model, which described how events are transmitted from informer participants to listener participants, we have considered a set of different constraints:

1. The middleware must support different protocols and technologies for delivering events in parallel. For instance, an in-process connection should fulfill high performance requirements in selected components through collocated optimization [VP07], while other parts of a system use usual network-based methods in parallel. Moreover, different network-based transports must be usable at the same time to support varying performance or interoperability requirements.

2. The payload of events is – by design – a user-defined programming language object of any type (cf. [Section 6.1.1](#) on page 55). Thus, no common (de-)serialization mechanism can be used and a more flexible *(un-)marshaling* [[Cou+12](#), p. 158; [ZKV04](#); [WS01](#), p. 41] approach is required.
3. If different transports are used, they may require different (un-)marshaling strategies.
4. The middleware must be able to create compatible serialization formats used on the wire from different domain objects to enable standardization and compatibility between different applications and libraries. For instance, an informer used in a C++ program might send OpenCV [[Bra00](#)] images and a Python application should be able to receive these images represented using the appropriate data type of the *Pillow* [[Pil](#)] library.
5. Depending on the application, client-level code may impose different QoS and performance requirements on the transmission of events.

To meet constraint 1, [RSB](#) has a notion of distinct *transport* mechanisms. The logically unified bus of the middleware is realized by these transports, which define the technology and (network) protocol for message exchange. Participants are connected to the bus through one or more *connectors*, which realize the transport. These connectors implement sending and receiving of events by transmitting (usually) serialized representations of these events, called notifications, via their protocol. This protocol imposes restrictions on the technical representation of notifications. For instance, one network-based transport might be able to send any sequence of bytes, whereas another network-based transport might only be able to cope with [Extensible Markup Language \(XML\)](#) data. This technical representation of serialized data is called *wire type* in [RSB](#).

Three transports are available in [RSB](#). For complex [distributed systems](#) over a network connection, a transport using the Spread Toolkit [[Spread](#); [AS98](#); [Ami+04](#)] can be used. A more lightweight network-based transport using [TCP](#) communication can be applied with less configuration overhead. Finally, an in-process transport implements efficient event exchange without serialization overhead for participants operating in a single operating system process.

Although the general structure of events is defined by the event model (cf. [Section 6.1.1](#) on page 55) and each transport can choose an appropriate marshaling scheme, the user-defined payload is of an arbitrary data type and cannot be serialized using generic methods. For this purpose, a *converter* interface exists in [RSB](#). Converters have the responsibility of (de-)serializing user data types to and from serialized representations that can be transmitted using a transport. They

vary in the following three dimensions and have to be chosen along these dimensions to successfully perform the marshaling task:

- The wire type.
- The [wire schema](#). Given a wire type, still different representations can be formed. For instance, the binary encoding or XML schema might differ. The wire schema is a string representation of the convention used to lay out the serialized data in a format compatible to the wire type.
- The user data type. For instance, OpenCV or *Pillow* [Pil] images.

Given a specific connector, only those converters matching its wire type can be used to (de-)serialize events. However, no clear selection criterion for a certain converter exists that also covers the other two aspects. For instance, the desired wire schema is a design decision of the overall system and, for deserialization, multiple converters can exist that produce different user data types. Therefore, RSB uses exchangeable [converter selection strategies](#) to select appropriate converters (following the *strategy* pattern [Gam+95, p. 315]). A default implementation is provided, which automatically selects a converter as long as no ambiguity exists. Another included strategy is based on a user-defined set of predicates to select the first matching converter. RSB includes a set of predefined converters for fundamental data types such as boolean, numbers, and strings. This set can be extended by the user to integrate user-defined payload types.

*converter selection strategy*

Converters are the technical solution to constraint 2 and together with converter selection strategies they meet constraints 3 and 4. Further details will be outlined in [Section 6.5](#) on page 66.

#### 6.1.4 Time model

Regarding the dynamics of transmitting events, different application requirements can exist. To meet these, RSB uses exchangeable strategies for sending and receiving, as well as explicit QoS parameters.

On the informer side, the default strategy realizes synchronous sending. However, if client code sometimes produces more events than a transport can handle immediately, and sporadic delays in the processing are acceptable, a queuing strategy can be more suitable to prevent blocking the client code. A comparable strategy interface also exists for the receiving side of the communication.

Available QoS parameters concern [reliability](#) and ordering of event submission. Both properties can be defined using a set of available options, which have a strict ordering so that, for example, a higher level of reliability includes all aspects of lower levels. QoS parameters are defined at the level of participants. So, the effective level is the minimum for each aspect of each pair of communicating informers

and listeners. For instance, specifying [reliable, ordered] at the sending side and [unreliable, ordered] at the receiving side will effectively result in [unreliable, ordered]. QoS parameters are realized by each transport independently so that the possibilities of each realizing technology can be used effectively.

#### 6.1.5 Observation model

Client code interested in receiving events uses listener participants to receive the unmarshaled notifications, which are represented as events again. RSB uses an asynchronous, push-based model for receiving events. By registering *handlers* at a listener, the client informs the framework about callback code to execute asynchronously on each received event. Multiple handlers can be registered on a listener. ◆ handler

Besides specifying the scope to indicate which events will be received – which is already defined by the listener – clients may require further filtering to avoid receiving irrelevant information. For this purpose, they can install dedicated *filters* at each listener to restrict the set of received events. All filters registered at a listener are interpreted as a conjunction. Thus, the first filter that rejects an event leads to a complete rejection of the event. Filters of a listener affect all registered handlers. ◆ filter

RSB implements content-based [Faio6, p. 84], client-side filtering, where filters are explicitly allowed to inspect the user-defined payload of events. Users can extend the available set of filters according to their requirements.

#### 6.1.6 Extension points

The preceding description of the architectural elements of RSB has shown that the user needs to specify selection criteria and provide extensions for entities such as converters and filters to match the application requirements. Thus, RSB provides explicit extension points to include the user implementations. Apart from the explicit configuration of these extensions via code instructions, a plugin mechanism and configuration files allow reconfiguration of some extension points without recompilation. Table 6.1 on the following page enumerates the available extension points, motivates their usefulness, and indicates how they can be configured.

## 6.2 INTROSPECTION

To support runtime analysis and debugging of systems, RSB has a distributed introspection system, which does not require a central name-server. Using the introspection mechanism, interested clients can get the following information about a running RSB system:

EXT. POINT	DESCRIPTION	C	P
converter	additional converters for (un-) marshaling of user data types		✓
converter selection	additional strategies for converter selection for special applications; configurable disambiguation for the default strategy	✓	
filter	additional filters for restricting the received events		
transport	addition of transports via plugins, configuration and selection in config file	✓	✓
sending/receiving strategy	customized event processing		

Table 6.1: Extension points in [RSB](#). Column *c* indicates whether an extension point can be configured using the configuration file (apart from inside the source code) and column *p* indicates whether extensions can be loaded via plugins.

**OPERATING SYSTEM PROCESSES** For all processes, the following information is available: a) command line, b) process start time, c) system user running the process, and d) [RSB](#) version.

**PARTICIPANTS** For all active [RSB](#) participants, the following information is available: a) unique ID, b) participant type, c) scope d) transports, e) payload type, and f) parent participant.

**HOSTS** For all hosts participating in the communication, the following data are available: a) operating system and version, b) [CPU](#) type, c) uptime, d) clock offset, and e) communication latency.

[Figure 6.6](#) on the next page visualizes the conceptual structure of how the introspection support is implemented in [RSB](#). Each process has a single `localIntrospection` object, which is responsible of collecting the necessary information. For this purpose, the `localIntrospection` object is notified about the creation and destruction of participant instances. It uses two additional objects to acquire host- and process-related information. To expose this information to interested remote clients, three participants are used. During normal operation, whenever a participant appears or is removed in a process, an event is broadcasted using the `broadcastInformer`. This event contains a respective `hello` or `bye` message with the required information. Therefore, under normal conditions, the introspection uses a differential protocol and interested clients need to keep track of the notified changes. However, when a new introspection client (`remoteIntrospection`) is created, it needs to know the initial state of the



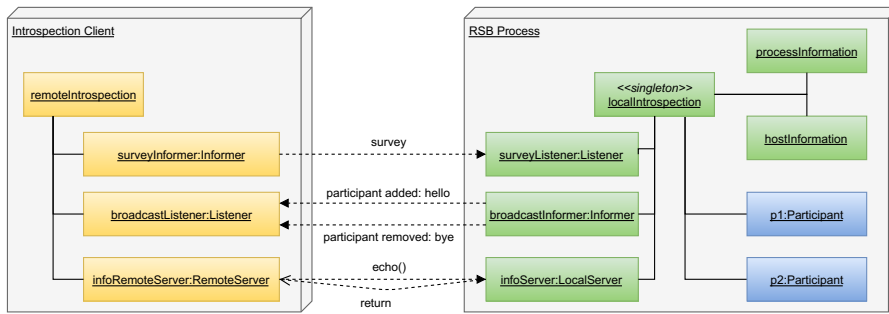


Figure 6.6: Structural view of the **RSB** introspection mechanism realization as a UML object diagram. Separated operating system processes are indicated through gray blocks. Objects of the local introspection information provider are colored in green and remote client objects are colored in yellow. Normal Participants are blue.

system. For this purpose, the `remoteIntrospection` object can broadcast an event to all running **RSB** processes using its `surveyInformer`. After receiving the survey request, each process will expose its current state by sending out `hello` events for all participants using the aforementioned mechanism. Finally, remote clients can use the `infoServer`, which is an RPC server, to periodically exchange ping-pong messages. These are used to compute the communication latency between the remote introspection client and each process and also to detect whether a process is still alive.

The exchanged events regarding the introspection are part of the normal bus communication below a reserved scope `/__rsb/introspection`. Therefore, they can be examined and recorded using the usual mechanism for debugging and dataset creation.

### 6.3 DOMAIN DATA TYPES: RST

One important aspect of component interoperability is an agreement on common data types. Even though the **RSB** architecture contains special means to overcome integration challenges imposed by diverging network protocols and data formats, the application of such special methods can be prevented if a common ground of established data types is used in the first place. Through the converter mechanism the core of **RSB** is agnostic to the agreed upon format and technology for serialization. However, systems constructed with **RSB** usually use data types from the *Robotics Systems Types (RST)* library. This is a reviewed library of domain data types which we have collected in our own robotics systems. Data types in this library follow a specified lifecycle and review process to migrate from a proposal stage in a sandbox area to stable data types, which may become deprecated. This process is set up to ensure that collected data types meet quality and reuse criteria before client software depends on them.

Robotics Systems Types (RST)

---

```

1  package rst.audition;
2
3  import "rst/audition/PhonemeCollection.proto";
4  import "rst/audition/SoundChunk.proto";
5
6  /**
7   * Objects of this represent a single utterance of speech.
8   *
9   * The data describes a single utterance in three different forms:
10  *
11  * * @ref .phonemes describes the utterance as a list of phone symbols
12  *   and durations (useful e.g. for lip animation).
13  *
14  * * @ref .audio is a @ref SoundChunk that can be played back on audio
15  *   devices containing the realization (e.g. by a TTS system)
16  *   of the included phoneme list
17  *
18  * * @ref .description is a textual description of the utterance for
19  *   debugging purposes.
20  */
21
22  message Utterance {
23
24     /**
25     * A collection of phonemes. Will be played back in the same
26     * ordering as given by @ref .Phoneme
27     */
28     required PhonemeCollection phonemes = 1;
29
30     /**
31     * A chunk of audio data that can be played back containing the
32     * realization (e.g. by a TTS system) of the included phoneme list
33     */
34     required SoundChunk audio = 2;
35
36     /**
37     * Textual representation of the utterance.
38     */
39     required string textual_representation = 3;
40
41  }

```

---

Listing 6.1: RST data type for speech utterances that have been produced by a text to speech (TTS) engine. The audio sample is annotated with phonemes, for instance, to control the lip motions of a humanoid robot.

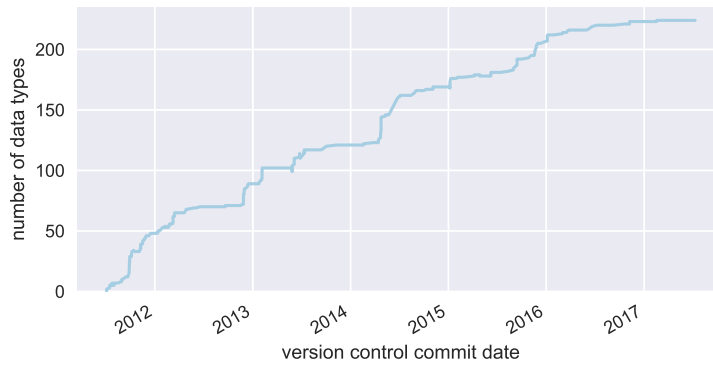


Figure 6.7: Evolution of the number of data types contained in [RST](#). Downward peaks are the result of using Git commit dates for the x-axis, which do not necessarily reflect the actual time of inclusion in the main line.

[RST](#) data types are defined using an [IDL](#). In contrast to other robotics middlewares, which have defined custom data formats or IDLs (for instance, `ros_msg` in ROS or the bottle format in YARP), we have decided to use an established IDL technology. This prevents further vendor lock-in on the middleware and its ecosystem, reduces the implementation efforts in [RSB](#), ensures a high code quality and maturity of the data type support libraries, and results in available tool support (syntax highlighting, code completion, validation). The [RST](#) library is based on *Protocol Buffers* [[Protobuf](#)]. This IDL is well-known outside the robotics community and is not inherently coupled to a broader framework, nor is the [RST](#) project itself limited to [RSB](#). [RSB](#) has a converter for *Protocol Buffers* [[Protobuf](#)] data types, which enables the usage of [RST](#) data types for communication.

[Listing 6.1](#) on the facing page visualizes an exemplary data type from [RST](#) that meets the intended quality requirements regarding naming, reuse, and documentation. At the time of writing, [RST](#) contained more than 200 data types. The evolution of this count over the years is presented in [Figure 6.7](#).

## 6.4 TOOL SUPPORT

Alongside the middleware core, a set of tools has been developed to carry out common tasks and to fulfill the initially stated requirements on a research middleware.

For debugging and analysis purposes, a set of command line tools has been developed. A *logger* allows intercepting the ongoing communication of a live system with different display options. Apart from printing the decoded event instances (optionally including the payload), timeline-based views can be used to continuously plot the dynamics of the communication. In case events or commands should be sent for debugging purposes or in scripting contexts, the *send* and *call*

tools enable this from the command line. A client for the *introspection* protocol implements the `localIntrospection` object (cf. [Section 6.2](#) on page 61) and allows examining the system structure at runtime. Besides a snapshot mode, which dumps the structure at a certain time, a live mode provides the possibility to continuously monitor a system using a tree-based view. The snapshot mode allows exporting in the [JavaScript Object Notation \(JSON\)](#) format as an established format. Finally, a *bridge* tool can be used to selectively map parts of the bus communication to other scopes or transports.

The task of recording datasets for scientific and debugging purposes is accomplished by the *rsbag* tool [MNW13]. *rsbag* attaches to the ongoing system communication through the usual listener participants and records all events on a set of specified scopes and all their sub-scopes based on the hierarchical structure of RSB. Thus, *rsbag* can be used to record the whole system communication or selected parts of it. Each recording session is stored in a single file, which can be replayed or transformed to other formats. For replay, different strategies are available. Apart from using the same timing as while recording, further strategies such as a fixed rate replay can be used. Moreover, control can also be achieved using the RSB RPC pattern, which allows remote controlling the emission of recorded event. Another mode permits exporting recorded events using user-specified conversions, for instance to [comma-separated values \(CSV\)](#) files, which can then be used for offline analyses. In both modes the scopes to process and additional filters to apply can be specified.

## 6.5 INTEROPERABILITY WITH OTHER MIDDLEWARES

With the availability of the different extension points, RSB provides unique opportunities for interoperability with other middlewares and systems. The availability of the extension point for transports allows communication with other ecosystems through implementations of their protocols. Due to the explicit representation and configurability of transports through plugins and the configuration system, no code changes are necessary to adapt a component from a native RSB transport to a foreign one. We have implemented and used transports for ROS topics and YARP ports [MNW13] as a verification of this approach.

Even if a network protocol of a different framework can be used for communication, ad hoc reuse of components in foreign frameworks is usually impossible without code modifications due to different data types and encodings. For instance, when using YARP with ROS, special data types need to be generated and used [YRT] or the native bottles need to structurally match ROS types [YRS]. Another common approach is to use explicit bridge components. However, these introduce an additional hop in the network communication, which

increases latency, [resource utilization](#), and system deployment effort. With an explicit interface for data type conversion, which is designed as a configurable extension point, [RSB](#) provides a systematic solution to this problem. Existing component can be configured to use special converters that perform the translation of data types from two distinct ecosystems. Consequently, interoperability in [RSB](#) is a combination of configuring the appropriate transports and the necessary converters.

Converters that translate between data types from different framework could be implemented manually. However, such an approach is error-prone and requires manual labor. To improve this situation, we have evaluated the possibility to represent data types from different framework using a common metamodel in Wienke et al. [[WNW12](#)]. The analysis of IDLs and data representation formats from different robotics ecosystems, as well as solutions used outside of the robotics domain, has shown that a core set of features can be identified, which can be used to represent the majority of data types formulated using the analyzed formats. From this feature set a metamodel and a tool for the automatic generation of conversion code were designed.<sup>1</sup> As data types often do not match structurally, a mapping between fields inside the data types is required to translate between them. For instance, field names might not match or one ecosystem represents angles in degrees, while another one uses radians. We have analyzed the required capabilities between [RST](#) and ROS data types to overcome such issues. The resulting feature set can be used to specify the mappings between data types in a declarative form. The created tool uses such a mapping definition and parsers for the data type definitions in both frameworks to construct conversion code. Generated converters directly translate between the serialized representation of the foreign framework and the native data type representation, in our case [RST](#) types encoded using *Protocol Buffers* [[Protobuf](#)]. This approach prevents unnecessary intermediate representations.

Converters created using the presented declarative approach, as well as manually created ones, can be compiled into a plugin for [RSB](#) and loaded to enable a seamless integration into foreign frameworks without code modifications. Such an approach, especially the ability to prevent intermediate representations, is impossible in frameworks that do not provide an explicit extension point for conversion.

## 6.6 APPLICATIONS

Since the initial release of [RSB](#), the middleware has been used in a variety of different application scenarios with different sizes, operating systems, and [performance](#) requirements. In the HUMAVIPS project,<sup>2</sup>

<sup>1</sup> The implementation has been done by Jan Moringen.

<sup>2</sup> <http://humavips.eu>

the humanoid robot NAO has been controlled in HRI scenarios using a distributed system based on RSB [Klo+11]. This scenario specifically included dealing with synchronized audio and stereo video signals [San+12]. Further applications of RSB and the NAO robot for HRI include different versions of a museum tour guide and related experiments [Dan+16; Geh+17]. Other HRI scenarios include a RoboCup@Home platform [Mey+15] and a system where NAO is used as a sport instructor [SGK17].

RSB has also been used to control other types of robotics platforms. This includes small embedded devices [Her+16; NRW12], the KUKA LWR IV robotics arm in different scenarios [TFR13; NWS15], and a continuum robot [Rol+15].

Another application area for RSB is smart homes. In the *Cognitive Service Robotics Apartment* (cf. Scenario CSRA on page 99). This is by far the largest system using RSB I am aware of with more than 20 computers and high throughput, for instance, from several Kinect devices.

In addition to such scenarios, RSB has also been used as a basis for other software. For instance, a framework for incremental dialog uses RSB for system communication [KKS14]. This system has been applied, for instance, for in-car dialog [Kou+14]. RSB has also formed the communication layer of a formal component model targeting compliant robots [NRW12].

The amount of systems based on RSB demonstrates that the middleware is a suitable platform for robotics and intelligent systems in production use cases. Most of the systems make active use of the provided introspection and recording capabilities for the ongoing research work as well as for debugging purposes. RSB has been updated constantly based on requirements and feedback from the application scenarios.

## 6.7 SUMMARY

The presented middleware RSB allows constructing robotics applications as a distributed system. Comparable to other common solutions, RSB systems are usually constructed following the microservice architecture style with isolated operating system processes per functional component. The availability of detailed runtime introspection and data logging facilities makes RSB a suitable platform for an implementation of *framework-level resource awareness* concepts. Such an implementation has to be built on top of data and abstractions generically available through the middleware or component model to avoid code changes. The hierarchical bus with the broadcast semantics and the introspection mechanism are directly available to acquire the data. With the unique interoperability concept provided through exchangeable transports and converters, the solutions developed in this thesis,

which are directly based on [RSB](#), can be integrated in other ecosystems without reimplementa-tion. Yet, the concepts of [RSB](#) are close enough to other common middlewares such as ROS or YARP (cf. [Section 5.4](#) on page 49) that no severe conceptual gap has to be expected. Therefore, a reimplementa-tion of the framework-level resource awareness concept directly for these frameworks is possible even without using the interoperability mechanisms. Thus, although this thesis primarily focuses on [RSB](#), solutions are conceptually compatible with other common frameworks and therefore a majority of research robotics systems as well as other systems employing microservice architectures.

The presentation of [RSB](#) has skipped several aspects such as benchmarks or optimization techniques. For these details, please refer to the original publication [[WW11](#)].





The previous chapter has already introduced dataset recording as an essential task for the scientific research process. Without appropriate datasets many qualitative and quantitative analyses are impossible. Furthermore, publicly available datasets support reproducibility and reliable benchmarking of scientific results [Pen11]. However, depending on the scenario and target system, creating datasets that have the required quality for the research tasks and public distribution is a challenging, complex, and time-consuming task. Merely recording the available system data using the [middleware](#) facilities will barely suffice to create a dataset that contains all required aspects. Moreover, simply redistributing such data in the proprietary format of each middleware will hinder the exploitation of the data in other ecosystems and offline research tasks. Instead, especially for datasets including [HRI](#) aspects, additional data sources – for instance, external video cameras or motion capturing – need to be recorded and combined with the middleware data to form the complete corpus. The creation process for such complex datasets often involves dealing with a multitude of devices and output formats. The data from these diverging sources have to be synchronized to be usable in combination and appropriate exports to globally accepted formats are required for distribution. Moreover, manual annotations may be required. All these generated data fragments must be managed consistently to prevent errors and to streamline this intrinsically complex process.

Driven by our own requirements to record such a complex HRI dataset for the EU FP7 research project HUMAVIPS, we have investigated how to establish such a streamlined and generic dataset creation process, which we have published in Wienke et al. [WKW12]. Parts of this chapter are based on this publication. The scenario of our own recording activities was chosen to provide data that can be used to improve the abilities of a robot interacting with a group of people through audio-visual integration. For this purpose, the setting of a small vernissage was chosen, where the humanoid robot NAO explained several paintings to a small group of visitors. This scenario was inspired by Pitsch et al. [Pit+11]. To ensure natural behavior, we invited naive participants to represent the visitors of the exhibition. The robot behavior was remote controlled, because the dataset was intended to provide the foundation for developing the necessary capabilities for autonomous interaction. These capabilities included addressee detection and [visual focus of attention \(VFOA\)](#) recognition. To acquire the necessary ground truth information, different exter-

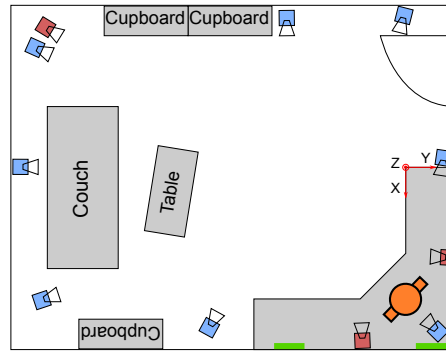


Figure 7.1: Overview of the recording setup. Orange: NAO, blue: motion tracking cameras, red: video cameras, green lines: paintings.

nal sensors had to be recorded in addition to the robot system data, which included cameras and microphones, as well as the remote control instructions. On the one hand, video cameras were required to obtain permanent recordings of the participants for the reliable annotation of speech and addressees. On the other hand, exact ground truth of the positions and orientations of the participants' heads was required, which was achieved by including a motion capturing system based on infrared markers. Figure 7.1 visualizes the resulting recording setup.

Comparable to the HUMAVIPS scenario, research in HRI often requires correlating data from multiple input streams for a joint analysis of system-level and interaction-level data [Loh+09]. Our proposed process specifically addresses the needs for creating and distributing multimodal HRI datasets that enable these research tasks. Because such datasets include data from all levels of the system and interaction, I will refer to this approach as a holistic dataset creation process.

### 7.1 CHALLENGES IN CREATING DATASETS

We have developed the process to address a set of key challenges in creating datasets that we have identified.

One of the primary issues when recording multimodal datasets is the *synchronization* of all modalities. For cameras or audio streams this could be done manually in a post-processing phase. However, synchronization is much more complicated with modalities which are less intuitive to observe for humans such as robot internal states. Moreover, manual synchronization takes time. Hence, one challenge is the *reduction of post-processing tasks*. This challenge can be addressed already in the recording phase through appropriately chosen devices and recording methods.

Another key challenge is achieving a high *integrity of the recorded data*. A high level of *automation* and the availability of automated *validation* mechanism already during the recording time address this

challenge. Another aspect related to this challenge is the *calibration* of recording devices. For instance, by calibrating all cameras with respect to a motion capturing system, previously unintended opportunities to use the dataset are preserved.

Regarding the integration of system-level data into the dataset, an important requirement is that the *gap between the recording and the production system* is as small as possible. On the one hand, this implies that the usual robot system should not need modifications to enable the recording task. On the other hand, recorded system data should be immediately usable for exercising the existing system [components](#). Yet, it is important that the dataset can be used even without the system integration. Thus, *export* facilities to established formats are necessary, for instance, common video formats or [CSV](#) files.

Finally, *established annotation tools* should be usable to allow an efficient progression in the labor-intensive manual annotation phase. This task should be supported by the *availability of system-level data for the annotation* inside the annotation tools. In any case, manual annotations should be prevented whenever it is possible to *generate annotations from system-level data* automatically.

## 7.2 DESCRIPTION OF THE HOLISTIC PROCESS

The fundamental concept of the proposed holistic dataset creation process is to use the middleware recording facilities (summarized in [Section 5.4](#) on page 49) as the primary technical solution for recording. Apart from being the easiest way to record system-level data, this decision has several beneficial consequences. First, all data recorded using the middleware is automatically synchronized as soon as all nodes forming the distributed robot system are time synchronized. This can easily be solved with established solutions such as [NTP](#). Additionally, this approach also ensures that recorded system-level data can easily be reused for offline experiments with system components through the middleware utilities (cf. [Section 6.4](#) on page 65 for examples of such tools).

Regarding recording devices external to the robot system, whenever possible, appropriate devices or methods (e.g., network cameras instead of camcorders, grabber programs) should be chosen to interface these data streams with the middleware. Once these data can be captured using the middleware tools, the aforementioned benefits – especially the automatic synchronization – directly apply here as well, and the need for manual post-processing is reduced. This decision is based on the assumption that for large enough datasets and potentially multiple dataset recordings using the same setup, selecting and integrating suitable hardware and recording software requires less time than manual synchronization and error fixing. In case such a direct integration is not feasible or possible, we proposed to reintegrate

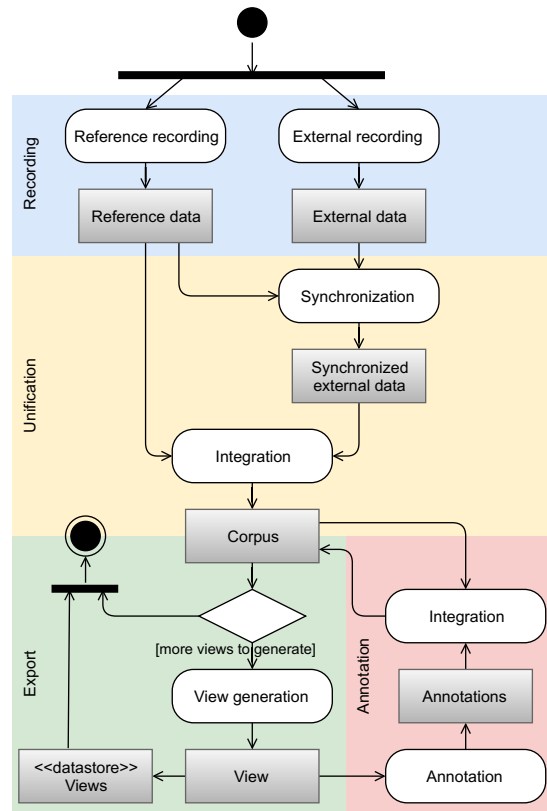


Figure 7.2: Schematic overview of the holistic dataset creation process as a UML activity diagram. Actions are depicted using white rounded boxes and produced data using gray boxes.

the data recorded using other technologies back into the middleware storage format in the post-processing phase. The same idea also applies to manually constructed annotations. As a result, the dataset is available in a single file format that is intrinsically synchronized. Therefore, all further tasks operate on a uniform storage format.

Figure 7.2 visualizes the proposed process as a set of necessary actions and their relations through generated data. In the holistic dataset creation process, the middleware data forms the *reference data* with intrinsic synchronization. Data that could not be gathered using this system is termed *external data* and the first post-processing step is to synchronize these data with the robot data. When designing the recording setup, it is advisable to think of automatic solutions for the synchronization to reduce the amount of manual work. In this approach this usually means that an additional component is added to the robot system that, at the same time, emits a system message and a physical signal noticeable by the external sensors (e.g., a beeping sound). After synchronization, the external data are then integrated into the middleware recording format through appropriate scripts, resulting in a unified corpus.

*reference data* ◆

*external data* ◆

To address the requirement of exporting parts of the dataset to common formats, we propose a *view-based approach* on the reference data. Views are immutable exports of (parts of) the datasets and their timing directly reflects the timestamps in the reference data. Apart from generating common export formats, further views are generated to serve as inputs for annotation. The manually created annotations are then fed back into the corpus by converting them to the middleware recording format. A further synchronization is not necessary as annotations are already based on the timing of reference data. View generation and annotation do not have to follow the strict ordering presented in the idealized process in [Figure 7.2](#) on the preceding page. Once all views have been generated and all annotations have been created and integrated, the result of the holistic dataset creation process is a corpus entirely represented in the middleware recording format, as well as a set of views for external use of the generated data.

### 7.3 REALIZATION BASED ON RSB

For the HUMAVIPS dataset we have realized the presented process based on the [RSB](#) middleware (cf. [Chapter 6](#) on page 51), which provides a suitable recording infrastructure with the *rsbag* tool (cf. [Section 6.4](#) on page 65). The following sections describe the realization of the process and the dataset recording.

#### 7.3.1 Data sources

All internal data from NAO as well as the control commands for remote operation have been recorded directly using the middleware without modifications to the system. The NAO robot, a control computer, and a recording server with sufficient disk space were synchronized using NTP to ensure time consistency.

Besides these system-level data, we used a Vicon motion capturing system<sup>1</sup> to acquire ground truth position data of participants and the robot. The tracking results of the motion capturing system were exported into the middleware with an adapter program that translated between a proprietary network protocol of the Vicon system and [RSB events](#). Additionally, we still recorded the complete source data of the motion capturing system using its proprietary software and storage format. These external data allow tuning the tracking parameters after the fact to correct potential tracking errors. This – unfortunately – is impossible with the direct export into the middleware. For an automatic synchronization of these external data in the unification phase, we started each recording trial with a clapperboard which carried markers for the Vicon system. The sound of the clapperboard was contained in the audio recordings of NAO’s microphones and

<sup>1</sup> <http://www.vicon.com>

therefore could be correlated with the clapperboard motion in the external recordings. These events can be detected automatically in both modalities (cf. [Section 7.3.3](#)).

For the additional cameras in the scene, we had to resort to camcorders due to technical restrictions with the available network cameras and webcams at the time of recording. Their external data can also be synchronized using the detection of the clapperboard sounds.

Finally, wireless close talk microphones carried by each participant were recorded directly in the reference data with an [RSB](#) adapter based on *GStreamer* [[Gst](#)].

### 7.3.2 Calibration

Besides the actual recording of the scenario with different participants, calibration sequences were recorded to unify the coordinate systems. During these recordings, a special Vicon marker was placed at different important positions such as the paintings. To automate the extraction of the appropriate measurements of the marker from the continuous detections, we used the clapperboard to mark the times when the marker was placed at the respective positions.

Besides this calibration aspect, we recorded a checkerboard pattern for all cameras (including NAO) so that distortions can be calibrated. Moreover, a special Vicon subject with 4 tracked markers has been presented to the external cameras and the Vicon at the same time. Hence, the location of each camera in the Vicon coordinate system can be computed if required.

### 7.3.3 Unification

After the recording sessions, the data were synchronized. To automatically synchronize the videos from the external cameras to the reference data, we calculate the cross-correlation peak of the cameras' audio channels with a reference audio channel from NAO's microphones. The audio processing tool *Praat* [[Praat](#)] was used to realize the required calculations. Based on the correlation peak we deduced the offset of the external videos to the audio from NAO, which in turn allowed us to compute the start time of the external videos in the [RSB](#) time frame.

Because of the expected tracking errors, the external data of the Vicon system had to be used instead of the automatically synchronized data. To fix the tracking errors manual parameter tuning in the Vicon software was necessary for each recorded trial. After this processing, an export of the tracking results to CSV as an intermediate format supported by the Vicon software was performed for later conversion to the [RSB](#) format. For synchronization, the detection of the clapperboard motion in the Vicon recordings was implemented by compar-

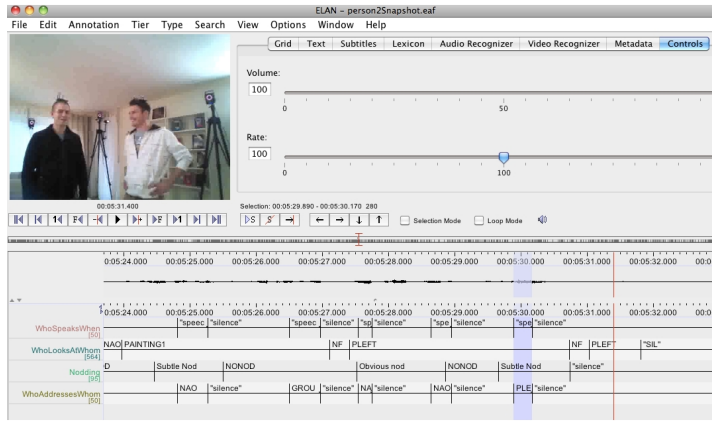


Figure 7.3: Export of one synchronized dataset trial to an *ELAN* [ELAN] project with added annotations.

ing the distances of the different markers attached to the clapperboard to select the moment the two board halves touched. The time of the clapperboard closing sound could have been detected using a template and cross-correlation again. However, because of issues with the detection accuracy, we had to resort to manual annotation using an audio editor.

#### 7.3.4 View generation and annotation

We decided to perform the manual annotation task using the established tool *ELAN* [ELAN; Wit+06]. For this purpose, we created a script to generate appropriate projects as views on the dataset. The script used the synchronized data created in the unification phase, converts video and audio to file formats compatible with *ELAN* using *FFmpeg* [FFmpeg], and automatically creates a project file to load in *ELAN* including system state information (cf. Figure 7.3).

In addition to the *ELAN* view, we have created several other views for analysis tasks. These include audio exports to wav format, video exports to MPEG-4, and CSV exports for the person tracking and the proprioception data of the robot.

## 7.4 SUMMARY

The presented holistic process for recording and post-processing datasets in robotics research takes a unique approach on the problem of recording and maintaining large datasets that comprise system and interaction data. Although others have presented systematic methods for acquiring datasets, some also in the form of frameworks (cf. Wienke et al. [WKW12] for a review of related work), none of the related approaches combines the benefits of automatic synchronization and replay abilities gained by using the middleware as the record-

ing framework. This decision improves the situation by reducing the manual effort required to perform the recording tasks. The proposed framework has been used for different scenarios – often though without the reintegration steps. Apart from the initial dataset published in Jayagopi et al. [Jay+13], this approach has also been used for a physical HRI study [Wre+13] and in the context of [Scenario CSRA](#) for automatic recordings of user interactions [Wre+17].

In the context of the work presented in this thesis, the holistic recording process is important for quantitative research tasks. These tasks require appropriate datasets, which have been recorded using the proposed process. Moreover, the method implemented for measuring the [resource utilization](#) of system components presented in the [next chapter](#) is influenced by the recording process and uses the middleware for the acquired measurements.



After introducing the basic building blocks of the systems this work is based on, this chapter now moves towards establishing [resource awareness](#). To realize any concept of resource awareness, the first necessary task is to uncover the [resource utilization](#) of the different system parts or [components](#) and to make the acquired [system metrics](#) available to the remaining system. Programs realizing this task are commonly called *collection daemons* or *collection tools* [[Full](#); [Telg](#); [Diam](#); [Cold](#)]. I have identified a set of [functional](#) and [nonfunctional requirements](#), which mandated the technological decisions for this task from the point of view of the [framework-level resource awareness](#) concept. From a functional perspective, the following requirements exist:

- FR1 System metrics need to be available at least at the level of operating system processes because in the targeted [microservice architectures](#) components are usually deployed as individual processes.
- FR2 Provided system metrics must cover at least the types of [resources](#) that are primarily affected by [performance bugs](#) as identified in [Section 3.3](#) on page 25. These are at least memory, [CPU](#), and network bandwidth.
- FR3 Data rates have to allow a correlation of utilization changes to potentially fast-paced task changes. For example, in [Scenario ToBi](#) on page 142 the object recognition component is activated sporadically for only one or two seconds. Thus, system metrics have to be updated at least every few seconds.
- FR4 Acquired system metrics must be made available following the ideas of the previously presented holistic dataset creation process (cf. [Chapter 7](#) on page 71) by exposing the acquired information via the [middleware](#). Apart from the aforementioned benefits for dataset creation, this approach also allows processing the gathered metrics at runtime using established and well-known interfaces and tools.

Additionally, I derived the following list of nonfunctional requirements from the framework-level resource awareness concept and further considerations:

- NFR1 The collection must be possible with the least amount of changes to existing systems or workflows (cf. [Chapter 4](#) on page 31). This, for instance, prohibits solutions that require an explicit instrumentation of each component.

NFR<sub>2</sub> Errors in the collection solution must not impair the [dependability](#) of the remaining system. For instance, a crash of a monitoring process must not crash functional components of the system. Software that is meant to improve the dependability of systems should obviously not impair it.

NFR<sub>3</sub> The processing overhead (own resource utilization) of the collection solution must be as minimal as possible. Any solution with a high processing overhead usually results in discussions with system developers and often ultimately refusal to use such a solution.

In the following sections, available technologies for system metric collection will be presented and evaluated regarding their applicability given the aforementioned requirements. I will primarily discuss solutions applicable to Linux because target systems of this thesis were Linux-based. After describing available collection methods, the established solution will be presented.

### 8.1 AVAILABLE SYSTEM METRIC SOURCES

The Linux operating system and its kernel offer different [application programming interfaces \(APIs\)](#) to acquire resource utilization information. These APIs form the potential [system metric sources](#) available in an unpatched Linux operating system without further specialized kernel modules or other extensions. Especially for the less widespread APIs, the documentation state is sometimes daunting. Man pages and the kernel documentation are often incomplete or outdated and refer to archived mailing list postings or the source code itself for undocumented aspects. Therefore, a solid assessment of the capabilities of the different system metric sources is a challenging task. For the following descriptions of the different available sources, [Table 8.1](#) on the facing page serves as a reference and tries to summarize the findings regarding the different solutions. APIs are described regarding the provided system metric categories, the scope at which information can be acquired (for a host, individual process, or event for threads/tasks), and further nonfunctional properties.

The oldest and most widespread system metric source is the proc file system [[Proc17](#)], which is a virtual file system that provides information about different internal data structure of the kernel in the form of pseudo-files. These files give access to a multitude of system metrics regarding the overall system as well as running processes and threads. As these metrics are spread across multiple files, bulk access to them is a rather costly operation and race conditions and inconsistencies might arise from the asynchronous, unsynchronized updates of the distributed information. Yet, the proc file system is the most established method to acquire system metrics and many common

		PROC	TASKSTATS	RUSAGE	CGROUPS	PCAP
SYSTEM METRICS	CPU	detailed	detailed	simple	simple	✗
	MEMORY	detailed	detailed	simple	detailed	✗
	I/O	detailed	detailed	✗	detailed	✗
	DESCRIPTORS	detailed	✗	✗	simple	✗
	BANDWIDTH	only host	✗	✗	✗	✓
	DELAYS	✗	detailed	✗	✗	✗
	TASKS	detailed	✗	✗	?	✗
SCOPE	HOST	✓	✗	✗	✗	✓
	PROCESS	✓	✓	✓	✓	✓
	THREADS	✓	✓	✓	?	✗
PROPERTIES	OVERHEAD	medium	low	low	medium	high
	CONSISTENT	✗	✓	✓	✓	✗
	DEPLOYMENT	easy	medium	difficult	difficult	medium
	AVAILABILITY	high	high	high	medium	high

Table 8.1: Comparison of common Linux system metric sources. Categories of available system metric: CPU utilization (CPU), memory usage (MEMORY), block device data throughput (I/O), information about open descriptors such as files and sockets (DESCRIPTORS), network throughput (BANDWIDTH), processing delays such as waiting for CPU or I/O (DELAYS), information about operating system tasks such as threads (TASKS). Scopes describe for which entities data are provided: the whole system (HOST), for a process (PROCESS), for individual threads/tasks (THREADS). Nonfunctional properties categorize: the processing overhead for getting data (OVERHEAD), whether data can be accessed without inconsistencies or race conditions (CONSISTENT), the required effort to use a solution (DEPLOYMENT), the availability across different systems (AVAILABILITY).

Linux tools use this mechanism (e.g., *top*, *ps*). Access to the provided information is usually available without requiring further authentication (as the root user) and *proc* is available on all common Linux distributions without further needs for configuration.

To reduce the overhead, avoid the consistency issues [Hol10], and to offer a unified interface for kernel accounting, the *taskstats* interface has been incorporated into the kernel [Lino6]. *taskstats* uses a netlink-based connection to distributed accounting information about Linux tasks (processes and threads). While the *proc* file system can use usual file system permissions to ensure that unprivileged users can only access detailed information about their own processes, a network-based protocol cannot use this mechanism. Therefore, only the root user can access this interface [Com11], which complicates its use compared to *proc*. Additionally, *taskstats* does not provide as many system metrics as *proc* and also only for tasks and threads, not for the host system.

Another means to acquire system metrics is the [Portable Operating System Interface \(POSIX\)](#) compliant system interface function `getrusage` [III08, p. 1068], which allows a process to acquire information about its own resource utilization or of its children. This complicates the application of this method for the intended use case because every component would need to be instrumented so that it can provide information about itself. An external approach is prevented by this method. Additionally, the provided set of system metrics is only limited.

Another modern approach to acquired system metrics for individual processes is termed *control groups* (cgroups) [Heo15]. Intended to restrict the resource utilization of individual or groups of processes, cgroups also allow measuring the utilized resources for each group. For this purpose, control groups for the desired aspects of resource utilization need to be created and processes have to be assigned to them. Afterwards, resource utilization can be measured for each group and a multitude of different aspects using a virtual file system with the same drawbacks that apply to `proc`. cgroups are a relatively modern feature, and they are primarily used in advanced scenarios such as virtualization. The situation regarding the way how control groups are handled and which front ends exist differs between Linux distributions and versions. Also, recently the API has been completely renewed [Ros16]. Therefore, using cgroups generically across different systems is not easily possible and efforts need to be made for each started process to assign it to an appropriate control group.

All APIs mentioned so far are closely coupled to the Linux kernel and undergo frequent extensions and changes. Especially, the interpretation of system metrics might change from one kernel version to the next as they are often direct reflection of the internal data structures of the kernel.

The aforementioned APIs did not provide information on the network bandwidth that is used per process. The `proc` file systems offers accumulated metrics for the host system, but not for individual processes. To acquire system metrics on this level, network packets need to be sniffed and attributed to the running processes. On Linux, the de facto standard method for performing packet sniffing is *libpcap* [Pcap; Pcap17]. Captured network packages can be tracked back to the sockets that were used to send or receive them and these descriptors of these sockets can be attributed to individual processes using the `proc` file system. While the task of sniffing all packages on a busy host is already computationally intensive, the additional parsing of the `proc` file system adds even more processing overhead. Moreover, the `proc` file system does not provide immediate notifications in case new processes are spawned. Thus, some packages might not be correctly related to their causing processes, resulting in slight

inconsistencies when measuring the actual network bandwidth per process. Nevertheless, this seems to be the only possibility to acquire these detailed system metrics. As sniffing all packages including their content is a potential security issue, this operation requires root permissions or special configurations, increasing the efforts for a generic application of this method.

From the presented system metric sources, only the `proc` file system allows fulfilling [NFR<sub>1</sub>](#). All other sources require special care when deploying a collection solution based on them. Moreover, `proc` offers the widest range of system metrics. Therefore, this source is the primary provider for system metrics for this work. To meet [FR<sub>2</sub>](#), which requests network bandwidth to be included in the uncovered system metrics, also `libpcap` has to be used, despite violating [NFR<sub>1</sub>](#), as no other solutions exists for this problem.

## 8.2 RESOURCE ACQUISITION TOOLS

A multitude of tools and libraries that realize the collection of system metrics exists. These tools cover different resources, provide them in different ways, and have varying footprints. Despite their availability, I have decided to implement the collection from scratch and the following paragraphs will explain the necessity of this task.

A first category of existing tools is command line tools for Linux/Unix that give access to a single kind of resource for manual system inspection such as `htop` [[Htop](#)], `iostat` [[Iostat](#)], `nethogs` [[Neth](#)], `netstat` from `net-tools` [[Nett](#)], or `lsof` [[Lsof](#)]. These tools already contain the necessary code to read out the respective sources (`proc` and `libpcap`). However, they only focus on a single aspect each and – apart from `nethogs`, which just recently started deploying its functions also as a shared library [[Bou16](#)] – are not intended to be integrated into different contexts such as a program that exposes the collected information via the [RSB](#) middleware.

Tools such as `nmon` [[Nmon](#)] or `Glances` [[Gla](#)] combine metrics for different resources with the aim to present information with a unified view. While `nmon` is also only designed as a standalone tool, `Glances` started to expose its results on the network via [JSON](#) or [Extensible Markup Language Remote Procedure Call \(XML-RPC\)](#) with version 2 [[Hen17](#)], which did not exist at the time this project started.

Another set of existing tools is primarily geared towards continuously acquiring and persisting system metrics related to the host system while ignoring individual processes. `sar` from `Sysstat` [[Syss](#)] is a daemon to continuously collect metrics and to store them in custom files. `collectl` [[Coll](#)] can be used for the same purpose. Moreover, it can distribute the collected metrics using a socket interface. These tools only focus on the host and are therefore not appropriate for the intended use case.

From the domain of server and infrastructure monitoring, collection daemons exist, which are able to provide resource utilization metrics apart from more specialized metrics. Exemplary tools comprise *collectd* [Cold], *Diamond* [Diam], *fullerite* [Full], and *Telegraf* [Telg]. They are primarily used to fill time series databases / metric stores used for visualization and anomaly detection purposes. Therefore, they assume an appropriate database to be running and network protocols are used to submit the data. Apart from the aforementioned restriction that these tools often only provide resource utilization information for the host system, data rates in these applications are usually lower.

Most of the presented tools have drawbacks for the intended use case such as:

- They often focus on a single resource type and a combination of multiple tools would be needed (increasing the deployment overhead, which conflicts NFR<sub>1</sub>).
- They are often not intended to be integrated into novel environments.
- Tools that provide network interfaces require adapters to translate the collected metrics to RSB events, which increases processing overhead (NFR<sub>3</sub>) and deployment efforts (NFR<sub>1</sub>) because of the additionally needed processes.
- Tools that are sufficiently flexible to integrate custom output sinks to implement an RSB export are highly generic and require a high configuration overhead (e.g., *fullerite* [Full], conflicts NFR<sub>1</sub>).

Therefore, I did not consider any of these tools to realize a system metric collection solution suitable for the easy application demanded by the framework-level resource awareness concept. Instead, I have created a custom implementation suitable to fulfill all stated requirements for the intended application in RSB-based scenarios.

### 8.3 IMPLEMENTATION

The system metric collection solution implemented for this thesis is structured into two distinct executables representing different collection scopes. On the one hand, the *host collector* realizes the collection of host-related system metrics while, on the other hand, the *process collector* is responsible of collecting per-process system metrics (FR<sub>1</sub>). This explicit distinction is suitable because the different reporting scopes imply different lifetimes and partially different requirements. The process collection task must be configured for each new process (representing a component) whereas the host collection is independent of the lifetime and is only needed once per host.

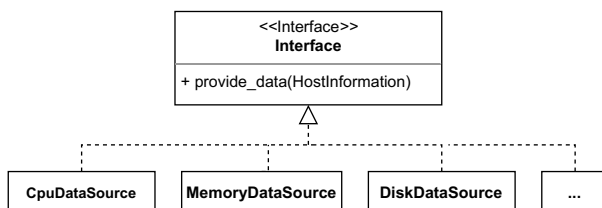


Figure 8.1: Data source structure of the host collector daemon as a UML class diagram.

Generally, both collectors use a periodic reporting scheme where current system metrics are reported with a configurable frequency. This frequency can be used to trade off between the desired temporal resolution of the data ([FR<sub>3</sub>](#)) and the imposed processing overhead ([NFR<sub>3</sub>](#)). System metrics are reported using appropriate [RST](#) data types via the [RSB](#) middleware to fulfill [FR<sub>4</sub>](#). The detailed list of collected host and process system metrics is available in [Appendix B.7](#) on page [225](#).

### 8.3.1 Host collection

Collecting system metrics for a host system is a relatively lightweight task and therefore no special care is needed to reduce the resource utilization of the collection daemon itself (cf. [NFR<sub>3</sub>](#)). Therefore, the *host collector* could be implemented as a Python program based on the established *psutil* [[Psut](#)] library. *psutil* provides an abstraction of common Linux system metric sources, especially the `proc` file system. It is an established library, which is available in many common Linux distributions and can easily be installed using the standard packaging tools (e.g., `pip`). Therefore, a dependency on it does not conflict with [NFR<sub>1</sub>](#). For the host itself, the `proc` file system provides all necessary metrics directly without the need for special configurations or root permissions and *psutil* wraps all required aspects. Hence, [FR<sub>2</sub>](#) can easily be fulfilled for the host level with this approach.

For reporting the data via [RSB](#), the `rst.devices.generic.HostInformation` [RST](#) data type is used, which groups host-related system metrics into different related units, for instance, network, CPU, and memory. As depicted in [Figure 8.1](#), the implementation of the host collector picks up this logical separation and is structured into different data sources. Each of them is responsible of acquiring the respective metrics, which are filled into an instance of `HostInformation` in the `provide_data` method. The orchestration code of the host collector periodically prepares such a message data structure, calls all data sources, and publishes the acquired metrics using the middleware.

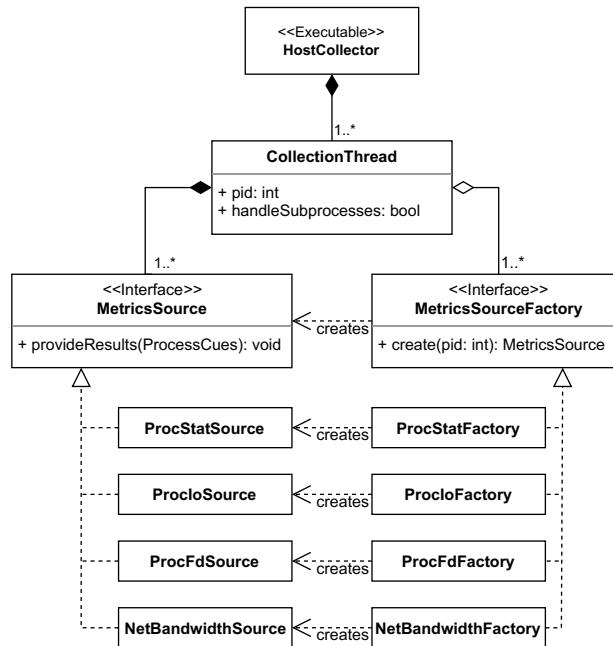


Figure 8.2: Structure of the process collection daemon represented as a UML class diagram. Metric sources are responsible of contributing coherent aspects to the reported data. The collection threads dynamically creates sources for subprocesses using factories.

### 8.3.2 Processes collection

For collecting system metrics on a process level, more care needs to be taken regarding the resource utilization the collection daemon itself imposes. The number of components and therefore processes is usually much higher than the number of hosts and therefore more data needs to be acquired and processed. Therefore, the *process collector* daemon has been implemented in C++ to reduce the own processing overhead (cf. [NFR<sub>3</sub>](#)). The process collector follows the same principles regarding dependencies and lock-in as the [RSB](#) middleware itself (cf. [Chapter 6](#) on page 51) and avoids unconventional dependencies that could prohibit an easy application ([NFR<sub>1</sub>](#)).

Structurally, the process collector daemon is organized in a comparable way to the host collector. [Figure 8.2](#) shows that a set of *MetricSource* instances is used to acquire the desired system metrics. In contrast to the host collection, these sources are not solely determined by their purpose but also by the system metric source they represent. The differences in system configuration and usage requirements (e.g., requiring root permissions) have influenced this decision. Hence, users can select appropriate sources matching the desired set of system metrics to collect and the effort required to configure the system ([NFR<sub>1</sub>](#)). Each source is responsible of placing the collected system metrics into an instance of the [RST](#) data type `rst.monitoring.ProcessCues`.



The daemon executable can be launched to monitor multiple processes in parallel. Each of the processes is then monitored by a single `CollectionThread`, which implements the fixed frequency sampling. The ability to handle multiple processes in parallel gives some freedom regarding the way the daemon is integrated into the orchestration of existing systems (e.g., one instance per component or a single instance per host; [NFR1](#)). Moreover, sources might be able to optimize their processing in case they operate for multiple processes in parallel ([NFR3](#)). This is especially important for the `NetBandwidthSource`, which collects per-process network bandwidth information using `libpcap`.<sup>1</sup> As explained in [Section 8.1](#) on page 80, this approach requires sniffing all network packages and relating them to the running processes. This computationally intensive work should be performed only once per host system and therefore a single instance of the collection daemon per host is best suited for this task. Moreover, the special configuration for packet sniffing only needs to be applied to a single daemon instance in this case. The combination of the available sources for the `proc` file system as well as for `libpcap` includes all resources demanded by [FR2](#).

The process collection daemon operates as an executable that runs in parallel to the monitored processes, which are identified by their [process identifiers \(PIDs\)](#). This method of operation is supported by the implemented sources and ensures that a potential crash of the collection daemon does not influence the running processes ([NFR2](#)).

### 8.3.3 Subprocess handling

The microservice architecture style used by potential target systems of this work does not impose restrictions on the individual components or service implementations. So, component implementations are not restricted and can use child processes to perform their operations. This situation already arises in case a process is started by a script which configures the actual binary to launch. In this case there will be a permanent process hierarchy for the whole runtime of the component with the root being the script executor and the actual implementation being a child process of the executor. In other situations, a component can temporarily launch child processes to perform some operations. Only collecting system metrics for the root process in the actual hierarchy of processes would therefore ignore important parts of the effective resource utilization of that component. Therefore, the process collection daemon explicitly includes measurements for the whole process hierarchy starting from the root component

<sup>1</sup> The implementation of the `NetBandwidthSource` is based on `nethogs` [[Neth](#)]. As the standalone `nethogs` library did not exist at the time the process collector was implemented [[Bou16](#)], this is a copy of the source code with optimizations for the specific use case.

PID. To avoid complicating the implementation metric sources with the subprocess handling, the `CollectionThread` itself is responsible of this task. Sources, on the other hand, are coupled to a single process and can ignore the hierarchy. At each measurement iteration, the collection thread determines the process tree for the monitored component, instantiates new sources for subprocesses that have appeared since the last iterations, and discards sources for subprocesses that have terminated. To enable this behavior, the thread instance is not directly parameterized with the `MetricsSource` instances themselves, but with `MetricsSourceFactory` instances that create the respective sources for individual PIDs.

#### 8.3.4 *Data representation*

For both collection daemons, I have decided to provide the acquired metric values directly in the way as gathered from the system metric source, even if this strategy prevents an abstraction from different sources providing comparable metrics, and also from the operating system itself. As already explained in [Section 2.1.2](#) on page 10, different system metric sources generate different kinds of measurement artifacts in the acquired metrics and a general method to filter these artifacts does not exist. Moreover, downstream computations may make use of specialized strategies to deal with these artifacts more efficiently than a general method can. Finally, many of the metrics provided by the Linux kernel are represented as increasing counters instead of current bandwidth measurements. While being unintuitive, these counters have many benefits for further processing. For instance, even if the aforementioned collection thread misses a new subprocess in one iteration due to race conditions in the `proc` file system, the next iteration still includes the complete resource utilization since the start of the new subprocess. Therefore, these counter representations prevent data loss for some concurrent operations and also more accurate subsampling after the fact without the need for integrating multiple subsequent measurements.

Regarding the representation of data for subprocesses, a similar strategy has been used. The whole process tree is represented with measurements attached to each process in the tree. This representation enables downstream computations to individually decide how to deal with the information.

#### 8.3.5 *System integration*

To facilitate the integration into existing architectures, the collection daemons can be attached to the system in multiple ways. [Figure 8.3](#) on the next page visualizes the possibilities. The process collector daemon provides two distinct modes of operation. On the one hand,

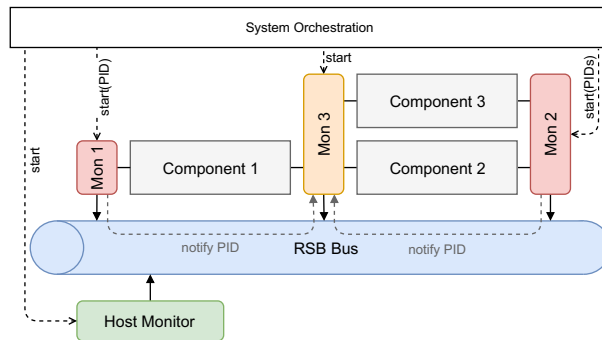


Figure 8.3: Integration of host and process collection daemons into a system. Process collection daemons can either be started for a fixed set of processes specified via PIDs as startup (Mon 1 and Mon 2), or without such an initial association (Mon 3). In that case, the PIDs to monitor is expected via [RSB](#).

the processes to monitor can be specified using command line arguments (Mon 1 and Mon 2 in [Figure 8.3](#)). The collector will then operate until all processes to monitor have disappeared. Commonly, a single collection daemon is attached to a single component to allow restarting of individual components (Mon 1), but also multiple components are supported (Mon 2). On the other hand, a remote control mode is implemented. In this case, the collector will be launched without specifying PIDs on the command line (Mon 3). Instead, processes to monitor are accepted on an [RSB](#) channel. Host collection daemons are started on each host and no association to processes is necessary.

In the systems used for this work, the command line mode has been used for proc file system based data and for every process a new instance of the collector was started. This was easy to integrate into the existing system orchestration by generically instrumenting the component start function. In contrast, network bandwidth collection has been realized by starting a single collector instance per host in remote control mode to avoid unnecessary overhead. Sending the appropriate PIDs to the remote controlled instances has been realized by letting the existing collector instances for proc-based system metrics send out the processes they monitor periodically via [RSB](#) (cf. [Figure 8.3](#)).

#### 8.4 SUMMARY

Based on a review of existing solutions and sources, I have realized daemons to acquire system metrics for components and the host system. These daemons follow the ideas of the framework-level resource awareness concept as they can be applied with a limited configuration overhead and without special instrumentation. Moreover, they integrate with the holistic recording process to support experimental research tasks, including the work pursued in this thesis. On cur-

rent operating systems, the selected approach unfortunately implies that the number of acquired system metrics is limited and the produced time series contain discretization artifacts. The following methods based on these data have to deal with these inaccuracies, which are of different severity for the different kinds of metrics.

## Part III

### DEVELOPER PERSPECTIVE

An important aspect of improving the dependability of robotics and intelligent systems is to establish an understanding among system developers on how the utilization of system resources behaves and is influenced by development decisions. Methods introduced in this part therefore aim to establish resource awareness by informing developers about these aspects and by providing methods to control the resource utilization during development work.



A first possibility to increase the [resource awareness](#) in robotics and intelligent systems is to make their developers aware of [resource utilizations](#). Through appropriate visualizations, [system metrics](#) can be made graspable at system runtime. In the infrastructure monitoring domain, this task is often accomplished by using browser-based [dashboards](#). Generally a *dashboard* has been defined as a “visual display of the most important information needed to achieve one or more objectives; consolidated and arranged on a single screen so the information can be monitored at a glance” [Few04]. Dashboards have found wide adoption in different domains (e.g., for business decisions [Eck11]) as a tool to quickly understand the current state of operations with the aim to derive necessary decisions. In the context of infrastructure monitoring, the types of dashboards that are typically used are called *operational dashboard* (in contrast to strategic or analytical dashboards) [Few06] with their focus on displaying primarily the current state of a system and the immediate past.

◆ *dashboard*

As shown in the next section, a large amount of solutions for realizing operational dashboards for infrastructure monitoring exists and these tools are in frequent use. Yet, up to my knowledge, no scientific study exists that systematically validates their usefulness. The wide adoption is an indicator that valuable insights can be achieved with these tools and Fatema et al. [Fat+14] conclude for monitoring solutions in general (in a cloud context) that they “have an important role [...] by allowing informed decisions to be made regarding resource utilisation” [Fat+14]. Therefore, I have decided to implement such a tool for the robotics domain with a focus on monitoring the resource utilization of the system and its [components](#). As described in [Section 4.2.5](#) on page 37, only few solutions exist in robotics. They focus on ROS-based systems and due to the extensive coupling to ROS, adaptation of these approaches for other [middlewares](#) was not easily possible. Therefore, I have realized the dashboard using generic tools from the infrastructure monitoring domain. In the following, I will review existing tools, introduce the architecture of the created solution, and present an evaluation for the usefulness of the approach.

### 9.1 AVAILABLE TOOLS

A vast amount of tools exists to realize the monitoring task for server infrastructure. Apart from the visualization of system-level [KPIs](#) using dashboards, these tools also pursue other tasks such as presenting

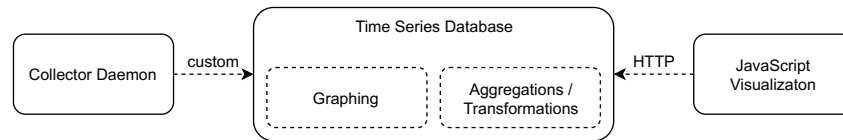


Figure 9.1: Common implementation scheme of dashboards.

business [KPIs](#) or automatic alerting in case of system issues. In the following discussions, I have only included tools that support the realization of operational dashboards for system metrics.

Generally, all solutions used to create resource utilization dashboards follow a common theme: A collector daemon acquires system metrics and stores them in a time series database. A front end is then constructed asynchronously based on queries to the stored data.

Early implementations of this approach are usually based on *RRDtool* [[RRD](#)], which is a fixed-size round robin database for time series with accompanying graphing tools. Examples for integrated monitoring solutions that provide dashboards based on *RRDtool* are *Munin* [[Munin](#)], *Cacti* [[Cacti](#)], and *Ganglia* [[Ganglia](#)]. The presented graphs in these dashboards are included as image files, which is the standard output format of the *RRDtool* graphing commands. Therefore, constantly updating these dashboards is an expensive task and direct user interaction with the graphs is limited.

Modern implementations usually separate the visualization from the database implementation by means of a remote [API](#). While the database provides data aggregation methods, the main visualization is browser-based. This results in the scheme depicted in [Figure 9.1](#).

Probably the oldest implementation of this approach is *Graphite* [[Graph](#)], which adds an [HTTP](#) API to its own *RRDtool*-like round robin database *whisper*. Besides raw data access, this API also provides a selection of commonly required transformation and aggregation functions for the data. Despite also providing integrated graphing code, *Graphite* is nowadays primarily used in combination with more advanced visualization front ends.

In contrast to the fixed-frequency round robin databases, a set of more flexible time series databases has evolved recently. These databases do not require data to be recorded at fixed rate and their data access and querying features are often designed more thoroughly. Popular candidates from this category include *InfluxDB* [[IDB](#)], *OpenTSDB* [[OTSDB](#)], *Prometheus* [[Prom](#)], and *KairosDB* [[KDB](#)].

On top of the time series databases, dashboards are commonly implemented as JavaScript front ends rendered in a web browser. One of the most prominent solutions is *Grafana* [[Grafana](#)],<sup>1</sup> which supports a variety backends and plugins for different graph types. Other, less known or maintained solutions include *Cyclotron* [[Cyclo](#)] and *gdash* [[Gdash](#)], or *Open MCT* for space missions [[TR16](#)].

<sup>1</sup> Close to 19000 stars on Github at November 14, 2017.



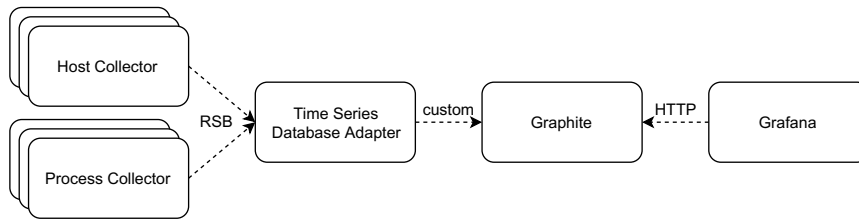


Figure 9.2: Data pipeline of the realized resource utilization dashboard.

Finally, some tools offer integrated solutions spanning multiple of the aforementioned tasks. In the open-source world, *netdata* [netdata] is such an example. Moreover, many commercial services exist.

Commercial services requiring a connection to an external host cannot be used in robotics settings with potentially unstable Internet links. Moreover, the necessity to acquire a license contradicts an easy application as required by the *framework-level resource awareness* concept. Therefore, I did not consider these solutions as candidates for my own implementation. I have also excluded solutions which are tightly integrated with an existing collection daemon. These existing daemons have not been considered for my work because they did not meet the requirements on the collection process explained in [Chapter 8](#) on page 79. Dashboards that are tightly integrated with their own collection daemons are hardly usable without them. I finally decided to use *Graphite* and *Grafana* [Grafana] to implement the resource utilization dashboard. Even though more modern time series databases provide interesting features to create suitable dashboards more easily, initial experiments with *InfluxDB* have shown that their flexible design imposes a higher resource utilization.

## 9.2 RESOURCE UTILIZATION DASHBOARD IMPLEMENTATION

In contrast to the general implementation scheme for dashboards presented in [Figure 9.1](#) on the preceding page, the existing collection daemons use the *RSB* middleware to publish system metrics instead of directly sending the data to target time series database as a consequence of supporting the holistic recording process. Therefore, an adapter is required to bridge between the *RSB event* bus and the proprietary data format and transmission protocol of the target database *Graphite*. The resulting processing pipeline is visualized in [Figure 9.2](#) with exact tool names.

### 9.2.1 Time series database adapter

The primary task of the time series database adapter is to translate between the *RSB* events containing *RST* data types and the custom protocol of the target time series database. Moreover, some raw sys-

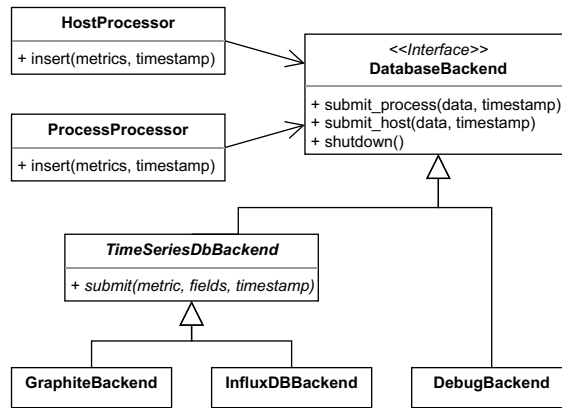


Figure 9.3: Architecture of the time series database adapter.

tem metrics contained in the [RST](#) data types are hard to visualize with the available solutions (e.g., aggregating data from a variable number of subprocesses). Therefore, the adapter also performs a pre-processing of the data before passing them to the database. For this purpose, the adapter implementation is structured as depicted in [Figure 9.3](#). Two distinct processor classes implement the necessary computation to prepare the data for storage in a database. Preprocessing of host system metrics is separated from process-level system metrics because of different data types and calculations that are necessary. After preprocessing, the data are handed over to an instance of a `DatabaseBackend`, which is responsible of storing the data in the actual database. Besides a debug backend which prints the generated data, two real storage backends have been implemented: one for the target database *Graphite* and one for testing purposes with *InfluxDB*. In both databases, metrics are stored as time series that are identified by string-based keys. The intermediate abstract class `TimeSeriesDbBackend` converts the structured data passed to the two `submit_*` methods to this string-based representation, before handing it over to the concrete implementations via the `submit` method.

For *Graphite*, the backend uses the Python *pickle*-based [\[PP17\]](#) protocol to send the data to the database using a socket connection. This protocol is the most efficient one and easy to implement in the Python-based adapter code.

The preprocessing performed by the two processor classes is best described for the process scope case (cf. [Figure 9.4](#) on the facing page). First, the data from potentially running subprocesses is integrated into the system metrics of the root process for a monitored component. This is achieved by summing up all numeric system metrics and creating a new virtual metric with the number of subprocesses. Afterwards, the data for the subprocesses is discarded. This operation is necessary because detailed plots on the level of potentially transient subprocesses of a component are usually not necessary and contradict the idea of dashboards as a quick overview. A detailed rep-

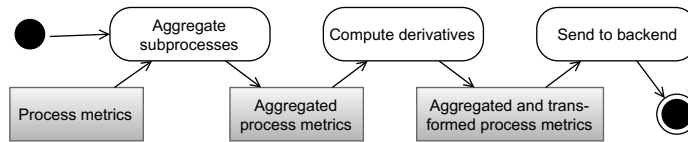


Figure 9.4: Processing steps performed by the time series database adapter represented as a UML activity diagram.

resentation of the variable number of subprocesses per component is hard to achieve in time series databases that are based on a fixed set of stored metric keys.

The second preprocessing step is to compute derivatives for all system metrics that are represented as counters. Computing this derivative on the fly for visualization purposes would be possible but costly. Moreover, the current value of a metric is usually what is visualized in a dashboard and not the accumulated counter. Therefore, this step reduces the computational demands of the dashboard rendering process and the complexity for creating the appropriate queries to visualize the data. Afterwards, the preprocessed data are handed over to the database backend.

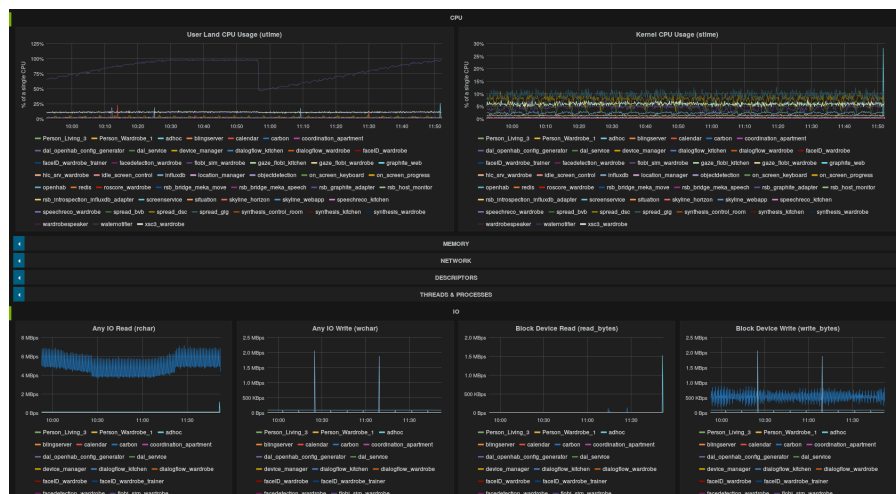
### 9.3 DASHBOARD DESIGN

Following the framework-level resource awareness concept, I have designed two *Grafana* dashboards that can be used generically on any system monitored using the proposed pipeline without requiring further configuration. One dashboard is responsible of displaying the host-level resource utilization (cf. [Figure 9.5a](#) on the next page) while the other focuses on the individual components (cf. [Figure 9.5b](#) on the following page). Both dashboards are structured along the primary types of [system resources](#) via distinct graph *rows* provided by *Grafana*. Each row contains separate plots for the different system metrics reflecting the [resources](#) and can be collapsed (as done for a few resources in [Figure 9.5b](#) on the next page). Within each graph, the history of the system metrics is plotted per process or host. The time range is configurable and individual plots can be focused by the user via a full screen display. Finally, the total amount of components and hosts can be filtered for the whole dashboard to temporarily limit the number of visible graphs.

Even though the generic dashboards provide an easily applicable solution – depending on the number of components or hosts – the visual load might become high. Therefore, more targeted and specially crafted dashboards might provide a more suitable and better graspable solution for individual application scenarios. By applying an established technology (*Grafana*), creating new dashboards or adapting existing ones is facilitated for end users. *Grafana* [[Grafana](#)] is well maintained, extensive help is available through online documentation



(a) Host-level dashboard



(b) Process-level dashboard

Figure 9.5: Screenshots of the generic dashboards for inspecting a system’s resource utilization with situations from [Scenario CSRA](#).

and community support, and many examples can be found on how to construct dashboard for specific use cases. Therefore, the support situation for end users is much better compared to a custom solution.

## 9.4 EVALUATION

The dashboards have primarily been used in two scenarios: In [Scenario CSRA](#) on the facing page they are in constant use to monitor the operations of the system. In [Scenario ToBi](#) on page 142 the dashboards were used as a validation while recording the dataset presented in [Chapter 12](#). In the following sections I will present qualitative and quantitative evidences to underline the usefulness of the approach for robotics and intelligent systems.

The **Cognitive Service Robotics Apartment (CSRA)** is a smart home environment operated within a research project of the Cluster of Excellence Cognitive Interaction Technology (CITEC) at Bielefeld University [Wre+17]. The Cognitive Service Robotics Apartment (CSRA) is an apartment equipped with a dense sensor and actuator network in which multiple embodied agents such as a service robot interact. On the one hand, research focuses on establishing a software architecture that enables the continuous operation of the system while providing the necessary facilities for qualitative and quantitative research work inside the apartment. For this purpose, the appropriate recording and processing capabilities have to be provided. On the other hand, learning and interactions in such an environment are analyzed, especially regarding the use of the embodied agents. The CSRA is one of the largest systems ever built at CITEC, comprising more than 200 component distributed across 22 hosts, connected using the **RSB** middleware (as of November 16, 2017).



The CSRA living room and kitchen area with different sensors, actuators, and agents (photo: CSRA, CITEC, reproduced with permission). The dashboard is constantly visible for developers on a large screen shown on the left side.

#### Scenario CSRA: The Cognitive Service Robotics Apartment (CSRA)

##### 9.4.1 Qualitative evidences

In both application scenarios, merely presenting the resource utilization to the developers has already helped to identify a set of previously unknown **performance bugs** in the system.

In the CSRA system, an unintended cyclic **CPU** utilization pattern could be identified with the approach. As visible in **Figures 9.5a** and **9.5b** on the facing page, the `flobi_sim` component, while theoretically idling, produced a long-running saw tooth pattern of CPU utilization. This issue was clearly identified once the system metrics were visualized in the dashboards.

A second issue in the CSRA system that could be identified using the dashboards was an interesting interplay between the way one component used the **RSB** middleware and the implementation of **RSB** itself. The developers of the system noticed that once the component was started, the system's **performance** started to degrade.

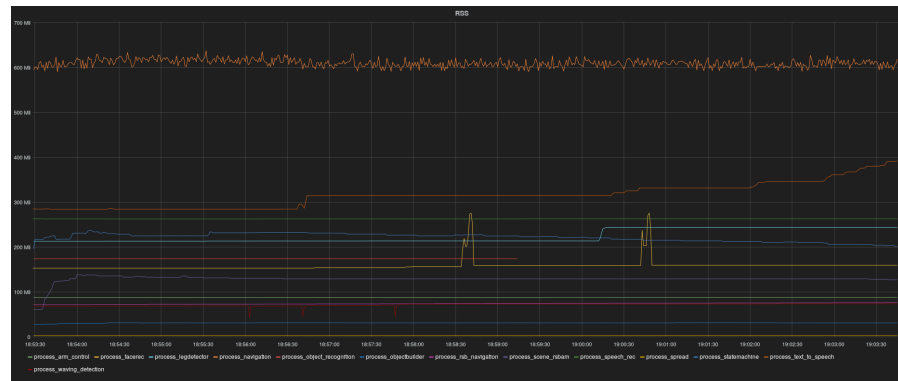
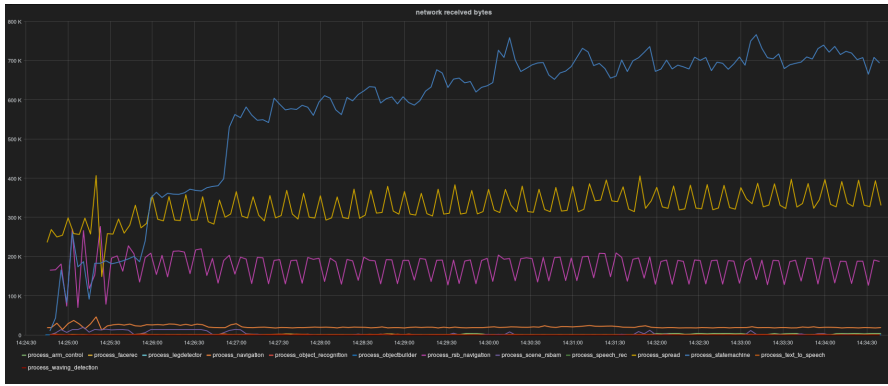


Figure 9.6: Memory leak of the TTS component in the *ToBi* system visible in the resident set size (RSS) graph of the process-level dashboard (TTS component: brown line, second one from the top).

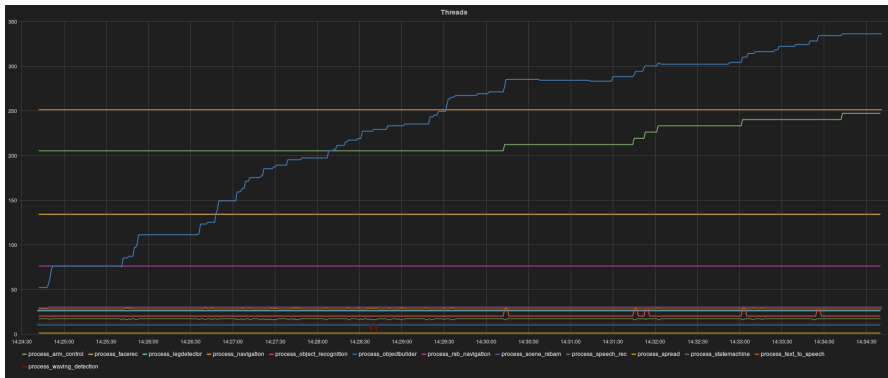
Through the dashboards it could quickly be revealed that this component opened more than 1000 network connections on start, which effectively overloaded the *Spread* [[Spread](#)] daemon that is used as the [transport](#) in this system. A 100% CPU utilization of the spread process was subsequently visible in the process-level dashboard. With another dashboard created to visualize details of the [RSB introspection](#) mechanism it could be verified that this component used a lot more [participants](#) than other components and because network connections were not shared between participants, the *Spread* daemon was flooded with as many connections as participants existed. After adding connection sharing in [RSB](#), this issue could be resolved.

In the *ToBi* system, two notable issues were identified based on the dashboards. First, a memory leak could be identified in the [TTS](#) component of the system. [Figure 9.6](#) shows a screenshot from the respective graph of the dashboard that helped to identify the leak.

A second issue in the *ToBi* system that was unnoticed before deploying the dashboard in the system was related to the handling of [RSB](#) participants in the state machine library *BonSAI*. This Java library contains code to automatically create [RSB](#) participants at the time a state or transition depends on them. Once created, these participants are stored in a hash map using a utility key class for retrieval in case a subsequent state requires a participant of the same type for the same [scope](#) again. This key class did not implement `hashCode` and `equals`. As a consequence, cached participants could not be looked up in the map. Instead, new instances were created and the old ones were not cleaned up. Consequently, more and more network connections piled up during runtime, which received the same data. Therefore, the total network bandwidth of this component increased over time. Additionally, as each receiving participant has its own thread in [RSB](#), also the number of threads of the component increased during the runtime. These effects, depicted in [Figure 9.7](#) on the facing page, were easily spotted using the dashboard.



(a) Incoming network traffic



(b) Number of threads

Figure 9.7: Screenshots of two dashboard graphs for an [RSB](#) participant leak in the state machine of the *ToBi* system (blue lines).

#### 9.4.2 Quantitative evaluation

Apart from the previously presented qualitative evidences, I have also tried to acquire quantitative data that underline the usefulness of the approach. For this purpose, I prepared an online questionnaire that was distributed across the developers participating in the CSRA scenario (cf. [Scenario CSRA](#) on page 99). The structure of the survey is documented in [Appendix C](#) on page 231 and in the following I will refer to the individual questions from this structure. Because of the limited size of the project, only 7 answers were submitted.

##### 9.4.2.1 Dashboard usage

A first set of questions was designed to understand in which situations the dashboards are being used and how useful they are in these situations. I first asked the participants to rate how often they consult the dashboards in different typical situations that appear in the scenario ([C.2.1](#)). Answers could be given on a five-point scale from *never* (0) to *always* (4). As shown in [Figure 9.8](#) on the next page, the most valuable situation is in case a problem has been detected. The

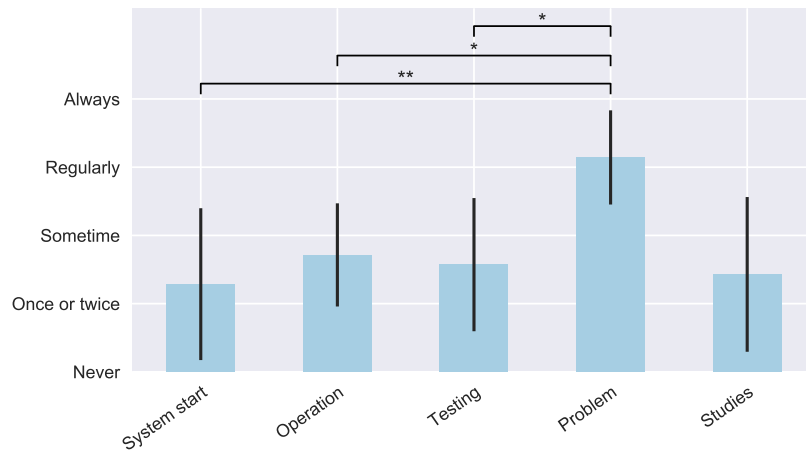


Figure 9.8: Usage frequency of the dashboards in different situations. Significances have been computed using a two-sample dependent t-test after checking for normality with a [Kolmogorov-Smirnov \(KS\)](#) test.

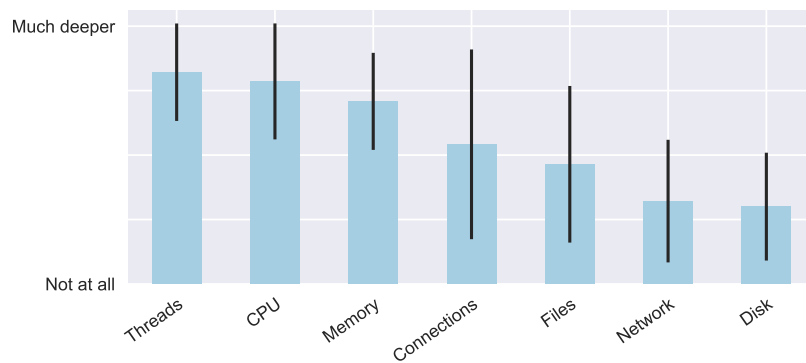


Figure 9.9: Rating of how much the dashboards improved the understanding about the utilization of different types of resources.

developers use the dashboards significantly more often in this situation than in most of the other situations they could rate. The largest individual differences exist for study situations and the system start.

To get an impression how much the dashboards help to get insights into the resource utilization of the system and its individual components, I asked participants to rate this on a five-point scale from *no insight at all* (0) to *better insights than before and with any other tool* (4) (C.2.3). For both, the hosts and the individual components, results are comparable with a mean of 3.14, which indicates that the CSRA developers have gained good insights by using the dashboards.

All participants indicated that they have a better understanding for resource utilization as a result of using the dashboards (C.2.3). I have further detailed this analysis by asking for individual resources (C.2.4). [Figure 9.9](#) visualizes the results. The rating of network bandwidth is probably spoiled as the per-process network bandwidth collection daemon was not running continuously in the CSRA system. Especially knowledge about the number of threads per process/com-



ponent seems to improve. This can potentially be explained because common command line tools such as *htop* usually do not display this system metric directly.

Finally, I asked whether the availability of the dashboards reduced the use of other tools for the purpose of understanding resource utilization (C.4.2). The answers to this question are mixed and no clear tendency can be determined.

#### 9.4.2.2 *Usefulness for debugging*

A second set of questions tried to evaluate the usefulness of the dashboards for debugging purposes.

With question C.3.1 I have asked how often detected bugs were visible in the dashboards. Participants could rate on a five-point scale from *never* to *always*. Mean and median reply agree on the result *sometimes*, which is expected, as not all bugs are performance bugs.

When asked whether the dashboards help to isolate the origin of bugs, all participants agreed that the dashboards help (C.3.2). Moreover, all participants indicated that they have found bugs they would not have found otherwise by using the dashboards (C.3.3). The types of bugs the participants reported they have found (C.3.4) can all be categorized as performance bugs, being either leaks or overloading of individual resources. Please refer to Appendix D.1 on page 235 for the exact answers.

## 9.5 SUMMARY

Despite the limited sample size of the presented survey, the results strongly suggest that dashboards focused on resource utilization aspects of a system are a valuable tool for developers and operators of robotic and intelligent systems. For all important aspects, participants of the survey have indicated that the dashboards have improved their understanding of resource utilization aspects. The dashboards are – as intended – used most often to identify and debug performance problems and the participants of the survey have clearly indicated that dashboards help to find the bug origins. The list of identified bugs acquired in the survey as well as the selected performance bugs described in Section 9.4.1 on page 99 demonstrate the usefulness as well as the necessity for such an approach in the robotics and intelligent systems community.



Apart from visualizing the [resource utilization](#) of systems and their [components](#), another important aspect that needs to be addressed by the [framework-level resource awareness](#) concept is the development process under which components are created and maintained. [Performance bugs](#) are usually the result of a developer making changes to the implementation of a component, who accidentally introduces a [performance regression](#). The effects of changes on the [performance](#) and resource utilization of the component are most likely unknown to the developer and will only eventually be detected when using the component in the integrated system, possibly during production use with severe consequences. Therefore, performance bugs should already be detected as early as possible, ideally even before an affected component revision or version is published and put into use. This can only be accomplished inside the development process that lead to new component revisions.

The software industry and the robotics community have developed established methods that are frequently used to detect [bugs](#) during development. These are methods such as unit testing at class level and integration testing at component level [[Lim+10](#); [Ben+09](#)], as well as domain-specific simulations [[Ste13](#); [LSL12](#); [Ben+09](#)]. Sometimes, also formal methods are applied to verify properties of components or systems [[Ben+09](#)]. These methods primarily verify the [functional requirements](#) and [nonfunctional requirements](#) such as the resource utilization are often ignored or only partially checked as side effects. Hence, the resource utilization is not systematically controlled during the development process and performance bugs stay unnoticed.

To address this situation, I have developed a framework for systematically testing components regarding their resource utilization profile in a fashion similar to unit and integration tests. In the following sections I will introduce this framework, which was published in Wienke and Wrede [[WW16d](#); [WW17a](#)]. Parts of the following text have previously appeared in or are based on these publications.

### 10.1 RELATED WORK

In contrast to the robotics and intelligent systems domain, systematically testing software for performance regressions is common practice in other disciplines, the most notable being large scale enterprise systems and website operation, where [APM](#) is applied. Performance testing in these systems is usually performed on a much coarser-grained

level with the whole system being deployed for testing as a monolithic unit. Tests are often performed based on mimicking or abstracting the human users of the systems (e.g., through HTTP interactions) and test runs can last up to several hours or days [Syd11]. Common tools such as *Apache JMeter* [JMeter] or research results such as Chen et al. [Che+08] reflect this. The outcomes of such tests are numerous KPIs for the system under test (SUT), which need to be analyzed by the performance engineers. Visualization and simple threshold checks are common techniques applied here.

A recent survey by Jiang and Hassan [JH15] provides a good overview of publications dealing with performance testing of large-scale systems. The authors separate the testing process into three successive steps: test *design*, *execution*, and *analysis*; and categorize publications along several axes inside each step. The review does not mention any work that specifically focuses on individual components as the unit of testing. Instead, most approaches follow the APM idea of analyzing the complete system as a single entity.

I will follow the separation of the testing process proposed in Jiang and Hassan [JH15] for the remaining analysis of related work. Regarding the *design* of performance tests, different methods exist. Tools such as *Apache JMeter* [JMeter] and *Tsung* [Tsung] test applications via network protocols such as HTTP or XMPP and provide methods to generate test interactions and data for these protocols. Often, recording capabilities exist to generate these interactions based on prototypical executions, and loops and parallel execution can be used to generate extended test loads using an abstract specification of the interactions. In *Apache JMeter*, interactions are specified primarily using a GUI, whereas *Tsung* uses an XML configuration file and command line utilities for defining tests. Other tools such as *Locust* [Locust], *NLoad* [NLoad], *The Grinder* [Grinder], and Chen et al. [Che+08] use the programming language level to define load tests. Tools such as *Gatling* [Gatling] are between these two categories by generating code from exemplary executions. Further related work based on domain-specific language (DSL) will be presented in Section 11.1. Most of the presented approaches provide a way to structure the performance or load test into distinct units such as test cases or test phases.

For test *execution*, frameworks have the duty to generate the load and to log metrics during the test. Depending on the framework, load can be generated from one or several hosts [Tsung; JMeter; Locust]. Most frameworks targeting web applications automatically log metrics such as response times for the issued requests. Additionally, some of them incorporate ways to also log the resource utilization of the SUT [Tsung; JMeter; Grinder].

For the *analysis* of performance tests with the aim to automatically detect performance regressions, several methods have been proposed. One common technique is the use of control charts [MHH13; Ngu12;

Ngu+12]. However, they assume a normal distribution of the measured values, which is usually not the case for [system metrics](#) such as CPU utilization under varying system states. Another category of approaches exploits the correlation of multiple [KPIs](#) in a test run. Changes in these correlations could indicate a performance regression. Moreover, correlations can be used to reduce the amount of metrics that needs to be analyzed. Foo et al. [Foo+10] and Žaléžničenka [Žal13] implement this approach with association rule learning techniques whereas Shang et al. [Sha+15] use clustering and regression. Additionally, Malik et al. [MHH13] present two other approaches based on clustering and [principle component analysis \(PCA\)](#).

Generally, the existing work mostly focuses on performance testing for integrated systems. Here, a key problem is to summarize the large amount of generated data [JH15]. Although similar tests are also desirable for robotics and intelligent systems, they are much harder to set up and maintain due to the complex interactions of robots with the real world and the nonstandard interfaces in contrast to protocols such as HTTP. Moreover, performance regressions detected in integrated tests cannot directly be attributed to individual components without further analyses. Therefore, testing the performance characteristics of individual components provides a parallel and currently more applicable method in robotics and intelligent systems.

In addition to data-driven methods, performance regressions can also be detected from appropriate software models. Relevant methods have already been discussed in [Sections 4.1.3](#) and [4.2.6](#).

## 10.2 PERFORMANCE TESTING FRAMEWORK CONCEPT

The design of the implemented testing framework follows the general idea of the framework-level resource awareness concept by focusing on individual components as the SUT (cf. [Section 5.5](#) on page 50) instead of complete systems. In addition to the considerations for the general concept, this decision is based on the following thoughts:

1. Testing a complete robotics system for performance regressions in an automated fashion is hard to achieve because of the interactions with the real world, e.g. via speech-based dialog or computer vision algorithms. Inputs for these interactions would need to be simulated or prerecorded and special interfaces to interact with a simulation or recorded data chunks are then required. Availability of such interfaces is limited.
2. The [middleware-based component interface](#) allows creating performance tests that are quite stable during component and system evolution. Although changes to the interface might occur, these should be infrequent. Otherwise, the integration of systems that use a respective component would be impaired.

3. Detected performance regressions can be attributed to individual components. This avoids complicated searches for the origin of a regression and the responsibility to fix the detected issues is clearly assigned to the component developer.
4. Component developers should have the most extensive knowledge about their components and the expected loads and behaviors. Therefore, developers are in a better position than system integrators to test the complete range of functionality and loads a component is intended to handle, and not only the requirements of a single target system. Moreover, it is also much easier to explore the space of potential loads on a component under test in isolation because the middleware inputs to the component do not need to be generated through several layers formed by the surrounding integrated system and ultimately the interactions of the system with the real world. The possibility for systematic exploration increases the test coverage for the individual components and ensures that important situations that might trigger exceptional resource utilization patterns are covered by the tests.
5. Components usually outlive individual robotics systems and might be used in several systems in parallel. Isolated performance tests per component ensure that the test suite does not need to be rebuilt with each new application and test results are continuously comparable despite changing application areas of the components.

Therefore, the developed testing framework is designed to test components individually.

*testing via middleware interface* Ⓞ

The general concept of this framework is visualized in [Figure 10.1](#) on the facing page. The middleware inputs of a component under test are replaced with inputs generated using the testing framework and the component is instrumented to acquire resource utilization information. The whole test progression consisting of testing metadata, component communication, and system metrics is recorded and stored. To decide whether the resource utilization of the current revision of the component under test has changed significantly compared to a previously recorded reference, the stored data is processed by a regression detection component of the framework.

The actual test cases constructed with the framework are maintained alongside the component similar to unit tests. This ensures that tests are kept up to date with the component by the component developers and test results are immediately available after changes.

*focus on vertical components* Ⓞ

Depending on the connectivity of a component with the remaining system or the underlying operating system and hardware, the complexity of testing via the middleware can vary. For example, a controlling state machine usually communicates with many other

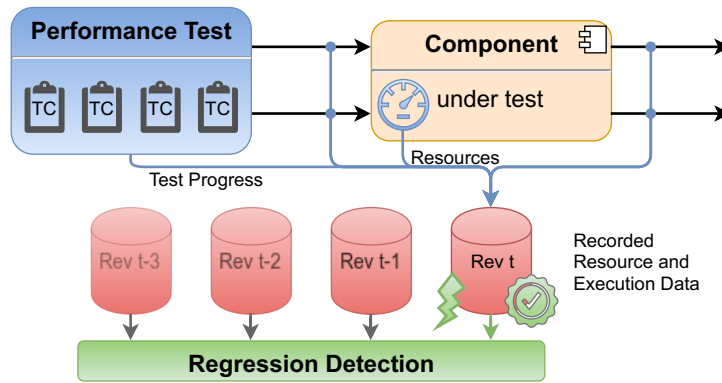


Figure 10.1: Visualization of the component-based performance testing concept. Black arrows represent the middleware interface and communication of the tested component. Blue arrows represent data that is recorded during test execution.

components in the system. Therefore, it is hard to test it in isolation. Although it is possible to test such a component (e.g., by implementing mock components for the tests), the testing framework primarily targets vertical components (cf. [Section 5.4](#) on page 49), which have isolated and well-defined component interfaces.

### 10.3 REALIZATION

In the following subsections I describe the realization of the testing framework. According to Malik et al. [[MHH13](#)], a common load or performance test (terms are often used interchangeably [[JH15](#)]) consists of “a) test environment setup, b) load generation, c) load test execution, and d) load test analysis” [[MHH13](#)]. I agree with this view and the detailed description of the framework follow this separation (with a changed order).

#### 10.3.1 Load generation

For vertical components, I assume that the resource utilization of the component at runtime is to a large extent related to the middleware communication the component is exposed to (including different aspects such as message size, message rate, communication channels, etc.). For instance, a face detection algorithm might utilize more or less CPU time depending on the rate and size of images it receives via the middleware. Similarly, a person tracker’s CPU and memory utilization might be related to the number of person percepts it receives. Finally, parameters inside the received messages might influence the processing, for instance, a desired quality criterion for a solver. The underlying hypothesis will be analyzed in more detail in [Chapter 13](#) on page 149. Therefore, generating load in terms of the middleware

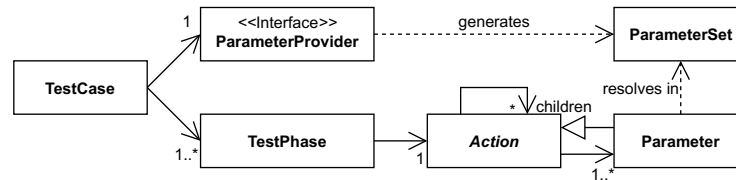


Figure 10.2: Class structure of the testing API.

communication provides a way to abstract the ongoing development changes inside the component while testing the aspects that are relevant to its use inside a robotics system.

To generate such middleware-defined loads, the testing framework allows specifying middleware interactions with components via a Java [API](#). I have chosen the Java language for realizing the framework because it provides the required performance to generate heavy loads (e.g., in contrast to Python) while providing a relatively easy to use programming language and environment (e.g., compared to C++), which is usable for most developers.

Inside this framework, a performance test consists of multiple *test cases* (cf. [Figure 10.2](#)). Each test case comprises one or more *test phases* and a *parameter provider*. A test phase is a named entity that consists of a tree of parameterized *actions* to perform via the middleware. Actions therefore specify the actual interaction of the test with the component. These actions have variable *parameters* such as a sending rate for messages or a number of faces to include in a face detection message. The parameter provider generates *parameter sets*, which specify the actual values for all variable parameters. Executing a test case means executing the action tree sequentially for all parameter sets, therefore generating different load levels on the component.

Parameters allow specifying the load profile of tests, whereas actions specify the structure of the interaction. Parameters can for instance be: a) communication rates, b) the number of generated messages, c) data sizes, d) sets of precomputed messages, or e) middleware communication channels. Separating these parameters from the structure of the interaction results in the following benefits:

- The influence of these parameters on the performance of the component can be analyzed systematically.
- Tests can be executed with different granularities. For example, this enables a developer to run a quick smoke test with a reduced parameter set on his own workstation before submitting changes to a component. The same test case can then be executed by a build server with a larger parameter set and longer runtime to generate detailed results.
- Test cases can be reused across different, functionally comparable components by changing the parameter providers.



DATA	
Parameter	Resolve a value from the current parameter set
StaticData	Return a predefined, static value
FLOW	
Sequence	Execute multiple actions sequentially
Loop	Loop an action n times or indefinitely
Parallel	Execute multiple actions in parallel
WithBackground	Execute one main action with multiple background actions. Background is interrupted when the main action finishes.
TIMING	
Sleep	Sleep for a specified time
LimitedTime	Execute an action for a limited time and then interrupt it
FixedRate	Execute an action at a fixed rate
MIDDLEWARE	
InformerAction	Send an <a href="#">RSB event</a>
RpcAction	Call an <a href="#">RSB RPC</a> method and optionally wait for the reply
WaitEvent	Wait for an event to arrive
BagAction	Replay prerecorded <a href="#">RSB</a> communication from a file
DynamicEvent	Construct an event (for Informer or RPC action)
ProtobufData	Generate protocol buffers event payloads from parameters

Table 10.1: Identified actions for constructing performance tests.

Test phases provide the ability to group operations to perform with the component under test. These phases are uniquely identifiable for a later analysis step. As test phases are executed sequentially inside each test case for all parameter combinations, they can also be used to realize the necessary actions for following the communication protocol expected by the tested component.

#### 10.3.1.1 *The action tree*

Actions that can be performed to interact with the component under test form a limited specification language suitable for the needs of performance testing. Each action generally is a function that takes the current parameter set as its input and optionally returns a result that may be processed by parent actions. [Listing 10.1](#) on the following page shows the simplified Action interface as well as an exemplary implementation stub. As visible, mandatory arguments to actions are provided by the return values the child actions' execute methods. Parameters are specializations of Action (cf. [Figure 10.2](#) on the preceding page) and their execute method resolves the current parameter value from the provided ParameterSet and returns it. This structure has the benefit that all configuration aspects of all actions can be con-

---

```

1 public interface Action<ReturnType> {
2     ReturnType execute(ParameterSet parameters);
3 }
4
5 public class Loop implements Action<Void> {
6     private final Action<?> action;
7     private final Action<Integer> iterations;
8
9     public Loop(final Action<?> action,
10                final Action<Integer> iterations) {
11         this.action = action;
12         this.iterations = iterations;
13     }
14
15     Void execute(ParameterSet parameters) {
16         final int iterations = this.iterations.execute(parameters);
17         for (int i = 0; i < iterations; ++i) {
18             this.action.execute(parameters);
19         }
20         return null;
21     }
22 }

```

---

Listing 10.1: Simplified Action interface and implementation of the Loop action displaying how subactions are used as parameters (iterations) and to specify side effects such as interactions with the component under test (action).

trolled via parameters if necessary. However, this also means that the special action `StaticData` has to be used whenever an argument of an action should be statically defined without using parameters. I have identified and implemented the actions shown in [Table 10.1](#) on the preceding page as a result of testing components from different systems.

One of the most frequent tasks to perform while testing is the generation of data to send via the middleware based on the current parameter values. The framework provides support for this task for the [RSB](#) middleware. The `ProtobufData` action is used to construct *Protocol Buffers* [[Protobuf](#)] messages from templates by scaling (repeated and string) fields based on parameters. [Listing 10.2](#) on the next page demonstrates how this action is applied, including the use of the *Protocol Buffers* [[Protobuf](#)] API for generating the template and the variable fragments. As a second possibility for generating data, the `BagAction` replays prerecorded data, optionally with modulations such as speed or channel selection. If a user requires further actions or methods for generating test data, custom implementations of `Action` can be added. Performance tests are created by forming a tree of these actions inside different test phases and test cases. [Figure 10.3](#) on page [114](#) visualizes the action trees that have been used for testing the leg detector component from [Scenario ToBi](#) on page [142](#).

---

```

1 public class LegGenerator implements DataGenerator {
2     @Override
3     public Object generate(final int index, final int totalEntries) {
4         final Random rand = new Random();
5
6         final Legs.Builder builder = Legs.newBuilder();
7         builder.setPair(rand.nextBoolean());
8         builder.setAngle(rand.nextFloat());
9         builder.setAngleVariance(rand.nextFloat());
10        builder.setDistance(rand.nextFloat());
11        builder.setDistanceVariance(rand.nextFloat());
12
13        return builder.build();
14    }
15 }
16
17 // when specifying the action tree
18 Parameter<Integer> LEG_NUMBER = new Parameter<Integer>(
19     "legNumber", Integer.class);
20 Action<?> action = new ProtobufData(
21     new StaticData<GeneratedMessage.Builder<?>>(
22         new BuilderValue(
23             LegDetections.newBuilder().setOrigin("legDetectorJava")),
24         new VariableRepeatedField("legs", LEG_NUMBER, new LegGenerator()))

```

---

Listing 10.2: API for varying *Protocol Buffers* data based on parameters.

---

```

1 new ParameterProduct(
2     Lists.<ParameterRange<?>> newArrayList(
3         new ParameterRange<Long>(REQUEST_LENGTH,
4             new FixedValue<>(50L, 500L, 10000L)),
5         new ParameterRange<Long>(WAIT_TIME,
6             new LongRange(500L, 2000L, 20000L)),
7         new ParameterRange<Integer>(REPETITIONS,
8             new FixedValue<>(5, 10, 100))),
9     Lists.<ParameterConstraint> newArrayList(
10        new ScriptConstraint(
11            "repetitions * (requestLength + waitTime) <= 70000"))));

```

---

Listing 10.3: Exemplary instantiation of a *ParameterProduct* for three parameters with a constraint specified using a Groovy script.

### 10.3.1.2 Parameters

Each test case is equipped with a parameter provider, which generates one or more sets of parameter combinations. Each parameter itself is a programming language object and has a printable name for the analysis (cf. line 18 in Listing 10.2). Moreover, it must be serializable by RSB (cf. Section 6.1.3 on page 58) because it will be reported using the middleware. The framework provides two implementations of parameter providers: a table, where the user manually specifies the row values, and a Cartesian product, where combinations of individual parameter values are created, optionally with constraints. For specifying these constraints, scripting languages such as Groovy can be used. Listing 10.3 demonstrate how a *ParameterProduct* can be instantiated.

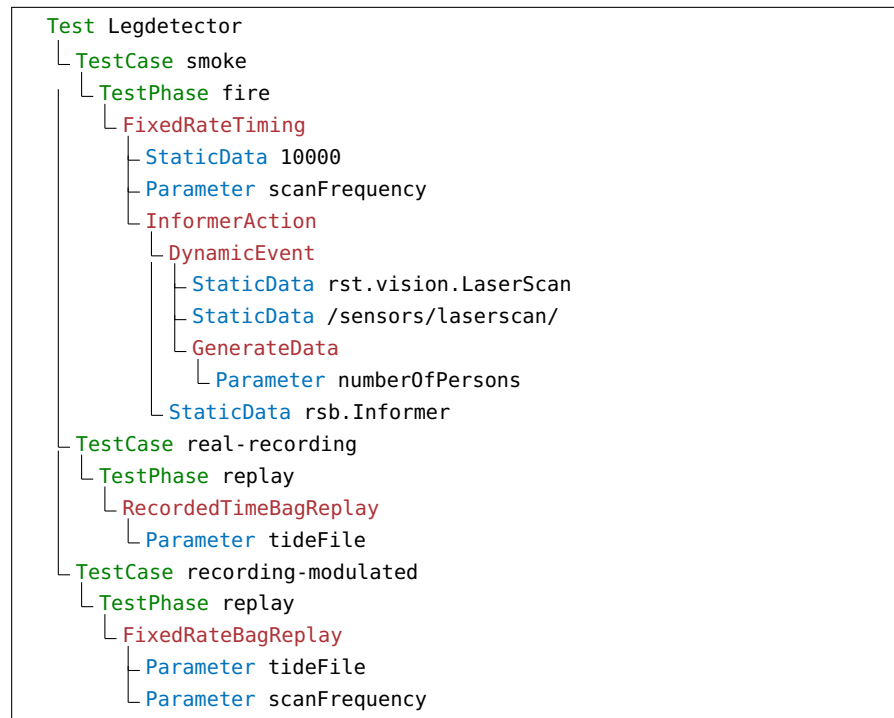


Figure 10.3: Structure of a performance test for the legdetector component from [Scenario ToBi](#) on page 142. The test consists of three distinct test cases, each with a single test phase. The general test case and phase structure is marked in green, actions are marked in red, and data-related actions are marked in blue.

### 10.3.2 Environment setup

For executing performance tests, the API contains a test runner. On test execution, the first task of this test runner is to set up the test environment based on a configuration file, which specifies the following aspects: a) locations of utility programs, b) the [RSB](#) configuration, c) processes that act as a test fixture (e.g. daemons, mock components), d) components to test, and e) test cases and their parameter providers. Using this configuration, the initialization of the test environment is performed in the following steps (cf. [Figure 10.4](#) on the next page):

1. Configuration of the middleware to ensure that test execution is isolated from the remaining system.
2. Creation of a temporary workspace for the test execution. The workspace is used as the working directory for executed processes and stores intermediate logs, which can be retained for debugging.
3. Start of all defined fixture processes. These could be daemons required for the middleware, database services used by the tested component, etc.

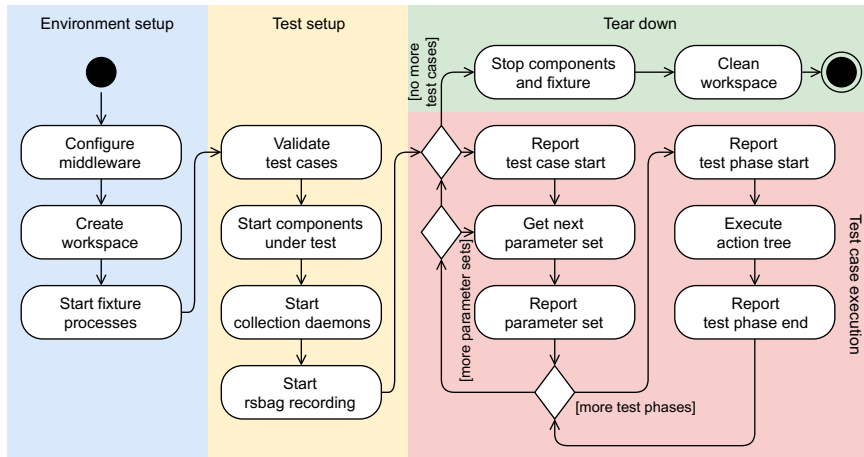


Figure 10.4: Visualization of steps performed to execute a performance test.

### 10.3.3 Test execution

Refer to [Figure 10.4](#) for a visualization of the following aspects.

#### 10.3.3.1 Orchestration

After the environment setup, the configuration is used to instantiate and execute the performance test. First, the configured test case and parameter provider instances are created and a static validation of parameter references is performed. If validation succeeds, the defined components to test are started. Although usually only a single component is started, it is also possible to test a combination of components. This could be the case if a small set of components is tightly coupled and creating mocks is harder than testing the set of components in combination. After starting all components, the test cases are executed sequentially. Inside each test case, the defined test phases are executed for all parameter sets returned by the parameter provider. Finally, all started components and the test fixture are terminated. I have decided to use a single execution of the component processes without intermediate restarts, for example, for each parameter set. On the one hand, test runs require more time with component restarts and, on the other hand, most robotics components usually operate for a longer time without restarting. Artificial restarts would make it harder to detect performance issues such as memory leaks, which slowly build up over time.

#### 10.3.3.2 Data acquisition & recording

To generate and record data for later performance analysis, the testing framework applies the previously introduced holistic dataset creation process (cf. [Chapter 7](#) on page 71) by focusing the recording on the middleware communication. For this purpose, the test runner

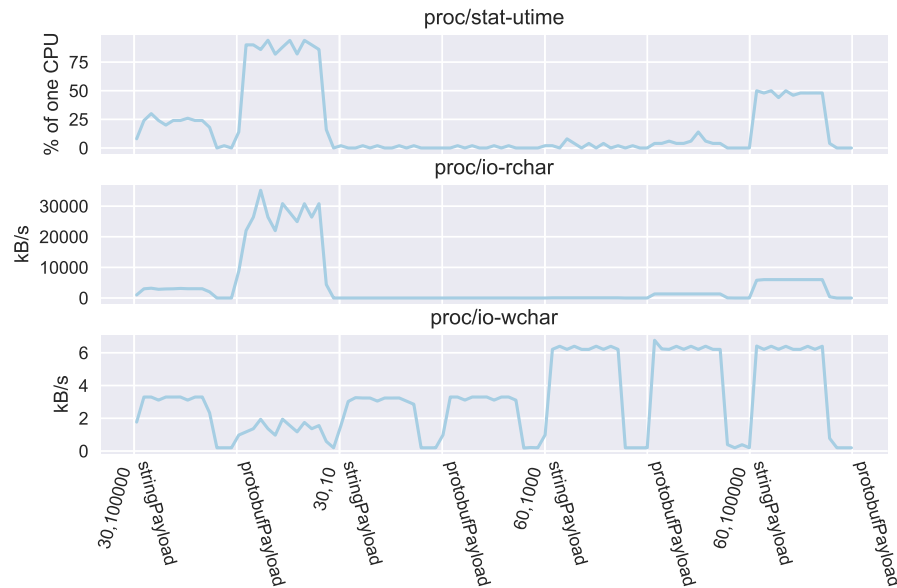


Figure 10.5: Excerpt from a test for a logging component. Two test phases are executed for different parameter combinations, in this case frequency and size of events to display by the logger. Parameter are marked with labels written from bottom to top and test phases with labels written from top to bottom.

launches the previously presented [resource](#) collection daemons (cf. [Section 8.3](#) on page 84) for all tested components so that the acquired system metrics are contained in the middleware communication. [Figure 10.5](#) shows an excerpt of a test case recording for a single test case with two test phases, which are executed for different parameters.

#### 10.3.4 Test analysis

The recorded performance test data have to be analyzed to decide whether a new component revision exhibits a different behavior regarding resource utilization compared to a previous revision. For this purpose, an analysis tool has been implemented, which performs the different tasks related to the detection of regressions. The tool is designed as a command line program to be integrated into shell scripts, for example, for a [continuous integration \(CI\)](#) server, so that the whole testing and analysis procedure can easily be embedded into existing development processes. In the following, the different tasks realized in this tool will be described.

##### 10.3.4.1 Data preparation

The output of a performance test is an `rsbag`-compatible file with all middleware events including the component communication, information about the test progress, and system metrics for the tested components. To analyze the performance of the tested components

across their revision, the recorded information needs to be persisted. However, depending on the amount and type of component communication, the resulting file size might prevent persisting these files for a longer time. Moreover, rsbag files are optimized for continuous replay and not for random access. Therefore, the recorded data are first transformed into [hierarchical data format 5 \(HDF5\)](#) files using the Python *pandas* library [McK10], which only contains the system metrics as well as the information about the test progress. In this step, information about the tested component revision is attached to the data in the form of a human readable title (e.g., a Git hash for tests per Git commit or a time stamp for nightly builds) and a machine-sortable representation (e.g., the Git commit date or an ISO 8601 formatted date [II04]) so that executions can be ordered. These HDF5 files are the artifacts that are usually persisted for each test execution.

#### 10.3.4.2 *Manual inspection*

As a first means of manually inspecting the performance of a component, the analysis tool supports generating different plots from the recorded data. The generated plots include the raw system metrics time series of a single execution, correlations between system metrics and numeric test parameters, and several plots, which show how system metrics have evolved with component revisions. For latter case, system metrics are summarized for each test case, test phase, and parameter set via mean and standard deviation, and they are plotted for each revision of the component. This allows one to track how the utilization of individual resources has evolved. [Figure 10.5](#) on the facing page is an example for a generated plot displaying the system metric time series of a single test execution.

#### 10.3.4.3 *Automatic regression detection*

To automatically detect changes in the resource utilization of a component, I have implemented three different methods in the analysis tool. All methods take one or more test execution results (as HDF5 files) and compare the observed resource utilization against a baseline from one or more other test executions. As no specific semantics are given regarding what is the baseline and what is the current test execution data, multiple testing modes can be realized with the same tools and analysis methods. These modes can be realized by providing different HDF5 files to the analysis tool. The following modes are common:

- *no-worse-than-before* principle [JH15]: the current revision is compared to the previous one (or a window of  $n$  previous revisions) to ensure that the current state is at least as good as the previous one.

- Comparison to a hand-selected baseline: all revisions can be compared to a manually selected baseline to ensure that the criteria of this baseline are met. The baseline needs to be re-selected to match intentional performance changes. In contrast to the previous mode, this mode is more likely to detect slow trends which are not detectable between consecutive revisions at the cost of requiring a manual baseline selection. For instance, a new feature per revision, each adding just a slight amount of resource utilization, can result in such a trend across multiple revisions.

Most existing methods for detecting performance regressions actually detect any change in the performance characteristics of the tested system. Although an automatic categorization whether a change is a regression or an improvement regarding the performance would be a desirable feature, this is often not easily possible. For instance, a new component revision might result in a higher CPU utilization for small workloads whereas the utilization improves for higher workloads. Therefore, the analysis tools detects any significant change of the resource utilization and the developer has to decide (e.g., based on the plots), whether the change is acceptable. This is often also the case for related existing methods. Therefore, I will continue to use the term *performance regression* to indicate those detected changes for the remainder of this chapter.

For the actual detection of performance regressions I have implemented multiple methods:

- The method proposed by Foo et al. [Foo+10] as an example for an association rule learning based approach. In this method, the KPIs are discretized and association rules are extracted. In case these rules show a low evidence on the new software revision, a performance regression is indicated.
- The method proposed by Shang et al. [Sha+15] as a reference for a recent method based on clustering and regression. Here, the KPIs are clustered and for each cluster one KPI is selected as a dependent variable. Then, a regression model is trained from the remaining KPIs to the dependent variable. In case this model creates a high prediction error for a new software revision, a performance regression is indicated.
- A basic two sample KS test for each system metric. For this purpose, the individual measurements of each system metric across the whole test execution time (of potentially multiple executions) are assumed to form an observation and the observation from the test executions is compared to the observation from the baseline. A threshold on the test statistic is used to determine whether a performance regression occurred or not.



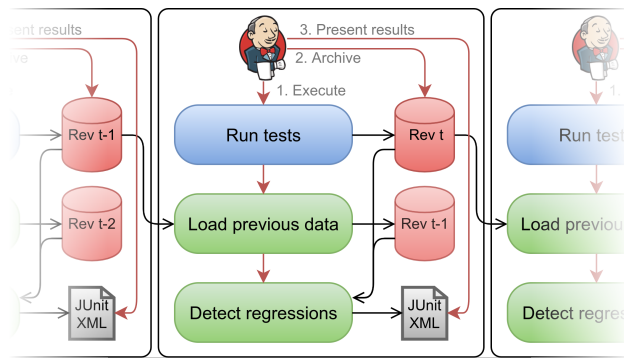


Figure 10.6: Integration of the testing framework in the *Jenkins* CI server. Each black rounded box with white background represent a single *Jenkins* build and therefore execution of the tests for a component. Red cylinders represent the stored data of the performance tests. Blue rounded boxes indicate the actual test run and green ones post-processing actions. Time and component versions progress from left to right.

The first two methods have been tested by their authors on websites and enterprise systems by testing the whole deployed system. They have not been tried on data for individual components. Here, I test their applicability for this purpose.

#### 10.3.5 Automation

The analysis tool reports results using *JUnit* [*JUnit*] XML files [*JXS*], which can be parsed by many automation tools, for example, the *Jenkins* [*Jenk*] CI server. This allows the integration of the approach with such tools, which can then automatically give feedback on potential performance regressions. Figure 10.6 visualizes how performance testing can be automated inside *Jenkins*. For each new software revision, a *Jenkins* build executes the performances tests, loads previous testing data (HDF5) from the job artifacts storage of *Jenkins*, and performs the regression detection. The resulting *JUnit* XML file is parsed by *Jenkins* using one of the available plugins and in case of regressions, developers can be notified, for instance, via mail. The *Jenkins* integration is an easy way to trigger a test execution for each new software revision with automatic notifications.

The system metrics recorded while executing performance tests are coupled to the execution platform. Therefore, a dedicated host should be used for all test executions and this host should be free from other tasks to avoid influences of [resource contention](#) in the measurements. For instance, this can be realized by adding a dedicated performance testing slave to a *Jenkins* server.

## 10.4 EVALUATION

To determine the accuracy of the automatic detection of performance regressions and to find out which of the detection methods introduced in [Section 10.3.4.3](#) on page 117 performs best, I have implemented a set of performance tests for several vertical components and infrastructure components from RSB-based systems. These are components for which I had in-depth knowledge available, so that ground truth information about performance changes could be gathered. The selected components cover a range of different tasks and programming environments to ensure that the evaluation does not overfit on a specific environment. In detail, the following components were tested:

- 2dmap: A Java-based visualization for person tracking results in the [CSRA](#) scenario (cf. [Scenario CSRA](#) on page 99).
- legdetector: A Java component for detecting legs in laser scans on the *ToBi* system (cf. [Scenario ToBi](#) on page 142).
- objectbuilder: A C++-based component that generates stable person hypotheses from detected legs and the SLAM position of the *ToBi* robot (cf. [Scenario ToBi](#) on page 142).
- logger-\*: A console-based logger for middleware events with different output styles indicated by the wildcard. Implemented in Common Lisp.
- bridge: An infrastructure component, which routes parts of the middleware communication to other networks (Common Lisp).

For all these components, tests have been written using the presented Java API and results have been processed using the aforementioned analysis methods. All tests could be generated with the provided actions, which suggests that the provided set of actions is generally sufficient for writing tests for such components.

For the evaluation I have tested the presented components using the *no-worse-than-before* principle by comparing each revision against the previous one. This is automatically possible without the need for a manual baseline selection and therefore the easiest method to apply in line with the requirements of the framework-level resource awareness concept. For the 2dmap, legdetector, and objectbuilder component I used all Git commits that I could compile without errors. For logger and bridge archived nightly builds were available instead.

To acquire ground truth information on the performance regressions introduced into the tested components, the generated plots displaying the evolution of system metrics across revisions have been manually examined and annotated. Additionally, the commit logs

	REVISIONS	EXECS	CHANGES
2dmap	25	4	10
legdetector	14	4	4
objectbuilder	23	4	7
logger-compact	306	2	16
logger-detailed	306	2	7
logger-monitor	228	2	6
bridge	176	2	6

Table 10.2: Available evaluation data per component.

have been used, especially in cases where a decision was not easily possible from the plots. Although I took great care with the annotations, I still expect some amount of errors because it is sometimes hard to decide whether visible changes are real performance regression or caused by other unknown influences. Especially for the nightly builds, the Git commit log was not sufficient to trace all possible changes, for instance, to the compilation environment used to create each build. Table 10.2 displays the amount of available data per component. Column REVISIONS shows the number of revisions that were tested per component and column EXECS indicated how often the test has been executed per revision. Finally, CHANGES shows how many revisions contained performance regressions in the manual annotations. The resulting dataset including ground truth information is publicly available [WW17b].

I have evaluated the performance of the different analysis methods by applying them to the whole history of each component. Given the binary ground truth annotation, the task to detect performance regressions has been treated as a classification problem. All analysis methods return multiple scores per test (Shang et al. [Sha+15] returns one score per cluster, Foo et al. [Foo+10] returns one score per frequent item set, and the KS test returns one score per system metric). I have decided to combine these individual scores using different functions and then to use the resulting number to compute an [area under curve \(AUC\)](#) score on the [receiver operator characteristic \(ROC\)](#) curve for the classification task, which is an established metric to assess a classifier [Faw06]. For aggregation of the individual scores, I have used the min, max, and mean functions as three straightforward options.

Based on the available data I receive the evaluation results visible in Table 10.3 on the next page, which shows the AUC score for each component, detection method, and the three aggregation methods. The highest scores per component are highlighted. As visible, for component tests the basic KS test is generally superior to the other methods. Only for some settings the method by Shang et al. shows comparable or slightly better scores. Especially the method proposed by Foo et al. does not work on this kind of data.

	FOO ET AL.			SHANG ET AL.			KS TEST		
	min	max	mean	min	max	mean	min	max	mean
2dmap	0.50	0.72	0.71	0.81	0.81	0.83	0.50	0.50	0.89
legdetector	0.50	0.51	0.47	0.75	0.97	0.97	0.50	0.50	0.97
objectbuilder	0.50	0.63	0.66	0.28	0.76	0.55	0.50	0.50	0.84
logger-compact	0.50	0.49	0.51	0.45	0.56	0.48	0.59	0.59	0.76
logger-detailed	0.50	0.39	0.39	0.46	0.60	0.57	0.61	0.50	0.84
logger-monitor	0.50	0.57	0.57	0.69	0.82	0.80	0.48	0.50	0.72
bridge	0.50	0.59	0.59	0.55	0.56	0.48	0.44	0.49	0.61
mean	0.50	0.56	0.56	0.57	0.73	0.67	0.52	0.51	0.81

Table 10.3: ROC AUC scores for different analysis and aggregation methods. For each component and the average across all components, the highest scores have been marked in red.

For the results shown in [Table 10.3](#), the tests have been executed multiple times (cf. [Table 10.2](#) on the previous page for the actual numbers). This has been done, because some aspects of the component performance characteristics differ across test runs of the same component revision. For instance, due to garbage collection timings in Java or Common Lisp programs, the memory footprint can be different across runs. Generally, I have observed that memory is one of the most common causes for false-positives due to such issues, and averaging across multiple runs provides a way to counteract this. In contrast to component restarts during test execution (e.g. for each parameter set and test phase) this is still faster to perform due to fewer restarts while retaining the ability to detect issues such as memory leaks. Additionally, effects of component initialization (warming up caches, loading files, and libraries, etc.) are less visible in the data. To quantify the effect of the number of test executions on the detection of performance regressions, I have varied the number of executions for all components that have been tested four times in total. [Figure 10.7](#) on the facing page shows the results for the most promising detection methods. Although a slight improvement can be observed for Shang et al. with more test executions, the results for the KS test are inconclusive. On the other hand, both methods already show a reasonable performance with a single execution of the tests.

## 10.5 SUMMARY

The presented framework introduces systematic performance testing in the robotics domain. While being established in other disciplines, the robotics and intelligent systems community largely lacks this practice and therefore a systematic way to control the resource utilization within the development process. I assume that one reason

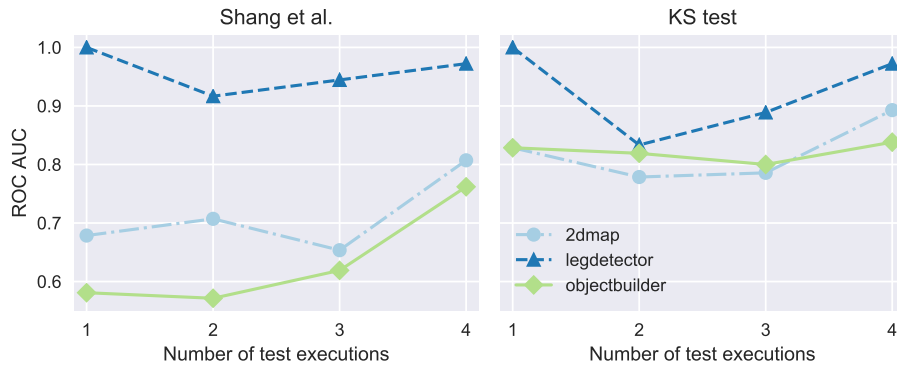


Figure 10.7: Influence of the number of test executions on the two most promising detection methods. For Shang et al. the max aggregation method was used and for the KS test the mean.

for this issue is the often less controlled development process in research settings. Thus, I have specifically designed the testing framework to meet the requirements of this domain by testing isolated components based on their middleware interfaces instead of integrated systems. This clearly assigns the testing responsibility to the individual component developers, allows a gradual adoption of the testing approach, and makes it resilient regarding system and component-internal changes. Testing components individually is also a novel perspective on performance testing in general and I am not aware of other work in this direction. Finally, I have designed the created testing tools such that they integrate with existing development processes without enforcing specific tool or automation requirements to enable an easy application of the approach.

Special care has been taken to provide abstractions suitable for vertical components and the required data generation tasks. The chosen concepts are the result of a domain analysis that I have carried out. Yet, the API is eventually meant only as a base tool for specifying performance tests. Java was chosen as a compromise between an acceptable coding experience and the required efficiency to generate load tests. However, the syntax is often verbose and requires many code-level constructs (e.g. *StaticData*, *Protocol Buffers Builder* pattern, Java generics declarations). Moreover, sometimes it is technically necessary to separate closely related aspects on the code level. For instance, the data generation code for *Protocol Buffers* data has to be disconnected from the actual instantiation of the action (cf. [Listing 10.2](#) on page 113) and it needs to be linked to its message field using an error-prone string-based identifier. Apart from explaining the API, I have also included the code listings in [Section 10.3](#) on page 109 to show these issue. Therefore, the next section introduces a DSL on top of the framework that abstracts the technical details that are mandatory in the Java API. With this DSL, the presented framework will be the technology mapping used in the model-based testing paradigm.



Although the performance testing framework presented in the previous chapter enables performance tests for robotics [components](#), their creation requires a reasonable amount of code. In [Section 10.5](#) on page [122](#) I have already delineated several code-level aspects that complicate writing performance tests from the point of view of a potential framework user and explained why they cannot easily be removed without sacrificing functionality. Following the ideas of the [framework-level resource awareness](#) concept, the existing situation is not ideal, because using the testing framework in the form of a pure Java implementation is sometimes more complicated than necessary and therefore might prevent adoption. With Wienke et al. [[Wie+18](#)], we have applied [MDS](#) techniques to improve on this situation by creating a [DSL](#) front end to the testing framework. Parts of this chapter are based on this publication. The aim of the DSL is to abstract all technical details so that testers can specify the performance tests in a concise way with all editing aids developers are used to from modern [integrated development environments \(IDEs\)](#).

When using DSLs to create software tests, one enters the domain of *model-based testing*, which is a term that is often used with slightly different meanings [[UL07](#)]. To clarify these diverse meanings, Utting and Legard [[UL07](#)] distinguish between four different types of model-based testing:

1. Generation of test input data from a domain model
2. Generation of test cases from an environment model
3. Generation of test cases with oracles from a behavior model
4. Generation of test scripts from abstract tests

[[UL07](#)]

The first two interpretations focus on (cleverly) generating test inputs from different descriptions of possible or common usage scenarios of the [SUT](#), which are then the models. The third interpretation adds knowledge about the expected reactions of the SUT so that effectively, the SUT is automatically tested against its specification. Finally, the fourth interpretation focuses on generating executable tests from abstract descriptions. With the work presented in this chapter, we have developed a solution that fulfills this last interpretation of model-based testing: abstract test descriptions are made executable by generating test programs realized using the testing framework presented in [Chapter 10](#).

In the following sections I will present related model-based testing approaches for the performance testing task, explain the design of the created performance testing DSL, and present an evaluation to demonstrate how it helps to reduce the effort required for performance testing.

### 11.1 RELATED WORK

For performance testing, several model-based approaches already exist in domains comparable to the related work presented for the testing framework in [Section 10.1](#) on page 105: large scale websites and enterprise operations. In the following paragraphs, I will shortly introduce important candidates, specifically those that apply DSLs for the purpose of specifying model-based tests following interpretation 4 of Utting and Legeard [UL07].

Not explicitly designed for performance tests, yet interesting because of the standardization efforts is the [UML Testing Profile \(UTP\)](#) [OMG13], which is a proposal for an application-agnostic language to define general software tests. Test behavior is modeled using extensions for [UML](#) behavioral diagrams (sequence, state machine). Because of the high abstraction level of UTP, in depth knowledge about UML is required for using the language efficiently and acceptance is limited [WSS11]. Moreover, UTP does not provide special support for performance tests.

Regarding performance testing, *Gatling* [Gatling] (already reviewed in [Section 10.1](#) on page 105) is related because the Scala language is used to provide an internal DSL for describing the test behavior. With the more concise syntax of Scala and some more flexibility, *Gatling* in the end provides a common programming [API](#) comparable to the testing framework introduced in this work, but with less code overhead.

Other approaches in this domain use external DSL to specify performance tests. The solutions presented in Sun et al. [Sun+16] and Cunha et al. [CMS13] mostly focus on specifying the test environment, measurements to take, and test workloads to execute without a detailed behavioral description. In Sun et al. [Sun+16] this is achieved through a [JSON](#)-inspired notation called *GROWL*, which is transformed into tests for *Apache JMeter* [JMeter]. Cunha et al. [CMS13], present the *CRAWL* DSL, which resembles [YAML Ain't Markup Language \(YAML\)](#) instead of [JSON](#) and the generation target is a custom test execution framework. Also, the approach presented in Jayasinghe et al. [Jay+12] primarily focuses on configuration aspects of performance tests while avoiding detailed behavioral descriptions. However, in contrast to custom DSLs, [XML](#) files are used, which are transformed to the required fragments such as executable scripts or configuration files by a multi-stage transformation process.



Dunning and Sawyer [DS11] present a DSL for specifying load tests for data management solutions. In contrast to reusing existing syntax, here, a declarative programming language tailored to the specific needs of the application scenarios has been designed which covers all aspects required to execute a load test. Test cases are declared in a single configuration file comprising all required configuration and behavior generation aspects.

With a surface syntax even less similar to programming languages, Bernardino Da Silveira et al. [BZR16; BRZ16] introduce the Canopus DSL for performance testing of server systems. Based on a prose-like representation, performance tests are specified as simulated virtual users for the tested system. Multiple aspects of the DSL allow modeling how the system is monitored, which scenarios are tested, and how virtual users behave.

Finally, more related to interpretation 2 of Utting and Legeard [UL07], the Wessbas-DSL presented in Hoorn et al. [Hoo+15] is used to specify probabilistic performance tests and their load profile based on Markov chains. Tests are transformed into configurations for a custom extension in *Apache JMeter* enabling Markov-based testing.

Most approaches presented here and in Section 10.1 on page 105 address testing complete systems based on HTTP interactions or comparable protocols. The test behavior in this case is often described by a combination of the Uniform Resource Locator (URL) to load, potential request parameters or POST data, and rates or probabilities for these interactions with the tested server. Some tools further allow formulating more complex behaviors of virtual users with multiple sequential interactions, called sessions [Gatling; BZR16]. However, more flexible declarative behavior specifications can rarely be found and targeting individual components is out of scope for existing tools. While UTP could be applied in such cases, further tooling would be necessary to execute the tests and the steep learning curve would prevent adoption in the fast-paced robotics research process.

Finally, as already mentioned in Section 10.1 on page 105, no performance testing framework exists for the robotics domain and therefore also no DSL for robotics performance testing is known. This is confirmed by a recent survey on DSLs in robotics, which does not list a single work for this purpose [Nor+16]. Therefore, work in this direction is a novel perspective for the robotics domain.

## 11.2 LANGUAGE DESIGN

To design a DSL targeting the performance testing framework, we have decided to use JetBrains *MPS* [Jet]. *MPS* is a *language workbench* [Fow05], which facilitates the whole process of creating and using DSLs by providing modern IDE features in an environment that allows language modularization and composition [Wig+17]. Through

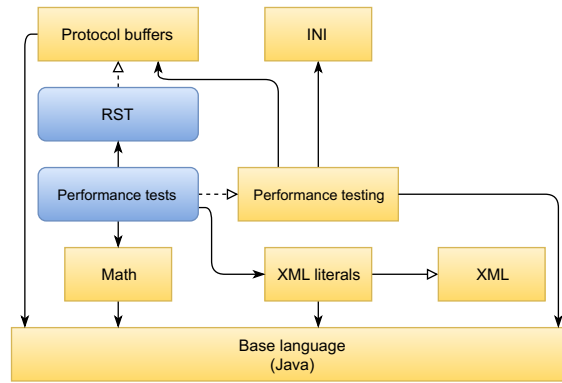


Figure 11.1: Modularization of the testing DSL. Languages are colored yellow and models in blue. Dashed arrows with hollow tips represent realizations of a language as a solution, solid arrows with hollow tips are language extensions, and all other arrows represent dependencies.

the reused of languages, duplication can be avoided, existing domain knowledge can easily be reused, and the syntax and editing aids are consistent and immediately available. For instance, a language for mathematical expression can be embedded into another language to specify system parameters. That way, computations for numeric parameters can be expressed using an established syntax without the need to reinvent the required logic.

*MPS* uses projectional editing, which means that the concrete syntax of a DSL is a direct representation (projection) of the [abstract syntax tree \(AST\)](#) of a model. Consequently, no parsing is required and models are always syntactically correct. Moreover, a model can be represented by multiple projections and the user can select the most appropriate one for the task at hand.

Apart from this, *MPS* provides many features of modern IDEs such as code completion, error checking, syntax highlighting [Voe13], and direct execution of created solutions. Therefore, *MPS* is a good candidate to create a DSL intended to simplify the creation of performance tests from a user perspective. It thus brings testing closer to the ideas of the framework-level resource awareness concept by reducing the necessary effort and knowledge.

In addition to the user-facing features, *MPS* provides a complete model of the Java programming language as a DSL termed *base language*. Other languages can target the *base language* using [model-to-model \(M2M\)](#) transformation to eventually generate executable Java code. This mechanism is used in the performance testing DSL to generate code against the Java-based API of the testing framework.

The performance testing DSL makes use of the language modularization capabilities of *MPS* and builds on several existing languages. [Figure 11.1](#) shows these languages and their relations. Their application will be described below when adequate.

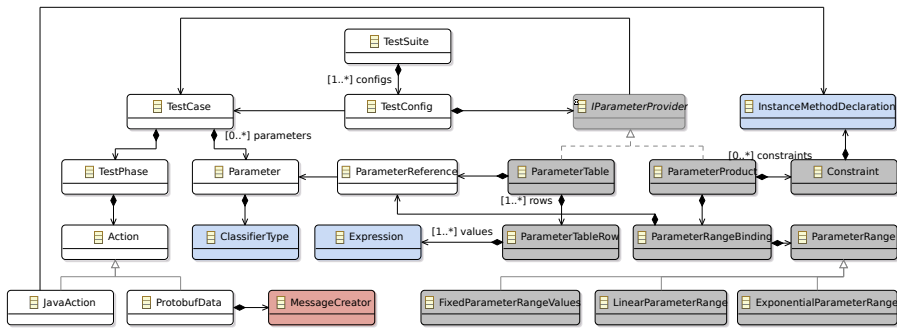


Figure 11.2: Metamodel of the performance testing DSL. Most Action implementations are not depicted for the sake of brevity. White and gray: performance testing DSL, red: *Protocol Buffers* DSL, blue: base language.

### 11.2.1 Metamodel

The metamodel of the performance testing language (cf. [Figure 11.2](#)) closely follows the class structure of the target framework and the separation of behavioral description with actions (white) and parameter generation (gray) is clearly visible. A `TestSuite` is the top-level entity that configures `TestCases` with appropriate `ParameterProviders`. As an exception to the generally mirrored structure, the Java framework represents test suites through configuration files instead of classes.

#### 11.2.1.1 Actions

Executable actions in the Java framework are represented by subconcepts of the abstract `Action` concept. These actions as well as their mandatory arguments (Actions themselves) are typed using the *MPS* type system, comparable to the Java Generics approach chosen in the target framework (cf. [Listing 10.1](#) on page 112). The general extensibility of the Java framework through custom `Action` (class) implementations (cf. [Section 10.3.1.1](#) on page 111) is realized in the DSL by the `JavaAction` concept, which embeds concepts from the Java base language to formulate the required behavior. Please refer to [Section 11.3.2](#) on page 133 for details on how this was realized. It is important to note that custom Java extensions are entirely modeled using the Java base language and no parsing is required here as well. As a consequence, common Java IDE features such as linting and code completion are available here, too.

#### 11.2.1.2 Data generation

To improve the generation of *Protocol Buffers* data in performance tests, the *protocol buffers* DSL has been included (cf. [Figure 11.1](#) on the preceding page), which represents these data types and provides design-time checks to ensure the consistency of the generated data.

The `ProtobufData` concept represents the generation of such data as an Action by composing a `MessageCreator` concept from the *Protocol Buffers* DSL. This `MessageCreator` represents the root of an editable data type. The concrete available data types in our system have been parsed from the *RST* data type definitions and are available as a model based on the *Protocol Buffers* language.

To generate data inside *Protocol Buffers* messages using Java code (cf. [Section 10.3.1.1](#) on page 111), such code can directly be embedded into the declarative data notation at the appropriate places. This prevents the implicit connection of the data generation code to the data fields using strings (cf. [Section 10.3.1.1](#) on page 111), which therefore avoids runtime errors. Please refer to [Section 11.3.1](#) on page 132 for further details on how this feature was realized.

#### 11.2.1.3 *Parameter specification*

For specifying the actual parameters values to for the test execution, both available `ParameterProviders` from the Java framework have been mirrored by concepts of the DSL (cf. gray parts of [Figure 11.2](#) on the previous page). The DSL representation of tabular data was equipped with consistency checks based on checking rules and the type system of *MPS* (cf. [Figure 11.4](#) on the facing page for an example). For instance, parameter values in each table row are checked for the correct type and each row needs to have the appropriate number of values matching the parameters declared in the table header.

For the Cartesian parameter product, comparable type system and checking rules have been implemented. Constraints on the generated parameters are encoded by embedding a method declaration from the base language which represents the constraint in a type safe fashion instead of a runtime-parsed Groovy expression. Please refer to [Section 11.3.2](#) on page 133 for further details on a type-safe realization for the embedded Java code.

#### 11.2.2 *Editors*

The projectional editors of the performance testing language for the behavior description have been designed to mimic a simple programming language familiar to programmers, who are the primary users of this DSL. Consequently, actions are represented in a way resembling simple function calls with named arguments (comparable to Python keyword arguments) and the tree structure of the actions is directly represented through nesting and indentation. However, actions that are not necessary to understand the structure of the test (i.e. `StaticData`) are not explicitly visualized and only their values are shown to reduce the visual complexity. Visible actions are displayed using a keyword color to be easily recognizable and potential complexity can temporarily be hidden using the code folding feature.

```

Phase : sendData
sendEvent (
  scope=/persons ,
  data=makeEvent (
    scope=/persons ,
    data=new rst.hri.PersonHypothesis {
      attention_targets : parameter numberOfTargets
      // immutable head
      public PersonHypothesisType.PersonHypothesis.AttentionTarget generate (
        final int index, final int totalEntries) {
        PersonHypothesisType.PersonHypothesis.AttentionTarget.Builder builder
          = AttentionTarget.newBuilder ();
        builder.setName (ByteString.copyFromUtf8 ("target" + index));
        builder.setProbability (new Random().nextFloat());
        return builder.build();
      }
    }
  age: new rst.hri.PersonHypothesis.AgeRange {
    age_max: 35.0
    age_min: 30.0
  }
  gender: rst.hri.PersonHypothesis.Gender.FEMALE
  name: "Mary"
}))

```

Figure 11.3: An exemplary test phase containing a representation of *Protocol Buffers* data with custom Java code for filling a repeated field.

```

TestSuite: TextToSpeechSuite

TestCases:
  TextToSpeechTest -> Cartesian Product {
    commands : linear range(10 -> 100 , step 10)
    words : linear range(10 -> 1000 , step 25)
    server : values["simple", "complex", "full"]

    constraints:
      // immutable head
      public boolean satisfied(final Integer commands, final Integer words,
        final String server) {
        if (server.equals("full")) { return words * commands < 10000; }
        return true;
      }
  }

```

TextToSpeechTest ->

commands	words	server
10	10	"full"
100	100	"complex"
1000	1000	"simple"

Error: type double is not a subtype of int

Figure 11.4: Test suite configuration with different parameter providers, in-line Java constraints, and design-time checks.

Figure 11.3 gives an impression of the general *syntax* of the designed editors. This figure also shows that the editors for *Protocol Buffers* data from the *Protocol Buffers* DSL were designed to match the well-readable text format representation already used for debugging purposes inside the *Protocol Buffers* implementations. Custom Java extensions are included in the editors directly using the provided Java-like representation at the appropriate places inside the AST. This avoids having to mentally connect visually disconnected concepts.

The editor for test suites (cf. Figure 11.4) maps a reference on an existing test case to an appropriate parameter provider, which is defined inline. Constraints for the Cartesian parameter product are defined inline using Java syntax (upper part of Figure 11.4). For the parameter table, a tabular presentation is used as an intuitive format.

Because all editors are direct projections of the underlying AST that represents a test case, context-dependent code completion, quick-fixes to automatically resolve inconsistencies, and design-time type checking are provided.

### 11.2.3 Code generation

For most aspects of the performance testing DSL, code generation ultimately targets plain Java source code implemented against the Java performance testing framework API presented in [Chapter 10](#) on page 105. For this reason, generators for test cases and actions, as well as the ones for parameter providers use M2M transformations into concepts of the Java base language from *MPS*. This base language eventually generates Java source code as text, which is automatically compiled so that compilation errors are prevented. The base language also forms the common generation target for other embedded languages that are used inside test cases (i.e. the Math and XML literals languages shown in [Figure 11.1](#) on page 128). The *Protocol Buffers* language itself is agnostic to specific programming language bindings. For this reason, appropriate generators were added that target the Java API of *Protocol Buffers* using M2M transformations. Depending on the types of concepts and their origins from different languages that are used to model a test case, appropriate generator pipelines are dynamically formed by *MPS*. They combine the necessary transformations to target Java in the end.

The second type of artifacts that are generated are INI configuration files. As in the testing framework test suites are represented through configuration files, instances of the *TestSuite* concept are transformed into concepts from the INI language using an M2M transformation. Test case classes and parameter providers are referenced using the generated class names. The INI language (cf. [Figure 11.1](#) on page 128) then provides the necessary text generation capabilities.

## 11.3 NOTABLE LANGUAGE FEATURES

The following subsections will explain in more detail, how a selected set of features has been implemented. These features all aim to reduce the complexity while creating performance tests and to ensure consistency as early as possible in the development of performance tests.

### 11.3.1 Inline data generation

One important feature of the testing framework is the ability to vary field values inside *Protocol Buffers* message depending on the current parameter values. To improve on the situation explained in [Section 10.5](#) on page 122, where the custom data generation code is disconnected from the data specification and linked by a string indicating the targeted field, we wanted to enable inline declarations at the exact position inside the data specification based on the *Pro-*

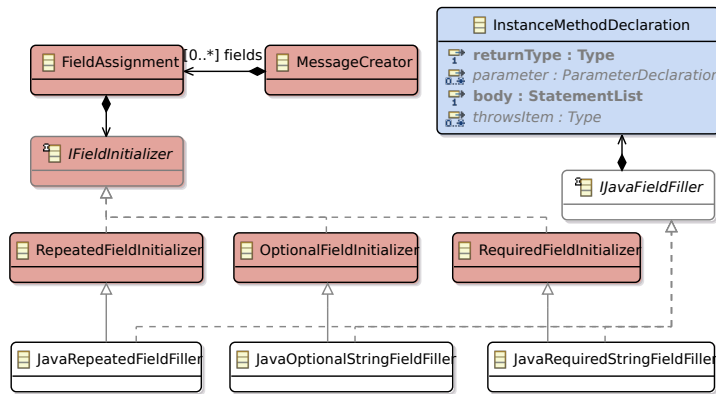


Figure 11.5: Integration of custom Java code into concepts of the *Protocol Buffers* DSL. Red: concepts from the *Protocol Buffers* DSL, blue: base language, white: performance testing DSL.

*Protocol Buffers* DSL. For this purpose, concepts from this DSL had to be extended to allow an integration of the Java-based data generation into the AST. In the *Protocol Buffers* DSL, a `MessageCreator` (cf. Figure 11.5) contains a set of `FieldAssignments`, which connect a field with its value represented using an instance of the `IFieldInitializer` interface concept. *Protocol Buffers* distinguishes between three different kinds of fields: repeated, optional, and required fields, which are represented by three different implementations of `IFieldInitializer`. In the performance testing DSL, a subconcept of each of these initializers has been created to integrate the data generation code into the AST. The created subconcepts compose a method declaration from the Java base language for the executable code. An *MPS* behavior is used to automatically assign the correct arguments and return types to this method based on the type of field that is addressed. As a result, the user is able to choose whether to use static values, custom code, or a combination of both to define *Protocol Buffers* data and string-based connections are avoided.

### 11.3.2 Type safety for embedded custom code

Custom Java-based action implementations, for instance, as depicted in Figure 11.6a on the next page, as well as constraints for parameter products can depend on parameters. In the testing framework, access to these parameters is performed through the generic `ParameterSet` (cf. Listing 10.1 on page 112), which eventually uses string-based lookup for parameter values. Consequently, a set of runtime errors is possible in case nonexisting parameters are referenced. To avoid such errors in the DSL, we pre-generate methods with an argument list matching the available parameters using a custom behavior. Moreover, the respective entry point methods (`execute` for actions and `satisfied` for constraints) are made immutable for the user ex-

```

TestCase : XmlProcessorTest

parameters:

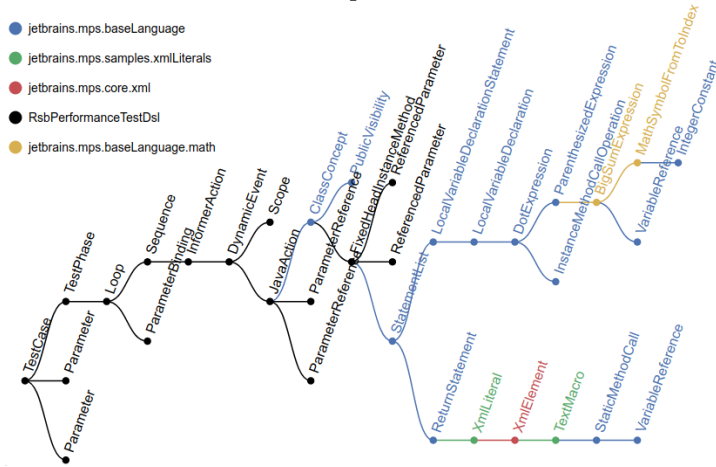
documentSize : Integer
sigma : Double
iterations : Integer

phases:

Phase : testStringData
Loop (iterations) {
  sendEvent(
    scope=/process_xml,
    data=makeEvent(
      scope=process_xml,
      data=JavaAction Generate(documentSize, sigma) -> Element {
        public Element execute(final Integer documentSize,
          final Double sigma){
          documentSize
          Double det = (  $\sum_{k=1}^{\text{documentSize}} \begin{pmatrix} 1 & k & 0 \\ 0 & 1.0 & 0 \\ 0 & 0 & \text{sigma} \end{pmatrix} \cdot \text{det}();$  );
          return <xml
            <value>${ String.valueOf(det) }</value>
          >;
        }
      }
    }
  }
}

```

(a) DSL representation



(b) Abstract syntax tree (AST)

Figure 11.6: DSL representation and corresponding AST of an exemplary performance test visualizing the use of concepts from different languages. Some elements in both representations were hidden for the sake of brevity. The AST displays the used concept names color-coded with their originating language.

cept for the contained statements. The generator is then responsible for correctly performing the string-based lookup of parameter values in the generated wrapper code. In case the available parameters change, checking rules indicate the mismatch and a quickfix in the IDE regenerates the method declaration.

### 11.3.3 Expressive custom code via embedding

We particularly designed the performance testing DSL to allow embedding of expressive DSLs for specialized domains. For instance, the user can use existing XML and math DSLs inside custom Java code to express these domains more naturally. Figure 11.6 shows an example for a test case embedding such concepts with an additional view



Case	TIME (s)			INTERACTIONS			INSTANCES		
	Java	DSL	Factor	Java	DSL	Factor	Java	DSL	Factor
1	399	95	0.24	914	274	0.30	124	37	0.30
2	305	91	0.30	731	229	0.31	215	92	0.43
3	150	55	0.37	436	123	0.28	49	9	0.18
4	78	26	0.33	258	83	0.32	41	6	0.15
5	232	86	0.37	668	236	0.35	307	183	0.60
6	415	95	0.23	1095	268	0.24	188	74	0.39
7	695	203	0.29	1707	530	0.31	289	105	0.36
8	355	165	0.47	1058	412	0.39	373	98	0.26
9	537	265	0.49	1292	736	0.57	508	147	0.29
10	516	136	0.26	1412	337	0.24	321	113	0.35
mean	368	122	0.34	957	323	0.33	242	86	0.33

Table 11.1: Comparison of DSL and Java API solutions.

as an AST. Other languages can be imported, too, depending on the use-case and requirements of the user to enhance the usability as well as to reduce the effort. The technical requirement is that generators exist for the embedded languages that eventually target the Java base language. In case of XML, these generators are provided by the *XML literals* language depicted in [Figure 11.1](#) on page 128.

#### 11.4 EVALUATION

The primary aim for creating the presented DSL was the reduction of the complexity for creating performance tests to better integrate the testing approach with the framework-level resource awareness concept. This complexity encompasses the necessary programming work as well as the mental complexity of understanding the Java API. Both should be minimized to simplify the creation and subsequent maintenance of performance tests.

To verify that the DSL helps to reduce the necessary work for creating new tests I selected 10 existing performance tests for components from both reference scenarios of this work ([Scenarios CSRA](#) and [ToBi](#)) as candidates for realistic tests. These tests were then recreated from scratch with the Java API and the DSL based on an abstract description of their contents. This description was constructed to neither resemble the DSL, nor the Java-based implementations. The recreation of the test cases was performed by myself as an expert user of both, the framework and the DSL, to prevent a bias towards one of the solutions and also to prevent learning effects during the experimentation. For creating the Java-based tests, *Eclipse [Ecl]* was used as a modern Java IDE with versatile code completion and refactoring features to compare the DSL-based approach against a state-of-the-art develop-

ment workflow in Java. Starting from an empty workspace, in both cases the time required to complete the reconstruction of the tests and input device interactions (key presses + mouse clicks) were measured as indicators of the required work. Custom Java code (actions or logic for filling *Protocol Buffers* data) was only reconstructed up to the necessary framing (e.g., class stubs which enable compilation), because an abstract description of the contents of these Java fragments would have been hard to achieve and therefore no comparable experiment setup could be determined. The results of these experiments can be found in the first two column blocks of [Table 11.1](#) on the previous page. The column “Factor” expresses the fraction of time or interactions required by the DSL solution compared to the Java framework. As visible, for both measurements, the DSL only requires one third of the effort of the Java-based solution.

To assess and compare the complexity of the test cases created using both methods, I first imported the existing Java test cases into *MPS* as models using the concepts of the Java base language. The assumption is that the complexity of a test case relates to the decisions and thoughts a developer has to perform when creating or understanding the test case. These decisions are reflected by the number of concept instances inside a model. Therefore, we have compared the sizes of the AST (number of concept instances) of the Java test implementations with their respective DSL representations. Please note that the Java base language of *MPS* automatically takes care of Java imports. Thus, such necessary helper constructs in the final text output are not represented in the AST of the solution and consequently also not counted for the results. The final numbers for this comparison can be found in the third column block in [Table 11.1](#) on the preceding page. Again, the DSL-based approach, on average, only requires one third of the concept instances of comparable Java implementations.

### 11.5 SUMMARY

With the presented DSL, the application of the performance testing framework for robotics components becomes much easier. The evaluation demonstrates that test cases can be created in approximately one third of the time and with only one third of the keyboard and mouse interactions compared to their Java pendants, in case of experienced users. Moreover, the complexity of the test cases in terms of concepts is also only on third of the Java solutions. Therefore, the DSL is an important step towards establishing a culture of systematic performance tests in robotics and intelligent systems. The proposed testing concept is generic and does not require modifications to existing components and therefore follows the general aims of the framework-level resource awareness idea. Nevertheless, more than for all other aspects presented in this work, applying this method still re-

quires manual labor and discipline and therefore the willingness of the component developers to provide these resources. With the intended integration of the testing framework into CI systems and the DSL as a means of reducing the testing effort, we have tried to minimize the required manual effort. Still, in the end, there is no free lunch. While for most parts of this work the additional efforts have been limited to sporadic activities such as software deployment or computational overheads at runtime, a method that specifically addresses the software development workflow can ultimately only be brought to life if developers accept it.



## Part IV

### AUTONOMY PERSPECTIVE

Resource awareness can also be exploited at system runtime to increase the dependability and autonomy without a direct intervention of the system developers. In this part, I will present methods that exploit the available information about the system's resource utilization at runtime to enable autonomous reactions that prevent critical situations such as resource starvation or performance bugs. These methods are data-driven and based on machine learning techniques.



**Resource awareness** can also be established at system runtime. If a robot or any other intelligent systems knows about the current **resource utilization** and has an expectation about it, this information can be used at runtime to improve the system's functionality, **dependability**, and autonomy. For instance, the planning layer of a system can use a prediction of the utilized **system resources** to avoid undesirable levels of **resource contention** or even **resource starvation**, both being able to severely degrade the system's functionality. Further, an expectation about the resource utilization can be used to decide whether **components** behave normally at runtime. In case the current resource utilization of a component differs from the expectation, this is likely unintended and the system should react before its processing becomes severely affected. This example shows that runtime **fault detection** can be based on, or be a part of a resource awareness concept. These two perspectives – resource utilization prediction and fault detection – will be addressed in this part of the thesis. However, before I can describe the details of these methods, an important question is how to develop and evaluate them. In the end, the presented methods are data-driven and a dataset with **system metrics** and **performance bugs** is required for a quantitative evaluation.

Others have addressed runtime fault detection in robotics systems before [SW05; PWW06; JED13; Gol+11]. However, the datasets used for their evaluations are not published and the research did not address resource utilization and performance bugs in component-based robotics systems. Therefore, I had to acquire a dataset that contained the required information with accurate ground truth on my own. Although performance bugs can often be observed during normal system operation, gathering usable data from these executions is usually impossible. Either the appropriate execution traces are not recorded at all or in an insufficient quantity. In case recordings exist, ground truth information about resource utilizations and observed **performance degradations** including their timings are missing. Therefore, reference datasets need to be explicitly created. With Wienke et al. [WMW16] we have created such a dataset, which I will briefly describe in the following sections. Parts of the descriptions are based on this publication.

*ToBi* (Team of Bielefeld) is CITEC’s mobile robot platform used for the RoboCup@Home competitions [Mey+15] since 2009. The robot consists of a mobile base with differential drive and two laser range finders with 360° coverage for distance data. Mounted on top of the robot are two RGBD cameras for object recognition, obstacle avoidance, gesture recognition, and scene interpretation, as well as an RGB camera for face recognition. For manipulation, a 5 DOF manipulator is used. The robot carries two Linux-based laptops, which are connected via Gigabit Ethernet. They run the distributed software system controlling the platform.

The software architecture of *ToBi* uses sensor components to provide extracted information about the scene, for instance, object recognition, person tracking, and speech recognition. Actuator components comprise navigation, text to speech, and grasping. All components communicate using the [RSB middleware](#). For coordination, the BonSAI framework [SW11] abstracts the different components as software sensors and actuators and allows modeling the system behavior as a finite state machine.



Left: *ToBi* interacting with a person (photo: CITEC/Susanne Freitag, reproduced with permission). Right: Sensors and actuators of the robot.

### Scenario ToBi: The *ToBi* system (Team of Bielefeld)

#### 12.1 RECORDING METHOD

To acquire a representative dataset, we have selected the *ToBi* system (cf. [Scenario ToBi](#)) as a reference for an established robot system, which has successfully participated at the international RoboCup@Home competitions. We selected a modified version of the restaurant task from the competitions in 2015 [Bee+15] as the recording scenario for the dataset. In this scenario, a robot that has no prior knowledge about the scene acts as a server in a restaurant. The plot consists of three phases. First, an operator sets up the robot and trains important locations. These comprise the place where the robot can pick up drinks as well as the tables drinks have to be delivered to. In the second phase, the robot waits for somebody waving to order. The robot approaches the person and asks for the name and the desired drink. After taking all orders, *ToBi* can be told to enter the third phase in which the orders get executed one after another. During these interactions, [SLAM](#) is used for navigation; face recognition and



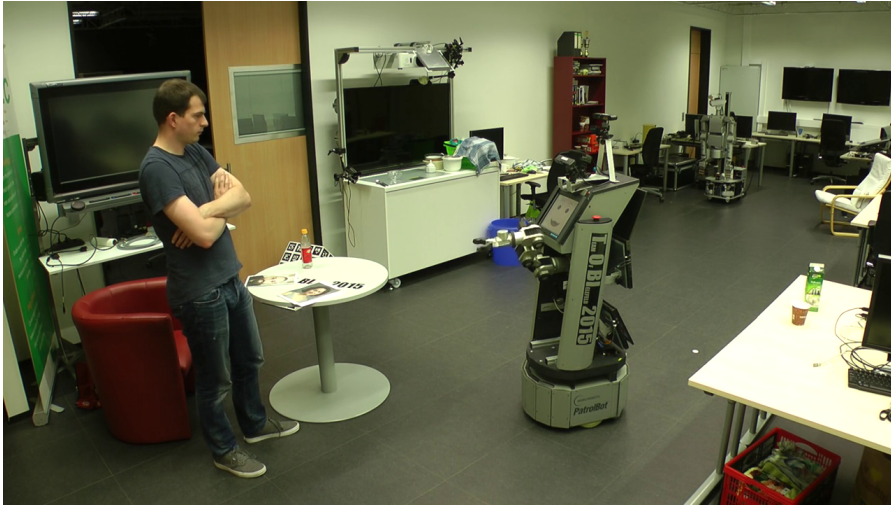


Figure 12.1: Scene from the dataset recording as captured by the external camera. Drinks to deliver are on the round table and the table on the right side. Different guests were simulated to reduce the recording effort using the photos on the round table.

object recognition are used to identify persons and objects; and RGBD sensors are used for grasping tasks. The scenario thus integrates a diverse range of behaviors and skills necessary for a mobile robot with a high complexity and variability, especially due to the involved [HRI](#). [Figure 12.1](#) shows the unstructured laboratory environment in which the dataset recording took place.

For recording the actual dataset trials, the holistic dataset creation process introduced in [Chapter 7](#) on page 71 was used. To include the required system metrics, the system was instrumented with the collection daemons presented in [Chapter 8](#) on page 79 and the [dashboard](#) (cf. [Chapter 9](#) on page 93) was used to manually inspect and verify the intended behavior of the robot system. In addition to the middleware-based *reference recording*, we added a camera as an *external recording* to provide an overview of the scene for manual inspection and potential unplanned annotations.

The software system was manually patched to contain a diverse set of performance bugs so that exact ground truth information was available. These performance bugs were selected to be [bugs](#) that result in a performance degradation only after a special condition. Permanent problems are easier to detect during development (for instance, using the testing framework) and less valuable to detect at runtime. The following sections explain in more detail, how the performance bugs were selected and included in the resulting dataset.

## 12.2 INCLUDED PERFORMANCE BUGS

When adding bugs to a system for creating a dataset used to evaluate fault detection methods, an important question to answer is whether the bugs are representative. To avoid tuning the dataset towards unrealistic or favorable conditions for the applied methods, we based the included performance bugs on the survey results presented in [Chapter 3](#) on page 21. We selected the types of implemented performance bugs to match the reported frequencies for the different categories and used the case study results for an inspiration of actual implementation errors to add.

Using these sources, the following performance bugs are included in the dataset. They are described along the categories from the survey and represent the 7 most frequent types of performance bugs in the survey results. Some of the included performance bugs do not directly influence system resources. Instead, they are causing delays or performance degradations visible primarily in the scenario progression and are therefore mostly visible in efficiency-oriented [KPIs](#). Please refer to [Appendix E](#) on page 237 for a detailed list of system components and how they are affected by the included bugs.

### 12.2.1 *Algorithms & logic*

These two categories from the survey were combined as it turned out to be hard to distinguish between them.

`btAngleAlgo` A mathematical error was added to a conversion between Euler angles and quaternion representations, which is used to compute the location of persons in front of the robot.

`armserverAlgo` The grasping controller for the arm performs unnecessary movements due to a bug in generating a trajectory in a graph of valid postures.

### 12.2.2 *Resource leaks*

`bonsaiParticipantLeak` The central state machine did not deallocate unused [RSB participants](#), resulting in a [TCP](#) connection leak.

`pocketSphinxLeak` The speech recognizer did not deallocate memory for the sound buffers, which results in a memory leak.

### 12.2.3 *Skippable computation*

`objectBuilderSkippable` The component that tracks persons transforms egocentric coordinates of detected person into global coordinates multiple times instead of only once per person.

`facerecSkippable` The throttling of the main loop in the face detection component was removed, which increases the CPU load.

`legdetectorSkippable` The detector for legs in the laser scans performed operations multiple times.

#### 12.2.4 Configuration

`bonsaiTalkTimeout` The configuration of the state machine used a wrong RSB scope to communicate with the TTS engine and had to wait for a timeout before resorting to a fallback scope.

`clockShift` To emulate a configuration issue with the clock synchronization via network time protocol (NTP), the clock of one computer was shifted at system runtime.

#### 12.2.5 Threading

`clafuSleep` An unnecessary sleep instruction was added to the object recognition component, which delayed the classification results for 5 s to simulate the effect of inefficient threading strategies in this component.

#### 12.2.6 Inter-process communication

`spreadLatency` The *Spread* daemon used by RSB was affected by constantly adding and removing a participant to the daemon network, which is a costly operation. As a consequence, RSB events had much higher latencies and jitter.

### 12.3 AUTOMATIC FAULT SCHEDULING

Given the implemented set of performance bugs, the next question is how and when to trigger them during the recordings. To acquire ground truth information about when the different performance bugs are active in each trial, the natural choice was to follow the holistic recording approach. Therefore, we decided to trigger the performance bugs via RSB events so that exact ground truth information is automatically contained in the recorded data.

Existing publications such as Golombek et al. [Gol+11] and Jiang et al. [JED13] used a straightforward strategy to induce faults into the system: at certain points in each trial a single bug is triggered manually and maintained until the end of the trial or until a system crash. Although this approach provides an easy to analyze dataset, it would be time-consuming to provide a statistically feasible amount of occurrences given the more complex scenario and number of bugs



Figure 12.2: Scheduling of induced performance bugs (blue). The time of a trial (horizontal axis, from left to right) is separated into slices (black lines with round markers) with an initial offset at trial start (red dashed) and pauses between slices (green dashed).

in our dataset. Therefore, we opted for a strategy where multiple reversible bugs were triggered during each recording trial to maximize the amount of occurrences being recorded. This means that each performance bug that was triggered via [RSB](#) could also be reverted back to a healthy system state as if it had never occurred. Such an approach forbids performance bugs that result in an actual system [failure](#). However, our main interest are bugs that are only visible as performance degradations without resulting in failures. Complete [system failures](#) would be detectable using other methods.

To prevent an operator bias on when to trigger the performance bugs or accidental correlations of the triggered bugs with the system behavior, we implemented an automatic scheduling component, which was added to the system. The algorithm used to trigger the different performance bugs is visualized in [Figure 12.2](#). Starting with the initial execution of the system state machine, each trial is separated into consecutive time slices of a fixed length, which are additionally separated by a fixed length pause. Within each of the slices, a single performance bug is scheduled for a fixed time interval. The bug instance is uniformly drawn from the set of available ones and its start time within the slice is also determined using a uniform distribution.<sup>1</sup> A uniform selection was chosen to provide the same statistical confidence for each performance bug. To further reduce potential correlations, the start of the first slice is randomly offset after the start of state machine using a uniform distribution up to 30 s. For the length of activation of each performance bug 80 s were used. We selected this duration so that a system expert was able to detect each performance bug in the robot’s behavior or appropriate visualizations,<sup>2</sup> but without resulting in a system failure. The length of each slice was selected to be 160 s to provide enough variation for the bug occurrence and the pause time between slices was set to 20 s as the minimum acceptable delay between consecutive performance bugs. This reflects the maximum time the recovery from any of the performance bugs did take (after the signal to recover normal state, heuristically determined) so that instances are correctly separated in any case. The start of each slice is exposed via [RSB](#) to include scheduling information in the dataset.

<sup>1</sup> Limited so that the intended execution time fits into the slice.

<sup>2</sup> In appropriate situations. For instance, the performance bug added to the grasping controller is only detectable in case the arm is used.

## 12.4 SUMMARY

With the explained method we have recorded a dataset which consists of 10 executions of the system without triggered performance bugs as a baseline and 33 successful trials with performance bugs. We recorded this number of trials so that at least 10 complete instances of each performance bug were recorded during execution. The total time of recordings is 8:16 h, which results in an average of 11:33 min per trial.

Apart from the 33 trials of the core dataset, 23 additional trials are available separately, which contain unexpected behaviors of the system. The number of trials with undesired faults is this high because during the recording session an actual hardware problem appeared. A loose screw in the gripper of the arm resulted in the system being unable to detect grasped objects. As a consequence, grasping often stopped at the point when the gripper was closed and the system could not recover from this state. We have annotated these situations, which results in an additional `fault` type being included in the dataset.

Within the lines of the holistic dataset creation process, the final corpus has been exposed using different views for external distribution as well as for my own needs. These views comprise `CSV` files and `pandas` [McK10] data frames for events and system metrics as well as metadata files for the system structure. Moreover, `ELAN` [ELAN] project files with automatically exported system communication tiers have been added to offer a basis for potential manual annotations. We released this dataset as open data for reuse and replication of research results [WW16a; WW16b].



The primary aim of this part of the thesis is to establish **resource awareness** at system runtime so that a system can autonomously judge about its **resource utilization**. Ideally, a system should inherently avoid critical situations by using expectations about the resource utilization to plan its actions appropriately. Moreover, if resource-aware planning is not available, fails, or in case of unexpected events, the system must be able to detect the arising critical situations before a **performance degradation** leads to a **system failure**. The least a system can do is to actively notify its users or developers about such situations. Users are then able to act appropriately and developers can debug the problem sooner and with more information about potential origins.

The task of notifying developers about unexpected resource utilization patterns is also a common task when operating servers or enterprise systems with high **reliability** requirements. Here, *alerting* systems are responsible of continuously checking different conditions that determine whether a system is healthy. These conditions often include **system metrics**, which are checked with simple averaging and threshold schemes. For instance, open-source solutions for monitoring and alerting such as *Prometheus*; *Grafana*; *Riemann* [**Prom**; **Grafana**; **Riem**] are primarily based on these basic rules. Even though this technique is applicable for some systems, it has an obvious drawback: information about the system state is not included in the decision logic. If a system exhibits different resource utilization patterns in different but expected application contexts, static thresholds can either be optimistic regarding the system's health in the sense that alerts are only generated in extreme cases, ignoring many undesired situations; or thresholds can be pessimistic resulting in many false positive alerts. Some off-the-shelf solutions try to solve this problem by avoiding absolute thresholds. Instead, the local history of a system metric time series is used to implicitly derive the state or context [**Skyl**; **Riem**].

None of these approaches is convincing in the context of robotics and intelligent systems. With the commonly observed flexibility and context-dependence of experimental systems such as the ones found in the scenarios used for this work, simply ignoring the system and environment state will lead to imprecise solutions. Implicitly deriving the system state and context from the raw system metrics will also fail. For instance, in contrast to websites, where many parallel users cause a constant load on the systems that behaves smoothly, system state and context changes in the target systems of this work are often

runtime resource  
awareness re-  
quires state and  
context information

discrete, especially on the level of system [components](#). For example, the *ToBi* robot described in [Scenario ToBi](#) on page 142 exhibits such discrete behavior changes. Once the robot has received all orders from the guests, the system switches to a completely different control state where it activates a vision component and the path planner for the arm to grasp the ordered drinks. These components have been idling up to this moment and now need to do their processing. Therefore, the system metric time series for these components will show sudden changes. An algorithm that operates only on these time series without context information cannot accept such a sudden change and will trigger and alert if it was not parameterized with high thresholds that make it completely useless. Therefore, I have decided that it is inevitable to use information about the system and environment state for enabling resource awareness at runtime. Yet, explicit modeling of such information contradicts the [framework-level resource awareness](#) concept. In addition to the question of how and in which formalism to specify how system metrics change with different states and contexts, a manual specification has many issues:

- Manual specification takes time and specifications need to be maintained continuously with the evolving system.
- The accuracy of the specified rules depends on the experience of the developer and his or her rigor when specifying the rules.
- It is questionable whether a suitable human-readable representation can be found to describe the relation of states, contexts, and resource utilization.

To realize resource awareness on the level of components in a way that prevents the presented problems and that complies with the framework-level resource awareness concept, I have devised the following hypothesis, which has already been used for the performance testing framework in [Chapter 10](#) on page 105:

**HYPOTHESIS 1** *For a component that is connected to the surrounding system only through the [middleware](#), the communication of this component contains all relevant state and context information and is therefore a suitable proxy to predict its resource utilization.*

This hypothesis is based on the idea that components, especially [vertical component](#), usually perform well-defined computational tasks, which are determined by the information or requests they receive. These tasks are assumed to have a predictable resource utilization. Their utilization is assumed to depend on parameters extracted from the received messages encoding the relevant system state and environment context, as well as on internal state of the component. This internal state is ideally visible in the outgoing communication of a component. In the following sections I will show that this hypothesis is valid for the systems examined in this work.



Under the assumption that [Hypothesis 1](#) is valid, I have decided to realize runtime framework-level resource awareness on a component-level using machine learning techniques. For each component a separate model is trained, which predicts the component's current resource utilization based on its middleware communication. These models can form the basis for different applications contributing to the system's resource awareness. I will later present potential applications. By constructing models from training data using machine learning, I avoid the aforementioned issues regarding manual specifications. Hence, the quality of the results does not depend on the experience of the developer providing specifications. The exact machine learning techniques used in this work are mainly intended to show that it is possible to realized such a runtime perspective on framework-level resource awareness. For this reason I have decided to approach the task as a well-understood regression problem for which a set of established methods is readily available.

In the following sections I will first explain how one can generically train a prediction model for each component based on the middleware communication. Afterwards, I will present an evaluation of how the proposed model performs on the different datasets to show the applicability of the chosen approach. This will also validate [Hypothesis 1](#). Finally, I will analyze one solution to the problem of acquiring suitable training data.

### 13.1 FEATURE GENERATION

To construct regression models for predicting the resource utilization of system components, their event-based communication has to be encoded into mathematical representations, which can be used as feature vectors for the learners. These feature vectors have to be synchronized with the acquired system metrics so that a valid mapping from features to metrics can be learned. In this work, the system metrics for each component are collected with a fixed frequency using the acquisition tools presented in [Chapter 8](#) on page 79. In contrast, the communication [events](#) that a component receives and sends are not required to follow any rules regarding their timing and therefore they are not intrinsically synchronized with the system metrics. Thus, the feature generation approach has to handle this flexibility.

In the following paragraphs I will present an extended version of the feature generation method that I have initially published in Wienke and Wrede [[WW16c](#)]. This approach operates individually for each system component and assumes a static connectivity of the component regarding the middleware [scopes](#) it uses for communication by requiring a fixed set of communication channels. For most of the components used in the scenarios of this work, this assumption is directly fulfilled. If so, all leaf scopes of the scope tree that a

component uses are declared to be the communication channels. If instead a component dynamically creates **participants** on changing scopes at runtime and a static configuration of this component with the aforementioned rule would not be possible, the scope hierarchy of the **RSB** middleware provides a means to reestablish static connectivity: by using the common parent scope of the dynamically created participants, the set of communication channels can be made static and independent of the component runtime. In the worst case, this means that the root scope / has to be used.

The set of communication channels of a component can be configured manually. However, to improve the applicability of the approach in terms of the framework-level resource awareness concept, another possibility to get the necessary configuration is to use the **introspection** mechanism of the middleware. The introspection can uncover the communication channels from a trial run of the system. For components that dynamically create participants on different scopes, this requires a pruning strategy. For the experiments in this work I have used the **JSON** export of the introspection tools (cf. [Section 6.4](#) on page 65) to specify the scopes of components.

Another important aspect is that the feature generation approach does not require special configurations depending on the kind of data exchanged by the component, which would increase the configuration and deployment overhead. Under these assumptions and constraints, the feature generation approach used in this work is mathematically defined as follows.

### 13.1.1 Accumulated event window features

*accumulated event window (AEW)* ♦

For the basic feature generation approach used in this work, which I termed *accumulated event window (AEW)*, I assume that for a system component  $c := \{S\}$  the set of communication channels  $S := \{s_1, \dots, s_\sigma\}$  is known, where  $S := S_i \cup S_o$  denotes the union of the component's input channels  $S_i$  and output channels  $S_o$ .

For a given execution of a component (inside a system), the acquired system metrics are encoded as a time series denoted by:

$$P := \{p_t : t \in T\} \quad (13.1)$$

where  $p_t \in \mathbb{R}^\varphi$  is the vector of  $\varphi$  system metric values at time  $t$  and  $T$  is the set of equidistant timestamps for the measurements, for instance at 1 Hz in [Scenario ToBi](#). Additionally, for each communication channel  $s \in S$  a time series with the communication events that occurred is required, which I have defined as

$$E_s := \{m(r_t) : t \in T_s\} \quad (13.2)$$

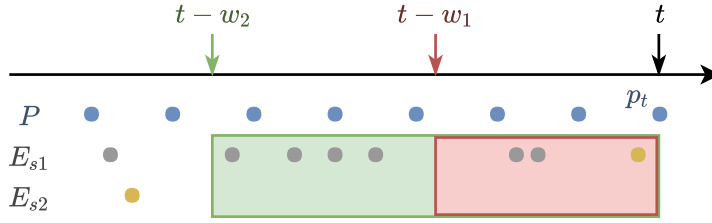


Figure 13.1: Visualization of the feature generation and synchronization approach for two scopes  $s_1$  and  $s_2$  with two temporal windows  $w_1$  (red) and  $w_2$  (green). Features are computed for the system metric measurement (blue)  $p_t$  at time  $t$ . For each scope,  $E_{s,t}^{w_i}$  comprises all events on that scope that are inside the rectangular region for the respective window.  $e_{s,t}^{last}$  is marked in yellow for each scope.

where  $T_s$  is of no specified frequency as explained before, and

$$m(r_t) : \mathcal{R} \mapsto \mathbb{R}^\mu := e_t \quad (13.3)$$

is an encoding function that converts each structured communication event from the set of raw events  $\mathcal{R}$  into a vector of dimensionality  $\mu$ . Although the previous descriptions were middleware-independent,  $\mathcal{R}$  is specific for the middleware and therefore the encoding function  $m(r_t)$  needs to be defined for each middleware. For [RSB](#), I use the following generic encoding function that respects the initially stated requirements and constraints:

$$m_{rsb}(r_{rsb}) := \begin{pmatrix} \text{Size of payload} \\ \text{Size of causal vector} \\ \text{Size of user info vector} \\ \text{hash(Data type)} \\ \text{hash(Method)} \end{pmatrix} \quad (13.4)$$

The variable payload is abstracted by its byte size to avoid type-specific mapping functions that would require manual configuration. Lists contained in the event metadata are abstracted using their size, too. For the string items *data type* and *method*, I use a hash value. Although it would be possible to use their length, I expect many collisions from this approach. For instance, for every data type in [RST](#), on average 6.5 other data types with a name of similar length exist. These types would be indistinguishable if only the string length was used. The hash is realized by the *Python* hash function.<sup>1</sup>

As the different event series  $E_s$  are not yet synchronized with the system metrics  $P$ , a synchronization is required. To create constant prediction results at a fixed frequency, I have decided to reuse the timing  $T$  of the system metrics  $P$  for this purpose and therefore the

<sup>1</sup> The hash function returns deterministic results for strings in *Python 2.7* [[Pet12](#)].

events will be conformed to this frequency. For each communication channel  $s \in S$  I generate a new multi-dimensional feature time series

$$F_s := \{f_t^s : t \in T\} \quad (13.5)$$

where each  $f_t^s$  is computed at the timestamps  $t \in T$  of the system metric time series  $P$  and has the following structure:

$$f_t^s := \begin{pmatrix} \left( \begin{array}{c} \text{mean}(E_{s,t}^{w_1}) \\ \text{count}(E_{s,t}^{w_1}) \end{array} \right) \\ \vdots \\ \left( \begin{array}{c} \text{mean}(E_{s,t}^{w_\omega}) \\ \text{count}(E_{s,t}^{w_\omega}) \end{array} \right) \\ e_{s,t}^{\text{last}} \end{pmatrix} \quad (13.6)$$

$W := \{w_1 \dots w_\omega\}$  are temporal windows of length  $w_i \in \mathbb{R}$ . Inside each of these windows,  $E_{s,t}^{w_i}$  denotes all events on communication channel  $s$  that are inside the window starting from timestamp  $t$  and going backwards in time until  $t - w_i$ , thereby fulfilling:

$$E_{s,t}^{w_i} := \{e_\tau^s \in E_s : \tau \in [t - w_i, t]\} \quad (13.7)$$

In case no events were communicated inside a window,  $E_{s,t}^{w_i}$  is empty.  $\text{mean}$  computes the element-wise mean of all event instances in  $E_{s,t}^{w_i}$ <sup>2</sup> and  $\text{count}$  computes the number of instances. Finally,  $e_{s,t}^{\text{last}}$ , indicates the most recent event in  $E_s$  that occurred before timestamp  $t$ . [Figure 13.1](#) on the previous page visualizes this approach. While the different windows represent information about different temporal horizons of the component communication,  $e_{s,t}^{\text{last}}$  is used to include information about the last communicated event, for instance, in case no communication took place within the length of the temporal windows.

To create the complete feature vector for a component, the channel feature vectors  $f_t^s$  are stacked to form:

$$F_{AEW} := \{f_t : t \in T\} \quad (13.8)$$

$$f_t := \begin{pmatrix} f_t^{s_1} \\ \vdots \\ f_t^{s_\sigma} \end{pmatrix} \quad (13.9)$$

By incorporating the temporal structure of the event-based communication already in the feature vector, the intended application of non-temporal models for learning becomes feasible. I have used temporal windows of length 2.5 s and 6 s in my experiments. These times

<sup>2</sup> Not a Number (NaN) is returned in case the window is empty.

COMPONENT	d
armcontrol	187
facerec	85
legdetector	34
objectbuilder	51
objectrecognition	51
rsbnavigation	306
scenersbam	85
speechrec	136
statemachine	765
texttospeech	51

Table 13.1: Dimensionality of the AEW features for components from the *ToBi* dataset.

were chosen empirically to match the expected communication frequencies for the components in the application scenarios. The whole feature generation approach only uses information about events sent up to the point in time of the decision and can thus be used online.

The dimensionality of the AEW features depends on the number of scopes a component uses for communication. This dimensionality  $d$  follows the formula

$$d := \sigma \cdot (\omega \cdot (\mu + 1) + \mu) \quad (13.10)$$

where  $\sigma$  is the number of communication scopes,  $\mu$  is the dimensionality of the encoding function  $m(r_t)$ , and  $\omega$  is the number of temporal windows. In case of the described encoding function for [RSB](#) with  $\omega = 2$  and  $\mu = 5$ , this results in

$$d_{rsb} := 17 \cdot \sigma \quad (13.11)$$

[Table 13.1](#) lists the size of the effective feature vectors created using this method for the component from the *ToBi* dataset.

### 13.1.2 Adding previous system metrics

Predicting the resource utilization of a component at system runtime from AEW features can be compared to an open loop control problem: The current resource utilization is ignored. However, at runtime, the previous values of the system metrics are available from the data acquired by the collection daemons. They can therefore be used in addition to the AEW features to provide further information for the prediction task. Thus, I have created a second type of features by

combining the existing AEW features with the current system metrics using a concatenation:

$$F_{\text{Combined}} := \{f_t^c : t \in T\} \quad (13.12)$$

$$f_t^c := \begin{pmatrix} f_t \\ p_t \end{pmatrix} \quad (13.13)$$

The applicability of the combined features has to be evaluated per application. Due to the included previous state, models might be able to simply track the actual behavior of a component, independent of the communication. For instance, for [fault detection](#) purposes this could be an undesirable effect. Other applications might not be able to provide the previous state. Thus, even though better results are to be expected with the additional information conveyed, applicability might be limited and the pure AEW features need to suffice for achieving usable results in many situations.

### 13.1.3 *Baseline: system metrics*

To understand how much information the AEW features convey, a comparison to a baseline approach is advisable. For this purpose, I use the current resource utilization of a component in form of the acquired system metrics as features to predict the future resource utilization:

$$F_{\text{Previous}} := P \quad (13.14)$$

This decision reflects the idea that most system metrics will evolve smoothly given an adequate sampling interval and that they are correlated over time [[Bey+09](#)]. Therefore, the AEW features must prove that they convey more usable information than what is visible from the immediate past of the system metrics.

### 13.1.4 *Preprocessing*

The part of an AEW feature vector representing a single scope  $f_t^s$  can contain values different from NaN only in case there has been communication on this scope. However, some components create more [RSB](#) participants than used in some scenarios. So, training samples gathered in these situations contain dimensions consisting only of missing values. Thus, the first step of preprocessing is the elimination of these dimensions from the feature vector. Afterwards, the second phase of preprocessing imputes the remaining missing values with the mean value of the respective dimension [[Buu12](#), p. 10]. Missing values at this point are the result of empty temporal windows  $E_{s,t}^{w_i}$  reflecting times when temporarily no communication appeared on a

communication channel. Finally, all dimensions are centered around zero and scaled to unit variance to prevent a dominance of the dimensions with the highest amplitudes in the learning task.

### 13.2 MODEL LEARNING

Based on the previously presented features, I have experimented with several regression methods, which I will briefly describe here. The learning problem solved by all approaches is to estimate the function:

$$r(f_t) : F \mapsto P \quad (13.15)$$

This function predicts the current system metrics based on a feature vector and reflects the decision to model the temporal constraints only implicitly. The target of this function, the system metrics  $p_t$ , are vectors with the dimensionality of all metrics collected for a component. I have realized  $r(f_t)$  through a set of independent learners per dimension of the output vector  $p_t$ .

For the learning task I have evaluated two established and readily available regression methods. The first method is [gradient boosted regression trees \(GBR\)](#) [HTF09, p. 359], which extends boosting to arbitrary loss functions. A grid search with cross validation was used to optimize the hyperparameters of this approach. The second method that I have tested is [kernel ridge regression \(KR\)](#) [Mur12, p. 492] with a [radial basis function \(RBF\)](#) kernel [Mur12, p. 480]. KR combines ridge regression [Mur12, p. 225] with the kernel trick [Mur12, p. 488] to overcome the linear relation of features and targets. It can be computed in a closed form. Also for this method the parameters of the kernel have been optimized using a grid search with cross validation. I have tested KR in combination with two different dimensionality reduction approaches: [feature agglomeration \(FA\)](#) and [feature selection \(FS\)](#). FA uses agglomerative clustering [Mur12, p. 895] to group together similar features and FS uses an extra trees regressor [GEW06] as a fast preliminary regression method and selects the most important features from this model. To summarize, I have tested the following learning methods indicated by the names used from now on:

GBR Gradient boosted regression trees

KR-FA Kernel ridge regression with feature agglomeration

KR-FS Kernel ridge regression with feature selection

MEAN Baseline that predicts the mean value for each system metric found in the training samples.

The main purpose of testing more than one method was to show that a suitable learner for this problem can be found.

APPROACH	RRSE
GBR	0.78
Mean	1.02
KR-FA	23.90
KR-FS	29.67

Table 13.2: Mean prediction errors on AEW features.

### 13.3 EVALUATION

The aim of the evaluation was to find out whether one of the presented approaches is able to learn the resource utilization from the encoded events and on which factors the achieved results depend. For this purpose, I have first conducted an evaluation on the *ToBi* dataset (cf. [Chapter 12](#) on page 141) as an example for a real scenario.

#### 13.3.1 Results on the *ToBi* dataset

The following results have been acquired by testing the different feature versions and regression methods on the trials without induced bugs or unexpected faults from the *ToBi* dataset presented in [Chapter 12](#). Methods have been tested using a five-fold cross-validation with random folds.

Presenting an overall quantitative evaluation for the introduced prediction task is challenging due to the different measurement units and scales of the different system metrics. Therefore, a straightforward averaging of the [root mean square error \(RMSE\)](#) across the different metrics is impossible because this well-interpretable error is represented in the unit of each system metric. Instead, I will use the [root relative squared error \(RRSE\)](#) for parts of the evaluation that average across the different metrics. RRSE is an established loss function for such cases [[HKo6](#)]. It weights the relative squared error with the one of the naive approach of predicting the mean of the true data and therefore returns a fraction that is independent of the unit. It is defined as

$$\text{RRSE} = \sqrt{\frac{\sum_{i=1}^N (\hat{\theta}_i - \theta_i)^2}{\sum_{i=1}^N (\bar{\theta} - \theta_i)^2}} \quad (13.16)$$

with  $\theta_i$  being the true value of the  $i$ -th data item,  $\hat{\theta}_i$  the predicted value, and  $\bar{\theta}$  being the arithmetic mean of all values. An error of 1 is achieved if the mean value of the data is predicted. Lower errors indicate actual success in the prediction task.<sup>3</sup>

<sup>3</sup> This is only true in case the statistical properties of the training and test set are equal. Otherwise, scores above 1 can still indicate success, but worse than predicting the mean of the test data.



SYSTEM METRIC	KR-FA	MEAN	DATASET
	RRSE	RMSE	MEAN
proc/fd-open_connections	3.85	2.52	19.08
proc/fd-open_fds	0.97	2.54	45.37
proc/fd-open_files	1.01	0.0008	16.80
proc/stat-num_threads	2.57	4.66	64.75
proc/stat-stime	0.90	0.83 %	0.44 %
proc/stat-utime	0.75	11.52 %	12.93 %
proc/stat-rss	1.53	29.65 MB	187.90 MB
proc/stat-vsize	298.68	170.45 MB	4719.43 MB
proc/io-rchar	0.78	87.05 kB s <sup>-1</sup>	3.30 kB s <sup>-1</sup>
proc/io-wchar	0.66	686.65 kB s <sup>-1</sup>	741.03 kB s <sup>-1</sup>
proc/io-read_bytes	1.24	17.55 kB s <sup>-1</sup>	0.32 kB s <sup>-1</sup>
proc/io-write_bytes	0.89	3.20 kB s <sup>-1</sup>	1.27 kB s <sup>-1</sup>
netbandwidth-received_bytes	0.90	13.53 kB s <sup>-1</sup>	21.76 kB s <sup>-1</sup>
netbandwidth-sent_bytes	0.85	97.47 kB s <sup>-1</sup>	297.50 kB s <sup>-1</sup>

Table 13.3: Detailed comparison of the KR-FA method to the baseline MEAN.

Table 13.2 on the facing page displays the mean RRSE across all components for the task of predicting the current resource utilization from the AEW features. The results show that only GBR is able to successfully learn from the results and improve on the RRSE achieved by the baseline approach of predicting the mean training values (not to be confused with the *naive baseline* used in the RRSE computation).

To understand why the other regression methods fail to learn models that are, on average, better than predicting the mean value, a closer look at the individual system metrics is necessary. The second column of Table 13.3 displays the RRSE of the KR-FA method. The highlighted rows represent system metrics for which the prediction is worse than the naive baseline of the RRSE computation (values greater than 1). Apart from `proc/io-read_bytes`, one important observation for these metrics is that they can already be predicted with a reasonable accuracy just using the mean of the training data. To visualize this, the third column of table Table 13.3 shows the RMSE of the baseline approach converted to graspable measurement units and the fourth column shows the average value of the metrics across the components of the dataset. In the marked cases, the baseline RMSE is only a small fraction of the absolute values. As the prediction of the mean value is already sufficient to achieve low errors in these cases, other methods can hardly improve the results. Moreover, especially KR seems to exhibit overfitting on the training data as it cannot resort to predicting the mean value.

The reasons why some learning methods cannot improve on the mean prediction can also be explained by the type of affected metrics (cf. Appendix B.7 on page 225 for a detailed description of the

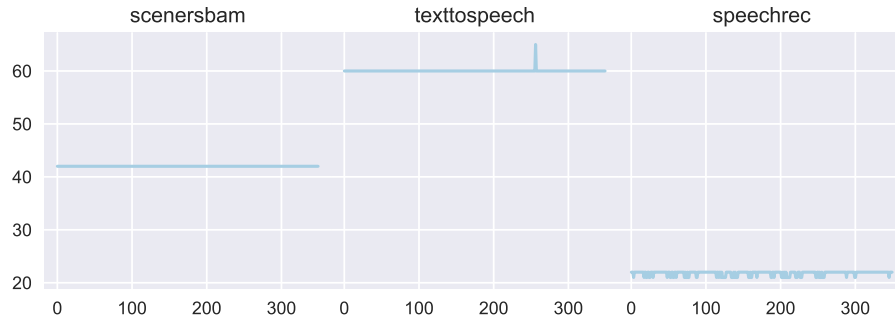


Figure 13.2: Examples for the `proc/fd-open_fds` metric from the *ToBi* dataset. The x-axis represents trial time in seconds and the y-axis is the number of open file descriptors of a component.

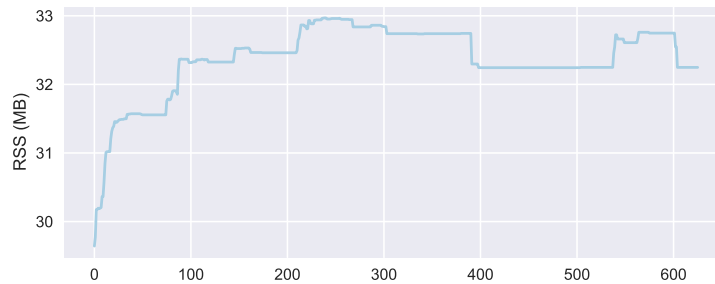


Figure 13.3: Progression of the resident set size (RSS) of the `objectbuilder` component in the *ToBi* dataset. The x-axis is time in seconds.

metrics). The first four metrics in Table 13.3 are counters for entities that usually do not fluctuate much. Figure 13.2 shows examples from the dataset of how the `proc/fd-open_fds` system metric behaves for some components. The mean value can hardly be beaten here. Apart from the previously mentioned exception of `proc/io-read_bytes`, the remaining two metrics measure the memory a component uses. As already explained in Section 10.4 on page 120 for the testing framework, memory often behaves unpredictably, for instance, because of garbage collection processes unrelated to the current processing of a component. Hence, the event-based communication cannot provide further information to improve the prediction.

Finally, another criterion exists that applies to all metrics where learning fails (previous exception applies): The time series of these metrics are characterized by longer periods with exactly the same measurement values, sometimes followed by an instantaneous shift to a different level where the pattern continues. Figure 13.3 shows an example for the RSS of a component. Such degraded time series can also be classified automatically by comparing the amount of change in the time series to the mean value of the time series using

$$\text{degraded}(m) := \frac{1}{|D^m|} \sum_{d \in D^m} d \leq 0.1 \quad (13.17)$$

APPROACH	RRSE
GBR	0.795
KR-FS	0.810
KR-FA	0.814
Mean	1.003

Table 13.4: Mean prediction errors on AEW features after removing degraded system metrics from the evaluation.

where  $m$  is the current metric and  $D^m$  is a boolean series indicating whether each value in a metrics time series is different from the previous value:

$$D^m := \{\text{bin}(|p_t^m - p_{t-1}^m|) : t \in T, t \geq 1\} \quad (13.18)$$

$$\text{bin}(x) := \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \quad (13.19)$$

In other words: if the time series contains less than 10 % entries with values different from their predecessor, the time series is degraded. By applying this heuristic to the aforementioned trials of the dataset, all four counter metrics (first four in [Table 13.3](#) on page 159), the two memory metrics, and `proc/io-read_bytes` are identified. Therefore, I have decided to ignore these metrics for the remaining evaluation because of their special nature. After exclusion, all models are able to achieve reasonable errors, as shown in [Table 13.4](#). This indicates that KR exhibits overfitting in case the best thing to do is to predict the mean value. On the other hand, GBR handles these cases gracefully and is therefore the best general-purpose method for the task, as long as training time is not important.

For further analyses, in [Figure 13.4](#) on the following page I have plotted the RMSE for the most promising two prediction methods individually for the remaining 7 system metrics. Apart from `netbandwidth/received_bytes` and `proc/io-rchar`, the AEW features result in much lower errors than the baseline approach and the errors are in an acceptable range. Apart from `netbandwidth-sent_bytes`, the prediction from the encoded events alone is also better than from previous metric values. The error that is achieved with the *Previous* features ( $F_{\text{Previous}}$ ) compared to the baseline varies for the different metrics and is a result of the average complexity of the utilization behavior of each metric. The exception for the network metric can be explained by the lower sampling frequency used for this metric in the original recording (0.5 Hz instead of 1 Hz). Before training the models, I have upsampled the network metrics to the common 1 Hz frequency. Thus, 50 % of the time of the upsampled data it is a perfect assumption to guess the previous metric value. The exception here is

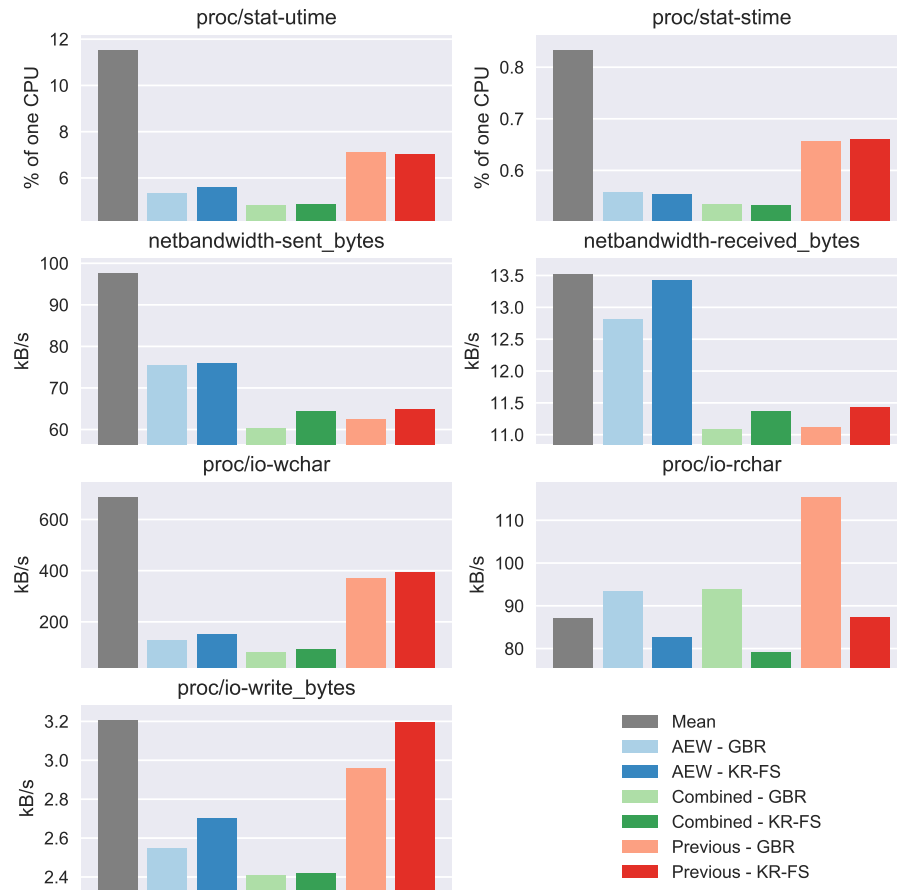


Figure 13.4: RMSE per system metric of different regression methods on the *ToBi* dataset. Note that the y-axes do not start at zero.

hence a result of the imperfect data recording and the post-processing and not systematic for the general task. Finally, the first mentioned two exceptions regarding `netbandwidth/received_bytes` and `proc/io-rchar` are also explainable. The components of the dataset have two options for communication with the central *Spread* [Spread] daemon of the system: C++ component on the same host as the daemon can use Unix domain sockets whereas Java component and all components on other hosts use network links. In the former case, the incoming communication to the components is reflected in the `proc/io-rchar` metric whereas in the latter case it is measured in `netbandwidth/received_bytes`. The resource utilization of the dataset components is primarily influenced by the incoming communication. Therefore, the regression models cannot predict these metrics because once the predictor events are received and available to contribute to the feature vectors, they have already been reflected in the metrics. Thus, such metrics reflecting the incoming communication are effectively unpredictable with any method based on the communication.

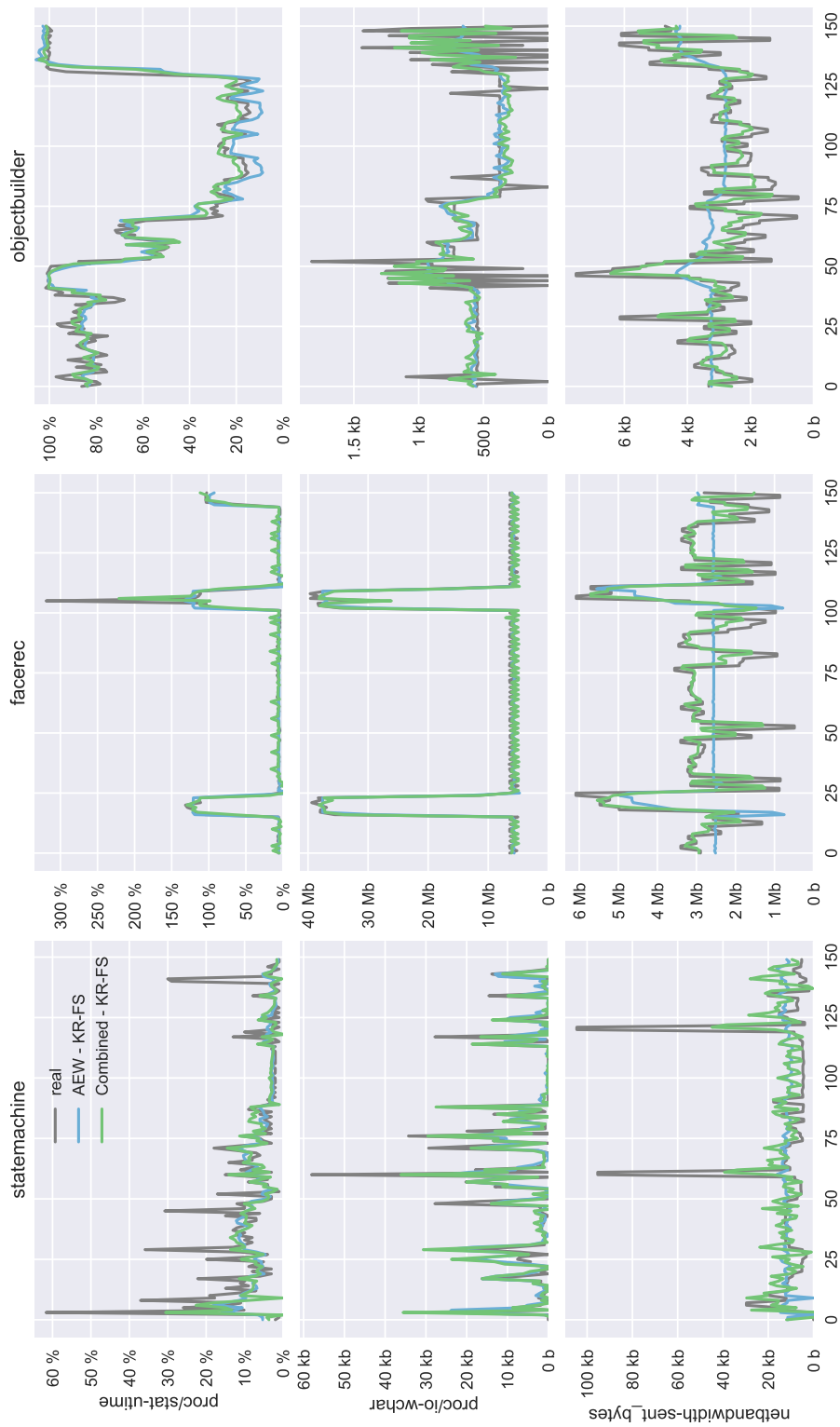


Figure 13.5: Examples for the prediction of three system metrics for different components. The x-axis shows time in seconds.

Finally, [Figure 13.5](#) on the previous page shows examples for the prediction of system metrics to visualize the achieved prediction performance. Here, models were trained on two trials of the dataset before predicting another trial. Already on the open loop AEW features, both regression methods are able to capture the essential behavior.

### 13.3.2 Influences of the component behavior

The previous evaluation has shown that the prediction error varies for different system metrics. However, also the structure of the components and the way their internal processing works will influence the results. I have constructed a set of mock components to systematically analyze this aspect. These components follow the [component interfaces](#) of components from the *ToBi* dataset so that I could feed them with realistic input data originating from the dataset. All mocks implement different processing strategies that primarily influence the CPU utilization. Therefore, the following analyses are limited to the `proc/stat-utime` metric as the metric with the highest variability. This also reduces the complexity of the analysis and for the interpretation. In detail, the mock components are:

**PEAK-LOADS** Modeled along the `facerec` component, this mock produces peaks of constant CPU utilization between longer idle times. The length of the peaks depends on the number of training phases triggered before. These training phases produce a constant load until the caller stops them. Thus, the mock is an example for a component doing short on-demand computations and long-term knowledge would be required to correctly predict all details of the utilization pattern.

**CONSTANT-INPUT** This mock scales its CPU utilization with the received inputs data. It models components performing constant processing without internal state such as vision algorithms.

**CONSTANT-STATE** Implements a comparable input dependence as the previous component, but the utilization mainly depends on an additional internal state that is periodically changed to different random values. The mock is thus an example for a more complex algorithm with constant processing activity. It is intended to analyze the influence of internal state on the prediction. For this purpose, two configurations have been tested by reporting or not reporting the state through output events.

**CONSTANT-COMPLEX** Realizes even more complex constant processing. In contrast to the previous mocks, this one depends on two periodic inputs in parallel, where one input results in a constant base load. The other input can be processed by a variable amount of worker threads. With a single thread, the incoming

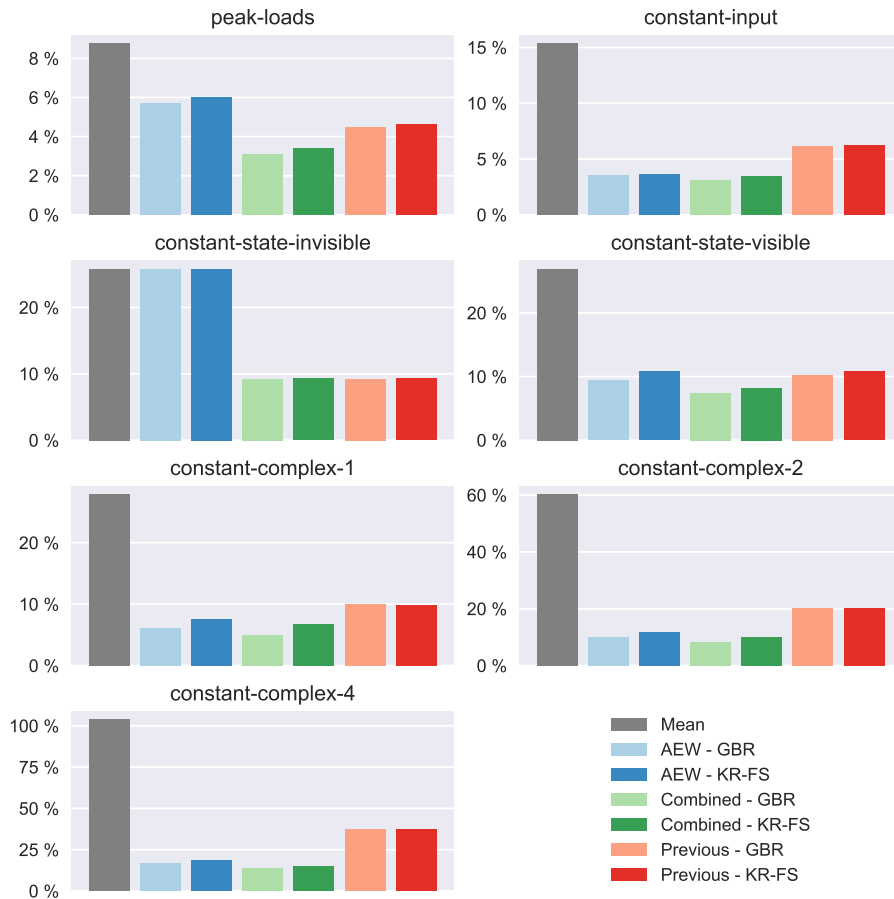


Figure 13.6: RMSE of different prediction methods on the `proc/stat-utime` metric for the different mock components. The measurement unit is the utilization of a single CPU core in percent.

work saturates the worker thread most of the time. However, the more threads are used, the more variability appears in the CPU utilization due to less saturation. The purpose of this mock is to understand how the prediction behaves with changing levels of variability for a constantly processing component. This mock has already been used in Wienke and Wrede [WW17a].

All mock components have been evaluated using a five-fold cross-validation with random folds on a concatenation of 5 fault-free trials from the *ToBi* dataset. Figure 13.6 shows the results for different approaches on the mock components. Moreover, examples for the achieved predictions can be seen in Figure 13.7 on the next page.

For the peak-loads component, predictions based on AEW features result in lower errors than the baseline. Hence, the regression methods are able to learn the occurrences of peaks. However, due to missing knowledge about the exact length and the resource utilization behavior resembling the degradations explained before with long times of the same measurements, predictions based on previous metric values are able to achieve lower errors. Predicting the previous values is



Figure 13.7: Examples for the predictions achieved for the mock components (proc/stat-utime). The unit of the y-axis is the utilization of a single CPU core in percent.

usually correct and only fails at the few times when the peaks start and stop. Still, a pure open loop prediction achieved reasonable results. The example in Figure 13.7 for this component underlines this analysis, but also demonstrates that the prediction accuracy based on AEW features is still good and captures the essential behavior of the component.

For all constantly processing components where the utilization is primarily determined by the inputs (constant-input and constant-complex), the AEW features dominate the prediction accuracy and knowledge about the previous metric values does not help much to further improve the results. Thus, components with such a direct input dependence and constant processing can be modeled with a high accuracy. The results of the different constant-complex configurations further show that the dominance of the AEW features for achieving low errors increases with more flexibility and less CPU saturation. With more threads, the difference between AEW errors and *Previous* errors increases.



Finally, for the constant-state component, the AEW features do not convey any information as long as the component does not expose its determining internal state. Once this information becomes available, the AEW features outperform knowledge from previous metric values.

These experiments show that components with constant processing can be predicted well as long as the necessary information are available in the communication. From an architectural point of view, this underlines the necessity for small and coherent components that avoid complex internal processing and state for a successful application of the proposed resource awareness method. For components with short on-demand activities, the general tendencies can be predicted with a reasonable accuracy. However, extensions to the features are necessary to capture complex interaction protocols such as the dependent [RPC](#) calls in the peak-loads example.

#### 13.4 LEARNING FROM PERFORMANCE TESTS

The evaluation has shown that it is possible to predict system metrics with a good accuracy using the presented approach. However, training the prediction models from real executions of a system has several implications that contradict the framework-level resource awareness concept. First of all, such a training requires the availability of a sufficient amount of representative execution data of the target system. Hence, every time a system is modified, it has to be run for some time to acquire training data. The resulting effort often conflicts with the idea of having methods that are easily applicable. Second, the resulting prediction models are only trained with interactions that appear in a single system. Thus, they do not capture correct operational states of components outside these interactions. Consequently, a model learned for one component is not resilient against changes to its surrounding system that would result in acceptable but previously unseen inputs to the component. For instance, a system developer might decide that it is better to capture image data from a camera with a higher frame rate or resolution. An existing model for a component processing these frames has to be retrained because no training data including this variability was available so far. From this point of view, the training data should ideally capture more variability than what is available in a single system instance.

The performance testing framework presented in [Chapter 10](#) on page [105](#) provides a potential solution to both aspects. By design, during test execution the complete event communication as well as the system metrics for the tested component are recorded. Therefore, the necessary information to construct the AEW features and to learn prediction models are available. Moreover, the framework is designed to systematically explore the complete range of expected input vari-

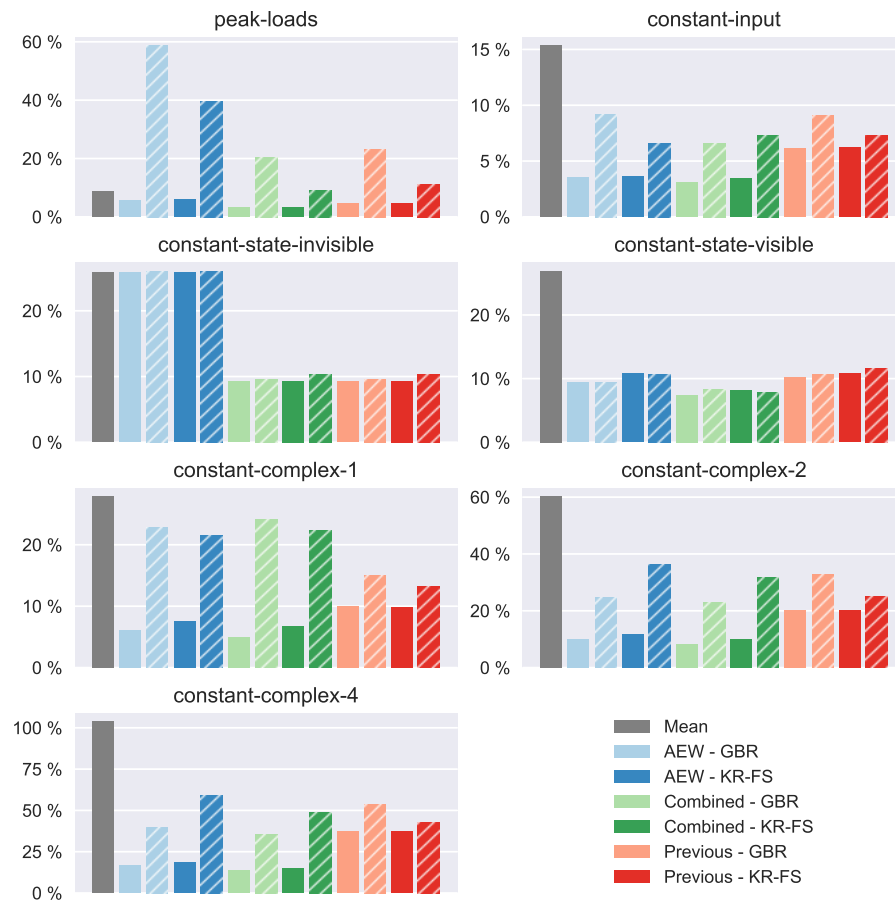


Figure 13.8: Comparison of prediction errors (RMSE for `proc/stat-utime`) between models trained on the dataset (solid, results from Figure 13.6) and models trained on performance tests (hatched).

ability for a component. Thus, the regression models trained on test execution data represent more variability in the application context of a component than from a single application. Hence, these models should be resilient against changes to the external system and in case they need to be retrained, the tests allow retraining models for individual components without needing data from the integrated system.<sup>4</sup> This approach, which I have initially explored in Wienke and Wrede [WW17a], will drastically reduce the required effort to apply the prediction method. However, the learning problem in this case is harder, because the underlying statistical properties of the features at training time differ from the ones at application time and the learner must generalize between these situations. Therefore, a drop in accuracy can be expected and the question is whether the acquired models are still usable.

<sup>4</sup> Training still has to be performed on the target computer due to the coupling of learned models to the executing hardware.

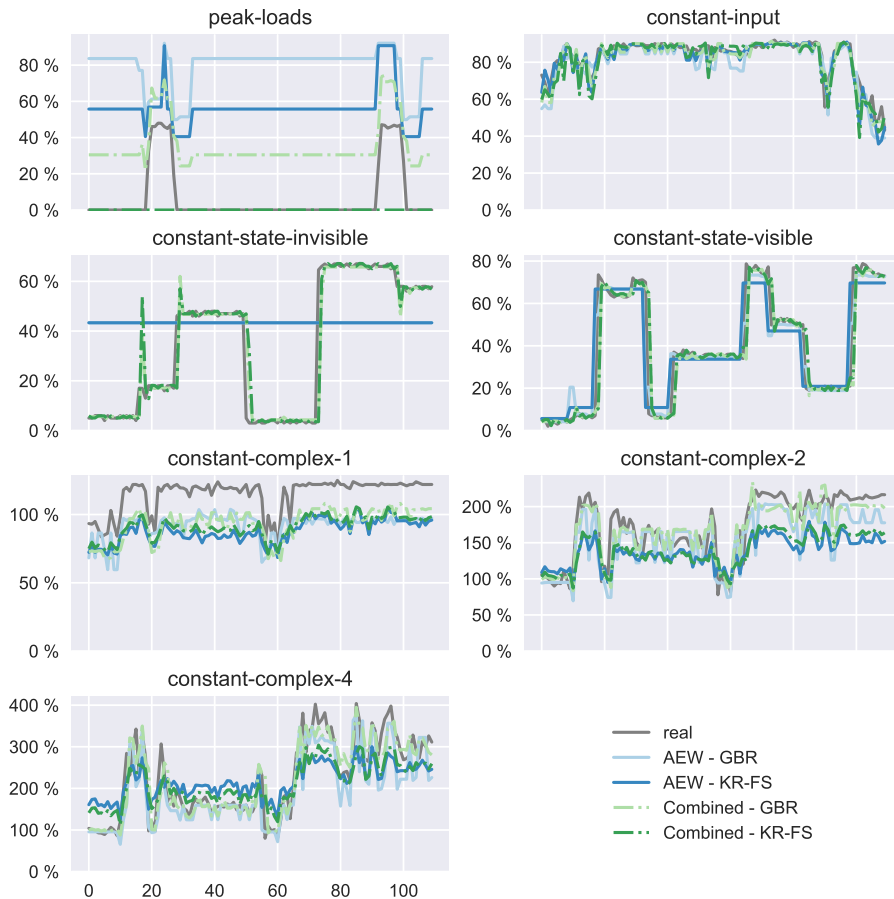


Figure 13.9: Examples for the predictions achieved for the mock components (proc/stat-utime) when training models from performance tests. Shows the same situations as Figure 13.7.

#### 13.4.1 Evaluation

To evaluate the presented idea, I have again used the mock components from the previous evaluation. As these components fulfill component interfaces of components from the real *ToBi* system, I could reuse the existing performance tests for *ToBi* components for acquiring training data. These tests were already used in Section 11.4 on page 135. Models trained from the acquired test training data were then tested on executions of the mock components with inputs from the *ToBi* dataset as a simulation of a real application scenario. The experiments were repeated 5 times and the presented results are the arithmetic means. The achieved errors as well as examples for predicted time series are depicted in Figures 13.8 and 13.9 on the preceding page and on this page.

The plots show mixed results. While the prediction generally works for components with constant processing – although with varying levels of success – it fails for the peak-loads component with its on-demand processing. The respective prediction example in Figure 13.9

demonstrates the problems. Even though the general tendency for changes in the prediction at the correct times is visible, the mean levels diverge from the real values. This phenomenon can be explained when comparing the inputs to the component generated by the performance test to the ones generated by real system executions. In the test, the pauses between the RPC calls are much shorter than in the real application to avoid extensive test runtimes. Thus, the properties of training data are too different compared to the test dataset and the learned models fail to generalize across this gap.

For the constant-input and the constant-state component with visible state, the prediction results are close to perfect or even slightly better than the results from the dataset. In these cases, training from the performance tests was able to generalize to the dataset situation. Finally, for the constant-complex component, the prediction results are generally acceptable, but the amplitude of the system metric is not reproduced correctly, resulting in higher errors. The increased complexity of the component could be one reason for the decreased accuracy of the prediction, but further analyses are required to verify this hypothesis.

#### 13.4.2 *Influences of the test structure*

Generally, the evaluation shows that the achievable error rates depend on how well the training data from the performance tests resembles – at least partially – the test data. In Wienke and Wrede [WW17a] I have analyzed this influence in detail for the constant-complex component. The component interface adopted from the objectbuilder component accepts global SLAM positions of the robot on one scope and a list of detected leg pairs on a seconds scope. The effective CPU utilization of the mock depends on three aspects encoded in these events. To analyze their influence on the prediction results, I have executed the existing performance test with combinations of three different configurations per aspect, resulting in 9 effective test configurations. The configurations for the aspects were:

LEG NUMBER The number of detected legs.

- d: Match the dataset and produce 1 to 22 detected legs in steps of 3.
- l: Produce less data by sending 1 to 13 detected legs in steps of 3 to test extrapolation.
- m: Produce more data by sending 1 to 28 detected legs in steps of 3 to test the influence of additional training data.

**LEG RATE** The rate of leg detection results.

- d**: Match the **d**ataset with leg detection results at 30 Hz.
- r**: Use a **r**ange of different production rates, but not exactly the target rate from the dataset, i.e. 10, 20, 40 and 50 Hz to test interpolation.
- dr**: Use a **r**ange of production rates including the **d**ataset target rate, i.e. 10, 20, 30, 40 and 50 Hz.

**POSE RATE** The rate of SLAM position results. Similar conditions:

- d**: 10 Hz
- r**: 2.5, 5, 15 and 20 Hz
- dr**: 2.5, 5, 10, 15 and 20 Hz

The results of these experiments are visualized in [Figure 13.10](#) on the next page for the most promising open loop prediction approach *AEW - GBR*. I did not select an approach including previous system metrics to specifically analyze only the influence of the event-based communication.

The results show that presenting more flexibility in the training data helps to learn a prediction model that generalize to the novel situation. For the event rates, the *d* and *dr* conditions produce lower error rates and expectedly the condition including the dataset rate (*dr*) results in the lowest average prediction error. For the leg number, which is a property of the data contained in the sent events, the condition with less variability than the dataset results in the lowest errors. Even though this is unintuitive at first, an explanation for this behavior is that most leg detection results in the dataset contain only a few pairs of legs and the theoretical maximum of 22 legs is only rarely reached. Thus, the *l* condition focuses on the most frequent cases of the dataset while ignoring seldom outliers and therefore allows training models that are more accurate for the majority of samples.

These analyses show that the structure of the performance tests has a noticeable effect on the achievable results. In case prediction models will be trained from tests, the test structure and parameters should thus be optimized so that at least the variability of potential application scenarios is contained in the training data to achieve better results. Further research will be necessary to devise useful guidelines on how to structure the performance tests for this purpose. With the performance testing [DSL](#) as a front end for specifying performance tests, these guidelines can be transformed into directly applicable editing aids realized through model checking and further [IDE](#) features. The model-driven engineering approach chosen here will therefore directly help to follow these guidelines and thus again limits the effort a developer has to put into applying the framework-level resource awareness ideas.

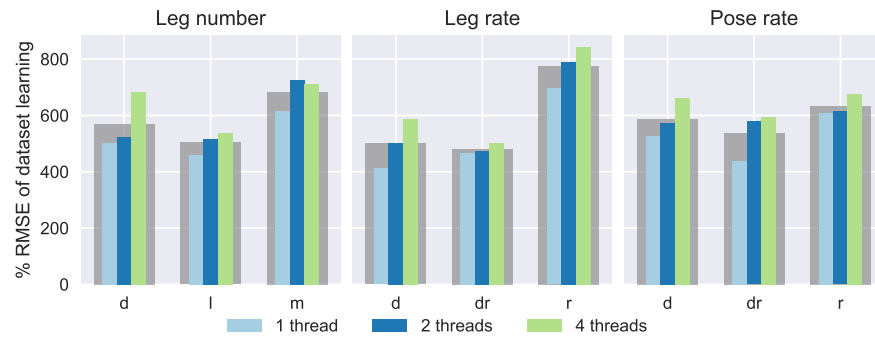


Figure 13.10: Prediction errors of the *AEW - GBR* approach for the constant-complex component for training based on different performance test configurations. Errors are expressed in percent of the RMSE a comparable model trained on dataset trials would make, which are the errors reported in Figure 13.8. Gray bars show mean values across all three thread configurations.

### 13.5 RELATED WORK

The presented approach for predicting the **resource** demands of robotics software components is a data-driven method. Such methods are also used in other areas, most notably, for the operation of large data centers for cloud environments or grid computing. Predictions of the future resource demands are necessary to provision just enough resources to an application so that **SLAs** are met while minimizing costs. The majority of data-driven methods in this domain realizes the prediction task as a time series forecasting problem by using past measurements of resource utilization to predict future demands without further information. For instance, Bey et al. [Bey+09] use fuzzy inference based on clusters of the historical time series data. Davis et al. [Dav+13] propose regression methods to predict the utilization of cloud hosts with adaptations to predict average and peak loads equally well. In contrast, Gong et al. [GGW10] approach the task with a mixture of two distinct methods. First, pattern mining is used to detect repeating patterns, which are then used for forecasting. If no such patterns can be found, the approach resides to Markov chains for forecasting short-term demands. Finally, Xue et al. [Xue+15] predict the resource utilization of data center applications using neural networks that use the history of the system metrics to predict their future values. Multiple networks are combined with bagging to improve the prediction. Further comparable forecasting methods are reviewed systematically in Gupta and Dinesh [GD17] and useful extensions are described. In contrast to the multi-step procedure of first forecasting the resource utilization and then deciding on how to provision resources so that SLAs are met, Xiong et al. [Xio+15] directly model the relation of available computing resources to the costs of potentially violating the SLAs using machine learning techniques.

A closely related problem is the prediction of runtimes and load profiles of CPU-bound jobs in high performance computing and grid settings. For instance, Seneviratne and Levy [SL11] introduce a prediction method for this problem based on a load profile of an application without [resource contention](#) and current measurements. Other approaches in this area are reviewed in Seneviratne et al. [SLB13].

With the advent of virtualized hosting several years ago, another interesting and related problem was the question of how application resource demands change when an application is transferred from a real system to a virtualized one. Wood et al. [Woo+08] have addressed this task by systematically benchmarking different virtualization solutions. From the acquired data, regression models were trained per virtualization solution. Each of them predicts the resource demands on the virtualized platform based on runtime data acquired while executing an application on real hardware. A generalized problem of predicting resource utilization of applications for previously unknown platforms is addressed in Shimizu et al. [Shi+09].

A related solution to the resource provisioning problem is to predict the requests that will arrive on an application for being able to scale the available resources accordingly. A common approach here is to construct generative models from past data to predict future requests [YKZ16; DVC13].

To summarize, most prediction methods for resource utilization operate primarily on larger units of deployed software with a horizon from a few minutes to hours or days using time series forecasting techniques. Instead, this work targets much smaller software parts in the sense of [microservices](#), which operate with more flexibility on the short-term time scale of interest. I am not aware of any other approach in this direction that uses the network-based communication as a primary feature for the prediction. Therefore, this is a novel perspective on the forecasting problem for resource utilizations.

## 13.6 SUMMARY

With the preceding evaluations I have shown that communication events can be used to predict the resource utilization for a set of system metrics. They convey enough information about the requested workload and the external and internal system state to predict the resource utilization of robotics components. Therefore, the initially stated [Hypothesis 1](#) on page 150 holds and this approach can be used as a tool for implementing the framework-level resource awareness concept. However, it is important to note that *prediction* in this work does not imply predicting future values. Instead, the trained regression models encode an expectation about the current resource utilization based on external information received through the event-based communication. Therefore, the question is how this basic method can

be exploited with actual value for the robotics system. First, the next chapter will show a runtime fault detection approach that is based on an expectation about the resource utilization of each component encoded through the presented regression models. It is the proof of concept for the applicability of the AEW features and the presented regression models. However, also other exploitation scenarios can be envisioned. As an additional runtime application, trained models can potentially be used for mid-term planing with the aim to avoid resource contention or [resource starvation](#) based on simulated communication events. For a larger set of components, the actual structure of the ongoing communication is either regular (inputs at a fixed rate) or determined by control commands of controlling state machines. Therefore, from the perspective of such a state machine, a simulation of future event sequences should be possible. This approach cannot use information about previous metric values and can only rely on the AEW features without additions for previous values. The evaluation has shown that such open loop predictions already result in good error rates and are thus sufficient for the task. Being able to train regression models from performance tests is an important step towards reducing the required effort for the application of such methods.

Going back to the developer perspective presented in Part III of this work, trained regression models could also be used as another method for detecting [performance regressions](#) in the testing framework introduced in [Chapter 10](#) on page 105. If the resource utilization of a new software revision deviates from the predictions made by the trained model from a former version, a [performance bug](#) might have been added. Both ideas as well as other exploitation scenarios required a verification in future work.

Two important restrictions apply to the presented work in this chapter. First, all trained models are coupled to the execution hardware the training data was acquired on. The availability and utilization of [system resources](#) will differ across platforms and this variation is currently not modeled. Second, the presented work does not explicitly model resource contention aspects. All analyzed systems were sufficiently sized so that no severe resource contention appeared. As the overall load on the host system is not part of the features or models, predictions will degrade in case the training data or the actual system execution is affected by resource contention. Both aspects have to be addressed in future work to make the resulting models more flexible.



## RUNTIME PERFORMANCE DEGRADATION DETECTION

---

The previous chapter has introduced a way to represent the relation between communication [events](#) and the [resource utilization](#) of system [components](#) using machine learning techniques. The learned models were meant as a foundation to realize different runtime [resource awareness](#) functions. In this chapter, I am going to explore an exemplary application by detecting [performance bugs](#) at system runtime. This work extends an initial publication of the method in Wienke and Wrede [WW16c].

Even though the previous part of this thesis has introduced methods to prevent performance bugs already at development time, such methods can never completely prevent [bugs](#) from ending up in a production systems, especially in cases where they are currently being adopted. Edsger W. Dijkstra already concluded in 1972: “program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence” [Dij72]. Therefore, performance bugs also need to be controlled at system runtime and the survey presented in [Chapter 3](#) on page [21](#) has underlined the need for such runtime detection methods. Detecting performance bugs allows reacting on them and repairing the system state before a critical situation arises or a [system failure](#) eventually renders the robot or intelligent system useless [Ste13; Zam+13]. Although other disciplines have started to adopt autonomous [fault detection](#) methods, only limited use can be observed in robotics research. Here, the largest part of the existing fault detection work focuses on specific areas such as sensor or actuator [faults](#) [cf. [Pet05](#); [SMWo6](#)]. However, bugs in the control software of intelligent systems and performance bugs have mostly been ignored so far. Thus, the presented fault detection approach for performance bugs – apart from demonstrating the usefulness of the previously introduced features and regression models – brings a novel perspective to the robotics and intelligent systems community that has not been addressed before. It is important to note that the presented approach cannot detect performance bugs directly. Instead, it can only detect their visible effects, namely the [performance degradations](#), as already explained in [Section 2.2.3.1](#) on page [16](#). Hence, the presented specific form of fault detection could also be called *performance degradation detection*. For the sake of brevity and relation to existing work, I will however continue to use the term fault detection.

### 14.1 RELATED APPROACHES

Fault detection, or more general, novelty or anomaly detection has a long tradition in several domains and many techniques have been developed. Chandola et al. [CBK09] and Miljkovi [Mil11] present thorough overviews about these general techniques, their categorization, and application domains.

Regarding existing approaches used in robotics, Pettersson [Pet05] gives a good overview on early research and distinguishes between analytical, data-driven, and knowledge-based approaches. The data-driven approaches are closest to my own work. However, most of them treat the fault detection problem as a classification problem with known faults. This requires annotated training data with recorded and known fault instances and I have already argued in Chapter 12 on page 141 that such data hardly exists in the robotics domain. Additionally, these methods cannot generalize to unexpected types of faults. Finally, none of the presented methods specifically focuses on faults in the control system or on performance bugs.

A method with focus on the control system has been presented in different development stages by Weber and Wotawa [WW06], Kleiner et al. [KSW08], and Zaman et al. [Zam+13]. This fault detection mechanism is based on observers, which either check invariants in the communication using Horn clauses or thresholds on common system properties. Observers are manually designed or derived automatically from reference executions in case of invariant checks. Due to the strict nature of the invariants, imperfect training data will likely affect the results of this approach. Based on the observer results, a diagnosis engine searches for contradictions of the current state with a predefined model of the system. Thus, despite being data-driven regarding invariants, this method also requires a system model.

In contrast to the previous approach, Khalastchi et al. [Kha+15] introduce an online fault detection purely data-driven approach that compares the correlations of automatically clustered redundant measurements using a sliding window. This approach does not specifically handle performance bugs. Instead, it is evaluated with sensor data. Because of using only the recent history of sensor readings, the method lacks explicit state knowledge and sudden behavior changes will likely result in false positive classification results (cf. Chapter 13 on page 149). Moreover, no explicit knowledge about the origin of a detected fault is available. With Khalastchi and Kalech [KK17] the authors have recently presented an improved version of this approach and incorporated the same idea as Zaman et al. [Zam+13] to use model-based diagnosis for deriving the root cause of a detected fault.

Jiang et al. [JED16] recently proposed an approach that uses the communication of the system to assess its health state. Invariants in the communication are learned based on templates and reference ex-

ecutions of the system. The presented evaluation shows the ability of the approach to detect and correct violations that originate from external disturbances of the expected environment of a UAV (e.g., increased wind speed). The method does not specifically address issues in the control software and detected faults cannot be attributed to system components.

Finally, Golombek et al. [Gol+11] describe a fault detection method that also utilizes the communication of the robot to assess the system state. Even though this method is a general fault detection approach, it was also able to detect some performance-related faults. However, only the complete system is inspected and the detection results do not carry information about the affected components.

The runtime detection of performance degradations is also of interest and more established in domains such as server administration or cluster computing. Because the variations in resource demand are often more stable than in robotics, many solutions here use time-series processing methods without incorporating context information such as event streams. Examples include *Skyline* [Skyl] and *Riemann* [Riem]. One noteworthy exception is Knorn and Leith [KLo8], where in addition to a base model of the resource utilization, an additive event model exists that adds utilization patterns for pretrained events. These event models need to be recorded beforehand, which prohibits the easy application of the approach in the robotics domain with many state changes and events that need to be modeled.

The review of the related work shows that a data-driven runtime detection of performance degradations for individual system components is a novel perspective in the robotics domain. Existing methods focus on sensor data or require extensive system models to provide a component-level report resolution and system state is often not explicitly modeled.

#### 14.2 RESIDUAL-BASED DETECTION OF PERFORMANCE DEGRADATIONS USING RESOURCE PREDICTIONS

To realize a fault detection method compatible with the [framework-level resource awareness](#) concept, an important influence is that training data with system execution traces containing annotated fault instances is not available and a structured acquisition of such data is infeasible. Thus, the fault detection method can only be realized as a novelty detection or one-class classification problem. In such a setting, the classifier has the task to decide whether a sample belongs to the class of known healthy system executions or not, without knowing counter-examples. To realize this classification task with a focus on performance bugs, I have decided to use the previously presented resource prediction models as model of the known and expected healthy system behavior and a residual-based fault detection scheme

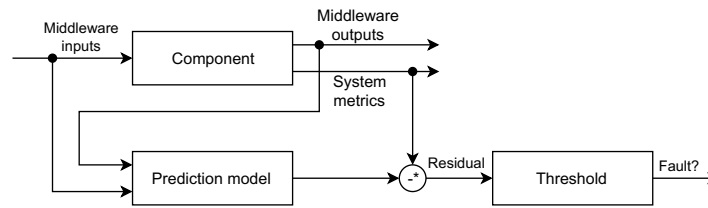


Figure 14.1: Scheme of the fault detection approach. Adaptation of the general residual-based fault detection scheme in Ding [Dino8, p. 7].

[Dino8, p. 7] based on the predicted and actual resource utilization to implement the decision logic. Thus, for each component of the system, a fault detection scheme as visualized in Figure 14.1 is used.

*residual-based fault detection scheme*

⊙ This scheme encodes the idea that a performance degradation for a single system component is a deviation of its current resource utilization from the predicted resource utilization above an acceptable threshold. As the uncovered and predicted system metrics are vector-valued, I have decided to use the Mahalanobis distance (indicated as  $-*$  in Figure 14.1) to compute a scalar residual that can be compared to a threshold. The Mahalanobis distance is an established method for fault detection in case of low-dimensional problems [LKK10; VF11], which is the case for the limited set of system metrics used in this work (cf. Appendix B.7 on page 225).

All aspects of the presented scheme can be learned from suitable training data. In the simplest case, the training corpus comprises reference executions [She+09] of the system in the intended scenario that have been classified to be free of visible faults by an expert user or developer of the system. This method cannot ensure that a healthy system is used for training in all cases. Therefore, the expert user needs to closely monitor the important KPIs (for instance using the monitoring tools presented in Chapter 9 on page 93) as well as the delivered service of the system to reduce the likelihood that the training data are severely impaired. Despite not being perfect, this method is used by several other publications in this area, for instance, Golombek et al. [Gol+11] and Jiang et al. [JED16]. Moreover, suitable modeling methods are able to deal with some amount of noise in the training data [CBK09].

Using the training data, the following three steps are performed for each system component:

1. Based on a first fraction of the training data, the resource prediction model as described in Chapter 13 on page 149 is trained.
2. A second fraction of the training data is used to estimate an empirical covariance matrix of the prediction error for the system metrics. This covariance matrix characterizes the ability of the prediction model to reproduce the different system metrics.

3. In the last step, the remaining fraction of the training data is used to estimate a suitable threshold for the Mahalanobis distances. One possibility to realize this step is to use gradient descend to achieve a desired false-positive rate on this part of the training data.

In addition to this fault detection method based on the previously presented regression models, I have also implemented another approach to validate the usefulness of the AEW features for fault detection purposes independent from the regression models. Here, a [one-class support vector machine \(OCSVM\)](#) is used as a standard novelty detection technique to detect faults as outliers in the joint space of features and system metrics represented by  $F_{\text{Combined}}$  presented in [Equation \(13.12\)](#) on page 156. This additional approach is intended to provide a second view on the ability to detect performance degradations based on the encoded event communication of components.

### 14.3 EVALUATION

To validate the proposed fault detection methods for performance degradations, I have performed two evaluations. After an initial evaluation on the dataset presented in [Chapter 12](#) on page 141, a detailed analysis based on a mock component is presented as a further validation step.

#### 14.3.1 Results on the ToBi dataset

For the initial evaluation of the fault detection approach I have used the *ToBi* dataset with its different components (cf. [Chapter 12](#)). Different prediction models as well as feature variations have been tested to find out how the combinations perform. Additionally, I have added a baseline method to compare against. This baseline is a threshold based on the Mahalanobis distance of the empirical covariance matrix for the vector of system metrics, independent of the AEW features. The threshold can be learned using the same procedure as described in step 3 of the training procedure presented before. However, it is directly performed on the real system metrics of a component instead of the predicted ones.

For each approach I have trained a fault detection model for all components of the *ToBi* system that communicate via [RSB](#). The models were trained on the 10 fault-free trials of the dataset. Afterwards, for all healthy trials with induced faults, each model was used to predict the current state of its component at each time step of the data (1 Hz). To compute aggregated scores, the temporal order of the time series has been ignored and the task was treated as a binary classification problem by combining all individual time series of predictions (per trial) into a single set of points. As all approaches offer a numer-

APPROACH	AUC
AEW - GBR	0.624
AEW - KR-FS	0.634
AEW - OCSVM	0.614
Combined - GBR	0.631
Combined - KR-FS	0.634
Combined - OCSVM	0.624
threshold	0.582

Table 14.1: Detection results for different fault detection approaches on the *ToBi* dataset as the mean value across all components.

ical value that is compared to a threshold for the final class decision,<sup>1</sup> I used the [ROC AUC](#) metric as an established and resilient metric for comparing the performance of the different approaches. Training and evaluation have been performed 5 times and results were averaged to account for randomness in the approaches.

A first assessment of how well the different approaches perform can be found in [Table 14.1](#). It shows that all approaches based on AEW features outperform the basic threshold model. However, the difference to the baseline is low. Therefore, a more detailed analysis for this reason is required. One aspect is how much the features help in the detection task for the different system components (cf. [Appendix E.1](#) on page 237 for a detailed list of components and their purposes and behaviors). [Figure 14.2](#) on the next page shows the scores for the most promising approaches per component. It becomes visible that the additional information from the feature vectors only help for some components while for others the threshold model or even a random classification (a score of 0.5) cannot be outperformed. Especially in case of the `rsbnavigation` and the `scenersbam` components, scores below 0.5 are reported. This suggests that these components show no stable resource utilization behavior and thus conclusions drawn from the training are invalid on the test data. The components where scores of the regression model based approach are better than a random classification and than the baseline are `armcontrol`, `objectbuilder`, `objectrecognition`, and `texttospeech`. Looking at the behavior of these components it becomes visible that these are components that show variability and state or input dependence in the way they utilize [resources](#). `armcontrol`, `objectrecognition`, and `texttospeech` implement an on-demand processing based on [RPC](#) method calls. Therefore, they usually do not cause any load on the system apart from the times they were called. `objectbuilder` is one of the few other components in the system that, despite exhibiting a constant processing scheme, does not constantly saturate a single [CPU](#). Therefore, in these cases, the approaches based on the feature vectors can make use of

<sup>1</sup> For the OCSVM the distance from the separating hyperplane can be used.

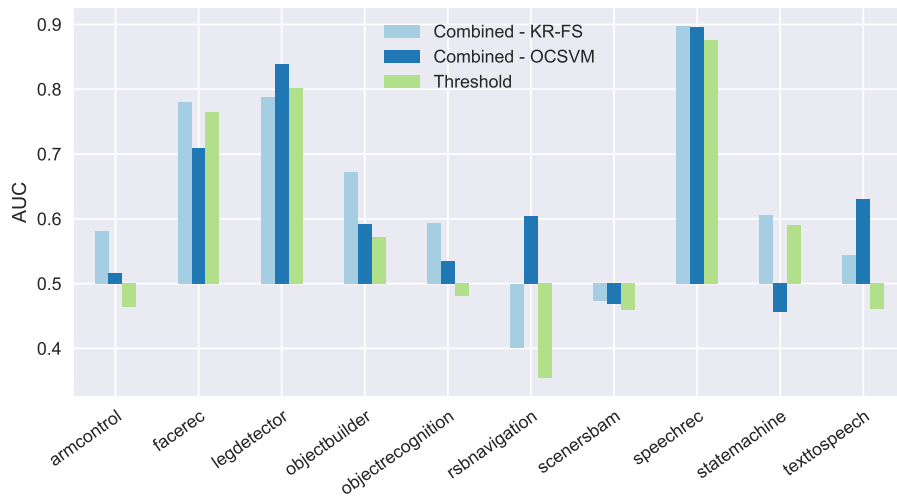


Figure 14.2: Detection scores of the *Combined - KR-FS* approach compared to baseline *Threshold* per component. The bars wrap at 0.5, which represents a random decision.

the encoded state information, whereas the threshold model can only learn a fixed threshold independent of the current processing state of the component. This basic threshold is sufficient for an accurate classification for components that show a constant load across the whole execution time, but not for the aforementioned components with more variability. Additionally, the OCSVM is able to achieve superior results for the *rsbnavigation* and *texttospeech* components. These are components that are only affected by the *spreadLatency* bug, which results in the delayed transmission of events. The OCSVM can handle bugs that manifest primarily in the events better than the approaches based on regression models, because the events directly contribute to the features on which the classification is performed and are not reduced to predicted system metrics before the classification.

Another perspective is to look at the different performance bugs contained in the dataset and how their visible performance degradations can be detected. For this analysis, I have used the slice structure of the dataset explained in [Section 12.3](#) on page 145. Each slice contains a single performance bug and each performance bug can affect one or more components in parallel.<sup>2</sup> Therefore, one way to measure the detection performance for individual performance degradations is to cut out the respective slices from the binary result time series for the affected components and to compute the metrics on these fragments. This procedure ignores most parts of the execution data of each component in which no performance degradation is expected. Additionally, in each included slice for a single performance bug it

<sup>2</sup> For instance, the shift of the clock on one of the two hosts affects all components operating on that host or the temporary interruption of the *Spread* daemon affects all components (cf. [Section 12.2](#) and [Appendix E.2](#) on page 144 and on page 238).

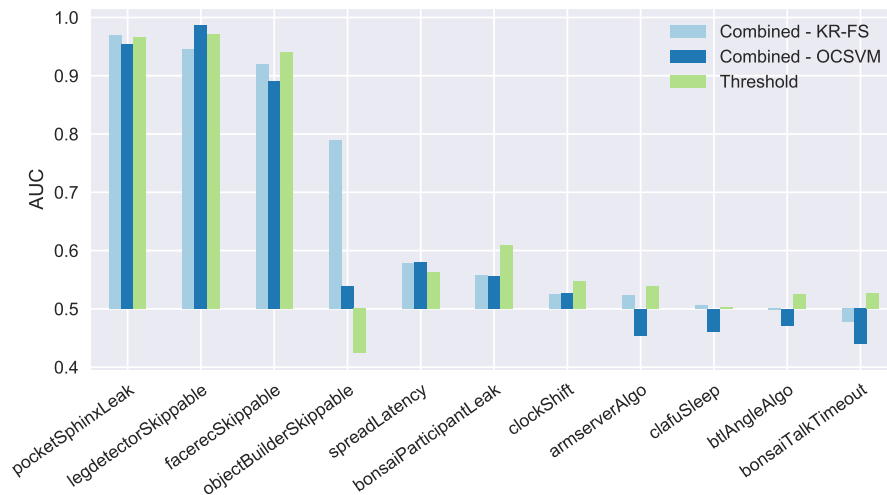


Figure 14.3: Detection scores of the *Combined - KR-FS* approach compared to the baseline *Threshold* per performance bug in the dataset.

is much more likely that at each point in time the respective performance degradation is visible. Thus, the analyzed detection problem in this kind of evaluation is different and therefore the resulting absolute scores are not comparable with the previous evaluation results. Figure 14.3 depicts the results of this evaluation. It is visible that most performance bugs that directly influence one of the system metrics (leaks and skippable computations) are easily detectable, even by the baseline approach. On the other hand, the remaining performance bugs contained in the dataset with low detection rates have only indirect and limited influence on *system resources*. Instead, they often affect timing of events or of the scenario progression. This information is not covered in the features and therefore expectedly hard to detect. Important exceptions from this rule are the `objectBuilderSkippable` and the `armserverAlgo` performance bugs. `objectBuilderSkippable` affects the previously mentioned `objectbuilder` component with its high variability in the resource utilization. Here, the available state information from the encoded events pays off and the proposed fault detection method achieves much higher scores. On the other hand, `armserverAlgo` represents a performance bug that could be detectable through unexpectedly long peaks of resource utilization of the component. However, because of the design of the dataset recording it is possible that inside a slice a performance bug is triggered, but the affected component never exhibits the required behavior to result in a visible performance degradation. In the scenario, the affected arm controller is used only for a few seconds in each trial and therefore the chance that the required component activity (motion planing in this case) and the respective slice appear together is low.

Thus – in retrospect – it becomes visible that the dataset, which has been recorded before developing the fault detection approach, is in some parts a severe challenge for the detection algorithms, whereas



other parts of the data are already well covered by the baseline approach. Another factor that affect the results is the effective communication graph of the components in the system. As explained in the previous chapter, prediction models trained only on the targets system do not generalize to unseen inputs to a component. Once an upstream component is in a faulty state during execution, its outputs might change and thus downstream components receive unexpected inputs. The fault detection models for these downstream components will likely report a fault in these cases, which will be interpreted as a false positive. These effects further reduce the achievable scores. Still, the evaluation shows that the developed method can correctly detect performance degradations that are visible in the resource utilization and the `objectbuilder` example shows that the regression model based method provides real benefit in case a component exhibits high variability and input dependence in its processing. In this case, also the generic OCSVM is outperformed.

#### 14.3.2 *Influence of component behavior*

To provide further hints whether the approach is beneficial in case of components with high variability, I have performed a second evaluation using the constant-complex component introduced in [Section 13.3.2](#) on page 164. The variability of resource utilization of this component can be influenced by the number of threads it is using to process incoming work. With more threads, less CPU saturation appears and the behavior becomes more variable (see [Figure 13.7](#) on page 166 for a visualization of the mock utilization behavior). Thus, this component is a good candidate to analyze the performance of the fault detection method for different levels of variability. I have added a bug to the component that results in more work being performed by the mock per input event than necessary for a few seconds before switching back to a normal state for some time. The evaluation has reused the input events from the fault-free trials of the *ToBi* dataset to simulate realistic inputs. The average ROC AUC scores for the approaches after repeating the model training and classification 5 times are depicted in [Figure 14.4](#) on the following page.

As expectable, a threshold is unable to detect performance degradations in the condition with one thread. The component already uses the complete range of possible CPU utilization values for its normal processing and any threshold either reports false positives most of the time or nothing at all. With the increasing number of threads, the threshold method gains more chances to correctly report faults for high CPU values. As such high values are used less often during normal operation of the component, the false positive rate decreases. Yet, the achievable scores are low. In contrast, most methods based on the prediction models are able to achieve much better scores apart

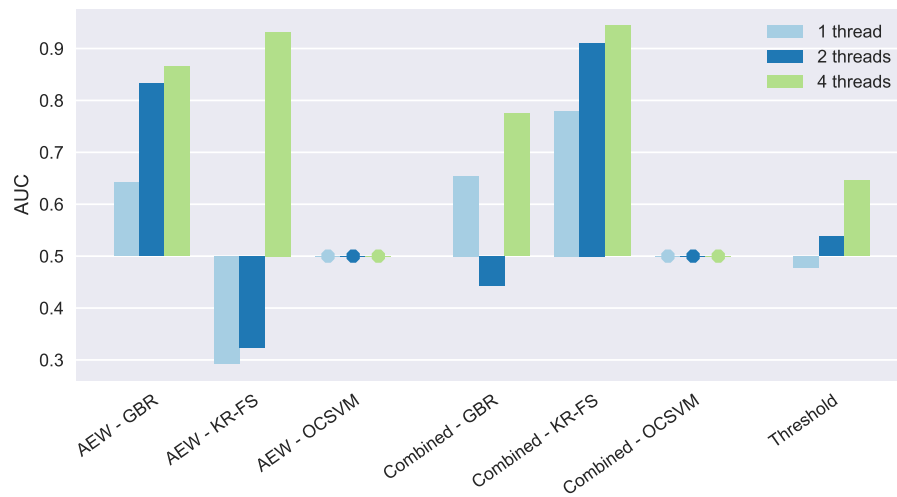


Figure 14.4: Fault detection ROC AUC scores on the different thread configurations of the constant-complex component.

from some combinations that seem to exhibit overfitting. As deviations from the expected CPU utilization can only be detected in case no saturation exists and these times are infrequent for the configuration with one thread, the scores are lower compared to configurations with more variability. Thus, the evaluation shows that this model of detecting performance degradations works best in these cases with high variability without saturation. Even with saturation, the approach can successfully detect performance degradations at the times where no CPU saturation would be expected. The most stable models for this experiment are *AEW - GBR* and *Combined - KR-FS*. Further analyses will be required to understand why the other combinations of features and learners do not generalize. Finally, the results show that the OCSVM is not able to capture the variability and is unable to provide useful detection results.

#### 14.4 SUMMARY

The presented fault detection approach enables systems to autonomously detect and handle performance degradations at runtime. With appropriate strategies that react on the detection results, the [dependability](#) of the systems can thus be increased despite the existence of performance bugs in the implementation of the constituting system components. Moreover, also the system developers will gain better insights into their systems and detect performance degradations provide more detailed information where to start debugging. This is a novel perspective on fault detection in the robotics domain, where most approaches are specialized on hardware faults or external disturbances to the system. Apart from these immediate benefits to-

wards more stable systems, I could also demonstrate the applicability of the prediction method presented in the previous chapter.

The evaluation of the fault detection method has shown that the achievable detection rates depend on the types of bugs and the characteristics of the components. Expectedly, only performance bug that affect the system metrics were detectable with good scores. For other types of performance bugs, other methods have to be applied. For instance, changes to expected event rates would be detectable by invariant learning techniques such as the ones presented in Weber and Wotawa [WW06]. Thus, a complete safety network requires different fault detection methods depending on the types of bugs that are expected or that should be covered. However, for the scope of this thesis, the presented fault detection methods has shown to be an effective tool to handle performance degradations at system runtime.



## Part V

### PERSPECTIVES

The last part of this thesis will summarize the contributions, assess their impact, and outline future opportunities for research.



## CONCLUSION

---

With this thesis, I have presented a set of methods that help to improve the [dependability](#) of robotics and intelligent systems as commonly found in research settings. By exploiting knowledge about the utilization of [system resources](#), the developed methods establish [resource awareness](#) throughout the design and the runtime of systems, which is a perspective on dependability often ignored in research systems. To address the requirements of the development process in this domain, the methods were designed to be applicable with limited effort and in loosely controlled environments. I have taken the predominant architectural style of component-based systems realized in [microservice architectures](#) as the foundation for the developed methods. By hooking into the existing extensions points and [introspection](#) abilities, labor-intensive modifications to the [components](#) can be avoided. Moreover, the presented techniques can be applied individually for different components to enable a gradual integration into production use. In contrast to existing approaches for resource awareness in these systems, the focus of this work on easy applicability, individual components, and its foundation in the [distributed system](#) architecture are a novel perspective on resource awareness, which I have termed *framework-level resource awareness*.

In contrast to others, I understand resource awareness as an overarching property of systems that can only be achieved if it is addressed during all phases of system design and operation. Thus, I have presented methods that target the development process and the autonomous runtime of robotics and intelligent systems. Successfully establishing resource awareness includes that system and component developers are aware of [resources](#), their utilization, and the effects modifications have on these properties. A resource-aware algorithm in a planner or scheduling component of a system cannot work around all implementation errors and oversights introduced into a system during development time, especially if a modification to the planner introduced a [performance bug](#) in its own implementation. Thus, the first set of methods presented in Part III is directed to the system developers. An appropriate visualization through a [dashboard](#) ensures that developers can gain an understanding of the [resource utilization](#) of their systems and a performance testing framework supervises changes to component implementations. Instead of requiring manual verifications or accidental discoveries, the testing framework systematically uncovers unexpected changes to the resource utilization of components and test execution is automated using a [CI](#) setup.

For enabling resource awareness at runtime, in Part IV I have explored the unique perspective of using machine learning methods to predict the resource utilization of components. Many existing approaches for runtime resource awareness either operate on the implementation level or require extensive modeling. The chosen way avoids these efforts and can instead be used as a plugin to existing systems. For instance, the presented [fault detection](#) method based on the learned prediction models is an additional service that can be applied selectively and without component modifications. Generally, the idea to use the [middleware](#) communication as a predictor for the resource utilization is a novel perspective. I am not aware of other approaches following this idea.

The existing microservice architectures with their realization as distributed systems based on a common middleware with introspection abilities have proved to provide enough information for enabling framework-level resource awareness. Without modifications to existing components or infrastructure I was able to realize my methods only through transparent additions to the systems. I am convinced that even without modifications, further methods for enabling resource awareness and improving the dependability in general can be developed and integrated. Future work in this direction is one factor that determines the maturity of robotics and intelligent systems and contributes to their applicability and success. From an architectural point of view, this thesis has shown that a fine-grained separation of systems is an important step towards enabling supervising functions such as the ones presented in this work. In case information are hidden in component implementations, they are not available to framework-level techniques. Especially hidden camera inputs and actuator control outputs have been a challenge in the systems I have been working with. Future system architects should separate the algorithmic processing from the hardware interface using component boundaries to support system analysis and supervision.

With the methods developed in this thesis I have shown that it is feasible to implement resource awareness in robotics systems on the framework level and that these methods provide a benefit for the systems and their developers. Ultimately, I hope that future systems will be implemented using a mixture of framework-level and implementation-level resource awareness methods. As intended, the framework-level methods offer an easy entry into enabling resource awareness, especially at runtime. However, some applications with their constraints and requirements can only be realized successfully if parts of the system also reflect resource awareness on the implementation level. For instance, severe resource constraints as the ones found on planetary rovers can only be met if a resource-aware planner is used. In such cases, framework-level methods can be used to gather the required expectations about resource utilizations of other parts of



the system, especially for legacy applications, while only a fraction of the components will require modifications on the implementation level. Moreover, the coverage of the development process is a unique perspective. Thus, I envision a coexistence of both approaches and both sides of resource awareness have their unique application areas.



OUTLOOK

---

Technical systems and solutions always offer potential for improvement and also this thesis has opened up perspectives for future development and change.

One obvious point for improvement is the application of the developed methods in more systems. This would give further insights into their strengths and limitations and would also validate their generality. With the dominance of ROS in robotics research, a good perspective to attract further users is porting the tools to this ecosystem. From a conceptual point of view, I have tried to limit the requirements of the methods on the underlying system with its [middleware](#) to the common set of features identified in [Section 5.4](#) on page 49. Therefore, I assume that no severe obstacles exist on the way to implementations for other frameworks. A verification is needed though. Technically, the new ecosystem requires comparable daemons to uncover the [resource utilization](#), which needs to be exposed using the middleware. With ARNI [[BHW16](#)] and Drums [[MWV14](#)] suitable bases exist and only limited modifications are required to realize the full metric collection concept. For the developer perspective, the performance testing framework will need additional actions to generate ROS payloads in addition to [RSB](#), and the DSL has to be extended for these additional actions including the related data types. Moreover, the test runner and the analysis tool have to be adapted to support *rosbag* [[ROSb](#)]. For the runtime perspective, apart from adapter code for the new middleware, the biggest difference is the limited availability of metadata on ROS messages compared to [RSB events](#). A different encoding function for constructing feature vectors is necessary that handles the reduced amount of information. As in [RSB](#)-based systems the payload size is usually the strongest source of information, I do not think that the missing remaining information are a severe obstacle, though. Thus, I am confident that the approach can be transported to other ecosystems.

Apart from moving to a different ecosystem, a lot of detail improvements to the developed methods are possible. However, on a broader perspective, larger features could make a real difference. For instance, to provide better support for the developer for resolving detected [performance bugs](#), an integrated development approach closing the loop from detection results to code modifications is an interesting perspective. If the developer automatically gets targeted feedback up to the level of code lines on [performance regression](#) detected while testing or after a detected [performance degradation](#) at runtime, chances will

become higher that these issues are quickly resolved. Thus, a connection of detected issues back to the code level, probably based on the commit history, would be an interesting research topic.

Another perspective is to combine the runtime [fault detection](#) with system simulations. As not all performance bugs can be avoided with testing, detecting them in simulations instead of production runs of the system will further increase the [dependability](#) and reduce the consequences of newly detected issues. Ideally, the simulations would automatically be constructed to systematically search through the state space of the [components](#). Similarly, an automatic generation of performance tests would reduce a lot of the currently necessary effort. Validating the suitability of generated test cases directly opens up the need for a coverage metric for performance aspects.

On a broader scale, it will be interesting to see how the provided methods integrate into more structured approaches to robotics development, where systems are rigorously modeled and code is generated. These approaches will reduce the need for some of the developed methods. However, similar questions regarding the resource utilization and the effect of changes can also be asked on the level of models. It will be interesting to see how existing knowledge can be transferred into model checking rules and simulations on this level.

In any case, many aspects of [resource awareness](#) for robotics systems have not been researched so far or have only been covered on a high level. Therefore, many possibilities for improvement remain to be explored.

Part VI

APPENDIX





## SURVEY: FAILURES IN ROBOTICS AND INTELLIGENT SYSTEMS

---

The following sections represent the structure of the online survey about performance problems in robotics systems, which is discussed in [Chapter 3](#) on page 21. This is a direct export of the survey structure without modifications. Please note that the questions do not use the unified terms introduced in [Chapter 2](#) on page 7.

### A.1 INTRODUCTION

Thank you very much for taking the time to participate in this survey. This survey is part of my PhD project with a focus on exploiting knowledge about computational resource consumption in robotics and intelligent systems, pursued at Bielefeld University. Therefore, in order to participate, you should be involved or have been involved in the development or maintenance of such systems. In case you have worked or are working with mutiple systems in parallel, please provide answers on the combination of all theses systems.

Participating in this survey should not take longer than 15 minutes. The survey consists of several questions and you are free to skip questions in case you do not want to answer them. Moreover, you can go back and forth between the questions you have already answered in order to revise them. All data you enter in this survey will be anonymized.

Johannes Wienke  
jwienke [at] techfak.uni-bielefeld.de

### A.2 MONITORING TOOLS

The first part of this survey addresses how robotics and intelligent systems are monitored at runtime in order to assess their health and understand the ongoing operations. Monitoring includes the ongoing collection of runtime data, the observation of operations as well as the assessment of system health.

A.2.1 *How often do you use the following kinds of tools to monitor the operation of running systems?*

Rate individually for:

- Operating system command line tools e.g. htop, iotop, ps (OS)

- Logfiles (LOG)
- Dashboard views e.g. munin, graphite, nagios (DASH)
- Inter-process communication introspection e.g. middleware logger (IPC)
- Autonomous fault or anomaly detectors (FD)
- Special-purpose visualizations e.g. rviz, image processing debug windows (VIS)
- Remote desktop connections e.g. VNC, rdesktop (RDP)
- Others (OTH)

ANSWER TYPE Fixed choice

- Never (0)
- Rarely (1)
- Sometimes (2)
- Regularly (3)
- Always (4)

A.2.2 *Please name the concrete tools that you use for monitoring running systems.*

Separate different tools with a comma.

ANSWER TYPE longtext (length: 40)

### A.3 DEBUGGING TOOLS

This part of the survey addresses tools that are used in order to debug systems in case a failure has been detected. Debugging is the process of identifying the root cause of an observed abnormal system behavior.

A.3.1 *How often do you use the following tools for debugging?*

Rate individually for:

- Console output e.g. printf, cout (PRNT)
- Logfiles (LOG)
- Debuggers e.g. gdb, pdb (DBG)



- Profilers e.g. kcacheGrind, callgrind (PROF)
- Memory checkers e.g. valgrind (MEMC)
- System call introspection e.g. strace, systemtap (SYSC)
- Inter-process communication introspection e.g. middleware logger (IPC)
- Network analyzers e.g. Wireshark (NWAN)
- Automated testing e.g. unit tests (TEST)
- Simulation (SIM)
- Others (OTH)

ANSWER TYPE Fixed choice

- Never (0)
- Rarely (1)
- Sometimes (2)
- Regularly (3)
- Always (4)

A.3.2 *Please name the concrete tools that you use for debugging.*

Separate different tools with a comma.

ANSWER TYPE longtext (length: 40)

#### A.4 GENERAL FAILURE ASSESSMENT

Please provide information about failures you have observed in the systems you are working with.

A.4.1 *Averaging over the systems you have been working with, what do you think is the mean time between failures for these systems?*

The mean time between failures is the average amount of operation time of a system until a failure occurs.

ANSWER TYPE Fixed choice

- < 0.5 hours (0)
- < 1 hour (1)

- < 6 hours (2)
- < 12 hours (3)
- < 1 week (4)
- > 1 week (5)

A.4.2 *Please indicate how often the following items were the root cause for system failures that you know about.*

Rate individually for:

- Hardware issues (HW)
- System coordination e.g. state machine (COORD)
- Deployment (DEPL)
- Configuration errors e.g. component configuration (CONF)
- Logic errors (LOGIC)
- Threading and synchronization (THRD)
- Wrong error handling code (ERR)
- Resource leaks or starvation e.g. RAM full, CPU overloaded (LEAK)
- Inter-process communication failures e.g. dropped connection, protocol error (COMM)
- Specification error / mismatch e.g. component receives other inputs than specified (SPEC)
- Others (OTH)

ANSWER TYPE Fixed choice

- Never (0)
- Rarely (1)
- Sometimes (2)
- Regularly (3)
- Very often (4)

A.4.3 *Which other classes of root causes for failures did you observe?*

Separate items by comma.

ANSWER TYPE text (length: 24)

#### A.5 RESOURCE-RELATED BUGS

The following questions deal with the consumption of computational resources like CPU, memory, disk, network etc.

A.5.1 *How many of the bugs you have observed or know about had an impact on computational resources, e.g. by consuming more or less of these resources as expected?*

Please approximate the amount with a percentage value of the total number of bugs you can remember. A quick guess is ok here.

ANSWER TYPE integer (length: 10)

#### A.6 IMPACT ON COMPUTATIONAL RESOURCES

The following questions deal with the consumption of computational resources like CPU, memory, disk, network etc.

A.6.1 *Please indicate how often the following computational resources were affected by bugs you have observed.*

A computational resource was affected by a bug in case its consumption was higher or less than expected, e.g. in comparable or non-faulty situations.

Rate individually for:

- CPU (CPU)
- Working memory (MEM)
- Hard disc space (HDD)
- Network bandwidth (NET)
- Number of network connections (CON)
- Number of processes and threads (PROC)
- Number of file descriptors (DESC)

ANSWER TYPE Fixed choice

- Never (0)
- Rarely (1)
- Sometimes (2)

- Regularly (3)
- Very often (4)

A.6.2 *If there are other computational resources that have been affected by bugs, please name these.*

ANSWER TYPE longtext (length: 40)

#### A.7 PERFORMANCE BUGS

The following question specifically addresses performance bugs. A system failure or bug is a performance bug in case it is visible either through degradation in the observed performance of the system (e.g. delayed or very slow reactions) or through an unexpected consumption of computational resources like CPU, memory, disk, network etc.

A.7.1 *Please rate how often the following items were the root causes for performance bugs you have observed.*

Rate individually for:

- Hardware issues (HW)
- System coordination e.g. state machine (COORD)
- Deployment (DEPL)
- Configuration errors e.g. component configuration (CONF)
- Logic errors (LOGIC)
- Threading and synchronization (THRD)
- Wrong error handling code (ERR)
- Unnecessary or skippable computation (SKIP)
- Resource leaks or starvation e.g. RAM full, CPU overloaded (LEAK)
- Inter-process communication failures e.g. dropped connection, protocol error (COMM)
- Specification error / mismatch (SPEC)
- Algorithm choice (ALGO)
- Others (OTH)

ANSWER TYPE Fixed choice

- Never (0)
- Rarely (1)
- Sometimes (2)
- Regularly (3)
- Always (4)

#### A.8 CASE STUDIES

For the following questions, please provide descriptions of any kind of bug that you remember.

A.8.1 *Thinking about the systems you have worked with so far, is there a bug that you remember which happened several times or which is representative for a class of comparable bugs?*

ANSWER TYPE Fixed choice

- Yes (Y)
- No (N)

#### A.9 CASE STUDY: REPRESENTATIVE BUG

Please briefly describe the representative bug that you remember.

A.9.1 *How was the representative bug noticed?*

Please explain the observations that were made and how they diverged from the expectations.

ANSWER TYPE longtext (length: 40)

A.9.2 *What was the root cause for the bug?*

Please explain which component(s) of the system failed and in which way.

ANSWER TYPE longtext (length: 40)

A.9.3 *Which steps were necessary to analyze and debug the problem?*

Please include the information sources that had to be observed and the tools that got applied.

ANSWER TYPE longtext (length: 40)

A.9.4 *Which computational resources were affected by the bug?*

Computational resources include CPU, working memory, hard disc space, network bandwidth & connections, number of processes and threads, number of file descriptors etc.

ANSWER TYPE longtext (length: 40)

#### A.10 CASE STUDIES

For the following questions, please describe any kind of bug that you remember.

A.10.1 *Thinking about the systems you have worked with so far, is there a bug that you remember which was particularly interesting for you?*

ANSWER TYPE Fixed choice

- Yes (Y)
- No (N)

#### A.11 CASE STUDY: INTERESTING BUG

Please describe briefly the most interesting bug that you remember from one of the systems you have been working with.

A.11.1 *How was the interesting bug noticed?*

Please explain the observations that were made and how they diverged from the expectations.

ANSWER TYPE longtext (length: 40)

A.11.2 *What was the root cause for the bug?*

Please explain which component(s) of the system failed and in which way.

ANSWER TYPE longtext (length: 40)

A.11.3 *Which steps were necessary to analyze and debug the problem?*

Please include the information sources that had to be observed and the tools that got applied.

ANSWER TYPE longtext (length: 40)

A.11.4 *Which computational resources were affected by the bug?*

Computational resources include CPU, working memory, hard disc space, network bandwidth & connections, number of processes and threads, nubmer of file descriptors etc.

ANSWER TYPE longtext (length: 40)

#### A.12 PERSONAL INFORMATION

As a final step, please provide some information about your experience with robotics and intelligent systems development.

A.12.1 *In which context do you develop robotics or intelligent systems?*

ANSWER TYPE Fixed choice

- Student (excluding PhD students) (STUD)
- Researcher at a university (PhD students, scientific staff) (RES)
- Industry (IND)
- Other (OTHER)

A.12.2 *How many years of experience in robotics and intelligent systems development do you have?*

ANSWER TYPE integer (length: 10)

A.12.3 *How much of your time do you spend on developing in the following domains?*

Please indicate in percent of total development time. Numbers may not sum up to 100.

Rate individually for:

- Hardware (HW)
- Drivers (DRV)

- Functional components (COMP)
- Inter-process communication infrastructure (COMM)
- Software architecture and integration (ARCH)
- Other (ANY)

ANSWER TYPE integer (length: 3) Hint: Percent of development time

#### A.13 FINAL REMARKS

Thank you very much for participating in this survey and thereby supporting my research.

In case you have further questions regarding this survey or the research topic in general, please contact me via email.

Johannes Wienke

jwienke [at] techfak.uni-bielefeld.de



# B

## FAILURE SURVEY RESULTS

---

### B.1 USED MONITORING TOOLS

The following table presents the results for question [A.2.2](#). The free text answers have been grouped into categories (caption lines in the table). For each answer that included at least one item belonging to a category, the counter of each category was incremented. Hence, the counts represent the number of answers that mentioned a category at least once. Additionally, for each category, representative entries have been counted the same way. Some of the answers include uncommon or special-purpose tools or techniques. These have not been counted individually and, hence, are visible only in the category counts.

Notes regarding entries:

- “Middleware tools” represents entries that are specific to the middleware-related aspects of an ecosystem. For instance, ROS\_ -DEBUG has not been counted here. Instead, this belongs to the “Manual log reading” category.

TOOL	ANSWER COUNT
<b>VISUALIZATION</b>	<b>27</b>
rviz	22
gnuplot	2
matplotlib	1
<b>MIDDLEWARE TOOLS</b>	<b>23</b>
ROS command line	14
rqt	5
RSB	4
<b>BASIC OS TOOLS</b>	<b>22</b>
htop	12
ps	7
top	7
acpi	1
du	1
free	1
lsof	1
procman (gnome)	1
pstree	1
screen	1
tmux	1

*Continued on next page*

TOOL	ANSWER COUNT
MANUAL LOG READING	13
REMOTE ACCESS	9
ssh	5
putty	1
rdesktop	1
vnc	1
CUSTOM MISSION-SPECIFIC	4
GENERIC NETWORK	2
netstat	1
tcpdump	1
wireshark	1
HARDWARE SIGNALS	1

## B.2 USED DEBUGGING TOOLS

The following table presents the results for question [A.3.2](#). The free text answers have been grouped into categories (highlighted lines in the table). For each answer that included at least one item belonging to a category, the counter of each category was incremented. Hence, the counts represent the number of answers that mentioned a category at least once. Additionally, for each category, representative entries have been counted the same way. Some of the answers include uncommon or special-purpose tools or techniques. These have not been counted individually and, hence, are visible only in the category counts.

Notes regarding entries:

- “Middleware tools” represents entries that are specific to the middleware-related aspects of an ecosystem. For instance, ROS\_ -DEBUG has not been counted here. Instead, this belongs to the “Manual log reading” category.

TOOL	ANSWER COUNT
DEBUGGERS	19
gdb	17
pdb	3
VS debugger	2
ddd	1
jdb	1

*Continued on next page*

TOOL	ANSWER COUNT
<b>RUNTIME INTROPSECTION</b>	<b>13</b>
valgrind	12
callgrind	2
kcachegrind	1
strace	1
<b>GENERIC</b>	<b>15</b>
printf, cout, etc.	14
logfiles	4
git	1
<b>MIDDLEWARE</b>	<b>12</b>
ROS command line	5
RQT	2
RSB	2
<b>SIMULATION &amp; VISUALIZATION</b>	<b>7</b>
gazebo	4
rviz	1
Vortex	1
stage	1
<b>FUNCTIONAL TESTING</b>	<b>6</b>
gtest	2
junit	2
cppunit	1
rostest	1
<b>IDES</b>	<b>4</b>
Qt Creator	2
KDevelop	1
LabVIEW	1
Matlab	1
Visual Studio	1
<b>GENERIC NETWORK</b>	<b>2</b>
wireshark	2
tcpdump	1
<b>DYNAMIC ANALYSIS</b>	<b>1</b>
Daikon	1

### B.3 SUMMARIZATION OF FREE FORM BUG ORIGINS

The following table presents all answers to question [A.4.3](#). Individual answers have been split into distinct aspects. These aspects have been assigned either to an existing answer category from question [A.4.2](#) or to new categories.

ANSWER	CATEGORY	
	EXISTING	NEW
unknown driver init problems (start a driver, and works only after second trial)		Driver & OS
environment noise (lighting condition variation, sound condition in speech recognition) hard to adapt to every possible variation		Environment
Insufficient Component Specifications	Specification	
Changed maps/environments		Environment
lossy WiFi connections	Hardware	
unreliable hardware	Hardware	
in Field robotics, the environment is the first enemy...		Environment
Environment changes		Environment
sensor failures	Hardware	
unprofessional users		Environment
Operation System / Firmware failure		Driver & OS
network too slow	Hardware	
Loose wires	Hardware	
other researchers changing the robot configuration		Config mgmt
coding bugs	Logic	
algorithm limitations		Environment
sensor limitations		Hardware lim
perception limitations		Environment
wrong usage		Environment
Failures in RT OS timing guarantees		Driver & OS

#### B.4 SUMMARIZATION OF OTHER RESOURCES AFFECTED BY BUGS

The following table presents the free text results of question [A.6.2](#). Answers have been split into distinct aspects and these aspects have been assigned either to one of the existing categories from question [A.6.1](#) or – if these did not match – new categories have been created to capture the answers. Parts of answers that did not represent [system resources](#), which have a [resource capacity](#) that can be utilized, have been ignored. These are marked as strikethrough text.

ANSWER	RESOURCE	
	EXISTING	NEW
USB bandwidth and or stability		IO bandwidth
locks on files/devices/resources permissions file system integrity	File descriptors	
interprocess communication queues, e.g. queue overflow		IPC
Files (devices) left open. <del>Wrong operation in GPU leads to restart.</del>	File descriptors	
Memory leak – not sure why or where	Memory	

## B.5 REPRESENTATIVE BUGS

The following subsections present answers to the questions for representative bugs (A.9). For the analysis, answers have been tagged for various aspects and types of bugs being mentioned in them. Raw submission texts have been reformatted to match the document and typographical and grammatical errors have been corrected.

### B.5.1 *Representativ bug 8*

OBSERVATION computer system unresponsive

CAUSE memory leak

DEBUGGING

- find faulty process
- analyze memory usage (valgrind/gdb)
- repair code

AFFECTED RESOURCES main memory

TAGS basic programming issue; performance bug

B.5.2 *Representativ bug 10*

OBSERVATION System got stuck in infinite loop.

CAUSE Unexpected infinite loop in the behaviour (state machine). Noise in the data caused the system to infinitely switch between two states.

DEBUGGING

1. Detection of which states were affected.
2. Detection of the responsible subsystem(s).
3. Detection of the responsible functions.
4. Recording data that caused the problem.
5. Analyzing the data and searching for unexpected situations.
6. Modification of the system in order to handle such situation correctly.

AFFECTED RESOURCES CPU

TAGS coordination; environment-related

B.5.3 *Representativ bug 14*

OBSERVATION high latency in spread communication

CAUSE wrong spread configuration/wrong deployment of components

DEBUGGING trial & error: reconfiguration, stopping and starting components, monitoring of latency via rsb-tools

AFFECTED RESOURCES network-latency

TAGS communication; performance bug

B.5.4 *Representativ bug 21*

OBSERVATION Incorrect response of the overall system according to requested task request. System thinks it did not grasp an object although it did and restarts grasping operation or cancels the task due to the missing object in hand.

**CAUSE** State machine design and/or logic error and/or untriggered event due to sensor not triggering as expected (hardware) or too much noise (environment noise). The root cause is often a case not being handled correctly in a big system with a lot of sensors and possible case.

**DEBUGGING** event logger analysis over XCF XML data, unit test of single sensor output to see noise level or false positives.

**AFFECTED RESOURCES** Hardware (noise in the sensor)

**TAGS** coordination; environment-related

#### B.5.5 *Representativ bug 26*

**OBSERVATION** Segfault

**CAUSE** Segfault

**DEBUGGING** gdb

**AFFECTED RESOURCES**

**TAGS** basic programming issue

#### B.5.6 *Representativ bug 30*

**OBSERVATION** Unexpected overall behavior.

**CAUSE** Wrong logic in the abstract level.

**DEBUGGING** Run simulation in the abstract layer.

**AFFECTED RESOURCES** None.

**TAGS** coordination

#### B.5.7 *Representativ bug 41*

**OBSERVATION** Failure to observe expected high-level output. More specifically, a map that was being built was lacking data.

**CAUSE** Congested wireless network connection. The amount of data could not be transmitted within the expected time frame.

**DEBUGGING** Logging of signals between modules on the deployed system to verify data was being produced and transmitted correctly, and logging of data received.

**AFFECTED RESOURCES** Network connection

**TAGS** communication; timing

#### B.5.8 *Representative bug 42*

**OBSERVATION** Because of timing mismatch the planning system was working with outdated data.

**CAUSE** Non-event based data transfer.

**DEBUGGING** Going through multiple log files in parallel to find the data that was transmitted in comparison to the data that was used in the computation.

**AFFECTED RESOURCES** Non. Mostly mismatch between specification and performed actions.

**TAGS** coordination; timing

#### B.5.9 *Representative bug 46*

**OBSERVATION** Navigation did not work correctly

**CAUSE** Algorithmic errors

**DEBUGGING** Dig in and verify steps in the algorithm

**AFFECTED RESOURCES**

**TAGS**

#### B.5.10 *Representative bug 60*

**OBSERVATION** delays in robots command execution

**CAUSE** supervision and management part of the framework

**DEBUGGING** benchmarking, profiling

**AFFECTED RESOURCES**



TAGS performance bug

#### B.5.11 *Representativ bug 69*

OBSERVATION memory leak

CAUSE resource management, dangling pointers

DEBUGGING check, object/resource timeline, usually start with resources that are created often and handed over regularly and therefore might have unclear ownership

AFFECTED RESOURCES memory, CPU

TAGS basic programming issue; performance bug

#### B.5.12 *Representativ bug 70*

OBSERVATION constantly increasing memory consumption

CAUSE Memory leaks

DEBUGGING Running the code in offline mode with externally provided inputs and observing the memory consumption pattern. Tools like valgrind or system process monitor helps to discover the problem

AFFECTED RESOURCES Working memory

TAGS basic programming issue; performance bug

#### B.5.13 *Representativ bug 76*

OBSERVATION Visually in system operation. In one case, elements within a graphical display were misdrawn. In another, command codes were misinterpreted, resulting in incorrect system operation.

CAUSE Variable type mismatch e.g. integer vs. unsigned integer – such as when a number intended to be a signed integer is interpreted as an unsigned integer by another subsystem.

DEBUGGING Debugger using single step and memory access.

AFFECTED RESOURCES None

TAGS basic programming issue; performance bug

B.5.14 *Representative bug 81*

OBSERVATION segfault

CAUSE C++ pointers

DEBUGGING gdb, valgrind

AFFECTED RESOURCES none

TAGS basic programming issue

B.5.15 *Representative bug 96*

OBSERVATION segmentation fault

CAUSE logical errors, bad memory management

DEBUGGING using debuggers, looking and studying code

AFFECTED RESOURCES working memory, number of process and threads

TAGS basic programming issue

B.5.16 *Representative bug 128*

OBSERVATION Robot software is not working / partially working (e.g. recognizing and grasping an object)

CAUSE Wrong configuration and/or API changes that hasn't been changes in all components (Problem with scripting languages like python)

DEBUGGING

- identify error message and component via log files / console output
- Think about what could have caused the problem (look into source code, git/svn commit messages/diffs)
- try to fix it directly or talk with other developers in case of bigger changes / out of my responsibility

AFFECTED RESOURCES none

## TAGS

B.5.17 *Representativ bug 135*

OBSERVATION middleware communication stopped / was only available within small subsets of components

CAUSE unknown

## DEBUGGING

## AFFECTED RESOURCES

TAGS not/accidentally solved; communication

B.5.18 *Representativ bug 136*

## OBSERVATION

1. Application/process hang.
2. 100% core usage on idle
3. Unbalanced load between cores (Monolithic code).

## CAUSE

1. Loose wire/couple (mostly USB)
2. Active wait  

```
while(1) while(!flag); process(); flag = 0;
```
3. A bad design. No threads were used, but time measurements to switch between tasks.

## DEBUGGING

1. Check everything, realize that the file-device is open but device is no longer present or has different pointer or has reseted
- 2/3. Check every code file. People use to make old-style structured programming when using C/C++

when you notice the performance go brick, check CPU/memory usage with OS tools and notice one process is using everything but is idle.

AFFECTED RESOURCES Mostly CPU

TAGS basic programming issue; performance bug

B.5.19 *Representative bug 156*

**OBSERVATION** Difficult to reproduce, random segmentation faults

**CAUSE** 90% of the time it has been either accessing unallocated memory (off-by-one errors) or threading issues

**DEBUGGING** When working with a system with many processes, threads, inter-process communications, etc., the standard tools (gdb, valgrind) are often not that helpful. If they can't immediately point me to the error, I'll often resort to print statement debugging.

**AFFECTED RESOURCES** Memory leaks, CPU usage

**TAGS** basic programming issue

B.5.20 *Representative bug 190*

**OBSERVATION** unforeseen system behavior, decreased system performance

**CAUSE** misconfiguration of middleware

**DEBUGGING**

- monitoring middleware configuration of concerned components
- checking log-files
- sometimes debug print-outs

**AFFECTED RESOURCES** CPU, network load

**TAGS** communication; performance bug

B.5.21 *Representative bug 191*

**OBSERVATION** Software controlling the robot crashed immediately after started in robot or robot stop to move when has to perform certain operation

**CAUSE** The error was caused by not checking range of allocated memory in some object's constructor, we used `sprintf` instead of `snprintf`

**DEBUGGING**

- gdb – did not find anything
- valgrind – did not find anything

Both tools were run on PC, where the error did not occur, but we did not use them on the robot's pc. The bug was found accidentally.

**AFFECTED RESOURCES** access to non-allocated memory lead immediately to crash of program.

**TAGS** basic programming issue; not/accidentally solved

## B.6 INTERESTING BUGS

The following subsections present answers to the questions for interesting bugs (A.11). Answers have been processed the same way as for [Appendix B.5](#) on page 211.

### B.6.1 *Interesting bug 5*

**OBSERVATION** There are too many to remember. A recent one got noticed by surprisingly high latency in a multithreaded processing and visualization pipeline.

**CAUSE** Sync to vblank was enabled on a system and due to a possible bug in Qt multiple GL widgets contributed to the update frequency. The maximum display update frequency dropped below 30 Hz.

**DEBUGGING** Compare systems and analyze timing inside the application. Google the problem.

**AFFECTED RESOURCES** None

**TAGS** driver & OS

### B.6.2 *Interesting bug 21*

**OBSERVATION** On an arm and hand system, with hand and arm running on separate computers linked via an Ethernet bus, timestamped data got desynchronized. This was noticed on the internal proprioception when fingers moved on the display and the arm did not although both moved in physical world.

**CAUSE** NTP not setup correctly. University had a specific NTP setting requirement that was not set on some computers. Could actually never synchronize.

**DEBUGGING** Looking at timestamps in the messages over rosbag or rostopic tools. Analysing system clock drift with command line tools.

**AFFECTED RESOURCES** working memory and CPU would be used more due to more interpolation/extrapolation computation between unsynced data streams.

**TAGS** configuration

### B.6.3 *Interesting bug 32*

**OBSERVATION** PCL segfaulted on non-Debian/Ubuntu machines when trying to compute a convex hull.

**CAUSE** The code was written to support Debian's libqhull, ignoring the fact that Debian decided to deviate from the upstream module in one compile flag that changed a symbol in the library from struct to struct\*. That way all non-Debian ports of libqhull failed to work with PCL, and instead segfaulted while trying to access the pointer.

**DEBUGGING**

- minimal example
- printf within the PCL code
- printf within an overlaid version of libqhull
- gdb
- Debian package build description for libqhull
- upstream libqhull package
- 12 hours of continuous debugging.

**AFFECTED RESOURCES** Well, segfault, the entire module stopped working. So basically everything was affected to some degree..

**TAGS** driver & OS; basic programming issue

### B.6.4 *Interesting bug 46*

**OBSERVATION** The robot kept asking someones name.

**CAUSE** Background noise in the microphone

**DEBUGGING** The bug was obvious: no limit on the amount of questions asked. Simply drawing/viewing the state machine made this very obvious.

**AFFECTED RESOURCES**

**TAGS** coordination; environment-related

#### B.6.5 *Interesting bug 60*

**OBSERVATION** signal processing in component chain gave different results after several months

**CAUSE** unknown

**DEBUGGING**

**AFFECTED RESOURCES**

**TAGS** not/accidentally solved

#### B.6.6 *Interesting bug 69*

**OBSERVATION** segfault

**CAUSE** timing and location of allocated memory

**DEBUGGING** memory dumps...many many memory dumps

**AFFECTED RESOURCES** it did not affect resources constantly, but system stability in general; maybe CPU and memory

**TAGS** basic programming issue

#### B.6.7 *Interesting bug 76*

**OBSERVATION** While operating, a robot system normally capable of autonomous obstacle avoidance would unexpectedly drop communication with its wireless base station and drive erratically with high probability of collision.

**CAUSE** The main process was started in a Linux terminal and launched a thread that passed wheel velocity information from the main process to the robot controller. When the terminal was closed or otherwise lost, the main process was terminated but the thread continued to run, supplying old velocities to the robot controller.

DEBUGGING top, debugger, thought experiments

AFFECTED RESOURCES None

TAGS coordination

B.6.8 *Interesting bug 83*

OBSERVATION Random segfaults throughout system execution.

CAUSE Bad memory allocation: malloc for sizeof(type) rather than sizeof(type\*).

DEBUGGING Backtrace with gdb, profiling with valgrind, eventual serendipity to realize the missing \* in the code.

AFFECTED RESOURCES Memory

TAGS basic programming issue

B.6.9 *Interesting bug 133*

OBSERVATION memory mismatch, random crashes

CAUSE different components using different boost versions

DEBUGGING debugger, printf. Finally solved after hint from colleague

AFFECTED RESOURCES

TAGS basic programming issue

B.6.10 *Interesting bug 149*

OBSERVATION Erratic behaviour of logic

CAUSE Error in mathematical modeling

DEBUGGING Unit tests

AFFECTED RESOURCES None

TAGS



B.6.11 *Interesting bug 150*

**OBSERVATION** An algorithm was implemented in both C++ and MATLAB exactly the same way. However, only the MATLAB implementation was working correctly.

**CAUSE** Difference in storing the float point variables in MATLAB and C++. MATLAB rounded the numbers, however, C++ cut them.

**DEBUGGING** Step by step tracing and debugging, and watching variables. Then, comparing with each other.

**AFFECTED RESOURCES** Working memory

**TAGS** basic programming issue

B.6.12 *Interesting bug 153*

**OBSERVATION** Control Program crash after a consistent length of time

**CAUSE** Presumably memory leak. Never knew for sure.

**DEBUGGING** Not sure

**AFFECTED RESOURCES** Not sure

**TAGS** basic programming issue; performance bug

B.6.13 *Interesting bug 156*

**OBSERVATION** Visualization window crashing 100% of the time I open it. Running the program inside of gdb resulted in the program successfully running 100% of the time.

**CAUSE** ??? Likely something internal to closed-source graphics drivers interacting with OpenGL/OGRE

**DEBUGGING** Was able to eventually generate a backtrace that pointed to graphics drivers.

**AFFECTED RESOURCES** CPU/Memory/GPU were all affected because I had to run the program inside of gdb

**TAGS** driver & OS

B.6.14 *Interesting bug 162*

OBSERVATION bad localization of a mobile robot in outdoor campus environment. Jump of the estimation

CAUSE Bad wheel odometry reading.

DEBUGGING Analyze log file

AFFECTED RESOURCES None. Loss of performance due to incorrect position tracking

## B.7 COLLECTED SYSTEM METRICS

B.7.1 *Host system metrics*B.7.1.1 *Memory**total (B)*

The total amount of working memory currently available in the system.

*used (B)*

The currently used amount of memory, including caches etc.

*usable (B)*

The amount of memory that can be used by processes without swapping. This can be a different value than total - used because of caches etc.

B.7.1.2 *Swap**total (B)*

The total amount of swap space currently available in the system.

*used (B)*

The amount of swap space currently being used.

B.7.1.3 *CPU*

CPU utilization is expressed using counters directly resembling the entries of the `proc` file system. These counters represent the total amount of time a CPU spent for one of the declared aspects since boot (or overflow of the counter). These counters are measured in *jiffies*, which is the length of one clock tick of the Linux software clock [MT]. The number of clock ticks per second is a kernel configuration parameter that varies between kernel versions and user choices [MT]. Therefore, the CPU measurements contain the length of one jiffy as a data item to compute back these values to real units:

*jiffy length ( $\mu$ s)*

The length of one software clock tick.

For each (virtual) CPU core of a host, the following counters are reported. Explanations are based on [Proc17; Per16]

*total (jiffies)*

The total time this CPU has spent in any mode, including idle state.

*idle (jiffies)*

The time this CPU has been idling.

*user (jiffies)*

The time spent in user mode. These are executions in user space, excluding other special calculations.

*user\_low (jiffies)*

Time spent in user mode with lower processing priority (nice).

*system (jiffies)*

Time spent for processing system calls.

*iowait (jiffies)*

Time spent waiting for [input/output \(I/O\)](#) operations to complete. This is usually an unreliable value because other operations are scheduled while waiting [[Proc17](#)].

*irq (jiffies)*

Time spent servicing hardware interrupts.

*softirq (jiffies)*

Time spent servicing softirqs, which are software-defined interrupts to handle important processing, for instance, required to fulfill real-time guarantees [[Cor17](#)].

*steal (jiffies)*

Time spent serving other operating systems in case the current system is running in a virtualized context.

*guest (jiffies)*

Time spent serving a virtualized guest operating system.

Ideally, *idle* + all other detailed fields sum up to the *total* value. However, no guarantee for this exists and new kernel versions might introduce new detailed counters. If they are not represented by the collection daemon, this calculation will fail.

In addition to these per-core [system metrics](#), the Linux load is measured, which is “the number of processes in the system run queue averaged over various periods of time” [[Gla16](#)]:

*load 1 (number)*

System run queue length average within the last minute.

*load 5 (number)*

System run queue length average within the last 5 minutes.

*load 15 (number)*

System run queue length average within the last 15 minutes.

B.7.1.4 *Disk*

For each file system partition, the following metrics are reported:

*space total (B)*

The total size of the partition.

*space used (B)*

The currently used space on the partition.

Additionally, throughput metrics are available for each (physical) block device. Due to implementation details of the kernel, these devices are not always mutually exclusive. For instance, in case *LVM* [*LVM*] or disk encryption using LUKS [*Crypt*] are used, the Linux kernel device mapper places virtual devices above the actual physical hardware so that *virtual resources* are created with system metrics coupled to the originating physical block devices. For each of these devices, the following metrics are collected.

*read/write count (number)*

The number of read/write operations performed on this device so far.

*read/write bytes (B)*

The number of bytes read from / written to this device so far.

*read/write time (ms)*

The time this device spent reading/writing data so far.

#### B.7.1.5 Network

For each network interface the following metrics are collected:

*bytes sent/received (B)*

The number of bytes this interface has sent/received so far.

*packets sent/received (packets)*

The number of network packets this interface has sent/received so far.

*send/receive errors (number)*

The number of send/receive errors that have happened so far.

*send/receive drops (packets)*

The number of network packets dropped while sending or receiving.

Additionally, the number of currently existing network connections is reported. These numbers are grouped by the address family used by the connection (e.g. IPv4/6, Unix domain sockets) and the used protocol (e.g. *TCP*, *UDP*). For all combinations of the two aforementioned axes, the current number of connections in different states is reported.

*num {state} (number)*

Current number of connections in state {state}.

Please refer to the `rst.devices.generic.NetworkState.Network-Connection` data type for the detailed list of states.

#### B.7.1.6 *Users*

The following metrics regarding currently logged in users of the host system are reported:

*users (number)*

Current number of distinct users logged in.

*sessions (number)*

Current number of login sessions opened by all logged in users. This represents the number of TTYs used by all users.

*hosts (number)*

Current number of distinct hosts users are logged in from.

#### B.7.1.7 *Processes*

Finally, the number of currently running processes on the host is tracked:

*processes (number)*

Current number of running processes.

#### B.7.2 *Process metrics*

For the process collection, the following metrics are collected. I will represent the metrics with the pattern `{source}-{metric}`, which I will be using throughout the rest of this work.

##### B.7.2.1 *Source proc/stat*

The `proc/stat` source provides metrics regarding CPU and memory utilization as well as the number of threads.

*proc/stat-utime ( $\mu$ s)*

Amount of time a process has spent in user mode since its start.

*proc/stat-stime ( $\mu$ s)*

Amount of time a process has spent in kernel mode (e.g. system calls) since its start.

*proc/stat-num\_threads (number)*

The number of operating systems threads (tasks) that a process currently has.

*proc/stat-vmem (B)*

The current virtual memory size of a process. This is the “total amount of virtual memory used by the task [(i.e. process or thread)]. It includes all code, data and shared libraries plus pages that have been swapped out and pages that have been mapped but not used” [Top17]. Virtual memory also comprises memory-mapped I/O.

*proc/stat-rss (B)*

The current **RSS** of a process. This is a “subset of the [...] [virtual memory] representing the non-swapped physical memory a task is currently using” [Top17].

As already explained in [Section 2.1.2](#) on page 10, the `utime` and `stime` metric are not updated periodically by the kernel. Instead, updates are made in relation to a process’ activity [Sta15]. Especially for processes with limited processing this leads to the described discretization artifacts in these metrics, especially in `stime`, because most processes spend less time in this state.

**B.7.2.2** *Source proc/io*

The `proc/io` source provides information regarding the I/O bandwidth a process imposes on a host system.

*proc/io-rchar (B)*

Number of bytes a process has caused to be read since its start. “This is simply the sum of bytes which this process passed to `read(2)` and similar system calls. It includes things such as terminal I/O and is unaffected by whether or not actual physical disk I/O was required” [Proc17].

*proc/io-wchar (B)*

Number of bytes a process has written since its start. Similar to `proc/io-rchar`, this is not physical I/O, but just the sum of all bytes passed to certain Linux system calls.

*proc/io-read\_bytes (B)*

By the respective Linux man page, this is described as an “attempt to count the number of bytes which this process really did cause to be fetched from the storage layer” [Proc17]. This is only accurate for block-based file systems and, for instance, for NFS, the state is effectively unknown [see notes in Kleo7]. Therefore, this has to be treated as a rough guess.

*proc/io-write\_bytes (B)*

Similar to `proc/io-read_bytes`, an “attempt to count the number of bytes which this process caused to be sent to the storage layer” [Proc17]. The same restrictions apply.

### B.7.2.3 Source *proc/fd*

This source provides metrics regarding opened files.

#### *proc/fd-open\_fds (number)*

The number of file descriptors a process has currently opened. This includes all kinds of files that are available in Linux including, for instance, sockets, terminal stream, and pipes.

#### *proc/fd-open\_files (number)*

The number of currently opened data files of a process, excluding special types such as sockets and pipes.

#### *proc/fd-open\_connections (number)*

The number of open network connections of a process.





## SURVEY: DASHBOARD EVALUATION

---

The following sections represent the structure of the online survey about the usefulness of [resource utilization dashboards](#) for robotics and intelligent systems, which is discussed in [Section 9.4.2](#) on page [101](#). This is a direct export of the survey structure without modifications.

### C.1 INTRODUCTION

Thank you very much for participating in this survey. The survey aims at validating aspects of the resource dashboards (processes and host) installed at the CSRA system.

Please mind that this specifically excludes the RSB introspection information.

Completing this survey should take approximately 10 minutes. All results will be handled anonymously.

### C.2 GENERAL

C.2.1 *Please rate, how often you consult the monitoring dashboard in different situations?*

Rate

Rate individually for:

- During system startup (START)
- During normal system operation (NORM)
- When testing new features / components (TESTING)
- In case of a system problem (PROBLEM)
- During studies (STUDY)

ANSWER TYPE Fixed choice

- Never (0)
- Once or twice (1)
- Sometimes (2)

- Regularly (3)
- Always (4)

C.2.2 *How much insight do you gain into the consumption and availability of computational resources (like CPU, I/O or memory) when using the dashboard?*

Please rate individually.

Rate individually for:

- the host system (computer) (HOST)
- individual system components (COMP)

ANSWER TYPE Fixed choice

- 0 No insight at all (0)
- 1 (1)
- 2 (2)
- 3 (3)
- 4 Better insights than before and with any other tool (4)

C.2.3 *Do you think you have a better understanding of the use of computational resource in the system as a result of the dashboard?*

ANSWER TYPE Fixed choice

- Yes (Y)
- No (N)

C.2.4 *For the different kinds of computational resources, how much did the dashboard improve your understanding of the consumption of these resources?*

Rate individually for:

- CPU usage (CPU)
- Memory usage (MEM)
- Disk usage and throughput (DISK)
- Network bandwidth / throughput (NET)
- Number of threads per process (THREAD)
- Open file descriptors (FDS)
- Established network connections (CONN)

ANSWER TYPE Fixed choice

- 0 Not at all (0)
- 1 (1)
- 2 (2)
- 3 (3)
- 4 Much deeper understanding (4)

C.2.5 *Please describe briefly, in which situation you find the dashboard most valuable.*

ANSWER TYPE longtext (length: 40)

### C.3 DEBUGGING

The following questions relate to the use of the resource dashboards in situations where issues like bugs or performance degradations have appeared in the system.

C.3.1 *How often are issues that you observe in the system visible in the dashboard?*

ANSWER TYPE Fixed choice

- Never (0)
- Rarely (1)
- Sometimes (2)
- Regularly (3)
- Always (4)

C.3.2 *Does the dashboard help to isolate the origin of bugs?*

ANSWER TYPE Fixed choice

- Yes (Y)
- No (N)

C.3.3 *Did you find bugs through the dashboard that you wouldn't have noticed at all or much later otherwise?*

ANSWER TYPE Fixed choice

- Yes (Y)
- No (N)

C.3.4 *Please briefly describe the bugs that you have found.*

ANSWER TYPE longtext (length: 40)

#### C.4 TOOLS

C.4.1 *Which tools do / did you use apart from the dashboard to understand resource utilization?*

ANSWER TYPE longtext (length: 40)

C.4.2 *Did the dashboard reduce the use of other tools for the purpose of understanding resource utilization?*

ANSWER TYPE Fixed choice

- Yes (Y)
- No (N)

#### C.5 END

C.5.1 *In case you have further comments or ideas regarding the performance dashboard, please indicate them here.*

ANSWER TYPE longtext (length: 40)

#### C.6 FINAL REMARKS

Thank you very much for participating in this survey and supporting my research!

# D

## DASHBOARD SURVEY RESULTS

---

### D.1 FOUND BUGS

Answers to question [C.3.4](#):

*Answer 3*

- huge amounts of threads
- dying java applications due to reallocation of big heap blocks
- lagging due to overloading a machine with too many components

*Answer 4*

- memory leaks
- infinite loops

*Answer 5*

- unusual CPU usage of components and correlation to other processes
- very high thread usage
- too high network connections

*Answer 6*

- threads have not been shut down correctly
- components induced general slowness due to heavy resource usage (network, i/o)

*Answer 7*

Threading problem.

*Answer 9*

Memory leaks and thread limits



## TOBI DATASET DETAILS

---

### E.1 INCLUDED COMPONENTS

The following list provides a short description of the purpose of each [component](#) of the *ToBi* system used in the [fault detection](#) dataset presented in [Chapter 12](#) on page [141](#). Components are indexed by their technical names used during analyses based on this dataset.

- armcontrol** Controls the Katana arm used as a manipulator for the system. It implements the required motion planing tasks and is directly connected to interface of the arm. Processing only happens on request via [RPC](#) methods.
- facerec** Implements a face recognition system. The component is directly connected to the used camera. Processing happens on RPC requests.
- legdetector** Continuously detect legs and leg pairs in laser scanner results provided via [RSB](#).
- objectbuilder** Implements tracking of persons in global coordinates based on the detected legs from the legdetector. Processing is continuous.
- objectrecognition** Realizes an on-demand object recognition system which is triggered via RPC calls. The component is directly connected to the used camera.
- rsbnavigation** Provides an [RSB](#) front end to the navigation stack of the mobile base. Some parts of this component operate only on request, while other parts operate continuously.
- scenersbam** A memory component that persists knowledge about the world. Processing happens on request.
- speechrec** A speech recognition component directly connected to the system microphone. Uses continuous processing.
- spread** The underlying communication daemon of the [RSB middleware](#).
- texttospeech** On-demand speech production directly connected to the system sound output.
- statemachine** The central controlling state machine of the system based on Siepmann and Wachsmuth [[SW11](#)]. Connects to most of the other system components.

## E.2 RELATION OF BUGS TO COMPONENTS

The following list explains which [performance bug](#) included in the *ToBi* dataset (cf. [Section 12.2](#) on page 144) affects which of the system components.

armserverAlgo armcontrol

bonsaiParticipantLeak statemachine

bonsaiTalkTimeout statemachine

btlAngleAlgo statemachine

clafuSleep objectrecognition

clockShift Affects all components running on the laptop where the clock was shifted: armcontrol, legdetector, objectbuilder, objectrecognition.

facerecSkippable facerec

legdetectorSkippable legdetector

objectBuilderSkippable objectbuilder

pocketSphinxLeak speechrec

spreadLatency Affects all components.



## ACRONYMS

---

### A

AEW

accumulated event window. *used on: pp. 152, 155, 156, 158, 159, 161, 164–167, 174, 179, 180*

API

application programming interface. *used on: pp. 80, 82, 94, 110, 112–114, 120, 123, 126, 132, 135*

APM

application performance monitoring. *used on: pp. 10, 105, 106*

AST

abstract syntax tree. *used on: pp. 128, 131, 133–136*

AUC

area under curve. *used on: pp. 121, 122, 180, 183, 184*

### C

CBSE

component-based software engineering. *used on: pp. 43, 45, 46, 51*

CI

continuous integration. *used on: pp. 116, 119, 137, 189*

CPU

central processing unit. *used on: pp. v, 7–11, 25, 35, 37, 38, 62, 79, 81, 99, 100, 107, 109, 118, 145, 164–166, 170, 173, 180, 183, 184*

CSRA

Cognitive Service Robotics Apartment. *used on: pp. 99, 101, 102, 120*

CSV

comma-separated values. *used on: pp. 66, 73, 76, 77, 147*

### D

DSL

domain-specific language. *used on: pp. 106, 123, 125–137, 171*

### F

FA

feature agglomeration. *used on: p. 157*

FDD

fault detection and diagnosis. *used on: p. 16*

FDI

fault detection and isolation. *used on: p. 16*

- FDIR**  
fault detection, isolation, and recovery. *used on: p. 16*
- FS**  
feature selection. *used on: p. 157*
- G**
- GBR**  
gradient boosted regression trees. *used on: pp. 157, 159, 161*
- GUI**  
graphical user interface. *used on: p. 106*
- H**
- HDF5**  
hierarchical data format 5. *used on: pp. 117, 119*
- HRI**  
human–robot interaction. *used on: pp. 3, 4, 68, 71, 72, 78, 143*
- HTTP**  
Hypertext Transfer Protocol. *used on: pp. 56, 94, 106, 107, 127*
- I**
- I/O**  
input/output. *used on: pp. 81, 226, 229*
- IaaS**  
infrastructure as a service. *used on: p. 33*
- IDE**  
integrated development environment. *used on: pp. 125, 127–129, 134, 135, 171*
- IDL**  
interface description language. *used on: pp. 47–49, 65, 67*
- IPC**  
inter-process communication. *used on: pp. 22, 24, 25, 27, 47*
- J**
- JSON**  
JavaScript Object Notation. *used on: pp. 66, 83, 126, 152*
- K**
- KPI**  
key performance indicator. *used on: pp. 10, 17–19, 32, 93, 94, 106, 107, 118, 144, 178*
- KR**  
kernel ridge regression. *used on: pp. 157, 159, 161*
- KS**  
Kolmogorov-Smirnov. *used on: pp. 102, 118, 121–123*
- M**
- M2M**  
model-to-model. *used on: pp. 128, 132*

**MARTE**

Modeling and Analysis of Real-Time Embedded Systems.  
*used on: p. 34*

**MDSD**

model-driven software development. *used on: pp. 38, 39, 125*

**MTBF**

mean time between failures. *used on: pp. xiii, 24, 28*

**N****NaN**

Not a Number. *used on: pp. 154, 156*

**NTP**

network time protocol. *used on: pp. 73, 75, 145*

**O****OCSVM**

one-class support vector machine. *used on: pp. 179–181, 183, 184*

**P****PaaS**

platform as a service. *used on: p. 33*

**PCA**

principle component analysis. *used on: p. 107*

**PID**

process identifier. *used on: pp. 87–89*

**POSIX**

Portable Operating System Interface. *used on: p. 82*

**Q****QoS**

quality of service. *used on: pp. 18, 33, 36, 39, 59–61*

**R****RBF**

radial basis function. *used on: p. 157*

**RMSE**

root mean square error. *used on: pp. 158, 159, 161, 162, 165, 168, 172*

**ROC**

receiver operator characteristic. *used on: pp. 121, 122, 180, 183, 184*

**RPC**

remote procedure call. *used on: pp. 47, 48, 55, 63, 66, 111, 167, 170, 180, 237*

**RRSE**

root relative squared error. *used on: pp. 158, 159*

## RSB

Robotics Service Bus. *used on: pp. xiii, xv, 51, 52, 54–63, 65–69, 75, 76, 83–86, 89, 95, 99–101, 111–114, 120, 142, 144–146, 152, 153, 155, 156, 179, 193, 237*

## RSS

resident set size. *used on: pp. 160, 229*

## RST

Robotics Systems Types. *used on: pp. xiii, xvi, 63–65, 67, 85, 86, 95, 96, 130, 153*

## S

## SLA

service level agreement. *used on: pp. 18, 172*

## SLAM

simultaneous localization and mapping. *used on: pp. 36, 142, 170, 171*

## SUT

system under test. *used on: pp. 106, 107, 125*

## T

## TCP

transmission control protocol. *used on: pp. 48, 59, 144, 227*

## TTS

text to speech. *used on: pp. 64, 100, 145*

## U

## UML

Unified Modeling Language. *used on: pp. 18, 34, 63, 74, 85, 86, 97, 126*

## URI

Uniform Resource Identifier. *used on: p. 57*

## URL

Uniform Resource Locator. *used on: p. 127*

## UTP

UML Testing Profile. *used on: pp. 126, 127*

## UUID

universally unique identifier. *used on: p. 56*

## V

## VFOA

visual focus of attention. *used on: p. 71*

## X

## XML

Extensible Markup Language. *used on: pp. 59, 60, 106, 119, 126, 132, 134, 135*

## XML-RPC

Extensible Markup Language Remote Procedure Call. *used on: p. 83*

*XMPP*

Extensible Messaging and Presence Protocol. *used on: p. 106*

**Y**

*YAML*

YAML Ain't Markup Language. *used on: p. 126*



## GLOSSARY

---

### B

#### *bug*

A software or hardware defect that potentially causes a [failure](#). *used on: pp. 14, 15, 17–19, 21, 24–29, 103, 105, 143–146, 158, 175, 181, 183, 185*

### C

#### *component*

A “component is a unit of composition, and it must be specified in such a way that it is possible to compose it with other components and integrate it into systems in a predictable way” [Crn+02]. See [Section 5.1](#) on page 43. In this work, used interchangeably with [microservice](#). *used on: pp. v, 4, 5, 21, 25, 29, 33–35, 37–39, 43–52, 58, 63, 66–68, 73, 74, 78–80, 82, 84, 86–89, 93, 96, 97, 99, 100, 102, 105–112, 114–123, 125, 127, 135–137, 141, 142, 144–146, 150–152, 154–157, 159, 160, 162–170, 172–175, 177–185, 189–191, 194, 237, 238*

#### *component interface*

“[A]ccess points [...] [which] allow clients of a component, usually components themselves, to access the services provided by the component” [Szy03, p. 42]. *used on: pp. 44, 49, 50, 107, 109, 164, 169, 170*

#### *component model*

A “component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution, and deployment” [WS01]. *used on: pp. 34, 44, 47, 48, 52, 68*

#### *component platform*

“[D]efines the rules of deployment, installation, and activation of components” [Szy03, p. 44]. *used on: pp. 44, 45, 47, 48*

#### *computational resource*

Abstract resources used to perform complexity analysis of algorithms. See [Section 2.1](#) on page 7. *used on: p. 7*

#### *connector*

Technical interface in [RSB](#) that realizes a [transport](#) and connects [participants](#) to this transport. *used on: pp. 59, 60*

#### *converter*

An object in [RSB](#) that is responsible for serialization between user-defined [event](#) payload and specific wire format, for instance a binary representation. *used on: pp. 59–63, 65, 67, 68*

*converter selection strategy*

Exchangeable and user-definable strategies to select appropriate [RSB converters](#). *used on: p. 60*

**D***dashboard*

“A dashboard is a visual display of the most important information needed to achieve one or more objectives; consolidated and arranged on a single screen so the information can be monitored at a glance” [Fewo4]. *used on: pp. xiii, 23, 28, 32, 38, 93–103, 143, 189, 231*

*dependability*

“The ability to avoid service failures that are more frequent and more severe than is acceptable” [Avi+04]. *used on: pp. v, 3–5, 12–17, 21, 23, 28, 31, 32, 80, 91, 139, 141, 184, 189, 190, 194*

*dependable computing*

Unifying term for research and methods dealing with the reliability, safety and security of computing systems. See [Section 2.2](#) on page 12. *used on: pp. 12, 14–17*

*distributed system*

“A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages” [Cou+12, p. 1]. *used on: pp. 22, 28, 29, 33, 38, 45, 59, 68, 189, 190*

**E***event*

In event-based systems, “a detectable condition that can trigger a [notification](#)” [Faio6]. If not stated otherwise, this refers to events in the [RSB middleware](#). *used on: pp. 54–63, 65, 66, 75, 84, 95, 111, 116, 120, 145, 147, 151–155, 158, 160–162, 164, 167, 171, 173–175, 179, 181–183, 185, 193*

**F***failure*

Short for [system failure](#). *used on: pp. v, 14, 17, 19, 21, 28, 29, 146*

*fault*

“An unpermitted deviation of at least one characteristic property (feature) of the system from the acceptable, usual, standard condition” [Iseo6, p. 20], which can potentially be observed through measurable *errors*. See [Section 2.2.2](#) on page 14. *used on: pp. 14, 15, 17, 18, 21, 147, 158, 175–179, 183, 184*

*fault detection*

“Determination of faults present in a system and the time of the detection” [IB97]. *used on: pp. v, 4, 16, 17, 23, 28, 141, 144, 156, 174–180, 182–185, 190, 194, 237*



*filter*

In [RSB](#), code that evaluates whether a received event should be delivered to registered [handlers](#). *used on: pp. 61, 62, 66*

*framework-level resource awareness*

A realization of the [resource awareness](#) concept in terms of reusable and generic methods that are provided by a software architecture environment. In contrast to [implementation-level resource awareness](#), the resource awareness is created by the framework and requires only a minimum amount of support from individual components and developers. *used on: pp. v, 5, 32–34, 37–39, 43, 50, 68, 69, 79, 84, 89, 95, 97, 105, 107, 120, 125, 128, 135, 136, 150–152, 167, 171, 173, 177, 189, 190*

*functional requirement*

Defines a “necessary task, action or activity that must be accomplished” [[US 01](#)] by software. *used on: pp. 3, 79, 105*

**H***handler*

In [RSB](#), a callback to be called on each received event. *used on: p. 61*

*hardware performance counter*

A special [system metric](#) directly provided by a hardware component of a system. See [Section 2.1.2](#) on page 10. *used on: p. 10*

**I***implementation-level resource awareness*

A realization of the resource awareness concept in terms of special modifications to or methods used for creating the functional components of a software system. For instance: anytime algorithms (cf. [Section 4.2.3](#) on page 36). *used on: p. 31*

*informer*

An [RSB](#) participant that sends events. *used on: pp. 54, 58–60*

*infrastructure as a service*

A service model of a cloud computing provider which enables a cloud user to use “processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications” [[MG11](#)]. *used on: p. 33*

*introspection*

A method to “obtain metadata about an object or application” [[Vino5](#)] at runtime. *used on: pp. 51, 58, 61–63, 66, 68, 100, 152, 189*

## K

*key performance indicator*

Synonym for [system resource](#) originating from [APM](#). *used on:* pp. 10, 17–19, 32, 93, 94, 106, 107, 118, 144, 178

## L

*listener*

An [RSB](#) participant that receives events. *used on:* pp. 54, 57, 58, 61, 66

## M

*mean time between failures*

The “expected or observed time between consecutive failures in a system or component” [[III10](#)]. *used on:* pp. 24, 28

*microservice*

An unit in a [microservice architecture](#) which is “built around business capabilities and independently deployable” [[LF14](#)]. In this work, used interchangeably with [component](#). *used on:* pp. 49, 50, 173

*microservice architecture*

An architectural style where applications are developed “as a suite of small services, each running in its own process and communicating with lightweight mechanisms” [[LF14](#)]. *used on:* pp. 49–51, 68, 69, 79, 87, 189, 190

*middleware*

A “software layer that provides a programming abstraction as well as [a] masking [of] the heterogeneity of the underlying networks, hardware, operating systems and programming languages” [[Cou+12](#), p. 17]. *used on:* pp. xiii, 45, 46, 48, 49, 51–54, 56, 58, 59, 65–69, 71, 73–75, 77–79, 85, 86, 93, 95, 99, 107–110, 112–116, 120, 123, 142, 143, 150–153, 190, 193, 237

*monitor*

A “software tool or hardware device that operates concurrently with a system or component and supervises, records, analyzes, or verifies the operation of the system or component” [[III10](#)]. *used on:* p. 16

## N

*nonfunctional requirement*

A “software requirement that describes not what the software will do but how the software will do it” [[III10](#)]. *used on:* pp. 79, 105

*notification*

In event-based systems, a notification is “an event-triggered signal sent to a run-time defined recipient” [[Faio6](#)]. *used on:* pp. 54–56, 59, 61

## P

*participant*

An object of the [RSB](#) middleware that forms the user-code entry point to communicate via the unified bus. *used on:* pp. 54–56, 58–63, 66, 100, 101, 144, 152, 156

*performance*

The “degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage” [III10]. *used on:* pp. 17–19, 21, 31, 33, 67, 99, 105, 110, 116–118, 122

*performance bug*

A programming defect that causes decreases the performance of a system in certain situations, but does not result in a service outage. *used on:* pp. xiii, xiv, 18, 19, 21, 25–27, 29, 31, 79, 99, 103, 105, 139, 141, 143–147, 174–177, 181, 182, 184, 185, 189, 193, 194, 238

*performance degradation*

The visible effect of a [performance bug](#), i.e. a reduction of performance of a system that still delivers the correct [service](#). *used on:* pp. 18, 19, 141, 143, 144, 146, 149, 175, 177–179, 181–185, 193

*performance regression*

A performance bug in existing functionality that was introduced unintendedly while modifying a system. *used on:* pp. 19, 105–108, 118–122, 174, 193

*platform as a service*

A service model of a cloud computing provider that enables a cloud user to “to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment” [MG11]. *used on:* p. 33

## Q

*quality of service*

The “description or measurement of the overall performance of a service, such as [a] telephony or computer network or [a] Cloud computing service” [Wik17]. *used on:* pp. 18, 33, 36, 39, 59–61

**R***reliability*

A “measure of the continuous service accomplishment” [Lap95] or the “ability of a system to perform a required function under stated conditions, within a given scope, during a given period of time” [Iseo6, p. 21]. *used on:* pp. 14, 21, 28, 60, 149

*resource*

Synonym for system resource. *used on:* pp. 7–11, 25, 31, 33–38, 79, 82, 84, 97, 102, 103, 116, 117, 172, 173, 180, 189

*resource awareness*

A software construction and operation concept that focuses on uncovering the usually hidden **resource utilization** of technical systems for developers and automated monitoring systems. See [Chapter 4](#) on page 31. *used on:* pp. v, 5, 7, 31–39, 50, 79, 91, 93, 139, 141, 149–151, 167, 175, 189–191, 194

*resource capacity*

A quantification of the availability of a system resource. See [Section 2.1](#) on page 7. *used on:* pp. 8, 9, 210

*resource contention*

Conflict of multiple processes requesting the same system resources in parallel. See [Section 2.1](#) on page 7. *used on:* pp. 8, 119, 141, 173, 174

*resource starvation*

Situation in which the **capacity** of system resources is not sufficient to fulfill a request that is therefore rejected. See [Section 2.1](#) on page 7. *used on:* pp. 8, 33, 139, 141, 174

*resource utilization*

Usage of system resource by processes running on a system. See [Section 2.1](#) on page 7. *used on:* pp. v, 4, 8, 9, 11, 12, 25, 31–35, 37–39, 41, 67, 78–80, 82, 84–88, 91, 93–95, 97–99, 102, 103, 105, 106, 108, 109, 116–118, 122, 139, 141, 149–151, 155, 156, 158, 159, 162, 165, 172–175, 177, 178, 180, 182, 183, 189, 190, 193, 194, 231

*Robotics Service Bus*

The robotics middleware used in this work. *used on:* pp. 51, 52, 54–63, 65–69, 75, 76, 83–86, 89, 95, 99–101, 111–114, 120, 142, 144–146, 152, 153, 155, 156, 179, 193, 237

*Robotics Systems Types*

A library of data types for robotics and intelligent systems defined using the *Protocol Buffers* [Protobuf] IDL. *used on:* pp. 63–65, 67, 85, 86, 95, 96, 130, 153

*robustness*

The “degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions” [III10]. *used on:* p. 14

## S

*scope*

In **RSB**, the notation used to express a hierarchical communication channel. *used on: pp. 57, 58, 61–63, 66, 100, 145, 151–153, 155, 156, 170*

*service*

A system's delivered function or, in other words: "the service delivered by a system (in its role as a provider) is its behavior as it is perceived by its user(s)" [Avi+04]. *used on: pp. 13–18, 178*

*software regression*

A **bug** in existing functionality that was introduced through a modification to the software. *used on: p. 17*

*system failure*

An "event that occurs when the delivered service deviates from correct service" [Avi+04]. See **Section 2.2.2** on page 14. *used on: pp. xiii, 18, 19, 24, 25, 28, 29, 146, 149, 175*

*system metric*

A measurable value exposed by an operating system, which loosely reflects aspects of resource utilizations. See **Section 2.1.2** on page 10. *used on: pp. xiii, 10–12, 18, 19, 23, 38, 79–87, 89, 90, 93–97, 99, 103, 107, 108, 116–121, 141, 143, 147, 149–164, 167, 170–173, 178, 179, 181, 182, 185, 226, 227*

*system metric source*

Technical interface to access system metrics. See **Section 2.1** on page 7. *used on: pp. xv, 10, 11, 80, 81, 83, 85, 86, 88*

*system resource*

(Physical) resources of a computer that can be allocated to a computational task, for example, the CPU, working memory, or network bandwidth. See **Section 2.1** on page 7. *used on: pp. v, xiii, 3, 7–12, 25–27, 31, 32, 35, 38, 91, 97, 141, 144, 174, 182, 189, 210*

## T

*transport*

A transport in **RSB** realizes the transmission of event notifications between participants. *used on: pp. 59–62, 66–68, 100*

## V

*vendor lock-in*

"Vendor lock-in, or just lock-in, is the situation in which customers are dependent on a single manufacturer or supplier for some product (i.e., a good or service), or products, and cannot move to another vendor without substantial costs and/or inconvenience" [Lipo06]. *used on: pp. 52, 56, 65*

*vertical component*

Vertical components “capture know-how in specific functional areas such as kinematics, motion planning, deliberative control, and address the requirements of target application domains such as service robotics, space robotics, or humanoid robotics” [BS09]. *used on: pp. 49, 108, 109, 120, 123, 150*

*virtual resource*

A system resource that is not directly coupled to hardware, but instead exists only as a software artifact of the operating system, potentially with relation to physical resources. For instance, the file system is a virtual resource with own properties and a relation to the disk as a hardware resource. *used on: pp. 12, 25, 227*

**W***wire schema*

String-based declaration of the encoding of serialized event payloads in [RSB](#). *used on: p. 60*

*wire type*

The technical representation in which an [RSB](#) transport expects serialized content for transmission, for instance, byte sequences or [XML](#) documents. *used on: pp. 59, 60*

## BIBLIOGRAPHY

---

### OWN PUBLICATIONS

- [Jay+13] Dinesh Babu Jayagopi et al. “The vernissage corpus. A conversational Human-Robot-Interaction dataset.” In: *Proceedings of the 8th ACM/IEEE International Conference on Human-Robot Interaction*. Ed. by Hideaki Kuzuoka et al. IEEE, 2013, pp. 149–150. DOI: [10.1109/HRI.2013.6483545](https://doi.org/10.1109/HRI.2013.6483545). used on: p. 78
- [Klo+11] David Klotz et al. “Engagement-based Multi-party Dialog with a Humanoid Robot.” In: *Proceedings of the SIGDIAL 2011 Conference*. Association for Computational Linguistics, 2011, pp. 341–343. used on: p. 68
- [San+12] Jordi Sanchez-Riera et al. “Online multimodal speaker detection for humanoid robots.” In: *12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*. IEEE, 2012, pp. 126–133. DOI: [10.1109/HUMANOIDS.2012.6651509](https://doi.org/10.1109/HUMANOIDS.2012.6651509). used on: p. 68
- [Wie+18] Johannes Wienke et al. “Model-Based Performance Testing for Robotics Software Components.” In: *Second IEEE International Conference on Robotic Computing*. IEEE, 2018, pp. 25–32. DOI: [10.1109/IRC.2018.00013](https://doi.org/10.1109/IRC.2018.00013). used on: p. 125
- [WKW12] Johannes Wienke, David Klotz, and Sebastian Wrede. “A Framework for the Acquisition of Multimodal Human-Robot Interaction Data Sets with a Whole-System Perspective.” In: *Multimodal Corpora: How Should Multimodal Corpora Deal with the Situation?* Ed. by Jens Edlund, Dirk Heylen, and Patrizia Paggio. 2012, pp. 46–49. used on: pp. 71, 77
- [WMW16] Johannes Wienke, Sebastian Meyer zu Borgsen, and Sebastian Wrede. “A Data Set for Fault Detection Research on Component-Based Robotic Systems.” In: *Towards Autonomous Robotic Systems*. Ed. by Lyuba Alboul, Dana Damian, and Jonathan M. Aitken. Lecture Notes in Artificial Intelligence 9716. Springer International Publishing, 2016, pp. 339–350. DOI: [10.1007/978-3-319-40379-3\\_35](https://doi.org/10.1007/978-3-319-40379-3_35). used on: pp. 21, 141
- [WNW12] Johannes Wienke, Arne Nordmann, and Sebastian Wrede. “A Meta-Model and Toolchain for Improved Interoperability of Robotic Frameworks.” In: *Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by Itsuki Noda et al. Lecture Notes in Artificial Intelligence 7628. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 323–334. DOI: [10.1007/978-3-642-34327-8\\_30](https://doi.org/10.1007/978-3-642-34327-8_30). used on: p. 67

- [WW11] Johannes Wienke and Sebastian Wrede. “A Middleware for Collaborative Research in Experimental Robotics.” In: *IEEE / SICE International Symposium on System Integration (SII 2011)*. IEEE, 2011, pp. 1183–1190. DOI: [10.1109/SII.2011.6147617](https://doi.org/10.1109/SII.2011.6147617). used on: pp. 51, 69
- [WW16a] Johannes Wienke and Sebastian Wrede. *A Fault Detection Data Set for Performance Bugs in Component-Based Robotic Systems*. 2016. DOI: [10.4119/unibi/2900912](https://doi.org/10.4119/unibi/2900912). Dataset. used on: p. 147
- [WW16b] Johannes Wienke and Sebastian Wrede. *A Fault Detection Data Set for Performance Bugs in Component-Based Robotic Systems - Sources*. 2016. DOI: [10.4119/unibi/2900911](https://doi.org/10.4119/unibi/2900911). Dataset. used on: p. 147
- [WW16c] Johannes Wienke and Sebastian Wrede. “Autonomous Fault Detection for Performance Bugs in Component-Based Robotic Systems.” In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2016, pp. 3291–3297. DOI: [10.1109/IRoS.2016.7759507](https://doi.org/10.1109/IRoS.2016.7759507). used on: pp. 151, 175
- [WW16d] Johannes Wienke and Sebastian Wrede. “Continuous Regression Testing for Component Resource Utilization.” In: *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. IEEE, 2016, pp. 273–280. DOI: [10.1109/SIMPAR.2016.7862407](https://doi.org/10.1109/SIMPAR.2016.7862407). used on: p. 105
- [WW17a] Johannes Wienke and Sebastian Wrede. “Performance regression testing and run-time verification of components in robotics systems.” In: *Advanced Robotics* 31 (22 2017), pp. 1177–1192. ISSN: 0169-1864. DOI: [10.1080/01691864.2017.1395360](https://doi.org/10.1080/01691864.2017.1395360). used on: pp. 105, 165, 168, 170
- [WW17b] Johannes Wienke and Sebastian Wrede. *Robotics Components Resource Utilization for Performance Regression Detection*. 2017. DOI: [10.4119/unibi/2913636](https://doi.org/10.4119/unibi/2913636). Dataset. used on: p. 121

## GENERAL

- [Ace+13] Giuseppe Aceto et al. “Cloud monitoring. A survey.” In: *Computer Networks* 57 (9 2013), pp. 2093–2115. ISSN: 13891286. DOI: [10.1016/j.comnet.2013.04.001](https://doi.org/10.1016/j.comnet.2013.04.001). used on: p. 32
- [AiC+04] Mitchell Ai-Chang et al. “MAPGEN: Mixed-Initiative Planning and Scheduling for the Mars Exploration Rover Mission.” In: *IEEE Intelligent Systems* 19 (1 2004), pp. 8–12. ISSN: 1541-1672. DOI: [10.1109/MIS.2004.1265878](https://doi.org/10.1109/MIS.2004.1265878). used on: p. 35
- [AL86] Algirdas Antanas Avižienis and Jean-Claude Laprie. “Dependable computing. From concepts to design diversity.” In: *Proceedings of the IEEE* 74 (5 1986), pp. 629–638. ISSN: 0018-9219. DOI: [10.1109/PROC.1986.13527](https://doi.org/10.1109/PROC.1986.13527). used on: p. 14
- [ALRo1] Algirdas Antanas Avižienis, Jean-Claude Laprie, and Brian Randell. *Fundamental Concepts of Dependability*. Tech. rep. 010028. Los Angeles, USA: University of California, 2001? URL: <http://www.idt.mdh.se/kurser/computing/DVA416/>



- [Lectures/avizienis01fundamental.pdf](#) (visited on 2017-05-04). *used on: p. 15*
- [Ami+04] Yair Amir et al. *The Spread Toolkit: Architecture and Performance*. Tech. rep. CNDS-2004-1. Johns Hopkins University, 2004. URL: <http://www.cnds.jhu.edu/pub/papers/cnds-2004-1.pdf> (visited on 2016-09-08). *used on: p. 59*
- [And+05] Noriaki Ando et al. "RT-middleware: distributed component middleware for RT (robot technology)." In: *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2005, pp. 3933–3938. DOI: [10.1109/IR05.2005.1545521](https://doi.org/10.1109/IR05.2005.1545521). *used on: pp. 46, 48*
- [AS98] Yair Amir and Jonathan Stanton. *The Spread Wide Area Group Communication System*. Tech. rep. CNDS-98-4. Johns Hopkins University, 1998. URL: <http://www.cnds.jhu.edu/pub/papers/spread.ps> (visited on 2016-09-08). *used on: p. 59*
- [ASK08] Noriaki Ando, Takashi Suehiro, and Tetsuo Kotoku. "A Software Platform for Component Based RT-System Development: OpenRTM-Aist." In: *Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by Stefano Carpin et al. Lecture Notes in Artificial Intelligence 5325. Berlin, Heidelberg: Springer, 2008, pp. 87–98. DOI: [10.1007/978-3-540-89076-8](https://doi.org/10.1007/978-3-540-89076-8). *used on: p. 48*
- [Avi+04] Algirdas Antanas Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing." In: *IEEE Transactions on Dependable and Secure Computing* 1 (1 2004), pp. 11–33. ISSN: 1545-5971. DOI: [10.1109/TDSC.2004.2](https://doi.org/10.1109/TDSC.2004.2). *used on: pp. 12–15, 17, 18, 246, 251*
- [Bar+96] Daniel J. Barrett et al. "A framework for event-based software integration." In: *ACM Transactions on Software Engineering and Methodology* 5 (4 1996), pp. 378–421. DOI: [10.1145/235321.235324](https://doi.org/10.1145/235321.235324). *used on: p. 54*
- [BBY16] Daniel J. Brooks, Momotaz Begum, and Holly A. Yanco. "Analysis of reactions towards failures and recovery strategies for autonomous robots." In: *25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*. IEEE, 2016, pp. 487–492. DOI: [10.1109/ROMAN.2016.7745162](https://doi.org/10.1109/ROMAN.2016.7745162). *used on: p. 3*
- [Bee+15] Loy van Beek et al. *RoboCup@Home 2015: Rules and Regulations*. 2015. URL: [http://www.robocupathome.org/rules/2015\\_rulebook.pdf](http://www.robocupathome.org/rules/2015_rulebook.pdf) (visited on 2016-09-06). *used on: p. 142*
- [Ben+09] Saddek Bensalem et al. "Designing autonomous robots." In: *IEEE Robotics & Automation Magazine* 16 (1 2009), pp. 67–77. ISSN: 1070-9932. DOI: [10.1109/MRA.2008.931631](https://doi.org/10.1109/MRA.2008.931631). *used on: p. 105*
- [Bey+09] K. Beghdad Bey et al. "CPU Load Prediction Model for Distributed Computing." In: *Eighth International Symposium on Parallel and Distributed Computing, 2009*. Ed. by Leonel Sousa and Yves Robert. Los Alamitos, California: IEEE, 2009, pp. 39–45. DOI: [10.1109/ISPDC.2009.8](https://doi.org/10.1109/ISPDC.2009.8). *used on: pp. 156, 172*

- [BHW16] Andreas Bihlmaier, Matthias Hadlich, and Heinz Wörn. “Advanced ROS Network Introspection (ARNI).” In: *Robot Operating System (ROS). The Complete Reference (Volume 1)*. Ed. by Anis Koubâa. Studies in Computational Intelligence 625. Cham: Springer International Publishing, 2016, pp. 651–670. ISBN: 978-3-319-26052-5. DOI: [10.1007/978-3-319-26054-9\\_25](https://doi.org/10.1007/978-3-319-26054-9_25). used on: pp. [38](#), [193](#)
- [BKR09] Steffen Becker, Heiko Koziolk, and Ralf Reussner. “The Palladio component model for model-driven performance prediction.” In: *Journal of Systems and Software* 82 (1 2009), pp. 3–22. ISSN: 0164-1212. DOI: [10.1016/j.jss.2008.03.066](https://doi.org/10.1016/j.jss.2008.03.066). used on: pp. [7](#), [9](#), [33](#)
- [Bou01] Tony Bourke. *Server load balancing*. Sebastopol, CA: O’Reilly, 2001. ISBN: 978-0-596-00050-9. used on: p. [33](#)
- [Bou16] Mohamed Boussaffa. *Support building as a library*. 2016. URL: <https://github.com/raboof/nethogs/pull/40>. used on: pp. [83](#), [87](#)
- [Bra00] Gary Bradski. *The OpenCV Library*. 2000. URL: <http://www.drdoobs.com/open-source/the-opencv-library/184404319> (visited on 2017-08-17). used on: p. [59](#)
- [Brio8] Robert Bringhurst. *The Elements of Typographic Style*. 3.2. Point Roberts, Wash.: Hartley & Marks, 2008. ISBN: 978-0-88179-206-5. used on: p. [285](#)
- [Bri16] Adrian Bridgwater. *Microservices are not the same thing as components*. 2016. URL: [https://www.theregister.co.uk/2016/01/06/inside\\_microservices/](https://www.theregister.co.uk/2016/01/06/inside_microservices/) (visited on 2017-07-31). used on: p. [49](#)
- [Bru+13] Herman Bruyninckx et al. “The BRICS component model.” In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. Ed. by Sung Y. Shin and José Carlos Maldonado. New York, New York, USA: ACM Press, 2013, pp. 1758–1764. DOI: [10.1145/2480362.2480693](https://doi.org/10.1145/2480362.2480693). used on: p. [46](#)
- [Bru01] Herman Bruyninckx. “Open robot control software: the ORO-COS project.” In: *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation*. IEEE, 2001, pp. 2523–2528. DOI: [10.1109/ROBOT.2001.933002](https://doi.org/10.1109/ROBOT.2001.933002). used on: pp. [46](#), [47](#)
- [BRZ16] Maicon Bernardino Da Silveira, Elder M. Rodrigues, and Avelino Francisco Zorzo. “Performance Testing Modeling: an empirical evaluation of DSL and UML-based approaches.” In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. Ed. by Sascha Ossowski. New York, New York, USA: ACM Press, 2016, pp. 1660–1665. DOI: [10.1145/2851613.2851832](https://doi.org/10.1145/2851613.2851832). used on: p. [127](#)
- [BS09] Davide Brugali and Patrizia Scandurra. “Component-based robotic engineering (Part I). Reusable Building Blocks.” In: *IEEE Robotics & Automation Magazine* 16 (4 2009), pp. 84–96. ISSN: 1070-9932. DOI: [10.1109/MRA.2009.934837](https://doi.org/10.1109/MRA.2009.934837). used on: pp. [43](#), [49](#), [252](#)

- [Buu12] Stef van Buuren. *Flexible imputation of missing data*. Chapman & Hall/CRC interdisciplinary statistics series. Boca Raton, FL, USA: CRC Press, 2012. ISBN: 978-1-4398-6824-9. used on: p. 156
- [BW14] Andreas Bihlmaier and Heinz Wörn. “Increasing ROS Reliability and Safety through Advanced Introspection Capabilities.” In: *Informatik 2014. Big Data - Komplexität meistern*. Ed. by Erhard Plödereder et al. GI-Edition Proceedings 232. Gesellschaft für Informatik. Bonn: Gesellschaft für Informatik, 2014, pp. 1319–1326. used on: p. 38
- [BZR16] Maicon Bernardino Da Silveira, Avelino Francisco Zorzo, and Elder M. Rodrigues. “Canopus: A Domain-Specific Language for Modeling Performance Testing.” In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 157–167. DOI: 10.1109/ICST.2016.13. used on: p. 127
- [Čap14] Karel Čapek. *R.U.R.* Dover Thrift Editions. Newburyport: Dover Publications, 2014. ISBN: 978-0-486-41926-8. used on: p. 3
- [Cas+06] Rebecca Castano et al. “Opportunistic Rover Science: Finding and Reacting to Rocks, Clouds and Dust Devils.” In: *2006 IEEE Aerospace Conference*. IEEE, 2006, pp. 1–16. DOI: 10.1109/AERO.2006.1656011. used on: p. 35
- [CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. “Anomaly detection: A survey.” In: *ACM Computing Surveys* 41 (3 2009), 15:1–15:58. ISSN: 0360-0300. DOI: 10.1145/1541880.1541882. used on: pp. 176, 178
- [CDF01] G. Cugola, E. Di Nitto, and A. Fuggetta. “The JEDI event-based infrastructure and its application to the development of the OPSS WFMS.” In: *IEEE Transactions on Software Engineering* 27 (9 2001), pp. 827–850. ISSN: 0098-5589. DOI: 10.1109/32.950318. used on: p. 55
- [Che+08] Shiping Chen et al. “Yet Another Performance Testing Framework.” In: *19th Australian Conference on Software Engineering (ASWEC 2008)*. Ed. by Farookh Khadeer Hussain. Los Alamitos, Calif.: IEEE Computer Soc, 2008, pp. 170–179. DOI: 10.1109/ASWEC.2008.4483205. used on: p. 106
- [Cis15] Cisco. *How Does Load Balancing Work?* 2015. URL: <https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/5212-46.html> (visited on 2017-06-29). used on: p. 33
- [CLo2] Ivica Crnkovic and Magnus Peter Henrik Larsson, eds. *Building reliable component-based software systems*. Artech House computing library. Boston: Artech House, 2002. ISBN: 978-1-58053-327-0. used on: p. 43
- [CMS13] Matheus Cunha, Nabor Mendonca, and Americo Sampaio. “A Declarative Environment for Automatic Performance Evaluation in IaaS Clouds.” In: *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE, 2013, pp. 285–292. DOI: 10.1109/CLOUD.2013.12. used on: p. 126

- [Com11] CVE-2011-2494. Common Vulnerabilities and Exposures. 2011. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-2494> (visited on 2017-10-25). used on: p. 81
- [Cor17] Jonathan Corbet. *Software interrupts and realtime*. 2017. URL: <https://lwn.net/Articles/520076/> (visited on 2017-11-10). used on: p. 226
- [Cou+12] George Coulouris et al. *Distributed Systems. Concepts and Design*. 5th ed. Harlow, Essex: Pearson Education, 2012. ISBN: 978-0-13-214301-1. used on: pp. 45, 59, 246, 248
- [Cou11] Steve Cousins. "Exponential Growth of ROS." In: *IEEE Robotics & Automation Magazine* 18 (1 2011), pp. 19–20. ISSN: 1070-9932. DOI: 10.1109/MRA.2010.940147. used on: p. 47
- [Crn+02] Ivica Crnkovic et al. "Basic Concepts in CBSE." In: *Building reliable component-based software systems*. Ed. by Ivica Crnkovic and Magnus Peter Henrik Larsson. Artech House computing library. Boston: Artech House, 2002. Chap. 1, pp. 1–22. ISBN: 978-1-58053-327-0. used on: pp. 44, 245
- [CW15] Dariusz Caban and Tomasz Walkowiak. "Prediction of the Performance of Web Based Systems." In: *Dependability Problems of Complex Information Systems*. Ed. by Wojciech Zamojski and Jaroslaw Sugier. Vol. 307. Advances in Intelligent Systems and Computing 307. Cham: Springer International Publishing, 2015, pp. 1–18. ISBN: 978-3-319-08963-8. DOI: 10.1007/978-3-319-08964-5\_1. used on: p. 34
- [Dan+16] Timo Dankert et al. "Engagement Detection During Deictic References in Human-Robot Interaction." In: *Social Robotics*. Ed. by Arvin Agah et al. Lecture Notes in Computer Science 9979. Cham: Springer International Publishing, 2016, pp. 930–939. DOI: 10.1007/978-3-319-47437-3\_91. used on: p. 68
- [Dav+13] Ian John Davis et al. "Regression-Based Utilization Prediction Algorithms: An Empirical Investigation." In: *CASCON '13: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*. Ed. by James R. Cordy, Kryztof Czarnecki, and Sang-Ah Han. Riverton, NJ, USA: IBM Corp., 2013, pp. 106–120. used on: p. 172
- [DB88] Thomas Dean and Mark Boddy. "An Analysis of Time-Dependent Planning." In: *Proceedings of the Seventh National Conference on Artificial Intelligence*. Ed. by Tom M. Mitchell and Reid G. Smith. Association for the Advancement of Artificial Intelligence. AAAI Press, 1988. used on: p. 36
- [Dhi91] Balbir S. Dhillon. *Robot Reliability and Safety*. New York, NY: Springer New York, 1991. ISBN: 978-1-4612-3148-6. DOI: 10.1007/978-1-4612-3148-6. used on: p. 3
- [Dij72] Edsger W. Dijkstra. "The Humble Programmer." In: *Communications of the ACM* 15 (10 1972), pp. 859–866. ISSN: 00010782. DOI: 10.1145/355604.361591. used on: p. 175

- [Dino8] Steven X. Ding. *Model-based fault diagnosis techniques. Design schemes, algorithms and tools*. 2nd ed. Advances in Industrial Control. Berlin: Springer, 2008. ISBN: 978-1-4471-6111-0. DOI: [10.1007/978-3-540-76304-8](https://doi.org/10.1007/978-3-540-76304-8). used on: pp. [16](#), [178](#)
- [Dobo4] Glen Dobson. *Quality of Service in Service-Oriented Architectures*. Tech. rep. Lancaster University, 2004. URL: <http://digs.sourceforge.net/papers/qos.pdf> (visited on 2017-05-17). used on: p. [18](#)
- [DS11] Shaun Dunning and Darren Sawyer. "A little language for rapidly constructing automated performance tests." In: *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering - ICPE '11*. Ed. by Samuel Kounev et al. New York, New York, USA: ACM Press, 2011, pp. 371–380. DOI: [10.1145/1958746.1958798](https://doi.org/10.1145/1958746.1958798). used on: p. [127](#)
- [Dup07] Lyn Dupré. *BUGS in writing. A guide to debugging your prose*. Rev ed., 10. print. Boston: Addison-Wesley, 2007. ISBN: 978-0-201-37921-1. used on: p. [285](#)
- [Dut+14] Ayan Dutta et al. "searchUCSG: A fast coalition structure search algorithm for modular robot reconfiguration under uncertainty." In: *Robotica* 32 (2 2014), pp. 225–244. ISSN: 0263-5747. DOI: [10.1017/S0263574714000095](https://doi.org/10.1017/S0263574714000095). used on: p. [36](#)
- [DVC13] Bruno Lopes Dalmazo, Joao P. Vilela, and Marilia Curado. "Predicting Traffic in the Cloud. A Statistical Approach." In: *Proceedings 2013 International Conference on Cloud and Green Computing*. IEEE, 2013, pp. 121–126. DOI: [10.1109/CGC.2013.26](https://doi.org/10.1109/CGC.2013.26). used on: p. [173](#)
- [Eck11] Wayne W. Eckerson. *Performance dashboards. Measuring, monitoring, and managing your business*. 2nd ed. Hoboken, N.J: Wiley, 2011. ISBN: 978-0-470-58983-0. used on: p. [93](#)
- [Fai06] Ted Faison. *Event-Based Programming. Taking Events to the Limit*. Apress, 2006. ISBN: 978-1-59059-643-2. used on: pp. [55](#), [56](#), [61](#), [246](#), [248](#)
- [Fat+14] Kaniz Fatema et al. "A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives." In: *Journal of Parallel and Distributed Computing* 74 (10 2014), pp. 2918–2933. ISSN: 07437315. DOI: [10.1016/j.jpdc.2014.06.007](https://doi.org/10.1016/j.jpdc.2014.06.007). used on: pp. [32](#), [93](#)
- [Faw06] Tom Fawcett. "An introduction to ROC analysis." In: *Pattern Recognition Letters* 27 (8 2006), pp. 861–874. ISSN: 0167-8655. DOI: [10.1016/j.patrec.2005.10.010](https://doi.org/10.1016/j.patrec.2005.10.010). used on: p. [121](#)
- [Few04] Stephen Few. "Dashboard Confusion." In: *Intelligent Enterprise* (2004). used on: pp. [93](#), [246](#)
- [Few06] Stephen Few. *Information dashboard design. The effective visual communication of data*. Beijing: O'Reilly, 2006. ISBN: 978-0-596-10016-2. used on: p. [93](#)

- [FHC97] Sara Fleury, Matthieu Herrb, and Raja Chatila. "GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture." In: *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications*. IEEE, 1997, pp. 842–849. DOI: [10.1109/IROS.1997.655108](https://doi.org/10.1109/IROS.1997.655108). used on: pp. 46, 48
- [FHM12] Sara Fleury, Matthieu Herrb, and Anthony Mallet. *GenoM User's Guide*. 2012. URL: <https://www.openrobots.org/distfiles/genom/genom.pdf> (visited on 2017-08-07). used on: p. 48
- [Foo+10] King Chun Foo et al. "Mining Performance Regression Testing Repositories for Automated Performance Analysis." In: *10th International Conference on Quality Software (QSIC)*. Ed. by Ji Wang, W. K. Chan, and Fei-Ching Kuo. Piscataway, NJ: IEEE, 2010, pp. 32–41. DOI: [10.1109/QSIC.2010.35](https://doi.org/10.1109/QSIC.2010.35). used on: pp. 107, 118, 121
- [Foo+15] King Chun Foo et al. "An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments." In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)*. Ed. by Antonia Bertolino. 2. Los Alamitos, California: IEEE, 2015, pp. 159–168. DOI: [10.1109/ICSE.2015.144](https://doi.org/10.1109/ICSE.2015.144). used on: p. 18
- [Fow05] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: <https://www.martinfowler.com/articles/languageWorkbench.html>. used on: p. 127
- [Fox+99] Dieter Fox et al. "Monte Carlo Localization: Efficient Position Estimation for Mobile Robots." In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), Eleventh Innovative Applications of Artificial Intelligence Conference (IAAI-99)*. Menlo Park, Calif. and Cambridge, Mass.: AAAI Press and MIT Press, 1999, pp. 343–349. used on: p. 36
- [GA07] Dora Luz Gonzalez-Bañales and Manuel Rodenes Adam. "Web Survey Design and Implementation: Best Practices for Empirical Research." In: *Proceedings of the European and Mediterranean Conference on Information Systems 2007*. 2007, pp. 1–10. used on: p. 21
- [Gam+95] Erich Gamma et al. *Design patterns. Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Boston, Mass.: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0. used on: p. 60
- [GBL09] Vahid Garousi, Lionel C. Briand, and Yvan Labiche. "A UML-based quantitative framework for early prediction of resource usage and load in distributed real-time systems." In: *Software & Systems Modeling* 8 (2 2009), pp. 275–302. ISSN: 1619-1366. DOI: [10.1007/s10270-008-0099-7](https://doi.org/10.1007/s10270-008-0099-7). used on: p. 34
- [GD17] Shaifu Gupta and Dileep Aroor Dinesh. "Online adaptation models for resource usage prediction in cloud network." In: *Twenty-third National Conference on Communications (NCC)*. IEEE, 2017. DOI: [10.1109/NCC.2017.8077082](https://doi.org/10.1109/NCC.2017.8077082). used on: p. 172

- [Geh+17] Raphaela Gehle et al. "How to Open an Interaction Between Robot and Museum Visitor?" In: *Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*. Ed. by Bilge Mutlu et al. New York, New York, USA: ACM Press, 2017, pp. 187–195. DOI: [10.1145/2909824.3020219](https://doi.org/10.1145/2909824.3020219). used on: p. 68
- [GEW06] Pierre Geurts, Damien Ernst, and Louis Wehenkel. "Extremely randomized trees." In: *Machine Learning* 63 (1 2006), pp. 3–42. ISSN: 0885-6125. DOI: [10.1007/s10994-006-6226-1](https://doi.org/10.1007/s10994-006-6226-1). used on: p. 157
- [GGW10] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. "PRESS: Predictive Elastic ReSource Scaling for cloud systems." In: *2010 International Conference on Network and Service Management (CNSM 2010)*. Ed. by Yixin Diao, Hanan Lutfiyya, and Deep Medhi. Piscataway, NJ: IEEE, 2010, pp. 9–16. DOI: [10.1109/CNSM.2010.5691343](https://doi.org/10.1109/CNSM.2010.5691343). used on: p. 172
- [GH94] Stephen Gilmore and Jane Hillston. "The PEPA workbench. A tool to support a process algebra-based approach to performance modelling." In: *Computer Performance Evaluation Modelling Techniques and Tools*. Ed. by Günter Haring and Gabriele Kotsis. Lecture Notes in Computer Science 794. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 353–368. DOI: [10.1007/3-540-58021-2\\_20](https://doi.org/10.1007/3-540-58021-2_20). used on: p. 34
- [Gla16] *getloadavg(3) - Linux Programmer's Manual*. 2016. URL: <http://man7.org/linux/man-pages/man3/getloadavg.3.html> (visited on 2017-11-10). used on: p. 226
- [Gol+11] Raphael Golombek et al. "Online data-driven fault detection for robotic systems." In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Ed. by Nancy M. Amato. Piscataway, NJ: IEEE, 2011, pp. 3011–3016. DOI: [10.1109/IR05.2011.6095034](https://doi.org/10.1109/IR05.2011.6095034). used on: pp. 141, 145, 177, 178
- [Gol13] Raphael Golombek. "Data-driven Fault Detection for Component Based Robotic Systems." Doctoral dissertation. Bielefeld: Bielefeld University, 2013. used on: p. 15
- [Gun+14] Haryadi S. Gunawi et al. "What Bugs Live in the Cloud?" In: *Proceedings of the 5th ACM Symposium on Cloud Computing*. Ed. by Edward D. Lazowska et al. New York, NY, USA: ACM, 2014, pp. 1–14. DOI: [10.1145/2670979.2670986](https://doi.org/10.1145/2670979.2670986). used on: pp. 21, 24, 26
- [HC01] George T. Heineman and William T. Councill, eds. *Component-based software engineering. Putting the pieces together*. Boston, Mass.: Addison-Wesley, 2001. ISBN: 978-0-201-70485-3. used on: p. 43
- [Hen17] Nicolas Hennion. *The Glances 2.x API How to*. 2017. URL: <https://github.com/nicolargo/glances/wiki/The-Glances-2.x-API-How-to> (visited on 2017-10-25). used on: p. 83

- [Heo15] Tejun Heo. *Control Group v2*. Linux. 2015. URL: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/cgroup-v2.txt?id=ae59df0349baf44c988b32a3b4dc21363d87df15> (visited on 2017-10-25). *used on: p. 82*
- [Her+16] Stefan Herbrechtsmeier et al. "AMiRo: A Modular & Customizable Open-Source Mini Robot Platform." In: *20th International Conference on System Theory, Control and Computing (ICSTCC)*. IEEE, 2016, pp. 687–692. DOI: [10.1109/ICSTCC.2016.7790746](https://doi.org/10.1109/ICSTCC.2016.7790746). *used on: p. 68*
- [HJ15] Reiner Hähnle and Einar Broch Johnsen. "Designing Resource-Aware Cloud Applications." In: *Computer* 48 (6 2015), pp. 72–75. ISSN: 0018-9162. DOI: [10.1109/MC.2015.172](https://doi.org/10.1109/MC.2015.172). *used on: p. 33*
- [HK06] Jiawei Han and Micheline Kamber. *Data Mining. Concepts and Techniques*. 2nd ed. The Morgan Kaufmann series in data management systems. Amsterdam: Morgan Kaufmann Publishers Inc., 2006. ISBN: 978-1-55860-901-3. *used on: p. 158*
- [HLN10] Veit Hoffmann, Horst Lichter, and Alexander Nyßen. "Processes and Practices for Quality Scientific Software Projects." In: *Proceedings of the 3rd International Workshop on Academic Software Development Tools*. 2010, pp. 95–108. *used on: p. 5*
- [Hol10] Michael Holzheu. *taskstats: Enhancements for precise accounting*. Linux Kernel Mailing List. 2010. URL: <https://lkml.org/lkml/2010/11/11/275> (visited on 2017-10-25). *used on: p. 81*
- [Hoo+15] André van Hoorn et al. "Automatic Extraction of Probabilistic Workload Specifications for Load Testing Session-Based Application Systems." In: *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*. Ed. by Moshe Haviv et al. ICST, 2015, pp. 139–146. DOI: [10.4108/icst.valuetools.2014.258171](https://doi.org/10.4108/icst.valuetools.2014.258171). *used on: p. 127*
- [HTF09] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning. Data Mining, Inference and Prediction*. 2nd ed. Springer series in statistics. New York: Springer, 2009. ISBN: 978-0-387-84857-0. DOI: [10.1007/978-0-387-84858-7](https://doi.org/10.1007/978-0-387-84858-7). *used on: p. 157*
- [HTW12] Guoqiang Hu, Wee Tay, and Yonggang Wen. "Cloud robotics: Architecture, Challenges and Applications." In: *IEEE Network* 26 (3 2012), pp. 21–28. ISSN: 0890-8044. DOI: [10.1109/MNET.2012.6201212](https://doi.org/10.1109/MNET.2012.6201212). *used on: p. 36*
- [IB97] Rolf Isermann and Peter Ballé. "Trends in the application of model-based fault detection and diagnosis of technical processes." In: *Control Engineering Practice* 5 (5 1997), pp. 709–719. ISSN: 0967-0661. DOI: [10.1016/S0967-0661\(97\)00053-1](https://doi.org/10.1016/S0967-0661(97)00053-1). *used on: pp. 12, 14, 16, 246*
- [IET14] IETF. *HTTP/1.1 Semantics and Content*. RFC 7231. 2014. URL: <https://tools.ietf.org/html/rfc7231> (visited on 2018-01-03). *used on: p. 56*



- [IET98] IETF. *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396. 1998. URL: <https://www.ietf.org/rfc/rfc2396.txt> (visited on 2016-09-08). used on: p. 57
- [IHE15] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodríguez, and Erik Elmroth. “Performance Anomaly Detection and Bottleneck Identification.” In: *ACM Computing Surveys* 48 (1 2015), pp. 1–35. ISSN: 0360-0300. DOI: [10.1145/2791120](https://doi.org/10.1145/2791120). used on: pp. 7, 8, 10, 17–19
- [IIo4] ISO and IEC. *Data elements and interchange formats — Information interchange — Representation of dates and times*. ISO/IEC 8601:2004. ISO, 2004. used on: p. 117
- [II12] ISO and IEC. *Information technology – Object Management Group – Common Object Request Broker Architecture (CORBA) – Part 3: Components*. ISO/IEC 19500-3. Geneva, Switzerland: ISO, 2012. used on: p. 44
- [IIIo8] IEEE, ISO, and IEC. *Standard for Information Technology—Portable Operating System Interface (POSIX®)*. IEEE 1003.1-2008. Version 7. New York, NY, USA: IEEE, 2008. used on: p. 82
- [III10] ISO, IEC, and IEEE. *Systems and software engineering – Vocabulary*. ISO/IEC/IEEE 24765. Piscataway, NJ, USA: IEEE, 2010. DOI: [10.1109/IEEESTD.2010.5733835](https://doi.org/10.1109/IEEESTD.2010.5733835). used on: pp. 13, 14, 16, 17, 44, 248–250
- [Iñi+12] Pablo Iñigo-Blasco et al. “Robotics software frameworks for multi-agent robotic systems development.” In: *Robotics and Autonomous Systems* 60 (6 2012), pp. 803–821. ISSN: 0921-8890. DOI: [10.1016/j.robot.2012.02.004](https://doi.org/10.1016/j.robot.2012.02.004). used on: p. 48
- [Ise06] Rolf Isermann. *Fault-Diagnosis Systems. An Introduction from Fault Detection to Fault Tolerance*. Berlin, Heidelberg, and New York: Springer, 2006. ISBN: 978-3-540-24112-6. DOI: [10.1007/3-540-30368-5](https://doi.org/10.1007/3-540-30368-5). used on: pp. 12–15, 17, 246, 250
- [ITUo8] ITU-T. *Definitions of terms related to quality of service*. ITU-T E.800. Recommendation. 2008. URL: <https://www.itu.int/rec/T-REC-E.800-200809-I> (visited on 2017-05-17). used on: p. 18
- [Jan+10] Choulsoo Jang et al. “OPRoS: A New Component-Based Robot Software Platform.” In: *ETRI Journal* 32 (5 2010), pp. 646–656. ISSN: 1225-6463. DOI: [10.4218/etrij.10.1510.0138](https://doi.org/10.4218/etrij.10.1510.0138). used on: pp. 46, 48
- [Jay+12] Deepal Jayasinghe et al. “Expertus. A Generator Approach to Automate Performance Testing in IaaS Clouds.” In: *Proceedings of the 2012 Fifth International Conference on Cloud Computing*. Ed. by Rong Chang. IEEE, 2012, pp. 115–122. DOI: [10.1109/CLOUD.2012.98](https://doi.org/10.1109/CLOUD.2012.98). used on: p. 126
- [JED13] Hengle Jiang, Sebastian Elbaum, and Carrick Detweiler. “Reducing failure rates of robotic systems through inferred invariants monitoring.” In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013, pp. 1899–1906. DOI: [10.1109/IROS.2013.6696608](https://doi.org/10.1109/IROS.2013.6696608). used on: pp. 141, 145

- [JED16] Hengle Jiang, Sebastian Elbaum, and Carrick Detweiler. “Inferring and monitoring invariants in robotic systems.” In: *Autonomous Robots* (2016). ISSN: 0929-5593. DOI: [10.1007/s10514-016-9576-y](https://doi.org/10.1007/s10514-016-9576-y). used on: pp. 176, 178
- [Jen14] Mike Jensen. *Reliability vs Robustness*. Mentor. 2014. URL: <http://blogs.mentor.com/mikej/blog/2014/10/07/reliability-vs-robustness/> (visited on 2017-05-08). used on: p. 14
- [JH15] Zhen Ming Jiang and Ahmed E. Hassan. “A Survey on Load Testing of Large-Scale Software Systems.” In: *IEEE Transactions on Software Engineering* 41 (11 2015), pp. 1091–1118. ISSN: 0098-5589. DOI: [10.1109/TSE.2015.2445340](https://doi.org/10.1109/TSE.2015.2445340). used on: pp. 19, 106, 107, 109, 117
- [Jin+12] Guoliang Jin et al. “Understanding and detecting real-world performance bugs.” In: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. Ed. by Jan Vitek, Lin Haibo, and Frank Tip. ACM Special Interest Group on Programming Languages. New York, NY, USA: ACM, 2012, p. 77. DOI: [10.1145/2254064.2254075](https://doi.org/10.1145/2254064.2254075). used on: pp. 19, 21, 24
- [JMC03] Merijn de Jonge, Johan Muskens, and Michel Chaudron. “Scenario-based prediction of run-time resource consumption in component-based software systems.” In: *6th ICSE workshop on component-based software engineering: automated reasoning and prediction*. Ed. by Ivica Crnkovic et al. 2003. used on: p. 34
- [JST12] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. “Modeling Resource-Aware Virtualized Applications for the Cloud in Real-Time ABS.” In: *Formal Methods and Software Engineering*. Ed. by Toshiaki Aoki and Kenji Taguchi. Lecture Notes in Computer Science 7635. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 71–86. DOI: [10.1007/978-3-642-34281-3\\_8](https://doi.org/10.1007/978-3-642-34281-3_8). used on: p. 33
- [JXS] Windy Road Technology. *JUnit-Schema*. 2016. URL: <https://github.com/windyroad/JUnit-Schema> (visited on 2017-11-28). used on: p. 119
- [Kar+11] Sertac Karaman et al. “Anytime Motion Planning using the RRT\*.” In: *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 1478–1483. DOI: [10.1109/ICRA.2011.5980479](https://doi.org/10.1109/ICRA.2011.5980479). used on: p. 36
- [KBT05] Tomáš Kalibera, Lubomír Bulej, and Petr Tůma. “Automated Detection of Performance Regressions: The Mono Experience.” In: *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Los Alamitos, California: IEEE Computer Society, 2005, pp. 183–190. DOI: [10.1109/MASCOTS.2005.18](https://doi.org/10.1109/MASCOTS.2005.18). used on: p. 19
- [KH04] Nathan Koenig and Andrew Howard. “Design and use paradigms for Gazebo, an open-source multi-robot simulator.” In: *2004 IEEE International Conference on Robotics and Automation*. Piscataway, NJ: IEEE, 2004, pp. 2149–2154. DOI: [10.1109/ROAS.2004.1389727](https://doi.org/10.1109/ROAS.2004.1389727). used on: p. 46

- [Kha+15] Eliahu Khalastchi et al. "Online data-driven anomaly detection in autonomous robots." In: *Knowledge and Information Systems* 43 (3 2015), pp. 657–688. ISSN: 0219-1377. DOI: [10.1007/s10115-014-0754-y](https://doi.org/10.1007/s10115-014-0754-y). used on: pp. 32, 176
- [KK17] Eliahu Khalastchi and Meir Kalech. "A sensor-based approach for fault detection and diagnosis for robotic systems." In: *Autonomous Robots* 84 (4 2017). ISSN: 0929-5593. DOI: [10.1007/s10514-017-9688-z](https://doi.org/10.1007/s10514-017-9688-z). used on: p. 176
- [KKS14] Casey Kennington, Spyros Kousidis, and David Schlangen. "InproTKs: A Toolkit for Incremental Situated Processing." In: *Proceedings of the 15th Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL)*. Ed. by Kallirroi Georgila et al. Association for Computational Linguistics, 2014, pp. 84–88. used on: p. 68
- [KLo8] Florian Knorn and Douglas J. Leith. "Adaptive Kalman Filtering for anomaly detection in software appliances." In: *IEEE INFOCOM workshops 2008*. Piscataway, NJ: IEEE, 2008, pp. 1–6. DOI: [10.1109/INFOCOM.2008.4544581](https://doi.org/10.1109/INFOCOM.2008.4544581). used on: p. 177
- [Kle07] Roland Kletzing. *[PATCH] Documentation for io-accounting / reporting via procfs*. 2007. URL: <https://github.com/torvalds/linux/commit/f9c99463b0cd05603d125c915e2886d55a686b82>. used on: p. 229
- [Kou+10] Samuel Kounev et al. "Towards Self-aware performance and resource management in modern service-oriented systems." In: *2010 IEEE International Conference on Services Computing*. IEEE, 2010, pp. 621–624. DOI: [10.1109/SCC.2010.94](https://doi.org/10.1109/SCC.2010.94). used on: p. 34
- [Kou+14] Spyros Kousidis et al. "A Multimodal In-Car Dialogue System That Tracks The Driver's Attention." In: *Proceedings of the 16th International Conference on Multimodal Interaction - ICMI '14*. Ed. by Albert Ali Salah et al. New York, New York, USA: ACM Press, 2014, pp. 26–33. DOI: [10.1145/2663204.2663244](https://doi.org/10.1145/2663204.2663244). used on: p. 68
- [Koz10] Heiko Koziolk. "Performance evaluation of component-based software systems: A survey." In: *Performance Evaluation* 67 (8 2010), pp. 634–658. ISSN: 0166-5316. DOI: [10.1016/j.peva.2009.07.007](https://doi.org/10.1016/j.peva.2009.07.007). used on: pp. 8, 18, 34
- [Kra09] Sacha Krakowiak. *Middleware Architecture with Patterns and Frameworks*. Unpublished. 2009. URL: <http://lig-membres.imag.fr/krakowia/Files/MW-Book/main-onebib.pdf> (visited on 2017-08-02). used on: p. 45
- [Krö+14] Manfred Kröhnert et al. "Resource Prediction for Humanoid Robots." In: *First Workshop on Resource awareness and adaptivity in multi-core computing*. Ed. by Frank Hannig and Jürgen Teich. Paderborn, Germany, 2014, pp. 22–28. used on: p. 37
- [Krö17] Manfred Kröhnert. *A Contribution to Resource-Aware Architectures for Humanoid Robots*. Vol. 1. Karlsruhe Series on Humanoid Robotics. Karlsruhe, Germany: KIT Scientific Publishing, 2017. ISBN: 978-3-7315-0632-4. DOI: [10.5445/KSP/1000065884](https://doi.org/10.5445/KSP/1000065884). used on: pp. 31, 36, 37

- [KSWo8] Alexander Kleiner, Gerald Steinbauer, and Franz Wotawa. "Towards Automated Online Diagnosis of Robot Navigation Software." In: *Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by Stefano Carpin et al. Lecture Notes in Artificial Intelligence 5325. Berlin, Heidelberg: Springer, 2008, pp. 159–170. DOI: [10.1007/978-3-540-89076-8\\_18](https://doi.org/10.1007/978-3-540-89076-8_18). used on: p. 176
- [Kub17] *The Kubernetes resource model*. Kubernetes. 2017-02-24. URL: <https://github.com/kubernetes/community/blob/e0cf34381e0842addf590bc43e62d669c25164ed/contributors/design-proposals/resources.md> (visited on 2017-04-26). used on: pp. 8, 9
- [Lap95] Jean-Claude Laprie. "Dependable Computing and Fault Tolerance: Concepts and Terminology." In: *The Twenty-Fifth International Symposium on Fault-Tolerant Computing. Highlights from Twenty-Five Years*. Los Alamitos, California: IEEE Computer Society, 1995. DOI: [10.1109/FTCSH.1995.532603](https://doi.org/10.1109/FTCSH.1995.532603). used on: pp. 12–14, 250
- [Laz16] Maria Lazarte. *Robots and humans can work together with new ISO guidance*. ISO. 2016. URL: <https://www.iso.org/news/2016/03/Ref2057.html> (visited on 2018-03-09). used on: p. 3
- [LF14] Jamers Lewis and Martin Fowler. *Microservices. a definition of this new architectural term*. 2014. URL: <https://martinfowler.com/articles/microservices.html> (visited on 2017-07-28). used on: pp. 44, 49, 248
- [Lik+05] Maxim Likhachev et al. "Anytime dynamic A\*: an anytime, replanning algorithm." In: *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling*. Ed. by Susanne Biundo. Menlo Park, California: AAAI Press, 2005, pp. 262–271. used on: p. 36
- [Lim+10] Jae-Hee Lim et al. "An Automated Test Method for Robot Platform and Its Components." In: *International Journal of Software Engineering and Its Applications* 4 (3 2010), pp. 9–18. ISSN: 1738-9984. used on: p. 105
- [Lino6] *Per-task statistics interface*. Linux. 2006. URL: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain/Documentation/accounting/taskstats.txt?id=4ae0edc21b152c126e4a8c94ad5391f8ea051b31> (visited on 2017-05-12). used on: pp. 10, 81
- [Lip06] The Linux Information Project. *Vendor Lock-in Definition*. 2006. URL: [http://www.linfo.org/vendor\\_lockin.html](http://www.linfo.org/vendor_lockin.html) (visited on 2017-08-14). used on: p. 251
- [LKK10] Raz Lin, Eliahu Khalastchi, and Gal A. Kaminka. "Detecting anomalies in unmanned vehicles using the Mahalanobis distance." In: *2010 IEEE International Conference on Robotics and Automation (ICRA 2010)*. IEEE, 2010, pp. 3038–3044. DOI: [10.1109/ROBOT.2010.5509781](https://doi.org/10.1109/ROBOT.2010.5509781). used on: p. 178

- [Loh+09] Manja Lohse et al. "Systemic interaction analysis (SInA) in HRI." In: *Proceedings of the 4th ACM/IEEE International Conference on Human-Robot Interaction*. Ed. by François Michaud et al. New York, NY, USA: ACM, 2009, pp. 93–100. DOI: [10.1145/1514095.1514114](https://doi.org/10.1145/1514095.1514114). used on: p. 72
- [LSL12] Florian Lier, Simon Schulz, and Ingo Lütkebohle. "Continuous Integration for Iterative Validation of Simulated Robot Models." In: *Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by Itsuki Noda et al. Lecture Notes in Artificial Intelligence 7628. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 101–112. DOI: [10.1007/978-3-642-34327-8\\_12](https://doi.org/10.1007/978-3-642-34327-8_12). used on: p. 105
- [LSS11] Alex Lotz, Andreas Steck, and Christian Schlegel. "Runtime monitoring of robotics software components. Increasing robustness of service robotic systems." In: *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 285–290. DOI: [10.1109/ICAR.2011.6088591](https://doi.org/10.1109/ICAR.2011.6088591). used on: p. 38
- [Luc10] David Luckham. *The power of events. An introduction to complex event processing in distributed enterprise systems*. 6th ed. Boston, Mass.: Addison-Wesley, 2010. ISBN: 978-0-201-72789-0. used on: p. 56
- [MCo4] Johan Muskens and Michel Chaudron. "Prediction of Runtime Resource Consumption in Multi-task Component-Based Software Systems." In: *Component-Based Software Engineering*. Ed. by Ivica Crnkovic et al. Lecture Notes in Computer Science 3054. Berlin and Heidelberg: Springer, 2004, pp. 162–177. DOI: [10.1007/978-3-540-24774-6\\_16](https://doi.org/10.1007/978-3-540-24774-6_16). used on: pp. 8, 9, 34
- [McCo4] Steve McConnell. *Code Complete. A practical handbook of software construction*. 2nd ed. Redmond, WA: Microsoft Press, 2004. ISBN: 978-0-7356-1967-8. used on: p. 24
- [McK10] Wes McKinney. "Data Structures for Statistical Computing in Python." In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stefan van der Walt and Jarrod Millman. 2010, pp. 51–56. used on: pp. 117, 147
- [Men+12] Shicong Meng et al. "Resource-Aware Application State Monitoring." In: *IEEE Transactions on Parallel and Distributed Systems* 23 (12 2012), pp. 2315–2329. ISSN: 1045-9219. DOI: [10.1109/TPDS.2012.82](https://doi.org/10.1109/TPDS.2012.82). used on: pp. 31, 32
- [Mey+15] Sebastian Meyer zu Borgsen et al. *ToBI - Team of Bielefeld: The Human-Robot Interaction System for RoboCup@Home 2015*. Tech. rep. Bielefeld University, 2015. URL: [http://robocup2015.oss-cn-shenzhen.aliyuncs.com/TeamDescriptionPapers/RoboCup@Home/RoboCup\\_Symposium\\_2015\\_submission\\_96.pdf](http://robocup2015.oss-cn-shenzhen.aliyuncs.com/TeamDescriptionPapers/RoboCup@Home/RoboCup_Symposium_2015_submission_96.pdf) (visited on 2016-09-06). used on: pp. 68, 142
- [MFNo6] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. "YARP: yet another robot platform." In: *Journal on Advanced Robotics* 3 (1 2006), pp. 43–48. ISSN: 1729-8806. DOI: [10.5772/5761](https://doi.org/10.5772/5761). used on: pp. 46, 48

- [MG11] Peter Mell and Timothy Grance. *The NIST definition of cloud computing*. Tech. rep. 800-145. Gaithersburg, MD: National Institute of Standards and Technology, 2011. DOI: [10.6028/NIST.SP.800-145](https://doi.org/10.6028/NIST.SP.800-145). URL: <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf> (visited on 2017-06-28). *used on: pp. 247, 249*
- [MHC12] Sheheryar Malik, Fabrice Huet, and Denis Caromel. "RACS: A framework for Resource Aware Cloud computing." In: *International Conference for Internet Technology and Secured Transactions*. Piscataway, NJ: IEEE, 2012, pp. 680–687. *used on: p. 33*
- [MHH13] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. "Automatic detection of performance deviations in the load testing of Large Scale Systems." In: *35th International Conference on Software Engineering (ICSE)*. Ed. by David Notkin, Betty H. C. Cheng, and Klaus Pohl. Piscataway, NJ: IEEE, 2013, pp. 1012–1021. DOI: [10.1109/ICSE.2013.6606651](https://doi.org/10.1109/ICSE.2013.6606651). *used on: pp. 19, 106, 107, 109*
- [Mic17] Microsoft. *Component Object Model (COM)*. 2017. URL: [https://msdn.microsoft.com/library/ms680573\(VS.85\).aspx](https://msdn.microsoft.com/library/ms680573(VS.85).aspx) (visited on 2017-08-01). *used on: p. 44*
- [Mil11] Dubravko Miljkovi. "Fault Detection Methods: A Literature Survey." In: *Proceedings of the 34th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2011)*. Ed. by Petar Biljanovic. Piscataway, NJ: IEEE, 2011, pp. 750–755. *used on: p. 176*
- [MNW13] Jan Moringen, Arne Nordmann, and Sebastian Wrede. "A Cross-Platform Data Acquisition and Transformation Approach for Whole-Systems Experimentation – Status and Challenges." In: *European Robotics Forum 2013. Working Session on Infrastructure for Robot Analysis and Benchmarking*. 2013. *used on: p. 66*
- [Mol15] Ian Molyneaux. *The Art of Application Performance Testing. From Strategy to Tools*. 2nd ed. Theory in practice. Sebastopol, CA: O'Reilly, 2015. ISBN: 978-1-4919-0054-3. *used on: pp. 8, 10, 13, 17, 18, 31*
- [Mör+10] Thomas Mörwald et al. "BLORT - The Blocks World Robotic Vision Toolbox." In: *Best Practice in 3D Perception and Modeling for Mobile Manipulation*. in conjunction with ICRA 2010. 2010. *used on: p. 36*
- [MPo8] Philippe Martinet and Bruno Patin. "PROTEUS: A platform to organise transfer inside French robotic community." In: *3rd National Conference on Control Architectures of Robots*. 2008. *used on: p. 46*
- [MSOo1] Mark Mitchell, Alex Samuel, and Jeffrey Oldham. *Advanced Linux programming*. Indianapolis, Indianapolis, USA: New Riders Publishing, 2001. ISBN: 978-0-7357-1043-6. *used on: p. 10*
- [MT] *time(7) - Linux Programmer's Manual*. 2016. URL: <http://man7.org/linux/man-pages/man7/time.7.html> (visited on 2017-11-10). *used on: p. 225*

- [Mur12] Kevin Patrick Murphy. *Machine learning. A probabilistic perspective*. Adaptive computation and machine learning series. Cambridge, Mass.: MIT Press, 2012. ISBN: 978-0-262-01802-9. used on: p. 157
- [MwRes] *Definition of resource*. Merriam-Webster. URL: <https://www.merriam-webster.com/dictionary/resource> (visited on 2017-04-26). used on: p. 7
- [MWV14] Valiallah Monajjemi, Jens Wawerla, and Richard Vaughan. "Drums: A Middleware-Aware Distributed Robot Monitoring System." In: *11th Conference on Computer and Robot Vision*. Ed. by Dave Meger. IEEE, 2014, pp. 211–218. DOI: 10.1109/CRV.2014.36. used on: pp. 38, 193
- [MY13] Brandon Malone and Changhe Yuan. "Evaluating Anytime Algorithms for Learning Optimal Bayesian Networks." In: *Proceedings of the Twenty-Ninth Conference Conference on Uncertainty in Artificial Intelligence*. Ed. by Ann Nicholson and Padhraic Smyth. Corvallis, Oregon: AUAI Press, 2013, pp. 381–390. used on: p. 36
- [Nes07a] Issa A. D. Nesnas. "CLARATy: A Collaborative Software for Advancing Robotic Technologies." In: *2007 NASA Science Technology Conference*. NASA, 2007. used on: p. 35
- [Nes07b] Issa A. D. Nesnas. "The CLARATy Project. Coping with Hardware and Software Heterogeneity." In: *Software Engineering for Experimental Robotics*. Ed. by Davide Brugali. 30. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 31–70. ISBN: 978-3-540-68949-2. DOI: 10.1007/978-3-540-68951-5\_3. used on: p. 43
- [Ngu+12] Thanh H.D. Nguyen et al. "Automated detection of performance regressions using statistical process control techniques." In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. Ed. by David Kaeli and Jerry Rolia. New York, New York, USA: ACM, 2012, p. 299. DOI: 10.1145/2188286.2188344. used on: pp. 19, 107
- [Ngu12] Thanh H.D. Nguyen. "Using Control Charts for Detecting and Understanding Performance Regressions in Large Software." In: *IEEE Fifth International Conference on Software Testing, Verification and Validation*. Piscataway, NJ: IEEE, 2012, pp. 491–494. DOI: 10.1109/ICST.2012.133. used on: p. 106
- [Nor+16] Arne Nordmann et al. "A Survey on Domain-Specific Modeling and Languages in Robotics." In: *Journal of Software Engineering in Robotics* (2016). ISSN: 2035-3928. used on: p. 127
- [NR11] Esha D. Nerurkar and Stergios I. Roumeliotis. "Power-SLAM: A linear-complexity, anytime algorithm for SLAM." In: *The International Journal of Robotics Research* 30 (6 2011), pp. 772–788. ISSN: 0278-3649. DOI: 10.1177/0278364910390539. used on: p. 36

- [NRW12] Arne Nordmann, Matthias Rolf, and Sebastian Wrede. “Software Abstractions for Simulation and Control of a Continuum Robot.” In: *Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by Itsuki Noda et al. Lecture Notes in Artificial Intelligence 7628. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 113–124. DOI: [10.1007/978-3-642-34327-8\\_13](https://doi.org/10.1007/978-3-642-34327-8_13). used on: p. 68
- [NTY08] Dor Nir, Shmuel Tyszberowicz, and Amiram Yehudai. “Locating Regression Bugs.” In: *Hardware and Software: Verification and Testing*. Ed. by Karen Yorav. Lecture Notes in Computer Science 4899. Berlin, Heidelberg: Springer, 2008, pp. 218–234. DOI: [10.1007/978-3-540-77966-7\\_18](https://doi.org/10.1007/978-3-540-77966-7_18). used on: p. 17
- [NWS15] Arne Nordmann, Sebastian Wrede, and Jochen J. Steil. “Modeling of movement control architectures based on motion primitives using domain-specific languages.” In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 5032–5039. DOI: [10.1109/ICRA.2015.7139899](https://doi.org/10.1109/ICRA.2015.7139899). used on: p. 68
- [OMG11] OMG. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. formal/2011-06-02. 2011. URL: <http://www.omg.org/spec/MARTE/1.1/PDF> (visited on 2017-08-01). used on: p. 34
- [OMG13] OMG. *UML Testing Profile (UTP)*. formal/2013-04-03. 2013. URL: <http://www.omg.org/spec/UTP/1.2/PDF> (visited on 2017-10-17). used on: p. 126
- [Pai+15] Brooks Paige et al. “Asynchronous Anytime Sequential Monte Carlo.” In: *Advances in neural information processing systems 27*. Ed. by Zoubin Ghahramani et al. Neural Information Processing Systems Foundation. Red Hook, NY: Curran Associates, Inc., 2015, pp. 3410–3418. used on: p. 36
- [Pau+14] Johnny Paul et al. “Resource-Aware Programming for Robotic Vision.” In: *First Workshop on Resource awareness and adaptivity in multi-core computing*. Ed. by Frank Hannig and Jürgen Teich. Paderborn, Germany, 2014, pp. 8–13. used on: p. 36
- [Pcap17] *pcap - Packet Capture library*. 2017. URL: <http://www.tcpdump.org/manpages/pcap.3pcap.html> (visited on 2017-10-25). used on: p. 82
- [Pen11] Roger D. Peng. “Reproducible research in computational science.” In: *Science* 334 (6060 2011). PMC3383002 Journal Article, pp. 1226–1227. ISSN: 1095-9203. DOI: [10.1126/science.1213847](https://doi.org/10.1126/science.1213847). eprint: [22144613](https://arxiv.org/abs/22144613). used on: p. 71
- [Per16] Performance Co-Pilot. *Understanding measures of system-level processor performance*. 2016. URL: <http://pcp.io/docs/howto.cpuperf.html> (visited on 2017-11-10). used on: p. 225
- [Pet05] Ola Pettersson. “Execution monitoring in robotics: A survey.” In: *Robotics and Autonomous Systems* 53 (2 2005), pp. 73–88. ISSN: 0921-8890. DOI: [10.1016/j.robot.2005.09.004](https://doi.org/10.1016/j.robot.2005.09.004). used on: pp. 175, 176



- [Pet12] Benjamin Peterson. *Second release candidates for Python 2.6.8, 2.7.3, 3.1.5, and 3.2.3*. Python-announce-list. 2012. URL: <https://mail.python.org/pipermail/python-announce-list/2012-March/009394.html> (visited on 2018-01-03). *used on: p. 153*
- [PGTo3] Joelle Pineau, Geoff Gordon, and Sebastian Thrun. "Point-based value iteration: An anytime algorithm for POMDPs." In: *Proceedings of the 18th international joint conference on Artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, pp. 1025–1030. *used on: p. 36*
- [Pit+11] Karola Pitsch et al. "Attitude of German museum visitors towards an interactive art guide robot." In: *Proceedings of the 6th ACM/IEEE International Conference on Human-Robot Interaction*. Ed. by Aude Billard et al. New York, NY, USA: ACM, 2011, pp. 227–228. DOI: [10.1145/1957656.1957744](https://doi.org/10.1145/1957656.1957744). *used on: p. 71*
- [PKK12] Yu-Sik Park, Hyung-Min Koo, and In-Young Ko. "A task-based and resource-aware approach to dynamically generate optimal software architecture for intelligent service robots." In: *Software: Practice and Experience* 42 (5 2012), pp. 519–541. ISSN: 00380644. DOI: [10.1002/spe.1074](https://doi.org/10.1002/spe.1074). *used on: pp. 31, 37*
- [PP17] *pickle* — Python object serialization — Python 3.6.3 documentation. 2017. URL: <https://docs.python.org/3/library/pickle.html> (visited on 2017-11-14). *used on: p. 96*
- [Proc17] *proc(5) - Linux manual page*. 2017. URL: <http://man7.org/linux/man-pages/man5/proc.5.html> (visited on 2017-05-12). *used on: pp. 10, 80, 225, 226, 229*
- [PWWo6] Bernhard Peischl, Jörg Weber, and Franz Wotawa. "Runtime Fault Detection and Localization in Component-oriented Software Systems." In: *17th International Workshop on Principles of Diagnosis (DX'06)*. 2006, pp. 203–210. *used on: p. 141*
- [Qui+09] Morgan Quigley et al. "ROS: an open-source Robot Operating System." In: *ICRA Workshop on Open Source Software*. 2009. *used on: pp. 38, 46*
- [Rat+14] Christoph Rathfelder et al. "Modeling event-based communication in component-based software architectures for performance predictions." In: *Software & Systems Modeling* 13 (4 2014), pp. 1291–1317. ISSN: 1619-1366. DOI: [10.1007/s10270-013-0316-x](https://doi.org/10.1007/s10270-013-0316-x). *used on: p. 34*
- [RC11] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)." In: *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011. DOI: [10.1109/ICRA.2011.5980567](https://doi.org/10.1109/ICRA.2011.5980567). *used on: p. 46*
- [Rey+06] Patrick Reynolds et al. "Pip: Detecting the Unexpected in Distributed Systems." In: *NSDI'06 Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*. Ed. by Larry Peterson and Timothy Roscoe. San Jose, CA: USENIX Association Berkeley, 2006. *used on: pp. 18, 33*

- [Rol+15] Matthias Rolf et al. "A multi-level control architecture for the bionic handling assistant." In: *Advanced Robotics* 29 (13 2015), pp. 847–859. ISSN: 0169-1864. DOI: [10.1080/01691864.2015.1037793](https://doi.org/10.1080/01691864.2015.1037793). used on: p. 68
- [ROS14] *ROS Users of the World*. 2014? URL: <http://metrorobots.com/rosmap.html> (visited on 2017-08-07). Year deduced from first GIT commit. used on: p. 47
- [Ros16] Rami Rosen. *Understanding the new control groups API*. 2016. URL: <https://lwn.net/Articles/679786/> (visited on 2017-10-25). used on: p. 82
- [RSS07] Szabolcs Rozsnyai, Josef Schiefer, and Alexander Schatten. "Concepts and models for typing events for event-based systems." In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*. Ed. by Hans-Arno Jacobsen. New York, New York, USA: ACM Press, 2007, pp. 62–70. DOI: [10.1145/1266894.1266904](https://doi.org/10.1145/1266894.1266904). used on: p. 55
- [RTT] *The Orocos Real-Time Toolkit*. URL: <http://www.orocos.org/rtt> (visited on 2017-08-07). used on: p. 47
- [RW97] David S. Rosenblum and Alexander L. Wolf. "A design framework for internet-scale event observation and notification." In: *Software Engineering — ESEC/FSE '97*. Ed. by Mehdi Jazayeri and Helmut Schauer. Lecture Notes in Computer Science 1301. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 344–360. DOI: [10.1007/3-540-63531-9\\_24](https://doi.org/10.1007/3-540-63531-9_24). used on: p. 55
- [Scho6] Christian Schlegel. "Communication Patterns as Key Towards Component-Based Robotics." In: *International Journal of Advanced Robotic Systems* 3 (1 2006), pp. 49–54. ISSN: 1729-8806. DOI: [10.5772/5759](https://doi.org/10.5772/5759). used on: p. 49
- [SGK17] Sebastian Schneider, Michael Goerlich, and Franz Kummert. "A framework for designing socially assistive robot interactions." In: *Cognitive Systems Research* 43 (2017), pp. 301–312. ISSN: 1389-0417. DOI: [10.1016/j.cogsys.2016.09.008](https://doi.org/10.1016/j.cogsys.2016.09.008). used on: p. 68
- [Sha+15] Wei Shang et al. "Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters." In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. Ed. by Lizy K. John et al. New York, New York, USA: ACM, 2015, pp. 15–26. DOI: [10.1145/2668930.2688052](https://doi.org/10.1145/2668930.2688052). used on: pp. 19, 107, 118, 121–123
- [She+09] Kai Shen et al. "Reference-driven performance anomaly identification." In: *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. Ed. by John Douceur et al. Performance Evaluation Review. ACM Special Interest Group on Measurement and Evaluation. New York, NY, USA: ACM, 2009, pp. 85–96. DOI: [10.1145/1555349.1555360](https://doi.org/10.1145/1555349.1555360). used on: pp. 18, 178

- [Shi+09] Shuichi Shimizu et al. "Platform-independent modeling and prediction of application resource usage characteristics." In: *Journal of Systems and Software* 82 (12 2009), pp. 2117–2127. ISSN: 0164-1212. DOI: [10.1016/j.jss.2009.07.020](https://doi.org/10.1016/j.jss.2009.07.020). used on: pp. 9, 173
- [Sig+10] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: <https://research.google.com/archive/papers/dapper-2010-1.pdf>. used on: p. 32
- [SL11] Sena Seneviratne and David C. Levy. "Task profiling model for load profile prediction." In: *Future Generation Computer Systems* 27 (3 2011), pp. 245–255. ISSN: 0167-739X. DOI: [10.1016/j.future.2010.09.004](https://doi.org/10.1016/j.future.2010.09.004). used on: p. 173
- [SLB13] Sena Seneviratne, David C. Levy, and Rajkumar Buyya. *A Taxonomy of Performance Prediction Systems in the Parallel and Distributed Computing Grids*. 2013. URL: <https://arxiv.org/pdf/1307.2380v2> (visited on 2016-07-13). used on: pp. 9, 173
- [SMWo6] Gerald Steinbauer, Martin Mörth, and Franz Wotawa. "Real-Time Diagnosis and Repair of Faults of Robot Control Software." In: *RoboCup 2005: Robot Soccer World Cup IX*. Ed. by Ansgar Bredendfeld et al. Lecture Notes in Artificial Intelligence 4020. Berlin, Heidelberg: Springer, 2006, pp. 13–23. DOI: [10.1007/11780519\\_2](https://doi.org/10.1007/11780519_2). used on: p. 175
- [SS11] Andreas Steck and Christian Schlegel. "Towards Quality of Service and Resource Aware Robotic Systems through Model-Driven Software Development." In: *Proceedings of the first international workshop on domain-specific languages for robotic systems (DSLRob 2010)*. Ed. by Ulrik P. Schultz, Serge Stinckwich, and Mikal Ziane. 2011. used on: pp. 31, 39
- [Sta15] *linux - /proc/[pid]/stat refresh period*. Stack Overflow. 2015. URL: <https://stackoverflow.com/questions/31219317/proc-pid-stat-refresh-period> (visited on 2017-05-11). used on: pp. 11, 229
- [Ste13] Gerald Steinbauer. "A Survey about Faults of Robots Used in RoboCup." In: *RoboCup 2012: Robot Soccer World Cup XVI*. Ed. by Xiaoping Chen et al. Lecture Notes in Computer Science 7500. Berlin, Heidelberg: Springer, 2013, pp. 344–355. used on: pp. 21, 24, 28, 105, 175
- [Sun+16] Yu Sun et al. "ROAR. A QoS-oriented modeling framework for automated cloud resource allocation and optimization." In: *Journal of Systems and Software* 116 (2016), pp. 146–161. ISSN: 0164-1212. DOI: [10.1016/j.jss.2015.08.006](https://doi.org/10.1016/j.jss.2015.08.006). used on: p. 126
- [Sun97] Sun Microsystems. *JavaBeans*. Ed. by Graham Hamilton. Version 1.01-A. 1997. URL: <http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html> (visited on 2017-08-01). used on: p. 44
- [SW05] Gerald Steinbauer and Franz Wotawa. "Detecting and locating faults in the control software of autonomous mobile robots." In: *16th International Workshop on Principles of Diagnosis (DX-05)*. 2005, pp. 13–18. used on: pp. 32, 141

- [SW09] William Strunk and Elwyn Brooks White. *The Elements of Style*. New York, NY: Longman, 2009. ISBN: 978-0-205-30902-3. used on: p. 285
- [SW11] Frederic Siepmann and Sven Wachsmuth. "A Modeling Framework for Reusable Social Behavior." In: *Works-In-Progress Track at International Conference on Social Robotics (ICSR 2011)*. Ed. by Ravindra Da Silva and Dennis Reidsma. 2011. used on: pp. 142, 237
- [SW99] Christian Schlegel and Robert Wörtz. "The software framework SMARTSOFT for implementing sensorimotor systems." In: *Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients*. IEEE, 1999, pp. 1610–1616. DOI: 10.1109/IROS.1999.811709. used on: p. 46
- [Syd11] Michael J. Sydor. *APM Best Practices. Realizing Application Performance Management*. Berkeley, CA: Apress, 2011. ISBN: 978-1-4302-3141-7. used on: pp. 10, 18, 106
- [Szy03] Clemens Szyperski. *Component software. Beyond object-oriented programming*. In collab. with Dominik Gruntz and Stephan Murer. 2nd ed. Addison-Wesley component software series. London: Addison-Wesley, 2003. ISBN: 978-0-201-74572-6. used on: pp. 43, 44, 245
- [TFR13] Lukas Twardon, Andrea Finke, and Helge J. Ritter. "Exploiting eye-hand coordination. A novel approach to remote manipulation." In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013, pp. 5463–5468. DOI: 10.1109/IROS.2013.6697147. used on: p. 68
- [TGo8] Mirco Tribastone and Stephen Gilmore. "Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile." In: *Proceedings of the 7th international workshop on Software and performance*. Ed. by Alberto Avritzer, Elaine Weyuker, and Murray Woodside. New York, New York, USA: ACM Press, 2008, p. 67. DOI: 10.1145/1383559.1383569. used on: p. 34
- [The15] The Economist. *Style Guide. The Bestselling Guide to English Usage*. 11th ed. New York: PublicAffairs, 2015. ISBN: 978-1-61039-538-0. used on: p. 285
- [Thr+00] Sebastian Thrun et al. "Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot Minerva." In: *The International Journal of Robotics Research* 19 (11 2000), pp. 972–999. ISSN: 0278-3649. DOI: 10.1177/02783640022067922. used on: p. 36
- [TLI11] Florent Teichteil-Königsbuch, Charles Lesire, and Guillaume Infantes. "A generic framework for anytime execution-driven planning in robotics." In: *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 299–304. DOI: 10.1109/ICRA.2011.5980289. used on: p. 36

- [TMW10] Mirco Tribastone, Philip Mayer, and Martin Wirsing. “Performance Prediction of Service-Oriented Systems with Layered Queueing Networks.” In: *Leveraging Applications of Formal Methods, Verification, and Validation. Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science 6416. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 51–65. DOI: [10.1007/978-3-642-16561-0\\_12](https://doi.org/10.1007/978-3-642-16561-0_12). used on: p. 34
- [Top17] *top - display Linux processes*. 2017. URL: <http://man7.org/linux/man-pages/man1/top.1.html> (visited on 2017-11-10). used on: p. 229
- [TR16] Jay Trimble and George Rinker. “Open Source Next Generation Visualization Software for Interplanetary Missions.” In: *SpaceOps 2016 Conference*. Reston, Virginia: American Institute of Aeronautics and Astronautics, 2016. DOI: [10.2514/6.2016-2348](https://doi.org/10.2514/6.2016-2348). used on: p. 94
- [Tur16] James Turnbull. *The Art of Monitoring*. v1.0.0. 2016. ISBN: 978-0-9888202-4-1. used on: p. 32
- [Uen+06] Ken Ueno et al. “Anytime Classification Using the Nearest Neighbor Algorithm with Applications to Stream Mining.” In: *Sixth International Conference on Data Mining (ICDM’06)*. IEEE, 2006, pp. 623–632. DOI: [10.1109/ICDM.2006.21](https://doi.org/10.1109/ICDM.2006.21). used on: p. 36
- [UL07] Mark Utting and Bruno Legeard. *Practical model-based testing. A tools approach*. Safari Tech Books Online. Amsterdam: Elsevier, 2007. ISBN: 978-0-12-372501-1. used on: pp. 125–127
- [US 01] US Army Department of Defense. *Systems engineering fundamentals*. Washington D.C., 2001. ISBN: 978-1-4841-2083-5. used on: p. 247
- [VF11] Ghislain Verdier and Ariane Ferreira. “Adaptive Mahalanobis Distance and k-Nearest Neighbor Rule for Fault Detection in Semiconductor Manufacturing.” In: *IEEE Transactions on Semiconductor Manufacturing* 24 (1 2011), pp. 59–68. ISSN: 0894-6507. DOI: [10.1109/TSM.2010.2065531](https://doi.org/10.1109/TSM.2010.2065531). used on: p. 178
- [Vino5] Steve Vinoski. “A Time for Reflection.” In: *IEEE Internet Computing* 9 (1 2005), pp. 86–89. ISSN: 1089-7801. DOI: [10.1109/MIC.2005.3](https://doi.org/10.1109/MIC.2005.3). used on: p. 247
- [Voe13] Markus Voelter. “Language and IDE Modularization and Composition with MPS.” In: *Generative and Transformational Techniques in Software Engineering IV*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. 7680. Berlin, Heidelberg: Springer, 2013, pp. 383–430. ISBN: 978-3-642-35991-0. DOI: [10.1007/978-3-642-35992-7\\_11](https://doi.org/10.1007/978-3-642-35992-7_11). used on: p. 128
- [Vol+00] Richard Volpe et al. *CLARAty: Coupled Layer Architecture for Robotic Autonomy*. Tech. rep. Pasadena, California: Jet Propulsion Laboratory, California Institute of Technology, 2000. used on: pp. 9, 35

- [Vol+01] Richard Volpe et al. "The CLARAty architecture for robotic autonomy." In: *2001 IEEE Aerospace Conference Proceedings*. Ed. by Robert A. Profet. Piscataway, NJ: IEEE Operations Center, 2001, pp. 1/121–1/132. DOI: [10.1109/AERO.2001.931701](https://doi.org/10.1109/AERO.2001.931701). used on: p. 35
- [VP07] M. Valente and R. Palhares. "Collocation optimizations in an aspect-oriented middleware system." In: *Journal of Systems and Software* 80 (10 2007), pp. 1659–1666. ISSN: 0164-1212. DOI: [10.1016/j.jss.2007.01.033](https://doi.org/10.1016/j.jss.2007.01.033). used on: p. 58
- [Wan+16] Jiafu Wan et al. "Cloud Robotics: Current Status and Open Issues." In: *IEEE Access* 4 (2016), pp. 2797–2807. ISSN: 2169-3536. DOI: [10.1109/ACCESS.2016.2574979](https://doi.org/10.1109/ACCESS.2016.2574979). used on: pp. 35, 36
- [WC17] Ruffin White and Henrik I. Christensen. "ROS and Docker." In: *Robot Operating System (ROS). The Complete Reference (Volume 2)*. Ed. by Anis Koubâa. 707. Cham: Springer International Publishing, 2017, pp. 285–307. ISBN: 978-3-319-54926-2. DOI: [10.1007/978-3-319-54927-9\\_9](https://doi.org/10.1007/978-3-319-54927-9_9). used on: p. 50
- [WF13] Alexandra Wander and Roger Förstner. "Innovative Fault Detection, Isolation and Recovery Strategies On-Board Spacecraft: State of the Art and Research Challenges." In: *Deutscher Luft- und Raumfahrtkongress 2012*. Deutsche Gesellschaft für Luft- und Raumfahrt. Bonn: Deutsche Gesellschaft für Luft- und Raumfahrt - Lilienthal-Oberth e.V., 2013. used on: p. 16
- [Wig+17] Dennis Leroy Wigand et al. "Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case." In: *Journal of Software Engineering in Robotics* 8.1 (2017), pp. 45–64. ISSN: 2035-3928. used on: p. 127
- [Wik16a] *Computational resource*. Wikipedia. 2016-02-02. URL: <https://en.wikipedia.org/w/index.php?oldid=702911944> (visited on 2016-07-26). used on: p. 7
- [Wik16b] *Hardware performance counter*. Wikipedia. 2016-02-10. URL: <https://en.wikipedia.org/w/index.php?oldid=704224910> (visited on 2016-07-26). used on: p. 10
- [Wik16c] *Resource contention*. Wikipedia. 2016-12-20. URL: <https://en.wikipedia.org/w/index.php?oldid=755891262> (visited on 2017-04-26). used on: p. 8
- [Wik16d] *Starvation (computer science)*. Wikipedia. 2016-12-09. URL: [https://en.wikipedia.org/w/index.php?title=Starvation\\_\(computer\\_science\)&oldid=753792573](https://en.wikipedia.org/w/index.php?title=Starvation_(computer_science)&oldid=753792573) (visited on 2017-04-26). used on: p. 8
- [Wik16e] *System resource*. Wikipedia. 2016-06-22. URL: <https://en.wikipedia.org/w/index.php?oldid=726423842> (visited on 2016-07-26). used on: p. 7
- [Wik17] *Quality of service*. Wikipedia. 2017-05-17. URL: <https://en.wikipedia.org/w/index.php?oldid=778156644> (visited on 2017-05-17). used on: pp. 18, 249

- [Wit+06] Peter Wittenburg et al. "ELAN: a Professional Framework for Multimodality Research." In: *Fifth International Conference on Language Resources and Evaluation (LREC 2006)*. 2006, pp. 1556–1559. *used on: p. 77*
- [Woo+08] Timothy Wood et al. "Profiling and Modeling Resource Usage of Virtualized Applications." In: *Middleware 2008*. Ed. by Valérie Issarny and Richard E. Schantz. Programming and Software Engineering 5346. Berlin, Heidelberg: Springer, 2008, pp. 366–387. DOI: [10.1007/978-3-540-89856-6\\_19](https://doi.org/10.1007/978-3-540-89856-6_19). *used on: p. 173*
- [Wre+13] Sebastian Wrede et al. "A User Study on Kinesthetic Teaching of Redundant Robots in Task and Configuration Space." In: *Journal of Human-Robot Interaction* 2 (1 2013), pp. 56–81. ISSN: 21630364. DOI: [10.5898/JHRI.2.1.Wrede](https://doi.org/10.5898/JHRI.2.1.Wrede). *used on: p. 78*
- [Wre+17] Sebastian Wrede et al. "The Cognitive Service Robotics Apartment." In: *KI - Künstliche Intelligenz* 31 (3 2017), pp. 299–304. ISSN: 0933-1875. DOI: [10.1007/s13218-017-0492-x](https://doi.org/10.1007/s13218-017-0492-x). *used on: pp. 78, 99*
- [Wreo8] Sebastian Wrede. "An Information-Driven Architecture for Cognitive Systems Research." Technische Fakultät. Doctoral dissertation. Bielefeld: Bielefeld University, 2008. *used on: p. 5*
- [WS01] Rainer Weinreich and Johannes Sametinger. "Component Models and Component Services: Concepts and Principles." In: *Component-based software engineering. Putting the pieces together*. Ed. by George T. Heineman and William T. Councill. Boston, Mass.: Addison-Wesley, 2001. Chap. 3, pp. 33–48. ISBN: 978-0-201-70485-3. *used on: pp. 44, 59, 245*
- [WSS11] Marc-Florian Wendland, Ina Schieferdecker, and Markus Schacher. "Testen mit dem UML Testing Profile." German. In: *OBJEKTSpektrum (Testing 2011)*. *used on: p. 126*
- [WW06] Jörg Weber and Franz Wotawa. "Using AI Techniques for Fault Localization in Component-Oriented Software Systems." In: *MICAI 2006: Advances in Artificial Intelligence*. Ed. by David Hutchison et al. Lecture Notes in Computer Science 4293. Berlin, Heidelberg: Springer, 2006, pp. 1139–1149. DOI: [10.1007/11925231\\_109](https://doi.org/10.1007/11925231_109). *used on: pp. 176, 185*
- [Xio+15] Pengcheng Xiong et al. "SmartSLA. Cost-Sensitive Management of Virtualized Resources for CPU-Bound Database Services." In: *IEEE Transactions on Parallel and Distributed Systems* 26 (5 2015), pp. 1441–1451. ISSN: 1045-9219. DOI: [10.1109/TPDS.2014.2319095](https://doi.org/10.1109/TPDS.2014.2319095). *used on: p. 172*
- [Xue+15] Ji Xue et al. "PRACTISE: Robust Prediction of Data Center Time Series." In: *Proceedings of the 11th International Conference on Network and Service Management (CNSM)*. Ed. by Mauro Totonnesi et al. IEEE, 2015, pp. 126–134. DOI: [10.1109/CNSM.2015.7367348](https://doi.org/10.1109/CNSM.2015.7367348). *used on: p. 172*
- [YKZ16] Min Sang Yoon, Ahmed E. Kamal, and Zhengyuan Zhu. "Requests Prediction in Cloud with a Cyclic Window Learning Algorithm." In: *IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2016. DOI: [10.1109/GLOCOMW.2016.7849022](https://doi.org/10.1109/GLOCOMW.2016.7849022). *used on: p. 173*

- [YRS] *Use YARP to talk to ROS services.* URL: [http://www.yarp.it/yarp\\_with\\_ros\\_services.html](http://www.yarp.it/yarp_with_ros_services.html) (visited on 2017-08-23). *used on: p. 66*
- [YRT] *Writing code to talk to ROS topics.* URL: [http://www.yarp.it/yarp\\_with\\_ros\\_writing\\_code\\_topics.html](http://www.yarp.it/yarp_with_ros_writing_code_topics.html) (visited on 2017-08-14). *used on: pp. 52, 66*
- [Yu+15] Jingjin Yu et al. "Anytime planning of optimal schedules for a mobile sensing robot." In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2015, pp. 5279–5286. DOI: [10.1109/IROS.2015.7354122](https://doi.org/10.1109/IROS.2015.7354122). *used on: p. 36*
- [ZAH12] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. "A qualitative study on performance bugs." In: *9th IEEE Working Conference on Mining Software Repositories (MSR), 2012*. Ed. by Michele Lanza, Massimiliano Di Penta, and Tao Xie. Piscataway, NJ: IEEE, 2012, pp. 199–208. DOI: [10.1109/MSR.2012.6224281](https://doi.org/10.1109/MSR.2012.6224281). *used on: pp. 19, 21*
- [Žal13] Žmicier Žaležničenka. "Automated detection of performance regressions in web applications using association rule mining." Master's thesis. Delft: Delft University of Technology, 2013. *used on: p. 107*
- [Zam+13] Safdar Zaman et al. "An Integrated Model-Based Diagnosis and Repair Architecture for ROS-Based Robot Systems." In: *2013 IEEE International Conference on Robotics and Automation (ICRA 2013)*. Piscataway, NJ: IEEE, 2013, pp. 482–489. DOI: [10.1109/ICRA.2013.6630618](https://doi.org/10.1109/ICRA.2013.6630618). *used on: pp. 175, 176*
- [Zha+15] Qi Zhang et al. "PRISM: Fine-Grained Resource-Aware Scheduling for MapReduce." In: *IEEE Transactions on Cloud Computing* 3 (2 2015), pp. 182–194. ISSN: 2168-7161. DOI: [10.1109/TCC.2014.2379096](https://doi.org/10.1109/TCC.2014.2379096). *used on: p. 31*
- [Zil96] Shlomo Zilberstein. "Using Anytime Algorithms in Intelligent Systems." In: *AI Magazine* 17 (3 1996), pp. 73–83. *used on: p. 36*
- [ZKV04] Uwe Zdun, M. Kircher, and M. Volter. "Remoting patterns: design reuse of distributed object middleware solutions." In: *IEEE Internet Computing* 8 (6 2004), pp. 60–68. ISSN: 1089-7801. DOI: [10.1109/MIC.2004.70](https://doi.org/10.1109/MIC.2004.70). *used on: p. 59*
- [ZVo8] Michael Zillich and Markus Vincze. "Anytimeness avoids parameters in detecting closed convex polygons." In: *The Sixth IEEE Computer Society Workshop on Perceptual Organization in Computer Vision*. in Conjunction with IEEE CVPR 2008. IEEE, 2008. DOI: [10.1109/CVPRW.2008.4562981](https://doi.org/10.1109/CVPRW.2008.4562981). *used on: p. 36*

## SOFTWARE PACKAGES

- [ACO] *academic-keyword-occurrence.* URL: <https://github.com/Pol-d87/academic-keyword-occurrence> (visited on 2017-08-02). *used on: pp. 45, 46*
- [Apache] *Apache.* Apache Software Foundation. URL: <https://httpd.apache.org> (visited on 2017-06-29). *used on: p. 33*



- [Cacti] *Cacti*. 2017. URL: <https://www.cacti.net/> (visited on 2017-11-14). used on: p. 94
- [Cold] *collectd. The system statistics collection daemon*. URL: <https://collectd.org/> (visited on 2017-10-26). used on: pp. 79, 84
- [Coll] *collectl*. URL: <http://collectl.sourceforge.net/> (visited on 2017-10-26). used on: p. 83
- [Crypt] *cryptsetup*. URL: <https://gitlab.com/cryptsetup/cryptsetup> (visited on 2017-11-10). used on: p. 227
- [Cyclo] *Cyclotron*. URL: <http://www.cyclotron.io/> (visited on 2017-11-14). used on: p. 94
- [DebT] *debtree*. URL: <https://collab-maint.alioth.debian.org/debtree/> (visited on 2017-08-14). used on: p. 53
- [Diam] *Diamond*. URL: <https://github.com/python-diamond/Diamond> (visited on 2017-10-26). used on: pp. 79, 84
- [Ecl] *Eclipse*. URL: <https://www.eclipse.org/> (visited on 2017-12-04). used on: p. 135
- [ELAN] *ELAN*. URL: <https://tla.mpi.nl/tools/tla-tools/elan/> (visited on 2017-08-29). used on: pp. 77, 147
- [FFmpeg] *FFmpeg*. URL: <https://ffmpeg.org> (visited on 2017-08-29). used on: p. 77
- [Full] *fullerite*. URL: <https://github.com/Yelp/fullerite> (visited on 2017-11-06). used on: pp. 79, 84
- [Ganglia] *Ganglia*. 2016. URL: <http://ganglia.sourceforge.net/> (visited on 2017-11-14). used on: p. 94
- [Gatling] *Gatling*. URL: <http://gatling.io> (visited on 2016-08-30). used on: pp. 106, 126, 127
- [Gdash] *gdash*. URL: <https://github.com/ripienaar/gdash> (visited on 2017-11-14). used on: p. 94
- [GDB] *GDB. The GNU Project Debugger*. URL: <https://www.gnu.org/software/gdb/> (visited on 2017-06-08). used on: pp. 23, 28
- [Gla] *Glances*. URL: <https://nicolargo.github.io/glances/> (visited on 2017-10-25). used on: p. 83
- [Grafana] *Grafana*. URL: <https://grafana.com/> (visited on 2017-11-14). used on: pp. 94, 95, 97, 149
- [Graph] *Graphite*. URL: <https://graphiteapp.org> (visited on 2017-07-04). used on: pp. 38, 94–96
- [Grinder] *The Grinder*. URL: <http://grinder.sf.net> (visited on 2017-10-22). used on: p. 106
- [Gst] *GStreamer*. URL: <https://gstreamer.freedesktop.org> (visited on 2017-08-18). used on: pp. 52, 76
- [HAProxy] *HAProxy*. URL: <http://www.haproxy.org> (visited on 2017-06-29). used on: p. 33
- [Htop] *htop. An interactive process viewer for Unix*. URL: <http://hisham.hm/htop/> (visited on 2017-10-25). used on: p. 83

- [IDB] *InfluxDB*. URL: <https://github.com/influxdata/influxdb> (visited on 2017-11-14). *used on: pp. 94–96*
- [Iotop] *iotop*. URL: <http://guichaz.free.fr/iotop/> (visited on 2017-10-25). *used on: p. 83*
- [iPCM] *Processor Counter Monitor (PCM)*. URL: <https://github.com/opcm/pcm> (visited on 2017-04-28). *used on: p. 10*
- [Jenk] *Jenkins*. URL: <https://jenkins.io/> (visited on 2017-11-28). *used on: p. 119*
- [Jet] *MPS*. Version 2017.2. JetBrains. URL: <https://www.jetbrains.com/mps/> (visited on 2017-10-19). *used on: pp. 127–130, 132, 133, 136*
- [JMeter] *Apache JMeter*. Apache Software Foundation. URL: <https://jmeter.apache.org/> (visited on 2016-08-08). *used on: pp. 106, 126, 127*
- [JUnit] *JUnit*. 2017. URL: <http://junit.org> (visited on 2017-11-28). *used on: p. 119*
- [KDB] *KairosDB*. URL: <https://kairosdb.github.io/> (visited on 2017-11-14). *used on: p. 94*
- [Locust] *Locust*. URL: <http://locust.io> (visited on 2016-08-30). *used on: p. 106*
- [Lsof] *lsof*. URL: <https://people.freebsd.org/~abe/> (visited on 2017-10-25). *used on: p. 83*
- [LVM] *LVM*. Version 2. URL: <https://sourceware.org/lvm2/> (visited on 2017-11-10). *used on: p. 227*
- [Munin] *Munin*. URL: <http://munin-monitoring.org/> (visited on 2016-09-08). *used on: p. 94*
- [netdata] *netdata*. URL: <https://my-netdata.io/> (visited on 2017-11-14). *used on: p. 95*
- [Neth] *nethogs*. URL: <https://github.com/raboof/nethogs> (visited on 2017-10-25). *used on: pp. 83, 87*
- [Nett] *net-tools*. URL: <https://wiki.linuxfoundation.org/networking/net-tools> (visited on 2017-10-25). *used on: p. 83*
- [nginx] *nginx*. Nginx Inc. URL: <https://nginx.org> (visited on 2017-06-29). *used on: p. 33*
- [NLoad] *NLoad*. URL: <http://www.nload.io> (visited on 2016-08-23). *used on: p. 106*
- [Nmon] *nmon*. *Nigel's performance Monitor*. URL: <http://nmon.sourceforge.net> (visited on 2017-10-25). *used on: p. 83*
- [OTB] *OROCOS TaskBrowser*. URL: <http://www.orocos.org/stable/documentation/ocl/v2.x/doc-xml/orocos-taskbrowser.html> (visited on 2017-08-09). *used on: p. 49*
- [OTSDB] *OpenTSDB*. URL: <http://opentsdb.net/> (visited on 2017-11-14). *used on: p. 94*
- [Pcap] *libpcap*. URL: <https://github.com/the-tcpdump-group/libpcap> (visited on 2017-10-25). *used on: p. 82*

- [Pil] *Pillow*. URL: <https://github.com/python-pillow/Pillow> (visited on 2017-08-17). used on: pp. 59, 60
- [Praat] *Praat. doing phonetics by computer*. URL: <http://www.fon.hum.uva.nl/praat/> (visited on 2016-09-08). used on: p. 76
- [Prom] *Prometheus*. URL: <https://prometheus.io> (visited on 2017-06-28). used on: pp. 32, 94, 149
- [Protobuf] *Protocol Buffers*. Google. URL: <https://developers.google.com/protocol-buffers/> (visited on 2016-08-17). used on: pp. 65, 67, 112, 113, 123, 129–133, 136, 250
- [Psutil] *psutil*. URL: <https://github.com/giampaolo/psutil> (visited on 2017-11-06). used on: p. 85
- [Riem] *Riemann*. URL: <http://riemann.io/> (visited on 2017-12-21). used on: pp. 149, 177
- [ROSB] *roscap*. URL: <http://wiki.ros.org/roscap> (visited on 2017-08-09). used on: pp. 49, 193
- [ROSt] *rostopic*. URL: <http://wiki.ros.org/rostopic> (visited on 2017-08-09). used on: p. 49
- [RRD] *RRDtool*. Version -. 2017. URL: <https://oss.oetiker.ch/rrdtool/> (visited on 2017-11-14). used on: p. 94
- [RRI] *rtt\_ros\_integration*. URL: [https://github.com/orocos/rtt\\_ros\\_integration](https://github.com/orocos/rtt_ros_integration) (visited on 2017-08-07). used on: p. 47
- [RTsh] *rtshell*. URL: <http://www.openrtm.org/openrtm/en/content/rtshell> (visited on 2017-08-09). used on: p. 49
- [RViz] *RViz*. URL: <http://wiki.ros.org/action/recall/rviz?action=recall&rev=79> (visited on 2017-05-24). used on: pp. 22, 23
- [Skyl] *Skyline*. URL: <https://github.com/etsy/skyline> (visited on 2017-12-21). used on: pp. 149, 177
- [Spread] *Spread*. URL: <http://www.spread.org/> (visited on 2016-09-08). used on: pp. 59, 100, 162
- [Sysstat] *Sysstat*. URL: <http://sebastien.godard.pagesperso-orange.fr/> (visited on 2017-10-25). used on: p. 83
- [Telg] *Telegraf*. URL: <https://www.influxdata.com/time-series-platform/telegraf/> (visited on 2017-10-26). used on: pp. 79, 84
- [Tsun] *Tsung*. URL: <http://tsung.erlang-projects.org/> (visited on 2016-08-23). used on: p. 106
- [Valg] *Valgrind*. URL: <http://valgrind.org> (visited on 2017-06-06). used on: pp. 23, 28, 29
- [WiSha] *Wireshark*. URL: <https://www.wireshark.org/> (visited on 2017-06-06). used on: p. 23
- [YarpDD] *yarpdatadumper*. URL: <http://www.yarp.it/yarpdatadumper.html> (visited on 2017-08-09). used on: p. 49



## DECLARATION OF AUTHORSHIP

---

According to the Bielefeld University's doctoral degree regulations §8(1)g: I hereby declare to acknowledge the current doctoral degree regulations of the Faculty of Technology at Bielefeld University. Furthermore, I certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. Third parties have neither directly nor indirectly received any monetary advantages in relation to mediation advises or activities regarding the content of this thesis. Also, no other person's work has been used without due acknowledgment. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged. This thesis or parts of it have neither been submitted for any other degree at this university nor elsewhere.

---

Johannes Wienke

---

Place, Date



#### COLOPHON

This thesis was typeset using the classicthesis typographical look-and-feel developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*" [Bri08]. The writing style has been influenced by Strunk and White [SW09], Dupré [Dup07], and The Economist [The15].