Technik
Informatik & Medien
Hochschule Ulm
University of
Applied Sciences

Hochschule Ulm - University of Applied Sciences

Department of Computer Science
Master's Course Information Systems

# Monitoring in Robotic Systems

Master Thesis
by
Alex Lotz

September 30, 2010


| | | |
|---|---|---|
| Thesis Adviser: | Prof. Dr. Christian Schlegel | Hochschule Ulm |
| Co Adviser: | Prof. Dr. Klaus Baer | Hochschule Ulm |
| External Adviser: | Dr. Michael Dorna | Robert Bosch GmbH, Corporate Research |

# Declaration of Originality

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Ulm, September 30th 2010 _____ (Alex Lotz)

# Abstract

Robotic systems and service-robotic systems in particular are complex and prone to errors. This lead to the problem that the cause of an error in such a system is difficult to analyse. This is a problem first during the development of such systems and second at runtime. During development a deeper in-view into particular components and their *states* is needed. At runtime a feedback from single components and from the system as a whole is needed as a first step towards robust and fault tolerant systems. These two needs are addressed under the term *Monitoring* in this thesis.

There are a lot of approaches in the field of robotics that partly address this problem. For example Logging is one common approach to analyse errors in a system during their development. However, most of them are either tightly connected to a specific *robotic middleware* (resp. framework), or they address only one specific part of the overall problem and do not provide generic solutions.

The first goal of this work is to analyse these approaches and to extract the base ideas behind them. The main goal in the main part of this work is to choose one of the promising robotic middlewares and to develop a concept for Monitoring that fulfill all the necessary requirements. For this purpose the ideas behind the approaches and own ideas must be combined to a valuable approach. An implementation at the end additionally confirms the ideas of the concept.

# Acknowledgements

I am very grateful for the advice and support of my thesis adviser, Prof. Dr. Christian Schlegel, for his feedback and for keeping me focused in my research. Thanks for his wisdom, understanding and patience.

My gratitude also goes to Robert Bosch GmbH and to Dr. Michael Dorna in particular. Our scientific discussions together with Prof. Dr. Christian Schlegel were an infinite source for inspiration and knowledge and greatly improved this work.

I would also like to thank my second reader, Prof. Dr. Klaus Baer, who made many helpful suggestions that led to improvements in this work.

My special thanks goes to the whole ZAFH team for their friendship, their support, the great scientific atmosphere and the numerous discussions with them. I would like to thank Andreas Steck for his help and discussions no matter time and day of the week and no matter how busy he was. I thank Dennis Stampfer for his support in writing this document and the numerous discussions with him. It is a honer to work with you all.

Finally, my special thank goes to my family who supported me all the way through this work.

# Contents

# 1. Introduction

Nowadays, Robotic Systems and Robotic Applications in particular are becoming more and more complex. Robots have to cope with complex, unstructured and dynamic environments. They have to act autonomously and make decisions on their own. The range of robot skills on the other hand grows continuously with rising requirements. Therefore the development of such systems becomes more and more complex. One common approach to cope with this problem is to structure the application as software components (or modules) and to use uniform (or standardized) communication mechanisms in the form of Frameworks (e.g. SMARTSOFT [Schlegel2004], ROS [Quigley2009], OpenRTM [Ando2005], etc.). These frameworks are often summarized under the term *Robotic Middleware*. Among other advantages, these frameworks allow to develop and to test parts of the whole system independently of each other. These parts can be then integrated to a complex system in a building blocks manner, depending on the scenario. This approach is also referred to as *Component-Based Software Engineering (CBSE)* [Schlegel2006]. Although, this is definitely a progress compared to monolithic systems, there are still some problems to be solved. For example due to the encapsulation of components it is difficult to locate the cause of a problem in a system, because the origin of the problem can be related to a specific part of one component or can be even caused by chained relations between components. Further, it is difficult to test components individually, offline and in advance. If the components are implemented correctly this does not implicitly mean that they will behave correctly later in the system. On the other hand, correctness is very important in particular for *human-robot-interaction*. Thus, errors in the system can lead to catastrophic effects either as damage to human health or even loss of lives. Common approaches to avoid these effects are extensive testing and simulation [Bensalem2009]. Due to the particular character of environments where robots have to interact it is highly difficult to test offline or to build convenient simulations. One promising approach in this area is presented by Schlegel et al. in [Schlegel2010]. Although it focuses on model-driven design it also depicts that it is often unavoidable to cope with errors first at run-time. Lussier et al. [Lussier2004] addresses this problem under the terms *robustness* and *fault tolerance*. Most of these approaches require information about the system (e.g. *current state* of the system) that is encapsulated in individual components and is not available on the system level. However, this information is important (and a precondition) to be able to handle errors and anomalies in a system at run-time and under proper context. The collection of data and the analysis of this data at run-time to determine the current state of the system is one important part of this work. To be able to collect the data it is first necessary to identify the origin of this data and to generalize the information which is also a part of this work.

## 1.1. Motivation

The main goals of Monitoring as introduced above are to improve the development process for building complex robotic applications and to improve the fault-tolerance and robustness aspects of the overall system. Improvements in these areas lead to the following advantages. The development process for single components and for the system composed out of components can be supported, accelerated

and simplified by means of Monitoring. Further, Monitoring makes the information about the state of the system generally available on system level. This makes it possible to implement advanced error handling routines and to use common fault-tolerance techniques. Additionally, Monitoring can be advanced by autonomous *diagnose* and *repair* functions as a first step towards *self-healing* systems. The information gathered by Monitoring can be used as *feedback* (from the system) either within the system itself (for example in a scenario control component) or in other tools like the MDSD Toolchain [Steck2010] of SMARTSOFT [Schlegel2004].

## 1.2. Definition of Monitoring

The term *Monitoring* is very generic and can be used as an umbrella term for a wide range of applications in different domains (some of them are mentioned in Chapter 2). Thus it is crucial for this work to find a clear definition for Monitoring.

Delgado et al. presents a feasible definition for a Monitor: *"A monitor is a system that observes the behavior of a system and determines if it is consistent with a given specification"* [Delgado2004]. The focus in this thesis is based on the first part of this definition, namely to monitor and to observe the behavior of a robotic system. Additionally, this must be done at runtime as will be evolved later in this work. The second part, the diagnose and analysis which can be applied by using the results of Monitoring as the source for information. The problem in the approaches as presented in the paper are that two different levels of abstraction are intermixed, which leads to static solutions which are not applicable in complex robotic applications.

There are many aspects that describe the behavior of a system and that can be *monitored* run-time. The first part of this work is to evolve these aspects (see next sections). The second part of this work shows the possibilities to monitor these aspects and to interpret them with a certain semantic (see Chapter 4).

A further constraint in this work is to focus on Monitoring of systems found in the service-robotics domain. With it, the focus lies on the needs in typical service-robotic scenarios and service-robotic applications. The determination of those needs is also a part of this work (see Chapter 3).

## 1.3. Overview of the Monitoring problem space

This section gives an overview of the problems which are relevant in this work. Figure 1.1 gives an overview for a system that includes Monitoring issues. The problem space can be divided into three main parts, namely the *component's internal view* ①, the *inter-component communication* ② and the *monitoring-system* ③.

First the *component's internal view* ① focuses on problems that are related to common structures and activities inside of components in various robotic applications. It is also of interest in which conditions a component can be at discrete time slots and what information is available inside of these components. Finally the main goal is to identify the information that is of interest for monitoring scenarios and must be therefore made accessible to the Monitoring system. One part of this is to identify recurring information that occurs in different components in a similar form and to generalize this type of information. For the customized information a generic user-interface must be developed.

The second part is the *inter-component communication* ②. Two additional aspects can be derived from this part, namely the customized communication between typical components in common robotic applications (also referred to as data-flow) and the specialized communication between those components and the monitoring system. For the former specific parameters can be monitored like

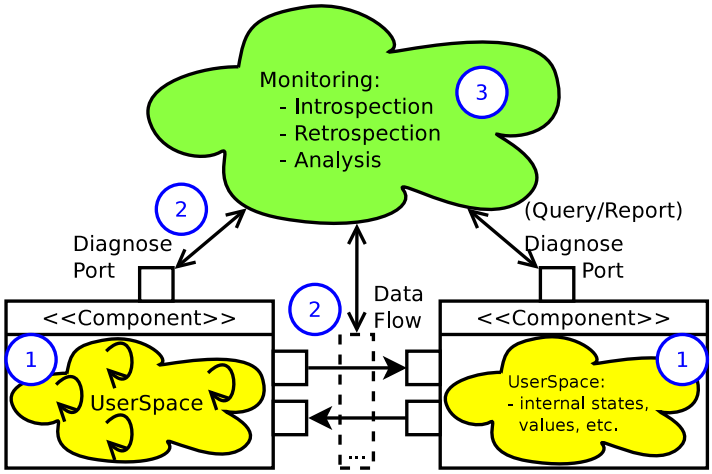


Figure 1.1.: Monitoring overview

plausibility of data, timing issues, etc. For the latter an important aspect is whether data must be queried from the monitoring system or whether components report information on their own. In addition the communication overhead related to Monitoring influences the system in different ways, which must be analyzed.

Finally, the third part shows problems related to the *monitoring system* ③ itself. The main focus here is to define the semantics and the interpretation possibilities for data that is collected from single components and from the communication between them. One part of this problem is to identify graphical representation possibilities (for example in a GUI) for this data. In the Monitoring three levels of granularity can be identified. The first level focuses on the problems related to internal aspects of running components in the system in terms of *introspection*[1]. In this case a snapshot on the state values (or the status) of a component is essential. The second level focuses on problems related to temporal issues in terms of *retrospection*[2]. Significant for this level is a chronological order of events in the system, respectively a list view with a history for occurrences of events in the system. On the third level the data coming either from single components or from the communication between them can be analyzed and interpreted to determine the information that indicates on the current state of the system (for example the health of the system).

## 1.4. General Use-Cases for Monitoring

The aspects presented in section 1.3 are still quite generic and can be applied to a wide range of applications. The use-cases described in this section (see Figure 1.2) focus more on the problems coming from typical scenarios in common robotic applications and the main challenges resulting from them. In terms of use-cases, a robotic application is seen as a *system*. Some *actors* (as shown in Table 1.1) interact with this system.

One of the common problems in building complex systems in the robotics domain is the limited insight into a running system. The reason lies in the nature of systems composed out of components. Components in such a system encapsulate functionality and data for a specific task. The communica-

[1] Introspection means that all relevant information about a component are gathered at a specific time-stamp as a snap-shot.
[2] Retrospection means that the same data as with introspection is gathered but with a chronological history.

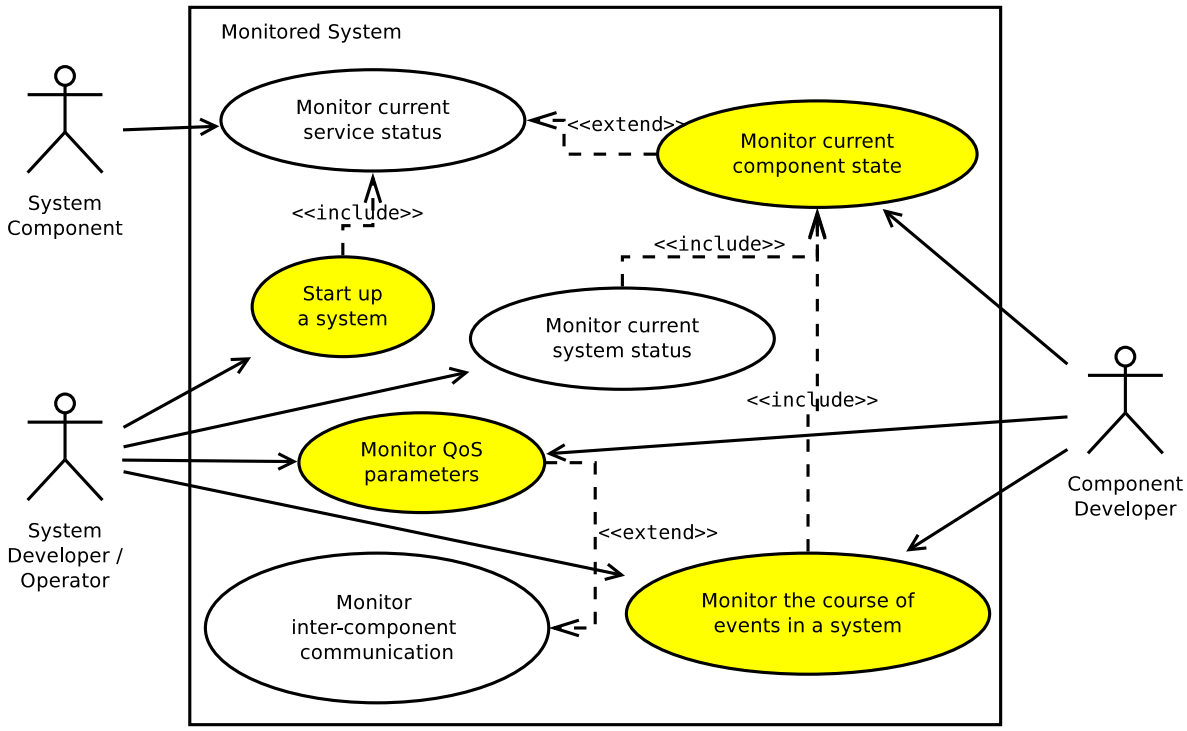


Figure 1.2.: General use cases

| Name | Role | Interested in |
|---|---|---|
| Component Developer | develops specific components (independently of the rest of the system) | current internal states, status and values of a specific component (internal view) |
| System Developer | builds up a system composed out of components (also component integration) | a summary view about parametrization and states of components (system view) |
| Operator | observes and controls a running system | the health status of the overall system |
| System-Component | runs independently of other components, but uses data from services of these components | information on the QoS aspects in communication objects |

Table 1.1.: Overview of the Actors in the system

tion between components is based on sophisticated communication mechanisms like communication patterns. The data that is communicated is defined according to a Canonical data model. Low level details of the internal structures and values inside of components are hidden from the system level. This is an important issue to reduce the overall complexity in the system, to reduce the tight coupling between components and last but not least to increase the reusability of functionality. However, first during development and foremost at run-time, some particular information is necessary to be made available outside of components (on the system level). For example to determine the current *health* from single components or from the whole system each component must sum up its vital values and publish them on a corresponding public interface. With this, some kind of *health-monitoring* is possible to inform a developer or an operator about the current state of the system.

Figure 1.2 shows the Actors (which are described in Table 1.1) and their interactions with the system. The main goal of the Actors is to monitor particular aspects of the system which is either under development or running on the target platform. These aspects are analysed with different levels of granularity namely the *system*, *component* and *service levels*. The figure shows the main use-cases (ellipses with yellow background). These use-cases are described in corresponding subsections. Additionally there are three sub-use-cases (ellipses with white background) which are parts of the main use-cases. They are described together with the main use-cases in the following subsections. Each of these subsections is structured as follows. First, a general overview of the current use-case is given. Second, one real scenario is presented where the Monitoring parts are highlighted. After that, some core challenges are presented. Finally, the referenced use-cases are listed and optionally some notes are shown.

### 1.4.1. Start up a system

The first step in running complex systems that are composed out of components is to start up (resp. to *initialize*) this system as a whole in a secure, predictive and robust manner.

At start-up of the system a component initializes its resources and its services. During this time the component is in the *Init* state. If all preconditions for a particular service in this component are fulfilled, this service is marked as ready to be used. If all preconditions of the component are fulfilled the component switches to the *Active* state. Preconditions could be that particular resources inside the component or remote services from other components are ready to be used[3].

#### 1.4.1.1. Scenario

A system developer triggers a start-up procedure (e.g. using a start-script). In this procedure, a set of components for a particular scenario (e.g. Joystick navigation example) is started all at once. Each component in the system initializes itself autonomously. A component that controls the behavior of the scenario monitors the current state of components in the system and starts the execution of the scenario at the time when all components are in the *Active* state. The advantage of starting a system in this way is that it becomes more predictive and secure.

A simplified version of the joystick-navigation scenario is illustrated in Figure 1.3. It consists of various base components for navigation like *LaserServer*, *Base*, *JoysticServer* and *CDL*[4]. The CDL component [Schlegel1998] implements collision free navigation and requires services from other components. For example, the *ScenarioControl* component could activate the *CDL* component only

[3] Services are ready to be used if they are ready to deliver proper data or are ready to process (resp. to handle) incoming requests.

[4] CDL: Curvature Distance Lookup

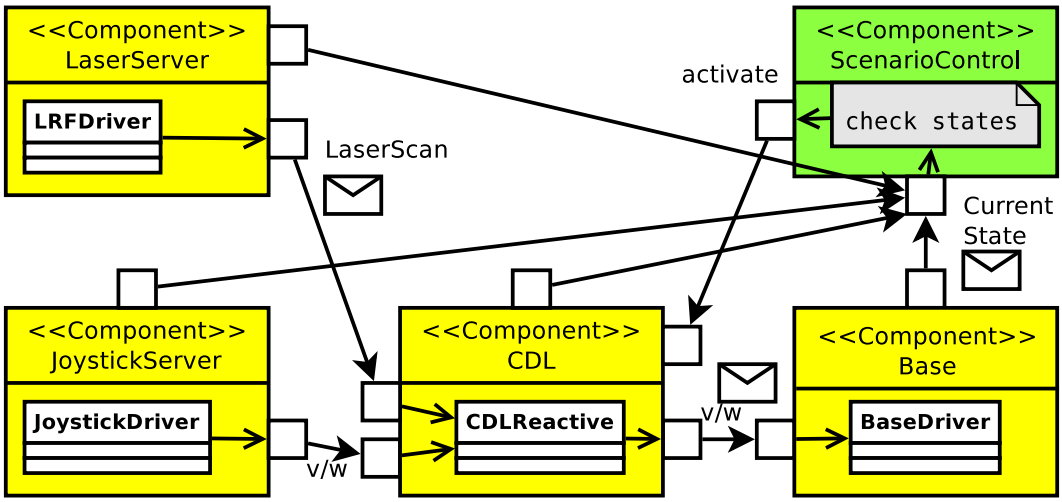


Figure 1.3.: Joystick navigation scenario

after all required components (like *LaserServer*, *Joystick* and *Base*) are fully initialised and are ready to deliver proper services. Thus, the *ScenarioControl* component uses the functionality of Monitoring to ensure proper initialisation.

**Main challenges** A component must be aware of the current state of its resources and must be able to determine whether all preconditions of all its services are fulfilled. Other components in the system must be able to recognize when a particular remote service becomes ready to be used.

**References** This use-case *includes* the case "Monitor current service status" as part of the precondition as described above.

**Notes** It is advisable to initialize the services in components as early as possible. With it, other components that rely on these services can react on their problems early.

### 1.4.2. Monitor current state of a component

A key part for robust and error resistant systems is the *awareness* of anomalies and failures in the system. A first step towards this goal is the knowledge about the current state of a particular component. Thus, it is necessary to monitor the current state of a component in a running system. As a second step the current state can be interpreted to a *health* value of this component. By analyzing the current state of all components in the system (for example inside of a specific *Monitor* component) it is possible to determine the current state of the whole system. This information can be interpreted further to the current *system health*.

#### 1.4.2.1. Scenario

A component developer triggers the monitoring process of a particular component. With it, the developer must start a *Monitor* component, the component that he currently develops and some dummy components which generate the data that is necessary to test the component under development. After

that, the developer sets-up the Monitor component such that it shows only the type of information the developer is interested in. In tern, the Monitor component sets-up its communication properties to react automatically on the state changes in the component under development and to update the corresponding view in the Monitor component.

**Main challenges** Each component in the system must provide some kind of *lifecycle-state* and it must be transmitted to the Monitor component. Additionally to the lifecycle some vital values of a component must be monitored either inside this component or on the system level. In both cases the results must be made available in the Monitor component. A general challenge is to identify both the data that is always available and the values that are dynamically available.

**References** This use-case itself is a part of the more general use-case "Monitor current system status" (as described above). Some of the vital values inside of each component are the dependencies (resp. preconditions for services) as described in the use-case 1.4.1 "Start up a system". For this reason, this use-case is included and extended here.

## 1.4.3. Monitor the course of events for the system states

To detect problems in a running system it is sometimes insufficient to observe only the current state of the system (or single components). Instead, it is often necessary to monitor the course of events in the overall system. The reason is that developers often want to identify the cause of an error and not only the occurrence of a specific error. Thus, a history of events available on the system level is necessary. For example, a Monitor component can provide a view with a list of events that occurred in the system in a chronological order. In other words the error becomes traceable. With that it is easier to find the event that led to an error and to develop appropriate error handling strategies for this case. An event in the system can be either a state change (as described in the use-case 1.4.2 "Monitor current state of a component") or a user defined message.

### 1.4.3.1. Scenario

A practical example for this use-case is the "face-recognition" scenario which consists of at least two components: an "*image-server*" component and a "*face-recognition*" component. The image server implements a video camera driver and provides captured images to other components in the system. The face recognition implements an algorithm to detect and to recognize faces within an image. The scenario is structured as follows. The image-server, the face-recognition and a Monitor component must be started (these components are schematically illustrated in Figure 1.4).

First, the face recognition delivers face-results if persons occur in front of the camera. After that the light conditions might change (for example the light could be switched off or the camera lens could be covered with an obstacle). The image server now provides black images and some kind of an image quality check algorithm inside of the image-server recognizes a problem with the current images. An appropriate message is sent to the Monitor component (including the current time-stamp of occurrence). The face-recognition component receives black images and fails to detect and to recognize any faces. A message that describes this problem could additionally be sent to the Monitor component. The Monitor component now provides the messages in a list including the first error (or warning) message from the image-server. Of course in this scenario the cause of the error is quite obvious and could be found by a developer without Monitoring. However, in real systems the cause is often much more difficult to identify. As a second step the developer could implement a strategy

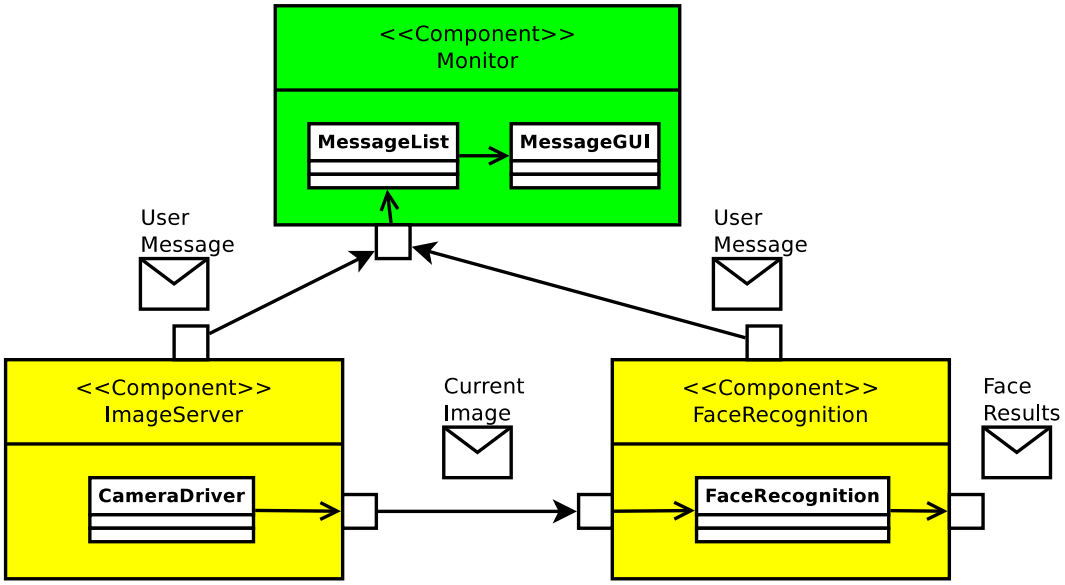


Figure 1.4.: Illustrated components for the face-recognition example

that the image server stamps current image-communication-objects with some kind of image-quality-check-failed flag. The face recognition component can react on this flag and ignore such images (which can be later a normal case in the execution).

**Main challenges** One of the main challenges is that the *events* - that come from different components which are distributed over a network and different PCs - must be (or appear to be) synchronous in a centralized Monitor component. Second, the history of events must be without *gaps*. The challenge is not to loose events in the system. Thus, a Monitor component must not oversee or miss any events. A further challenge is to minimize the effects from Monitoring on the regular execution of components (i.e. effects that are caused from communication overhead).

**References** One of the event types that can be captured in the Monitor component is the state-change of other components in the system. Therefore, this use-case *includes* the case 1.4.2.

### 1.4.4. Monitor inter-component communication and its QoS parameters

The previous use-cases focused mainly on the information that is generated and is available *inside* of specific components. Another possibility is to eavesdrop the communication *between* the components in the system. The first advantage is that the plausibility of data can be analysed. In a second step one could generate some simple statistics from these data and determine some *quality of service* (QoS) aspects. Simple statistics are minimum-, maximum- and average values of raw data, frequencies of messages, etc. Quality of service is for example a definition of boundaries for one or some of these statistics.

#### 1.4.4.1. Scenario

Again, the "*face-recognition*" example (as presented in the subsection 1.4.3 and as illustrated in Figure 1.4) is a convenient scenario. In this case, the communication between the *image-server* and the *face-recognition* components can be intercepted. Let's assume that the image-server is configured to deliver 30 image-frames per second. Let's further assume that the frame grabbing, the transportation of frame-messages and the face-recognition algorithm need a lot of CPU time. If other components in the same system produce some peak loads, the image-server can not get enough CPU time and will drop some frames. The frame rate of course will decrease.

A Monitor component could count the number of image-communication-objects per second which in fact is the real frame rate. A system developer could analyse the frame rate and decide on the configuration of the system such that the image-server and face-recognition get as much CPU time as possible at the time when they need it the most (for example when the robot is in front of a person). An additional example is that a scenario-control component reacts on the frame rate fluctuations if for example the human-robot-interaction becomes more important than the actions that are currently executing (depending on the scenario). In this case the scenario-control component could pause some components that are currently less important to save CPU resources. This is also an example for *resource-awareness* in a system. A more general and broader view on these aspects can be found in [Steck2010].

**Main challenges** The communication between components is typically optimized to produce as small communication overhead as possible. Thus, the communication protocol usually does not provide enough meta-information to reconstruct the communication objects from raw data (taking only the byte-stream into account). The main challenge is to find the right level in the system to intercept and to eavesdrop the communication between components. A second challenge is to keep the communication overhead for monitoring as low as possible. For example a doubling of all communication in a component is in most cases not acceptable, because it would exceed the limits of restricted resources in a typical mobile robot platform.

**References** The use-case "To Monitor the inter-component communication" and to derive and monitor the quality of service parameters are directly related. Thus, this use-case *extends* the inter-component communication.

## 1.5. Bird's eye view of the Monitoring concept and solution

This section presents a bird's eye view of the concept and the solution as developed in this work. The main goal here is to present a common theme that is used throughout this work in different chapters at different levels of abstraction and detail. The solution as shown here is discussed later in Chapter 4 in detail.

Figure 1.5 shows a solution for Monitoring in robotic applications. This solution will be developed piece by piece during this work. The main idea is to use a local **black-box** in each component that collects and stores particular information in the component. As will be shown later there are different types of information inside of each component that is of interest to be monitored. One information category is related to middleware-specific aspects which can be often collected in a generic way. For this information the black-box provides the GenericInterface (as shown in the figure). Additionally

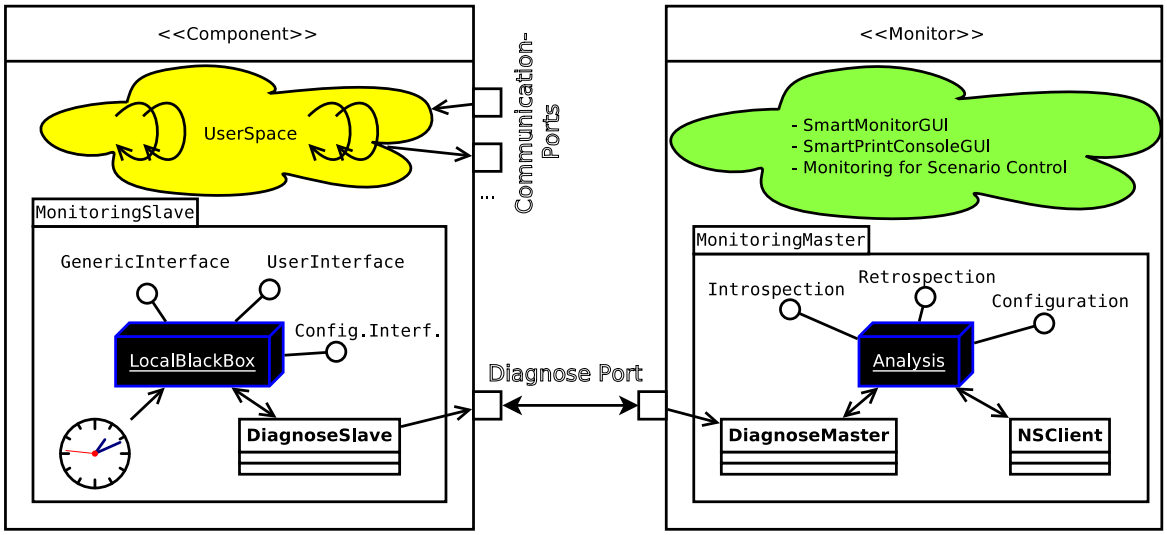


Figure 1.5.: Bird's eye view of the Monitoring concept

each component typically provides a user-specific part that implements a particular task in this component. This part (the UserSpace) is illustrated as yellow cloud in the figure. As there are typically several components in the system they must communicate with each other and exchange data (resp. information). This is done by using specific interfaces, namely communication ports. The UserSpace typically consists of different algorithms, libraries and threads. It is often necessary to observe (resp. to monitor) particular values from them. For this information the black-box provides the UserInterface. There are different possibilities to handle data in the black-box. To control and parametrize these possibilities (resp. the behavior) of the black-box, the ConfigurationInterface is provided. To determine the time when a particular event in a component occurs it is necessary to synchronize the local clock in this component with a global system time.

As shown in 1.3 there are different levels in the Monitoring system that can be examined. A solution for the first one - namely the components in the system - is presented above. A second level represents specialized Monitor components. There are two main aspects for Monitoring that can be examined. The first addresses a snap-shot view semantic. For this information type the Introspection interface is provided. The second aspect addresses temporal issues in the system. For it, the Retrospection interface is provided. Additionally the Configuration interface provides the possibility to parametrize the level of information that the Monitor component currently needs. One of the crucial parts in each Monitor component is the interaction with the NamingService (which is illustrated as NSClient in the figure). Finally, most of the Monitor components must implement a visualisation (i.e. a GUI) to display monitoring results. This is illustrated as a green cloud in the figure.

The third level is the communication between components in the system and the Monitor component. This communication is illustrated as the DiagnosePort in the figure. The DiagnosePort is a generic interface that each Monitor component can use to access particular information from one (or several) component(s) in the system. The interaction between a Monitor component and other components in the system represent a Master/Slave relationship. The reason for this will be shown later in this work.

Finally, section 4.3 will zoom into both black-boxes (LocalBlackBox and Analysis) and present their contents in terms of white-boxes.

# 2. Related Work

Monitoring is used in engineering, in medicine and in science to observe all kinds of systems. In engineering the current state of a technical system (i.e. mechanical-, electrical- or software-system) is observed to control this system. In medicine any kind of biological systems are observed to cure a disease for example. In science a system is everything (being real or abstract) that is currently on focus, for example to prove a theorem. Thus many ideas can be found in a wide range of different domains. It would be an impossible task to mention all of them in this chapter. More reasonable is to show first some examples from non-robotic applications which are helpful to understand the ideas that come later in the main part of this work. Second, some relevant robotic middlewares are analysed with the focus on aspects relevant to the type of Monitoring as understood in this work. Some of these aspects can be picked up later in the main part of this work.

## 2.1. General non-robotic applications for Monitoring

The term Monitoring is very natural for scientists, because it is very common to observe and to control any kind of automatic or autonomous systems (i.e. mechanical, electrical, software, biological and other systems). This chapter picks up a few of them that provide helpful aspects for the ideas as described later in this work. Some of these aspects are in common with most of these domains (as shown in the following). There is always some kind of a system that runs autonomously (or automatically). This system is prone to errors and failures. Thus, this system must be observed from time to time to assure that it works correctly. If the behavior of the system deviates from the expected or wanted behavior some kind of correction or repair of this system is performed to ensure that afterwards the system works correctly again [Lussier2004].

In *Medical Science* it is common to monitor *biological* systems (like a human body for example). A heart is one of the vital parts of the human body and can be easily monitored to ensure the liveliness of a body. There are some aspects that are in common with a software system that is composed out of software components. These aspects are described in 2.1.1.

In *Avionic Industry* a well-established system is a *flight-data-recorder* (also referred to as *Black-Box*). Some problems and ideas for this system are similar to the problems for Monitoring in a robotic application. The semantics and some ideas are shown below in 2.1.2.

In *Automotive Industry* it is common to monitor the controllers in a car under operational conditions (by using a so called *diagnose controller*). Afterwards the *error codes* are read-out from the diagnose controller - by connecting a diagnose device to a diagnose socket - to find problems or broken parts in a car or to ensure correct functionality. More details on this topic are shown in 2.1.3.

Finally some other examples for Monitoring in non-robotic applications are shown in 2.1.4.

### 2.1.1. Heartbeat analogy

A human body can be seen as a complex system that relies on a correct cooperation of different organs - analogous to “components” in the context of this work. One of the simplest impartial methods to check the *liveliness* of a body is to check its *heartbeat*. An absence of a heartbeat indicates a serious

problem. An unexpected or unusual heartbeat can indicate some anomalies in the body. Hence, a heartbeat can be used as a measurable unit for diagnose purposes. Of course the heartbeat has also its limitations. For example a correct and proper heartbeat does not say anything[1] about correct (resp. reasonable) brain functions.

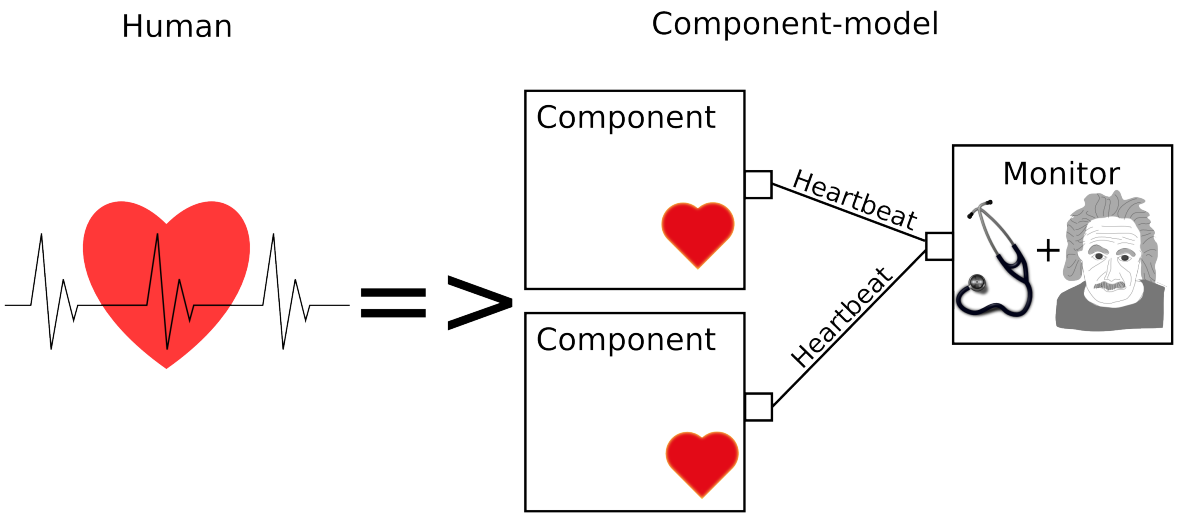


Figure 2.1.: Analogy to the heartbeat

This semantic can be simply adopted to components in a distributed robotic application (see Figure 2.1). In this case each component plays the role of a human body and triggers a periodic signal that represents some kind of a component-heartbeat. Similar to the organs in the body a component is composed out of parts that are vital to the services of this component. Possible interpretations of a heartbeat are listed in Table 2.1. Similar to the heartbeat of a human body the one in components also have its limitations. A correct heartbeat from a component can only indicate on the reactiveness of this component and not on the overall correctness of services that this component provides.

| ***Human Body*** | ***Component System*** |
|---|---|
| continuous pulse indicates *liveliness* of a human | continuous heartbeat indicates *liveliness* (or execution) of a component |
| too fast and too slow pulse indicates a problem | heartbeat rate that is out of range indicates an anomaly or an error |
| a heartbeat is only possible if some vital functions of a body (like respiration) are working correctly | a component should trigger a pulse only if some vital functions response or are in the right state |
| at specific stress levels of a body different ranges of heartbeat rates can be expected | at different states of a component different ranges of heartbeat rates can be expected |

Table 2.1.: Semantics of the heartbeat-analogy

[1]The analogy is simplified at this place and ignores the cases where some drugs or narcotics can have an effect on both the heartbeat and the brain.

### 2.1.2. Flight data recorder

One quite common and well-established approach which is similar to the idea of Monitoring as described in Chapter 1 is related to the concept of a *flight data recorder* in an aircraft (also referred to as *Black-Box*). The idea is not new and can be found in different applications.

For example [Staples1997] presents a rather radical solution to add a *software black-box* to every software. The core idea is that a software black-box is delivered together with every new consumer software. The black-box monitors states and events of the software, and stores this data in corresponding files. The main effect is that a broad number of users find (or provoke) the problems and errors in a new software - that are probably overseen by developers - very fast. The files created by the black-box can be transmitted back to developers, who can in turn analyse the information, deduce the cause of an error and develop appropriate software patches.

Compared to the robotics domain, the principle is similar with the difference that the errors are found not because of the users but because of the high dynamics in the environments and the broad range of scenarios.

A similar idea is presented in [Elbaum2000]. In this case, a framework with a *Software-Black-Box (short SBB)* is presented. This framework provides two key features, namely to capture interesting aspects of a software behavior and a reconstruction mechanism that helps to detect the cause of an error. In this paper a software is seen to be composed out of modules, which inherit specific functionality. It is assumed that at different times different modules are active and the change of activity can be traced with *transitions*. In case of an error, a possible scenario can be generated that is likely to provoke this error. Although this approach presents a very comprehensible analogy to a black-box in an aircraft, some assumptions can not be taken in robotic applications. For example the modules, respectively components in robotics, are often active concurrently and solve a specific task in cooperation. For this reason, it is difficult or even not possible to generate a transition matrix as described in the paper. Also the distribution of components and the communication between them are not addressed by this approach.

Notwithstanding the simplicity of the ideas behind the two approaches as described above lead to the assumption that a *software black-box* in components in a robotic application could offer a remarkable value for the development of such systems. The main challenges are to identify the right level of abstraction in a robotic application and to focus on the characteristics that are common in robotic scenarios (for more details see Chapter 4).

### 2.1.3. Car diagnosis system

A modern car has a lot of controllers that are distributed over many places in this car and perform different tasks with different requirements on the security and dependability. For example some controllers are responsible for functions in the motor and a failure in one of them can lead to financial damage. Other controllers are responsible for more security-relevant parts like the airbag system or the brake system. These controllers must be highly dependable and a failure in one of them can lead to the loss of lives. In turn, other controllers that are responsible for the entertaining system in the car are less important and a failure of one of them can lead in a best case to reduction of comfort for passengers.

All these controllers are interconnected via a network and must guarantee the operability of the whole car as a system. The safety relevant controllers are often designed with redundancy and a monitoring controller or component observes the functionality and in case of an error of one of the controllers a monitor can switch to (resp. choose) another controller. This approach is quite similar to

that in robotic applications. Components that are critical for the mission of a robot are often designed with redundancy. Again, a monitor observes the functionality of one of these components and in case of a failure in this component - for example a service can not be provided anymore - the monitor switches to another component which for example implements another algorithm for the same task and is probably more likely to succeed.

A further controller that is of interest for Monitoring is the *diagnose controller*. It is typically interconnected with all other controllers in the car and monitors their current state. All errors, failures and anomalies in the system are logged in the diagnose controller as so called *error codes*. These codes can be read-out afterwards (for example in a garage) with a special Diagnose Device which is connected to a specialized *Diagnose Port*. A mechanic can deduce on the problems and anomalies in the car and can replace broken parts by new ones. There are a lot of standards - defined by the automotive industry - that describe the functionality of the network and the diagnose controller [ISO14229] and other parameters like the mechanical dimensions of the diagnose plugs [ISO/DIS-15031], etc. Additionally, in Europe for example a diagnose controller must log the emissions produced by the car as defined by the law (as defined in the emission-standards Euro-5, Euro-6, etc.).

There are both similarities and differences between a system in a car that is composed out of controllers and a robotic system that is composed out of software components. On the first hand the controllers in a car can be compared with software-components in a robotic application as shown in the following. In both cases a certain task is implemented with certain separation of concerns. Also in both cases the communication is performed over a network and the controllers (resp. components) are distributed over different places (resp. PCs). A diagnose controller (or a diagnose system in more general) consists of distributed parts (one for each controller) in the system to provide access to the current state of this controller and one part that is inside of the diagnose controller itself, that collects the data from the distributed parts, enriches this data with further information (like time-stamp) and logs it in a local storage. Analogous to it, a diagnose system in a robotic application can consist of distributed parts in each component and a centralized part (e.g. in a Monitor component). This centralized part can be seen as a *black-box* which collects the information about the system and provides it for further analysis for example to visualize this information in a GUI or to analyse it on-line and decide on repair functions if necessary.

On the other hand, the focus and some requirements in a robotic application are different to those in a car-system. The topology in a car is typically more static and predictable than in a robotic application where high dynamics in the environment and dynamic wiring of connections are common. Therefore, the ideas - from a diagnose system in a car - must be adjusted to the robotic needs which are also described in Chapter 3.

In recent years, a very interesting standard appeared in the automotive industry namely the *AUTOSAR*[2] architecture. Some of the approaches presented there (i.e. the generation of modules) seems to be interesting on the first view. However, AUTOSAR provides a different architecture level that focuses more on low level modules. The application of those ideas to the robotics domain is not clear and the documentation do not provide hints on such aspects. For these reasons, AUTOSAR is considered as a work in progress that could lead to some interesting ideas by further investigate on this approach. The core idea - to define an open architecture that is available for every automotive manufacturer as common ground - is of course very appealing not only for the automotive industry but also for robotic applications. However, in robotics the development of general architectures and patterns is an ongoing process and is still fairly far away from standardisation. Nevertheless this would be a valuable goal that of course is beyond the scope of this work.

[2] AUTOSAR: AUTomotive Open System ARchitecture (http://www.autosar.org/)

### 2.1.4. Other examples for Monitoring

There are many other examples for Monitoring in different domains. For example *WireShark*[3] is a popular tool to eavesdrop communication between endpoints in the Internet. The base idea of this tool seems to be attractive for the robotics domain, where the communication between components could be eavesdropped in a comparable way. Therefore, this approach seems to be worth to be analysed in the main part of this work. Many other approaches lead to interesting analogies, but have often different focus and do not additionally lead to significant value.

One particular class of approaches is related to the robotics domain, which is the main focus in the following section.

## 2.2. Component based robotic middlewares

In the field of robotics, a wide range of robotic middlewares and frameworks can be found. Some of them originate from certain research groups. Most of them differ in the focus on requirements that were identified to be important in this group for robotic applications. The reason for that is twofold.

On the one hand the number and complexity of tasks that a service robot must perform (or at least is meant to be able to perform) in a wide range of scenarios grows continuously. There are many state of the art tasks like mobile manipulation, object recognition, etc. which are work in progress. Therefore, the number and kind of requirements to a robotic middleware grows and changes continuously as well. This makes the development of such a middleware to an ongoing process.

On the other hand, the development of robotic applications is still a young field of research compared to the development of traditional software. Thus, there is a lot of discussions in the robotic community about several aspects of robotic middlewares. This makes it difficult to find the one middleware that inherits the most of them.

A lot of different case studies on robotic middlewares like [Kortenkamp2008] or [Brugali2010] can be found in the literature. However, they are difficult to compare because the authors observe the middlewares from their individual (and sometimes subjective) point of view. The conclusion changes yearly and some of the middlewares are considered to be more important where others seem to loose significance (sometimes related to the activity of the researchers group). For these reasons it is not considered in this work to make a comparative study of these middlewares but to pick some of them and focus on aspects that are seen to be most valuable for Monitoring as understood in this work.

### 2.2.1. SmartSoft

SmartSoft is a component based framework, first described in [Schlegel1999]. One of the merits of SmartSoft comes from its sophisticated communication capabilities. From the experience of at least one decade a set of seven *communication patterns* evolved which are part of the SmartSoft idea [Schlegel2004]. These communication patterns are the key towards interoperable, reusable and composable systems as shown in [Schlegel2006].

> *"A major purpose of communication patterns is to relieve the component builder from error-prone details of distributed and concurrent systems by providing approved and reusable solutions. Communication patterns have to provide patterns for higher level component interactions."* [Schlegel2006]

[3] Online available at http://www.wireshark.org/

SMARTSOFT provides a certain terminology [Schlegel2006] which is used later in the main part of this work in chapter 4. This terminology consists of the following definitions:

**Component:** *"A component can contain several threads and interacts with other components via predefined communication patterns that seamlessly overcome process and computer boundaries. Components can be dynamically wired at runtime."* [Schlegel2006]

**Communication Patterns:** *"assist the component builder and the application builder in building and using distributed components in such a way that the semantics of the interface is predefined by the patterns, irrespective of where they are applied. A communication pattern defines the communication mode, provides predefined access methods and hides all the communication and synchronization issues. It always consists of two complementary parts named service requestor and service provider representing a client/server, master/slave or publisher/subscriber relationship."* [Schlegel2006]

**Communication Objects:** *"parametrize the communication pattern templates. They represent the content to be transmitted via a communication pattern. They are always transmitted by value to avoid fine grained inter-component communication when accessing an attribute. Furthermore, object responsibilities are much simpler to manage with locally maintained objects than with remote objects. Communication objects are ordinary objects decorated with additional member functions for use by the framework. Generality of the approach is achieved by using arbitrary and individual communication objects."* [Schlegel2006]

**Service:** *"Each instantiation of a communication pattern provides a service. Generic communication patterns become services by binding the templates by types of communication objects."* [Schlegel2006]

Communication patterns provide a clear semantic for the communication capabilities between components. This semantic provides an ideal basement for the Monitoring concept which is introduced in 4.3. All communication patterns can be implemented by using different kinds of communication mechanisms (like CORBA, message passing, or even directly on TCP/IP sockets) [Schlegel2006]. Even the usage of middleware-specific communication (like *Data-Ports* in RT-Middleware, *Topics* in ROS, etc) as a basis is possible. For this reason, the usage of communication patterns as a basis in the Monitoring concept do not restrict it to the SMARTSOFT framework. In fact, communication patterns allow a clear division between the communication related issues and the core of the Monitoring concept. For these reasons, SMARTSOFT is considered to be the most promising framework as a basis in this thesis.

#### 2.2.1.1. Components in SMARTSOFT

In the last decade of using SMARTSOFT many components[4] have been developed, which are used in several state-of-the-art scenarios like those in RoboCup@Home[5]. Such scenarios show some problems related to Monitoring as understood in this work. These problems and the abstracted version of the scenarios are presented as use-cases in section 1.4.

[4] Public available on SourceForge http://sourceforge.net/projects/smart-robotics/
[5] http://www.robocup.org/robocup-home/

#### 2.2.1.2. Model Driven Software Development (MDSD) in SMARTSOFT

Model Driven Software Development (resp. Model Driven Software Engineering) is considered to be one of the most promising approaches in service-robotics to cope with rising complexity in state-of-the-art robotic applications. Some recent advances in this area are presented in [Schlegel2009]. Thereto SMARTSOFT provides a "MDSD Toolchain"[6] as described in [Steck2010].

#### 2.2.1.3. Implementations of SMARTSOFT

As mentioned above the SMARTSOFT idea can be implemented by using different communication mechanisms as a basis. At the time of writing two reference implementations are publicly available on SourceForge. One implementation - the CORBA/SMARTSOFT[7] - is based on CORBA. This package consists of the framework and all public available components, that were developed with SMARTSOFT. A second implementation - the ACE/SMARTSOFT [Lotz2010] - is based on the *Adaptive Communication Environment (short ACE)*[8] and uses message passing as a basis for communication. This implementation additionally provides the new concept for a *life-cycle* state automaton in a component. This concept is shown in detail in 3.2.3. Because these kind of states are important for the Monitoring concept, ACE/SMARTSOFT is considered as a basis for the implementation as introduced in 4.4.

### 2.2.2. Robot Operating System (ROS)

ROS is a package[9] that consists of a library for a robotic middleware and a set of tools which support the development of components for this library. An introduction for the ideas and design criteria of the framework can be found in [Quigley2009].

The framework consists of a base library and components (also referred to as *stacks*) which are implemented using the base library. The base library represents an OS-abstraction with the following parts. First there are *Nodes* that are connected to one *Master*. The Master represents a naming-service that stores the addresses of Nodes. With that, Nodes can find each other. A Node represents the address of a component. The communication between components is realized with *Topics* and *Services*. The addresses of Topics and Services are stored (together with Nodes) in the Master. Topics represent a *publisher-consumer* relationship with a $n : m$ cardinality. This communication mechanism is based on message passing. Services represent a request-response (or a query) communication, which is based on communication of messages.

The tools in ROS can be classified according to their purpose. For example there is a set of tools like `roscd`, `rosmake`, etc. that are similar to common bash tools on Linux. In fact these tools are often mappings of the original tools with a certain configuration (like the environment variables needed by ROS components). Further there are tools that represent stand alone programs with a certain goal in mind. For example there are a number of tools for *visualization* and *monitoring* purposes as also depicted in [Quigley2009]. Some of these tools are analysed in the following subsections.

[6] MDSD Toolchain is public available at http://sourceforge.net/projects/smart-robotics/
[7] Online available on SourceForge at http://sourceforge.net/projects/smart-robotics/
[8] ACE is online available at http://www.cs.wustl.edu/~schmidt/ACE.html
[9] ROS is online available at http://www.ros.org/wiki/

#### 2.2.2.1. ROS RXCONSOLE

The tool RXCONSOLE (see screen-shot in Figure 2.2) implements some kind of a remote console for components in the system. This tool addresses one particular problem that appears during development of components. A component - that is under development - typically prints a lot of information and debug messages to the standard output stream (to the console window). These print-outs can be additionally created from different threads in the component, which leads to cluttered and mixed-up lines in the console.

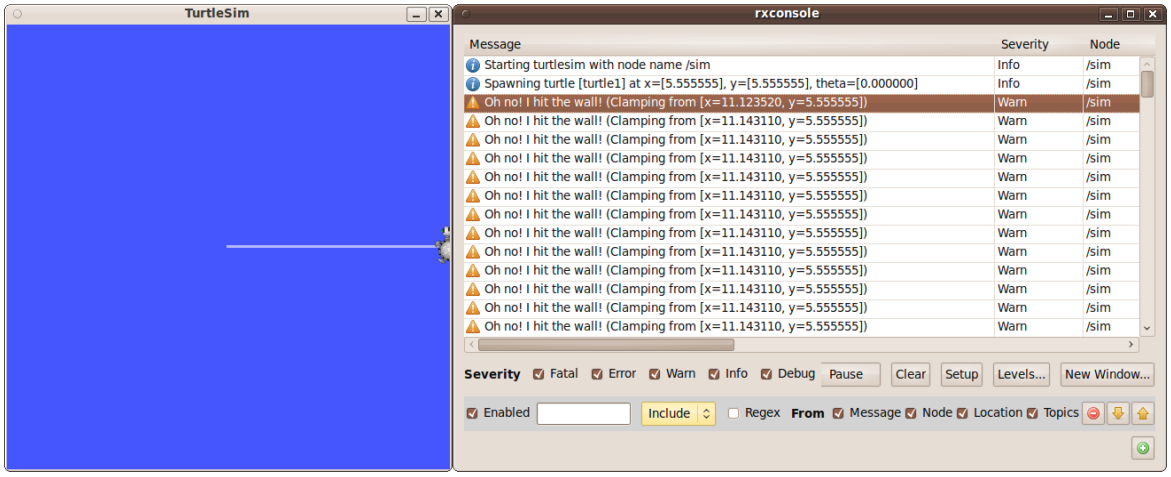


Figure 2.2.: ROS Console (RXCONSOLE)

As shown in Figure 2.2 the GUI of the RXCONSOLE tool shows messages that are created in a particular component. For that the messages are generated in this component with an interface similar to the `printf(...)` function of the standard C library. Then these messages are redirected from standard output to a specific message port. This port is connected to the `rosout topic`. The RXCONSOLE tool connects to this topic and lists the messages in its GUI. Added value comes from the filtering of the messages. For that the GUI can activate/deactivate specific types of messages and activate a filter - which is based on regular expressions - for the content of messages.

This tool provides the following two advantages. First, it offers a generalized access possibility for information that is generated somewhere in the system. Second, different message types can be masked out to reduce the amount of messages in the GUI to be able to see only relevant information (that is of current interest). The tool can be started multiple times to see information from different components.

On the other hand, this tool has some limitations. First, the topic *rosout* is a bottleneck for communication. All messages that are generated in the system are transmitted to the topic independent of the configuration of the filter in the GUI of RXCONSOLE. There is no clear semantic for the severity of the messages. For example it is not clear which messages are of the *Info* and which of the *Debug* type.

Taking all pros and cons into account, this tool provides a great value for development of components as it is. However, solving the disadvantages would make the tool even more useful, which makes the ideas behind this tool very interesting for this work.

#### 2.2.2.2. ROS RXBAG

The tool RXBAG implements at first glance a *logging* functionality. It offers the feature to connect to several *topics* - which are publishers in the system - as additional clients and to log the data in a specialized buffer labeled a *BAG*. The data stored in a BAG can be displayed in the RXBAG GUI (see screen-shot in Figure 2.3) and can be even replayed back into the system by creating additional (virtual) publishers.

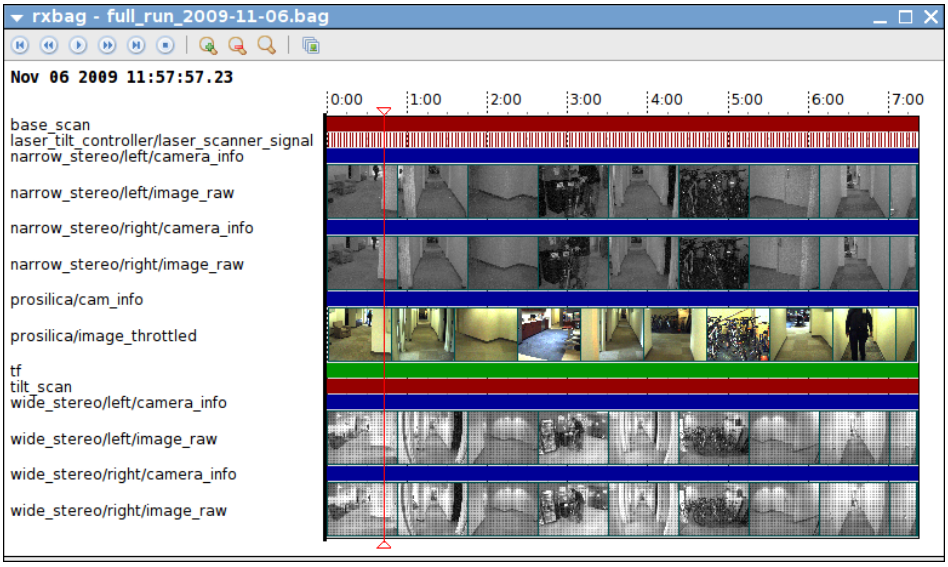


Figure 2.3.: ROS RXBAG logging tool[10]

Although the tool shows some interesting features it has also some limitations. First, unfortunately it works only with topics, ignoring the *services* in ROS. This is a considerable drawback above all for the reply feature of this tool. The visualization capabilities are reduced to primitive data-types and a few hard coded data-types like images of a specific format. The RXBAG tool runs internally a high precision clock and stamps each incoming data set with a time stamp. Unfortunately the time when data arrives in the RXBAG tool is different to the time when the same data arrives in other clients which leads to different timings in case of replaying the data. In complex scenarios with intense communication between many participants this can lead to a completely different overall behavior with deviation between real world and replayed executions.

To summarize, the tool seems to be driven by those requirements which are easy to implement in this framework. Therefore, the tool leaks on generality for the problems that are related to logging and replying of data in a distributed system. Nevertheless, this problem-cluster is of interest for this work and is valuable to be evaluated further.

#### 2.2.2.3. ROS RXPLOT

The tool RXPLOT (see screen-shot in Figure 2.4) implements a feature to plot data coming from *topics* over time. Unfortunately this feature is very limited to only simple data types like integers, floats, etc. Typical robotic applications often show more complex communication structures (like laser

[10]Online available at http://www.ros.org/wiki/rxbag

scans, binary data from sensors, current states, etc.), which require specialized visualization GUIs in most cases. Additionally, even simple data types seldom show some kind of sinus curves, but have fluctuating values where other aspects like minimal, maximal and average values or communication frequencies, etc. are of more interest. Finally, this tool has the same limitation as `RXBAG` namely it can only handle data coming from *topics* (ignoring services).

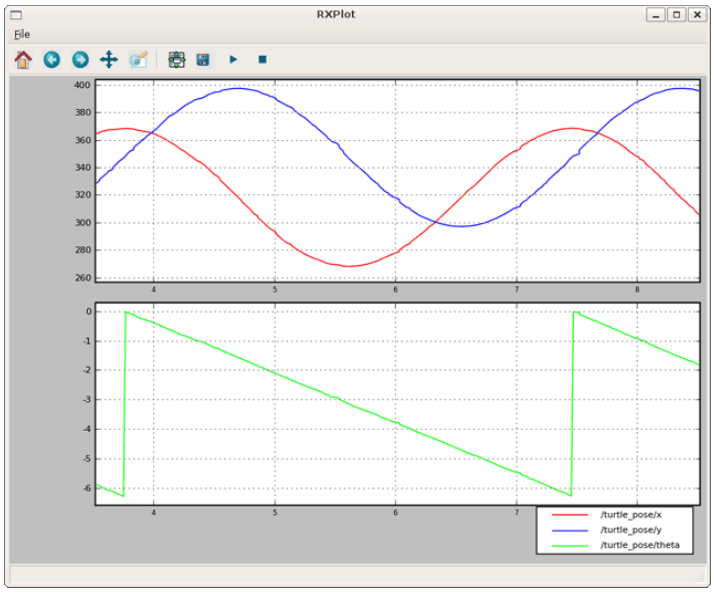


Figure 2.4.: ROS RXPLOT tool[11]

The problems of this tool as described above lead to the conclusion that it is not valuable enough to support developers during development. However, information that depends on temporal issues is still relevant in this work and must be analysed further.

#### 2.2.2.4. ROS RXGRAPH

The tool `RXGRAPH` shows the connections between publishers and consumers (clients) in a graph representation (see screenshot in Figure 2.5). Similar to the tools described above, `RXGRAPH` works on *topics* (ignoring *services*). However, in this case this limitation is not vital, because *services* are not connection-oriented and thus can not be visualized in a graph manner. For topics on the other hand this tool offers a helpful feature to visualise all connections in a graph representation, where the nodes are visualized as *ellipses* and the connections are drawn as directed *edges* between the ellipses. The direction of an edge (arrow) goes from a server to a client.

This tool seems to bring a considerable value during development of a system composed out of components which communicate through connection oriented protocols. In particular if the middleware allows to change the connections at run-time (rewiring, dynamic wiring, etc.) this feature can lead to a great value for developers of such systems and is worth to be considered further in this work.

[11] Online available at http://www.ros.org/wiki/rxplot
[12] Online available at http://www.ros.org/wiki/rxgraph

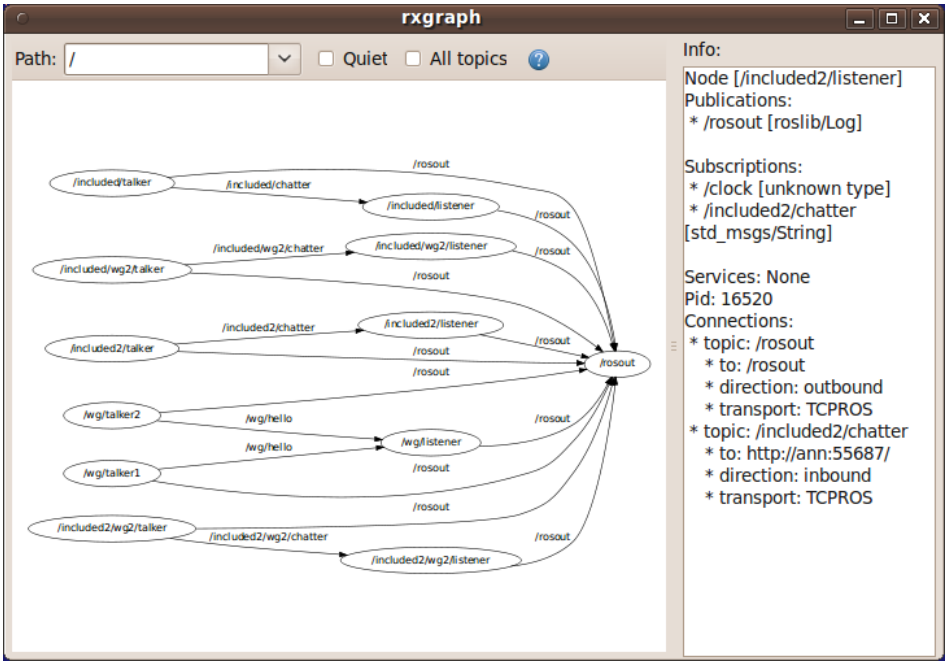


Figure 2.5.: ROS RXGRAPH connection-graph tool[12]

### 2.2.2.5. RVis and Gazebo

Two more tools are mentionable in ROS. First, the *Gazebo* 3D simulation is included in ROS. Gazebo is originally available with the Player/Stage/Gazebo project[13]. Gazebo provides a powerful 3D rendering engine with the possibility to visualize different robot platforms (with its movable parts) and even complete environments (with walls, furniture, etc.). Additionally a physics engine simulates the dynamics of robot platforms in the environment. A similar 3D simulation can be found in Microsoft Robotics Studio[14].

Second, ROS provides the *RViz* tool (see screenshot in Figure 2.6). This tool focuses on the visualisation of particular data-structures that are communicated via topics in ROS. Whether the visualization of services is possible is not clear, because only topics can be configured in the GUI. This tool allows to visualize sensor values from particular topics of appropriate components. It presents a valuable feature during development of components, because it allows to see the data values that are communicated directly at run-time. The tool allows to visualize around 15 predefined data structures[15] (like Map, Pose, PointCloud, etc.). This approach has also its limitations. The data-values can only be viewed and can not be measured or further analysed. At the date of writing it is not possible to define own structures for visualization (which are called plugins in RViz), the homepage says:

> *If you're a programmer and are looking to write a plugin, there is no documentation at this time. This is on purpose, as the plugin API is not considered stable, and so is not yet ready to be supported.*[16]

[13] Online available at http://playerstage.sourceforge.net/gazebo/gazebo.html
[14] http://www.microsoft.com/robotics/
[15] http://www.ros.org/wiki/rviz/DisplayTypes
[16] Online available at http://www.ros.org/wiki/rviz/UserGuide (last viewed on 7th of September 2010)

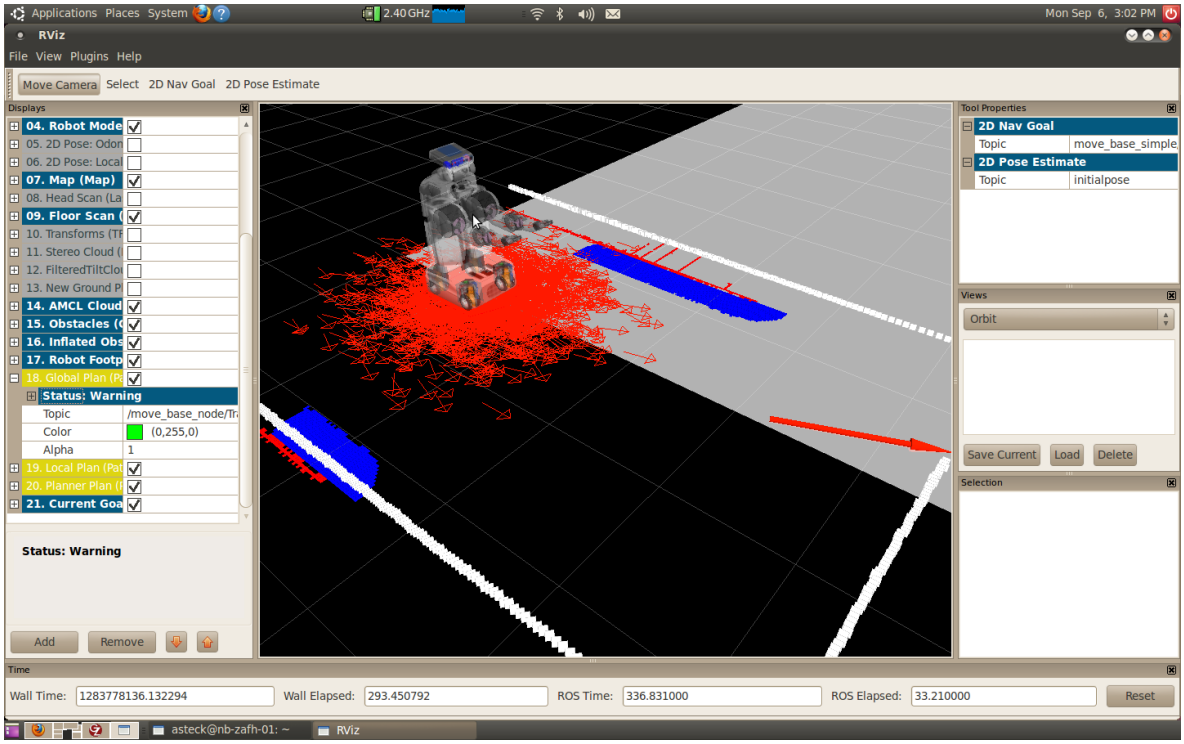


Figure 2.6.: RViz visualization tool

One particular aspect is of further interest, namely the *status* that is displayed in RViz (see Figure 2.7). This status shows whether a plugin is connected correctly to a specific topic and whether the communication provides correct messages. The latter can be determined by calculating a check-sum and by comparing the value with the received one. Additionally some hardware-analytics similar to those mentioned in the subsequent subsection are shown as a status. Unfortunately the documentation raises many open questions on the semantics and the meaning.

#### 2.2.2.6. Other tools

Additionally to the tools - as presented above - ROS provides the stack *diagnostics*, which consists of some tools for run-time analysis and diagnosis of hardware for the PR2 platform. The main focus of these tools is to monitor the hardware that is used in different components and to present this information in the *runtime-monitor*. A hardware can be a simple sensor or actor as well as more complex embedded systems. Again, the documentation raises many open questions on the functionality. In this work hardware-monitoring is considered to be a local problem inside of a components where only the results of local hardware monitoring are transmitted to a Monitor component.

### 2.2.3. Robot Technology Middleware (RT-Middleware)

RT-Middleware[17] [Ando2005] is a specification for a robotic middleware that is developed at National Institute of Advanced Industrial Science and Technology (short AIST). The main goal is to develop

[17]RT means Robot Technology in this context not real-time!

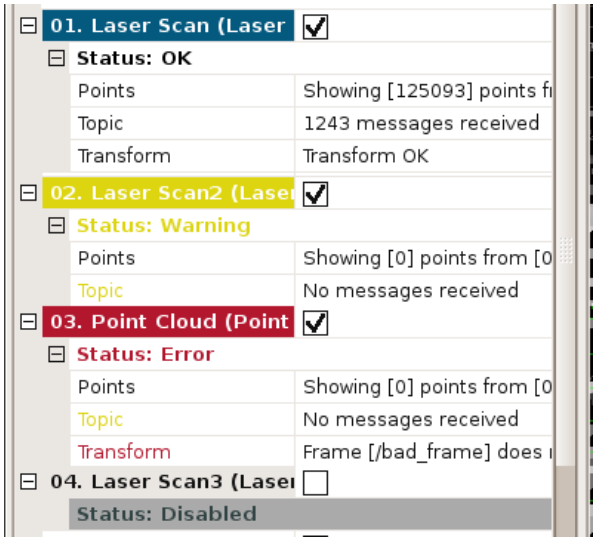


Figure 2.7.: Display status of one particular plugin in RViz[16]

several standards for robotic applications and make it publicly available at OMG[18]. One outcome is the *Robot Technology Component (short RTC)* specification[19] at OMG. A publicly available reference implementation for this specification is the OpenRTM-aist[20] project. This project implements the RTC specification using CORBA as basis.

#### 2.2.3.1. Architecture overview

The architecture of RT-Middleware provides a set of interfaces and structures for each component (as shown in Figure 2.8).

First the communication between rt-components can be realized by using *Data-Ports* and *Services*. It is further distinguished between *input ports* (InPort) and *output ports* (OutPort). This allows to realize a publisher-subscriber relationship which is based on messages. This kind of communication is referred to be active. This means that an OutPort must be actively filled with data using the `put` method and that the data in InPort must be actively retrieved using the `get` method. Contrary thereto a *Service* is seen to be passive. It is also distinguished between *service-provider* and *service-consumer*. A service-consumer requests a service from a service-provider. A service-provider can process the service request in the same scope without using an own thread of control (hence passive).

From the scope of this work, the more interesting parts of the rt-middleware architecture are the internal *state automaton* in each component and the *generic interface* on each component hull. The state automaton is described in more detail in the next subsection 2.2.3.2. The generic interface further consists of the three parts namely *Configuration Interface*, *RTC Interface* and *RTCEx Interface* (as shown in Figure 2.8). These Configuration Interfaces are further described with their user interfaces in the toolchain of rt-middleware in subsection 2.2.3.3. The other two interfaces are described - together with their relations to the state automaton - in the next subsection 2.2.3.2.

[18] OMG: Object Management Group
[19] http://www.omg.org/spec/RTC/1.0/
[20] http://www.openrtm.org/
[21] http://www.openrtm.org/OpenRTM-aist/html-en/Documents2FRT-Middleware20Overview.html

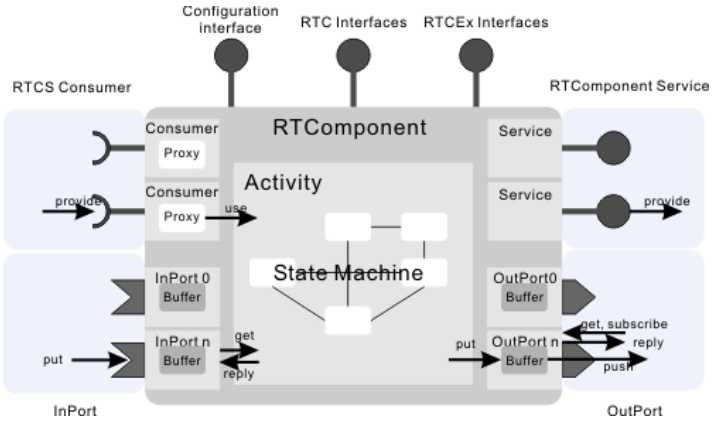


Figure 2.8.: Architecture overview of RT-Middleware[21]

#### 2.2.3.2. State automaton

The state automaton in rt-middleware is a key part of every component. It describes both the main structure of a component and the main instruments of control for a component and for its thread of control.

The state automaton in a rt-component (as shown in Figure 2.9) has different levels. The top level provides a *life-cycle* of the component with the states: *Created*, *Alive* and *Finish*. A rt-component typically steps through these three states from its creation to its abortion. The state *Alive* is the main state for rt-component's regular execution. It consists of the following two parts (which are shown as separate regions in the Alive state in Figure 2.9).

The top region consists of the sub-states *Stopped* and *Running*. These sub-states control the current state of the *execution-context* which is an abstract representation of a thread in rt-middleware. In the *Running* state the execution context (resp. its thread) is started and triggers the method `onExecute` either periodically or in a loop (depending on the configuration). The method `onExecute` inherits the user code of the corresponding component. An execution context can trigger the `onExecute` method either of one particular component or of a composed component. In the later case the `onExecute` methods of all components inside of the composed component are called *sequentially*. In the state *Stopped* an execution context is blocked and does not trigger the `onExecute` method(s).

The bottom region consists of the states *Inactive*, *Active* and *Error*. These states represent the current run-time state of a rt-component. In the *Inactive* and the *Error* states, the method `onExecute` of this component is not triggered by an execution context (independent of the state of this execution context). The relationship between rt-components and execution contexts is as follows. A rt-component owns one particular execution context. This connects the life-cycle of a component with an execution context. Thus the creation (resp. destruction) of a component also controls the creation/destruction of a particular execution context. However, as described above an execution context can trigger multiple components independently of the ownership. In rt-middleware this relationship is called *participation*.

One interesting aspect is that both regions can be controlled from the outside of a rt-component through *generic interfaces* as mentioned in 2.2.3.1 and shown in figure 2.8. The interface *RTC Interface* controls the lower region (which is the state automaton of the rt-component) and the interface *RTCEx Interface* controls the top region (which is the *owned* execution context of this component).

[22] http://www.openrtm.org/OpenRTM-aist/html-en/Documents2FProgramming20RT-Component.html

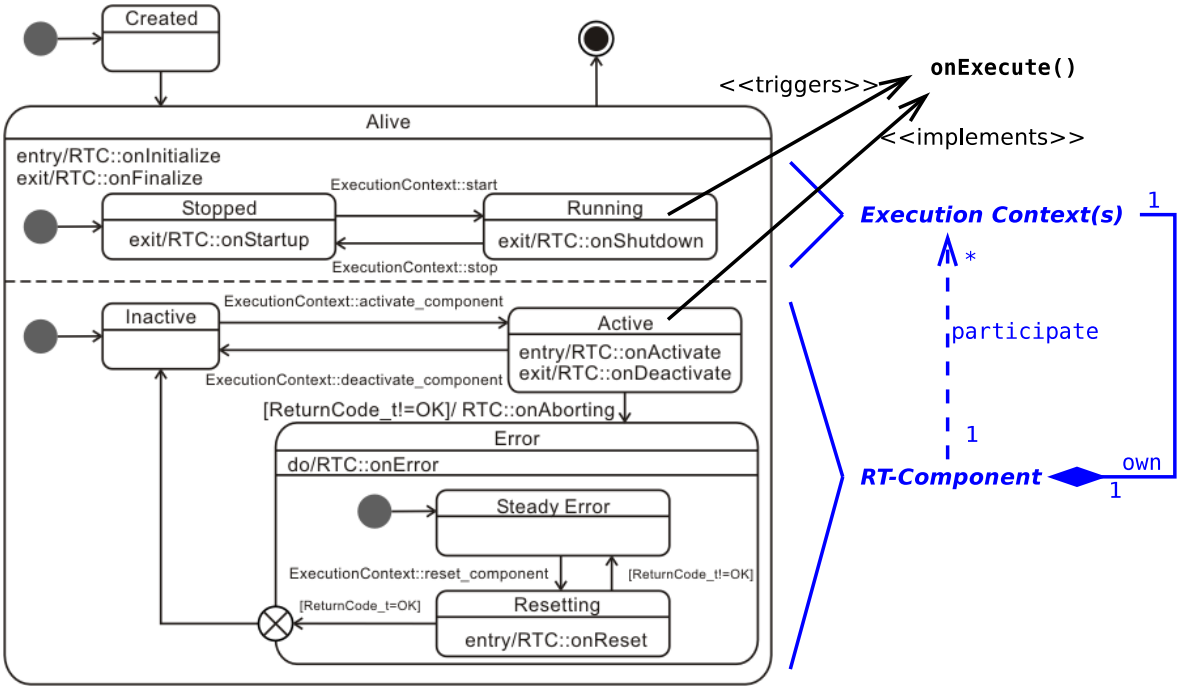


Figure 2.9.: State automaton of RT-Middleware[22]

Both automatons can be controlled independently of each other.

### 2.2.3.3. Toolchain overview

Besides the framework library OpenRTM provides a toolchain that implements some important aspects of rt-middleware. The toolchain can be divided into two main parts. First, the *offline* part is the *RTC Builder* tool which is used to create the structure for new components (see Figure 2.10). Second, the toolchain provides an *on-line* part the *RT System Editor* as shown in Figure 2.12. This tool controls the configuration and other parameters of components in a running system.

The *RTC Builder* tool (as shown in Figure 2.10) provides a *template* window to set-up all necessary parameters and configurations for a new rt-component. The *BuildView* shows a live overview of the currently configured component. The template allows to configure different parts of the component like *Component Profile* (the *Basic* frame), *Data Ports*, *Service Ports* and *Parameter Configuration* (see Figure 2.11).

The *RTC Builder* tool uses the information from its template to generate the source code for the rt-component's hull. The source-code consists of both the structure of the component and its initial parameters. The parameters which are set-up in the *Configuration* frame for parameter definitions as shown in the Figure 2.11 can be used later at run-time either inside of the component or outside on the component's hull through the *Configuration Interface* as mentioned in 2.2.3.1 and shown in Figure 2.8. Before starting the component a developer must include the source-code for this component into the `onExecute` method. After that the components of the system can be started and the system can be controlled with the *RT System Editor* tool (as shown in Figure 2.12).

The run-time part of the rt-middleware, namely the *RT System Editor* as shown in Figure 2.12, consists of the following parts. First, the *Naming-Service* view ① shows addresses of components that are

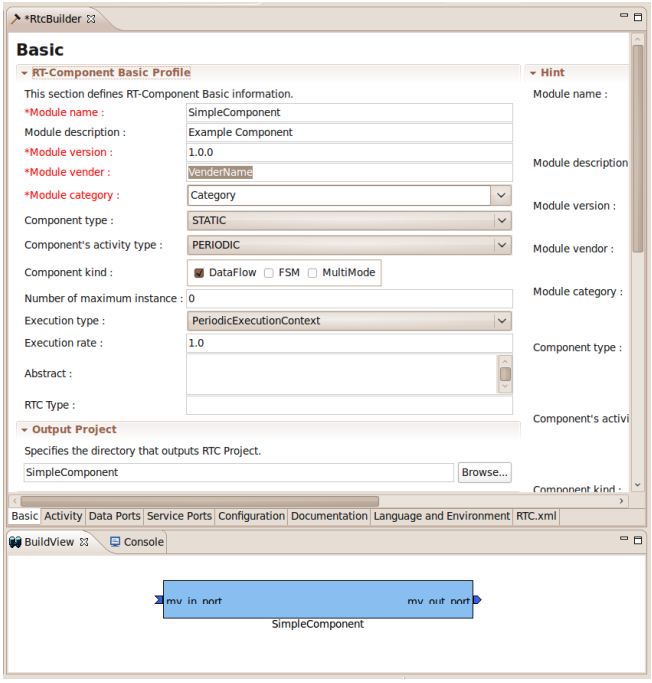


Figure 2.10.: RTC-Builder window

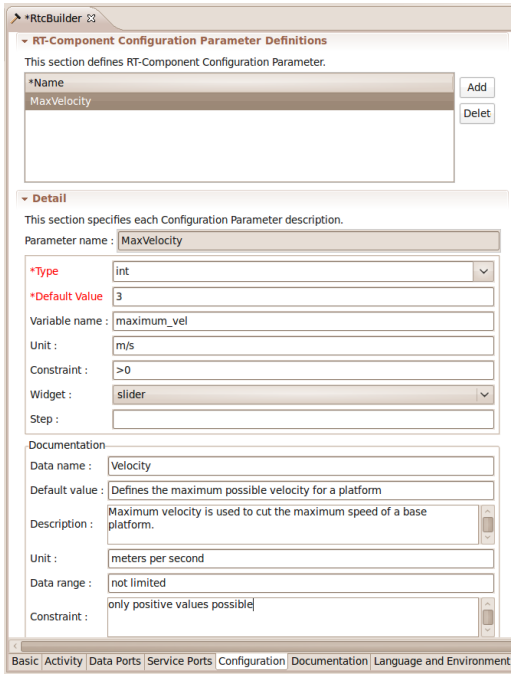


Figure 2.11.: Configuration window

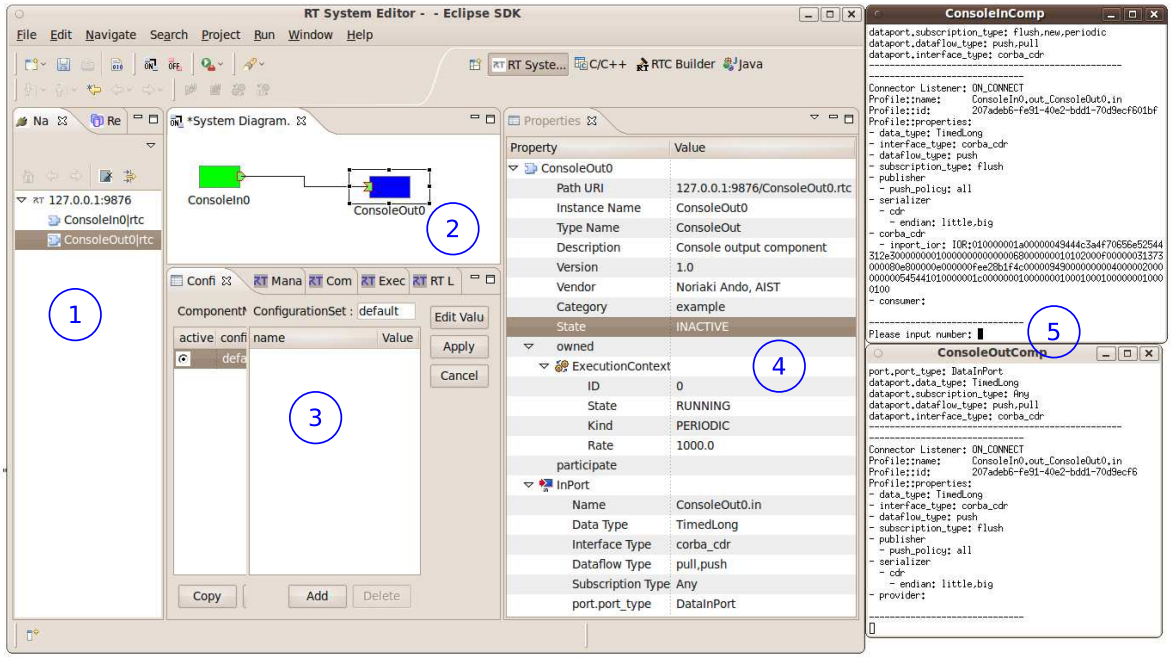


Figure 2.12.: RT-Middleware Toolchain

registered in a particular *naming-service*. By selecting one of the components in the list its properties are shown in the *Properties* view ④. Several components can be dragged from ① and dropped in the *System Diagram* view ②. After that, the connections between these components can be established. Additionally the view ② allows to group the components either as *ECShare* (all components share the same execution context) or *Grouped* (each component is triggered from a separate execution context). The *Configuration* view ③ provides a GUI for the *Configuration Interface* of a selected component. The parameters which are defined in the configuration window as shown in Figure 2.11 can be visualised and modified at run-time in this view ③. Finally, each component is started on Linux as separate process and runs typically in a console (this is shown as ⑤ in Figure 2.12).

#### 2.2.3.4. Conclusion

Though the idea of standards in the robotic domain is highly desirable the specification seems not been fully accepted as a basis for robotic applications. One of the reasons may be related to the level of granularity for components that can be constructed with RT-Middleware. As for example mentioned in [Schlegel2009], components need to be constructed at different levels of granularity. Although RT-Middleware offers the feature of building composed components, this is not sufficient. The reason is that it is not possible to hide (and encapsulate) internal structures inside of composed components which is however an important issue to reduce complexity of the overall system and to use a component as a black box in different systems.

Generic interfaces (as shown in 2.2.3.1) can be used to monitor the current state (as shown in 2.2.3.2) of the component. With that it is possible to build some kind of a Monitor component that automatically connects to all *RTC Interfaces* of the components running in a system and to monitor its *lifecycle* and *activity states*.

The toolchain as shown in 2.2.3.3 provides a simple code generator that is based on scripting. This makes the generation process very static without the possibility of adding customized aspects to the system. A better solution is to use more sophisticated approaches from MDSD[23] like the OAW[24] toolchain as also described in [Steck2010]. The information that is collected in the *RTC Builder* tool like *Component Profile*, *Port Profile(s)*, etc. is important as a basis for monitoring as understood in this work and is thus worth to be considered later in the main part. However, this information should be generated by a MDSD toolchain, as there is most of this information available and accessible.

### 2.2.4. Microsoft Robotic Studio

*Microsoft Robotics Studio* (short MSRS) provides a service oriented architecture that allows to build components which communicate among each other by using web-services [Jackson2007]. There are different levels for communication possible like shared memory for the case where components run on the same node or SOAP[25] for the distributed case. Using SOAP provides a simple way to observe the communication between service-nodes in a browser. Additionally to the library, a rich tool-chain is provided. Among others there is an IDE for development of applications, a 3D simulation (with a physics engine) and the *DSS Log Analyzer* tool (as shown in Figure 2.13) in particular. With DSS Log Analyzer it is possible first to log the messages from the communication between services in separate files for each service-node. Afterwards the messages can be loaded in the tool and can be visualized

[23] MDSD: Model-Driven Software Development
[24] OAW: Open Architecture-Ware (http://www.openarchitectureware.org/)
[25] SOAP: originally defined as Simple Object Access Protocol

in two different views, namely the *Message Flow* and the *Message Details*.

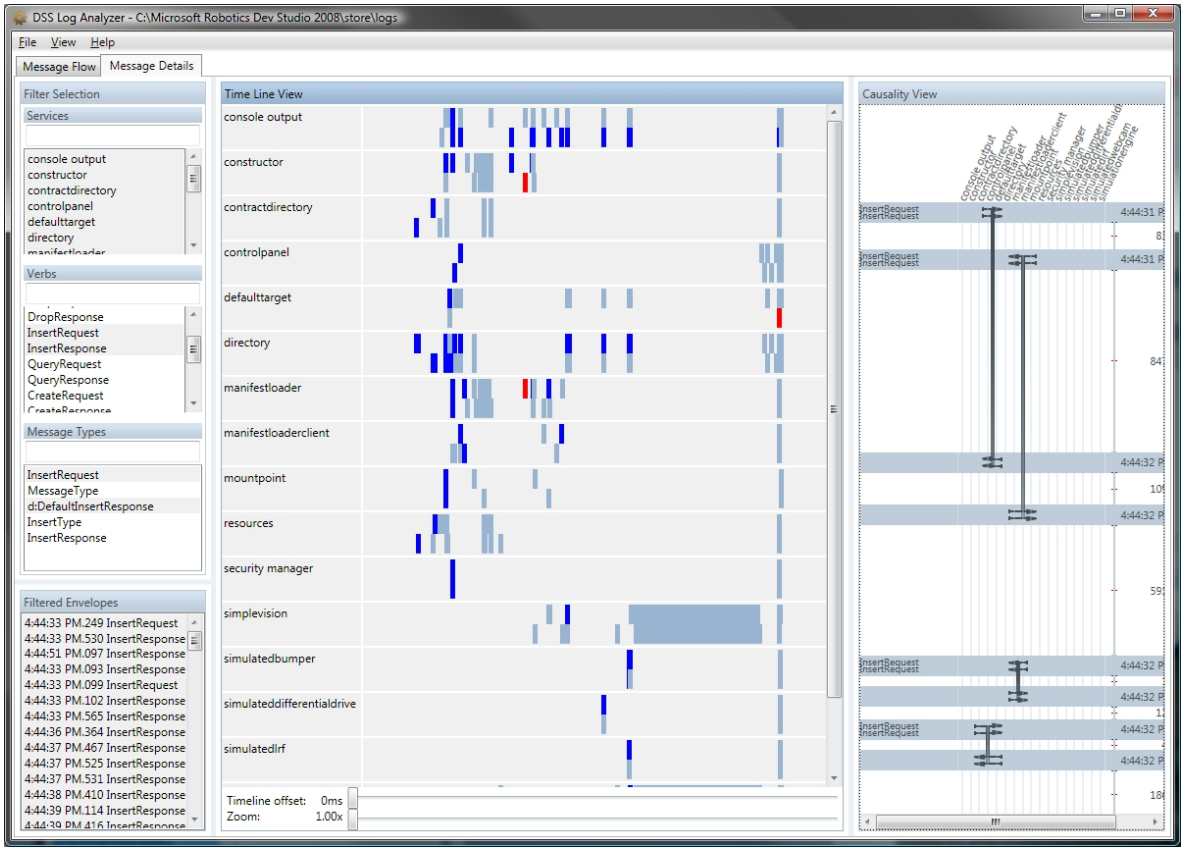


Figure 2.13.: DSS Log Analyzer of Microsoft Robotic Studio[26]

The former provides a graphical representation of all nodes and the messages that are communicated between these nodes for the captured period of time. The latter provides further details for single messages in a selected node. In a time-line a graphical representation shows the occurrences of messages (see screen-shot in Figure 2.13). It can be further distinguished between different message types like request/response or error messages. Finally, some simple (and predefined) visualization possibilities for single messages are provided (i.e. a WebCam visualization).

As also depicted in the [Jackson2007] there are two main limitations. First, this approach is not suitable for real-time applications. For these cases some kind of a gateway to the real-time system is suggested. Second, all services require a full .NET Framework as a basis, which enforces to port existing implementations of components. From the Monitoring point of view, MSRS provide a further problem - the DSS Log Analyser works only if SOAP is chosen for communication. This is a considerable restriction that is contrary to the requirement of efficient communication in many robotic scenarios and is thus not feasible as a basis in this work.

[26] http://msdn.microsoft.com/en-us/library/dd939178.aspx

### 2.2.5. Other Robotic Middlewares

There are many other robotic middlewares which can be found in the filed of robotics. Two of them are noteworthy. The first is the *Open Robot Control Software (OROCOS)* framework [Bruyninckx2001]. This framework provides for example the *TaskBrowser*. This is a feature in each component that allows to query for particular information of a component directly from a console (where the component is started). In the broadest sense this can be seen as a local Monitoring. On the one hand this seems to be a valuable feature, because the information can be queried in a way that is similar to a bash console (which is familiar to many developers). On the other hand the documentation raises many open questions about the usage of this information on the system level.

One further mentionable framework is the *Generator of Modules (GenoM) and GenoM3 in particular* [Mallet2010]. Although it provides an interesting feature called *Control Posters* which represent the messages from a component (resp. a module). These messages represent the current status information from a component. Again, even the most recent paper leaves many open questions on the clear semantic of this control posters. Thus it is not considered in this work.

Finally, there are other robotic middlewares like YARP, Orca, Marie, CLARATy, PlayerStage, CARMEN, etc. which are however not considered to be relevant in this work due to missing Monitoring capabilities or due to experimental implementations.

# 3. Fundamentals

This chapter presents the information that is necessary to understand the main part of this work in the next chapter.

## 3.1. The need for Monitoring in the robotic domain

The need for Monitoring in the robotic domain is twofold. First there are special characteristics that are common to robotic applications and define the scope for such an application including Monitoring aspects. Second there is a gap between the development of components and the integration of robustness and fault-tolerance into existing robotic applications. The gap is related to the absence of specific data outside of components that is needed by some approaches like fault-tolerance. This chapter goes into detail on these two aspects and shows some possible interfaces to these systems.

### 3.1.1. General needs in typical robotic applications

One of the common approaches to cope with complexity in robotic applications is to build up these applications of loosely coupled components with local responsibilities. This approach is also referred to as *Component Based Software Engineering (short CBSE)* as shown in [Brooks2005]. However, CBSE alone is not enough to build reusable and generic components. A clear semantic and developer friendly interfaces for the communication mechanism between components are the key parts towards component-based robotics as shown in [Schlegel2006]. In recent time the shift from code-driven to model-driven designs becomes additionally more and more important [Schlegel2009]. A recent approach for *Model Driven Software Engineering (short MDSE)* is presented in [Steck2010]. All these advances improves and simplifies the development of robotic applications. However, there is still a gap between CBSE and MDSE that is related to the Monitoring as understood in this thesis.

Systems that are composed out of encapsulated components hide low level details from the system level. On the one hand this lead to several advantages. It is possible to separate concerns in the system according to individual tasks (which are in turn encapsulated in single components). For example, the encapsulation limits the spreading of errors to the boundaries of one particular component. Thus, it is easy to isolate possible errors.

On the other hand the encapsulation comes with a price. It becomes more difficult to get an in-view into running components. However, this is sometimes necessary as shown in some use-cases at the beginning of this thesis. It is not reasonable to drill holes into the system to satisfy this needs, because this is directly conflicting with the CBSE approach. It is rather reasonable to develop a generic mechanism that collects those kind of information as depicted in the use-cases. Additionally to the use-cases this information can be used as an additional source of information for the MDSE related issues. For example it is possible to use the information coming from Monitoring as input for the *deployment* in the “MDSD Toolchain” which is described in [Steck2010]. Thus, this information can be used as a feedback from the system in the MDSD Toolchain. Finally, Monitoring can be used as a generic source of information to improve robustness and fault-tolerance as introduced in the following.

### 3.1.2. Robustness and Fault-Tolerance

As the significance of service robotics in the industry grows, the need for robust and fault-tolerant systems becomes more and more important. Many approaches for this field can be found in the literature.

A general definition and discussion on dependable and secure systems is given in [Avizienis2004]. This paper gives the main definitions relating to dependability and security and evolves some generic approaches which can be applied to different domains. The definitions of the terms *robustness* and *fault-tolerance* are used by Lussier et al. in his paper [Lussier2004] and are confirmed to be crucial parts of complex autonomous systems. Though both terms describe similar problems there are some important differences between them. Robustness describes the property of a system to be robust against unexpected situations (e.g. difficult lighting conditions like changing illumination, or dynamics in the environment like moving obstacles). Fault-tolerance on the other hand describes the ability of a system to cope with errors and failures coming from the inside of a system. This means that the system has to detect and to analyse errors and failures as well as to run appropriate error handling strategies. In many cases it is difficult to examine these two terms separately, because they are directly or indirectly related. For example many errors inside of the system result from sensors producing unexpected data coming from sensing in unexpected environments. This data causes unexpected behaviors in algorithms that in turn can lead to errors. On the other hand the system is often more robust if the algorithm implements extensive error handling mechanisms.

In general, the approaches for fault tolerance and robustness have the following consequence. There are different levels of abstraction. A low level addresses the problems related to local identification of errors (often on the source code basis). A high level addresses issues related to the analysis and diagnosis of faults and errors on the system level. By applying these approaches to the robotic needs a gap between this two approaches can be identified. The results of the local error detection must be generalized and provided on the system level. This is the level where Monitoring as understood in this work is placed. Having this information on system level (or in Monitor components in particular) allows to apply different approaches for error and fault diagnosis and to react in a secure and robust manner.

### 3.1.3. Testing

Testing is one of the common means for component- and system development. There are different types of tests possible. For example *black-box testing* can be used for component testing. In this case a component is seen as a black-box which has entry- and exit data and does some calculations on it. However, components for typical robotic scenarios are sometimes difficult to test. The reason is that environments where robots have to interact are very complex and often cannot be admissible simulated to get comparable testing environments. Thus, it is difficult to build appropriate test oracles to cope with real environmental behavior. Due to the high dynamics in the environment it is also not possible to test all conditions.

For that reason a common approach is to distinguish between algorithm testing and system testing. Algorithm testing can be done locally inside of the component by using typical software development (resp. software testing) approaches. For example if a scenario behavior is modelled using MDSD (in particular StateCharts), its model can be tested by using *model verification*. Model verification allows for example to test whether dead ends, infinite loops, etc. exist. On the other hand system testing is usually done directly on the target platform. For that all components for a particular scenario are executed directly on that platform. Each component is seen as a black-box (similar to black-box testing).

The data that is communicated between components can be either visualized in specialized components or tools like *RViz* (see 2.2.2.5), or can be monitored and analysed by using various techniques as shown in the main part of this work in chapter 4.

## 3.2. Common structures in applications build with SMARTSOFT

There are many architecture styles and suggestions on how to structure a system with its components. These architectures are implemented in various robotic middlewares as shown in Chapter 2 “Related Work”. However, one particular framework is seen to be most valuable for this work, namely SMARTSOFT and its structures as also presented in [Steck2010]. Figure 3.1 gives an overview of the key parts that are common for components in SMARTSOFT.

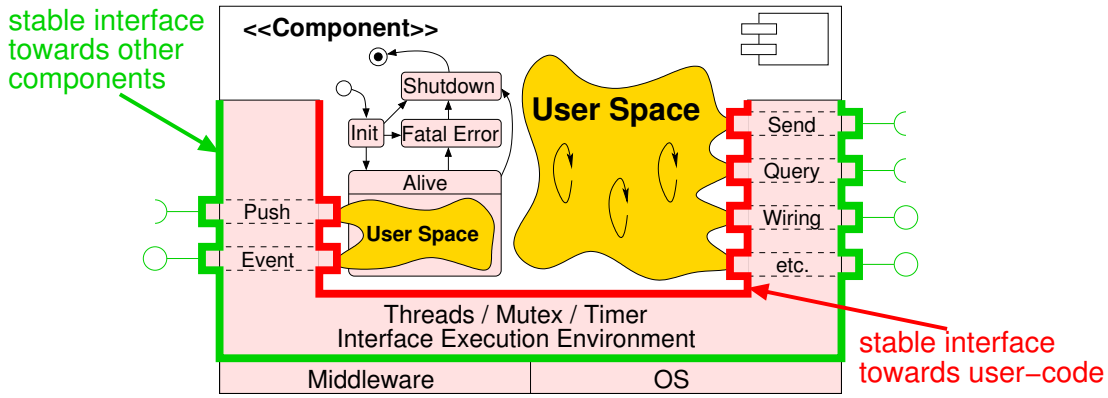


Figure 3.1.: Structure of a general component in SMARTSOFT [Steck2010]

Each component in SMARTSOFT consists of the following *generic* parts (a full description is available in [Schlegel2004] and [Steck2010]):

1. A *user-code* part - which is independent of SMARTSOFT - implements the core functionality for this component. Concurrent execution can be easily implemented by using Tasks. Arbitrary libraries and hardware can be used here. There are no restrictions on the structure. However, the communication with other components must be implemented by using *communication ports*. There are ports that are used in the component to request (resp. require) a specific service *from* other components and ports that provide a specific service *to* other components.

2. Additionally each component initializes a state-automaton that implements the *life-cycle-state* of this component. A current life-cycle-state can be *changed* from the inside of this component (from the user-code) and can be *requested* at different levels, namely inside of this component, in communication ports and outside of this component on the system level.

3. The first two parts represent the internal structure in the component (this is illustrated as a red container in Figure 3.1). The *communication* between components is realized with so called *communication ports*. Each communication port consists of the following three key parts:

   - An internal interface towards user-code. This is an abstract interface, that is independent of the underlying communication mechanism and of the framework implementation.

- An abstraction for the underlying communication mechanism. This is fully hidden from the user (i.e. no CORBA code on the interface to user-code) and from the system level (communication uses so called *communication objects*).
- An outer interface towards other components, that implements a clear semantic defined as *communication patterns* and communicates data that is well structured as *communication objects*. This interface is completely independent of any internal communication framework details.

4. The *component-hull* with its communication ports (shown in Figure 3.1 as a green, outer container) is stable towards other components and is independent of the internal implementation. This structure faces the following advantages:
   - It is possible to define components for specific tasks with stable definition of services.
   - The container with user-code inside of a specific component can be exchanged by another component with a similar signature.
   - The container with user-code can be also put in another component hull - which can use another SMARTSOFT implementation (i.e. with another communication mechanism) - without affecting the user-code part.

Of course this is only a very brief in-view into some core features of SMARTSOFT. As a conclusion the following advantages can be identified. First, the structures in SMARTSOFT are clear separated and the level of abstraction focus directly on robotic needs, hiding low level details (coming from the underlying communication middleware and operating system). The framework provides a manageable number of communication patterns, which allow to fully describe the behavior and the semantic of the communication between components in the system. Additionally, communication patterns like the StatePattern and the EventPattern allow to describe even complex communication behaviors in a simple and uniform way. All these aspects make the SMARTSOFT framework and in particular its abstraction level ideal as a basis for the Monitoring (which is presented in section 4.3). As will be shown the StatePattern and the EventPattern considerably reduces the complexity of the concept for Monitoring in robotic applications.

### 3.2.1. Three layer architecture

In the last decade one particular software architecture style showed its usefulness in robotic applications namely the *three layer architecture* ([Gat1998] and [Schlegel2004]). This architecture distinguishes between the *reactive*, *sequential* and *deliberative* layers.

The *reactive layer* (also called *skill layer*) is responsible for executing low level behaviors. Representative components for this layer are those that behave in a *sense-act* (or a *feedback-control*) paradigm and are sometimes stateless. However, this does not mean that these components must be simple. For example a Mapper and a Planner are also skill components. Characteristic for this layer is that such skill components are composed and controlled from a higher level, namely the *sequencing layer*.

The *sequential layer* (also referred to as *sequencing layer*) is responsible to control a set of skills. In doing so the skills are connected and parametrized to enable a certain high level behavior. Components at this level often have to take former states into account to decide on the next steps. This structure allows to implement complex and flexible behaviors that are composed out of multiple skills.

Finally the *deliberative layer* is responsible for the high level reasoning at a symbolic layer. In principle all kinds of high level planning can take place on this level. Specific for this level are components that build continuously a model of the environment and do *symbolic planning* on this model. With that it is possible to make a hypothesis on the future of the environment. In particular knowledge based and learning approaches are common at this level.

One disputed aspect is where the actual control of the system is implemented, at the sequencing or the deliberative layer. The deliberative layer can either take the role of the center of intelligence in the system or act as an advisor for the sequencing layer.

The consequence of this separation for Monitoring is as follows. On the one hand the communication from the sequencing layer to the skill components is typically implemented by using the Parameter ports (see the following subsection). The communication in the reverse direction is typically implemented by using the EventPattern. The latter provides the possibility to implement a feedback from skill components with customized events. The Monitoring as understood in this work can be seen as a *generic* feedback from skill components, that can be used either on the skill layer or on the sequencing layer. This does not collide with the regular events, because the focus is different. EventPatterns provide specific information like low battery event or path not found, etc. which are tightly connected to a particular component on purpose. Monitoring on the other hand provides information like a component is not in the alive state or component is unable to deliver a service. This type of information is typically applicable to all (skill) components in the system. Even a consideration of problems on higher levels is imaginable. However, the use cases for the latter idea are hard to define and must be further evaluated, which is out of scope of this work.

### 3.2.2. Recurring structures inside of typical SMARTSOFT components

Additionally to the *generic* parts the user-code part inside of components in typical robotic scenarios show some recurring structures which can be analyzed with the focus on Monitoring. Figure 3.2 shows an abstract view for a typical structure.

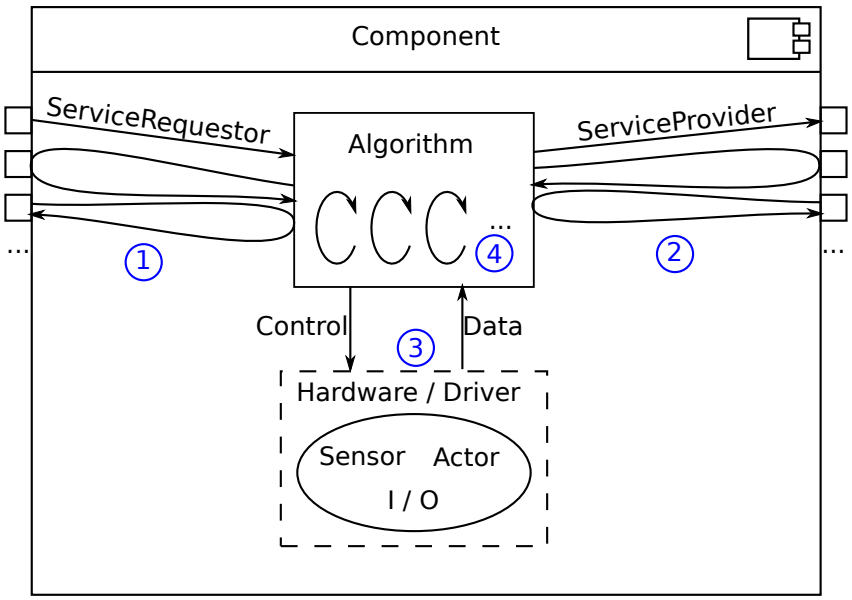


Figure 3.2.: Internals of a general component

First there is some kind of an algorithm, specific to the skill or behavior of this component. This algorithm can be divided in any number of tasks. Additionally there are sometimes input or output devices like sensors, actors or other I/O devices. In this case the algorithm often makes some transformations or filtering of data. The algorithm often needs some data from other components (shown as service-requestor input-ports) used for calculation or transformation purposes. The outcome of an algorithm is often published to other components through the service-provider ports. This structure is very common in many kind of robotic components.

Some of these components use additionally generic communication ports like *Wiring*, *State* and *Parameter*. Wiring provides a service on each component to reconnect the service-requestors from this component. This allows to rewire the net of connections between components at run-time depending on a specific scenario. StatePattern allows to control the activities of the component from the outside. Activities are controllable tasks that run in this component and that can be blocked and unblocked from other components by setting a particular MainState in the StatePattern (some further details are shown in 3.2.3). Finally, Parameter ports are used to reconfigure *skill* components from the *sequencing layer* (as described in 3.2.1). Parameters in this context are specific commands or parameter values (like thresholds, initialization values, etc.). Parameters are typically sent on a Parameter Port in a generic way. These three communication ports enable some run-time flexibility such that it is possible to reconfigure the overall system. With it, the system behavior and system properties can be changed at run-time.

### 3.2.3. Lifecycle state automaton in StatePattern

One recent development that is highly interesting for Monitoring can be found in [Schlegel2009] and is soon available in the ACE/SMARTSOFT implementation at [Lotz2010]. The core idea is quite simple. Each component in the system consists of a state automaton which represents the life-cycle of this component. This state automaton is illustrated in figure 3.3. The state automaton is implemented inside of the StatePattern in ACE/SMARTSOFT and provides two different abstraction levels. These levels are described in the following.

#### 3.2.3.1. Life-cycle automaton

A high level represents the life-cycle with the MainStates: *Init*, *Alive*, *FatalError* and *Shutdown*. These states are fix for every component in the system. Each component is responsible to set the current MainState of its state automaton. Allowed transitions are illustrated as arrows in the figure. Additionally, it is possible to command the *Shutdown* MainState from the system level by using the public interface of the StatePattern. A component is at the beginning (during initialisation) in the *Init* MainState. After the initialization is over the component changes into the *Alive* MainState. In fact, Alive is not a real MainState, but a pseudo state. Alive means that a component is neither in the initialization, nor has a fatal error nor is in the process of destruction. If a component detects a critical error (that this component is not able to handle) during initialization or at runtime, the component can switch to the FatalError MainState. The only escape from this state is to command this component to shut down. Finally a component is in the Shutdown MainState if this component is currently shutting down and is cleaning up its resources for example. The main idea is that a component can deliver proper services only if this component is in the Alive state.

As mentioned, Alive is a pseudo state. Here, the second abstraction level comes in. The Alive state can consist of various customized MainStates which can be individually defined for each component during its initialization. By default the MainState *Neutral* is defined, which indicates that this compo-

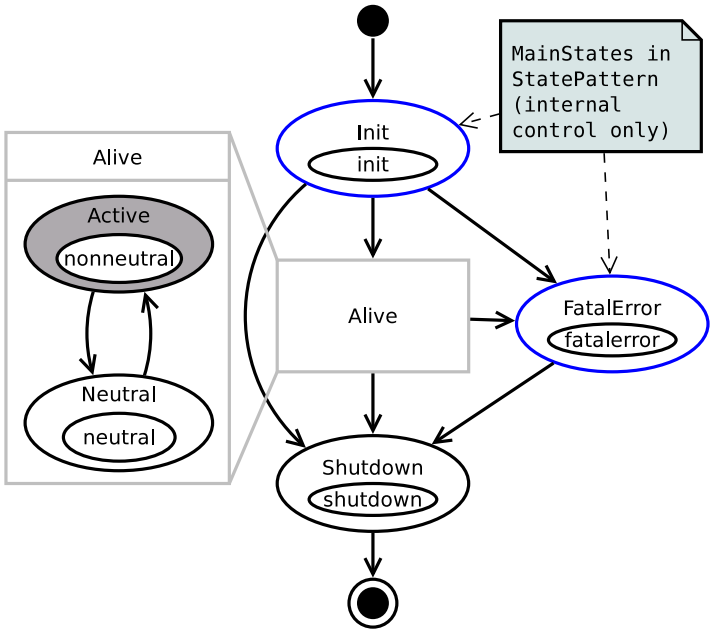


Figure 3.3.: Lifecycle automaton inside of state pattern

nent is not actively executing its task(s). This is typically the state to reconfigure this component - for example to change its parameters using the Parameter port or to change the connections of its services using the WiringPattern [Schlegel2009]. Besides this, there can be an arbitrary number of MainStates, in which the component can be at runtime.

#### 3.2.3.2. StatePattern

The base idea behind the StatePattern is described in [Schlegel2004]. Further details can be found in [Schlegel2009]. As stated there the StatePattern is different to other popular state machines (i.e. the finite state machine). The focus is different. The StatePattern focuses directly on the robotic need to control the activities (resp. the services) of a component.

> "*The state pattern is used by a component to manage transitions between service activations, to support house-keeping activities (entry/exit actions) and to hide details of private state relationships (appears as stateless interface to the outside).*" [Steck2010]

That is, the StatePattern implements a Master/Slave relationship to manage the activation and deactivation of particular activities (SmartTasks in SMARTSOFT) and therewith the services of a component. In contrast to a finite state machine in a StatePattern the Master commands MainStates directly (not by sending events). These MainStates represent a mask for a set (combination) of SubStates, which are defined on the Slave of the StatePattern. The advantage of the StatePattern is that the synchronization issues during activation (resp. deactivation) is fully transparent to the Master and are implemented inside of the Pattern. This offers considerable value for various scenarios like Mapper and Planner in SMARTSOFT for example.

#### 3.2.3.3. Conclusion

The StatePattern and the life-cycle state automaton in particular provide a generic representation of the current state of a component. This is ideally convenient for Monitoring purposes. Thus, this pattern is considered to be crucial for the Monitoring concept which is presented in 4.3. The current life-cycle state of a component can be directly used as a source of generic information that is to be monitored in a component.

# 4. Method

This chapter presents the main part of this work. It is structured into four sections. First, the research problem of this work is presented in section 4.1. This includes the collection and description of the base problems that come from use-cases as presented in chapter 1 and problems that can be found in various applications as presented in chapter 2. After that some potential approaches and their possible solutions are discussed in section 4.2. The main focus there is to evaluate the effort and the merits for the individual solutions and to decide on a sub-set of them. In section 4.3 first the requirements are derived from this sub-set. After that a holistic concept that inherits all the valuable approaches is presented. The main goal is to describe the architecture and the details of the concept. Finally, section 4.4 presents some details on the implementation. The focus there is to face the current state of the implementation and to show some details of the implementation.

## 4.1. Research Problem

The examination of the research problem is twofold. First, some dimensions for investigation are derived from the problem space (as introduced in section 1.3). Second, the main challenges and problems are identified and generalised from the use-cases (as introduced in section 1.4). These challenges and problems are grouped into some problem clusters. These clusters are used later in the subsequent section as a basis.

### 4.1.1. Dimensions of investigation

As shown in section 1.3 there are a wide range of problems that have to be analysed in this work. The first step is to define some classification parameters that help to investigate on single problems from different views. These parameters are called *dimensions* in this subsection. Knowing the dimensions it is easier to classify the problems and challenges and to group them into reasonable clusters (see 4.1.2).

The dimensions are listed in Table 4.1. The table is structured as follows. In the first column the dimension name is presented. The second (middle) column shows some possible values for this dimension. These values have different meanings. Some of these values can represent particular state-values of this dimension that cannot be compared with each other (in terms of that the one is bigger than the other, more complex, etc). Others can be directly represented as an axis. This axis has a direction that represent for example a rising complexity. This semantic is shown in the right column of the table.

### 4.1.2. Analysis of problem clusters

The problem-space as described in section 1.3 and the main challenges from use-cases as presented in section 1.4 span a bulky problem-scope which is hard to analyse. One possible solution to relax this issue is to classify the problems and to group them into reasonable clusters. This helps to focus on particular aspects of the whole Monitoring problem and to break down its overall complexity into

| *dimension* | *possible dimension values* | *axis* |
|---|---|---|
| Locality | • internals of a component are of interest<br>• communication between components is of interest<br>• internals of a Monitor component are of interest | not comparable, separation of concerns |
| Stakeholder | • important for Component Developers<br>• important for System Developers<br>• important for components with local diagnosis | rising complexity if important for more than one |
| Temporal | • independent of time (e.g. something happened or not)<br>• ordered events (but can be delayed)<br>• synchronous event occurrences in time (real-time) | rising temporal dependency |
| Autonomy | • manually initialized data that is then analysed by a developer<br>• automatically generated data that is then analysed by a developer<br>• automatically generated data that is then automatically analysed at run-time and the results are visualized in a Monitor component<br>• automatically generated data that is then automatically analysed at run-time and trigger some repair functions if necessary | expansion of the autonomy of data |
| Completeness | • only current snap-shot of the system/component is necessary<br>• current snap-shot and its history are needed<br>• current snap-shot, its history and the projection in the future are necessary | expansion of the completeness of data |
| Generality | • generic data (e.g. component liveliness-status)<br>• system specific data (e.g. Communication Objects)<br>• very component / algorithm specific data (e.g. current local values, smart-prints, etc.) | rising complexity to access and to analyse the data |
| Distribution | • local information that is inside of a component available<br>• information that is system wide available (but still at one specific location)<br>• information that is system wide available and distributed among several components | rising distribution, rising complexity to collect the data |

Table 4.1.: Dimensions

handy peaces. Hence, the problem clusters as presented in the following represent islands of problems that belong together.

#### 4.1.2.1. Vital parts in a component

**Problem statement:** *A component must know its **vital parts** and must monitor the **current state** of them.*

*Vital parts* in a component are those parts that are essential for the execution of one particular service from this component.

| ***category*** | ***examples*** | ***responsible for*** |
|---|---|---|
| Hardware resources | sensors, actors and other I/O | consume / deliver data |
| Software resources | libraries | calculate / require data |
| Communication ports | service requestors | use a remote service from other components |
| Activities | Tasks (with SubStates from StatePattern) | calculations for a particular service |

Table 4.2.: Examples for vital-parts inside of a component

Some examples for such vital parts are listed in Table 4.2. As shown there, most of these parts are related to data-flow inside of the component. The *generality* dimension can be used to identify different levels of data. One of the generic data types for most of the resources is their current state. For example a hardware or a software resource can provide its operational state that shows whether this resource is initialized and is ready to deliver (resp. receive) data. Further, a task can also show its operational state (i.e. running or stopped) or even whether a corresponding *SubState* of a StatePattern is activated. Finally, a Client Port can provide its connection state that indicates that this client is ready to receive (resp. request) data or a service from a remote Server Port in another component.

Most of these data types can be seen as generic data (of the *generality* dimension). Less generic data types are those that cannot be automatically generated. In this cases the component developer must program appropriate routines which provide the information that for example a particular library is initialized and is ready to be used.

The *stakeholder* dimension shows in this case that this information is used by all of them, the component and system developers or by other components in the system. The kind of stakeholders influence other dimensions like *locality*, *autonomy* and *distribution*. In case that the stakeholder is a component developer, the information is generated automatically (or manually) inside of a particular component and is transmitted to a Monitor component (where it is typically visualized). The developer can now get the full information detail and can use it for example for testing purposes.

In case the stakeholder is one of the components in the system, the information must be generated automatically inside of the component of interest and must be made available on the communication level (also referred to as service level). Those components in the system that depend on the service of the component of interest must react on this information in a customized way. For example such components can wait till the service is available or choose another component with the same or similar service. This is a base functionality for starting up a system autonomously as presented in the correspondent use-case in section 1.4. This use-case is additionally of interest for system developers which trigger appropriate routines.

#### 4.1.2.2. State of a component

**Problem statement:** *A component must provide its **current main state**.*

In the former subsection the information about particular parts of a component is analysed. The information about the current state of the whole component is twofold. On the one hand the information that a component is fully initialized and is ready to run requires to deduce that all its services are ready to run. On the other hand the information that a component is not ready to run (any more) can be deduced easier. For that the StatePattern in SMARTSOFT can be used. The StatePattern provides the standard MainState *Neutral* that indicates that at least some of the controlled tasks are blocked and thus the services which rely on these tasks are not able to process their service-requests. This information is important in particular for component developers which use it for example for component testing as presented in the correspondent use-case in section 1.4.

Additionally to the standard MainStates (like *Neutral*) the StatePattern provides the *life-cycle* state of the component as described in subsection 3.2.3. This allows to monitor whether the component is basically ready to run (this is the case in the *Alive* MainState) or whether the component is in the process of initialization (resp. shutdown). This is the case in the *Init*, resp. *Shutdown* MainStates.

The analysis of the *locality* and *stakeholder* dimensions shows that although the information is generated inside of particular components it is mostly needed in specific Monitor components which are then used by component and system developers. One example where the state of components is used in the system is some kind of scenario control component which uses the same communication mechanism as a Monitor component to monitor and to control different components in the system according to the scenario. In this case the highest level of the *autonomy* dimension is needed (where the data is generated and analysed autonomously).

In most cases the current (newest) data is needed (*completeness* dimension). The data is generic for standard MainStates and is customized for component specific MainStates of the StatePattern (*generality* dimension). Finally the state information is available at the system level at a specific location (*distribution* dimension). So far the *temporal* dimension is mostly ignored which adds another complexity.

#### 4.1.2.3. History of events

**Problem statement:** *The **history of events** in a system must be **without gaps** and in correct **chronological order**.*

One of the important information-types for system developers (*stakeholder* dimension) is the history of events. The first question is what events are and what type of events are possible in a system. For that the *generality* dimension shows different levels. First, there are generic events possible which indicate that one of the generic data types (i.e. current component state) as described in the foregoing subsection has changed its value. With that the behavior of a component can be monitored over time. Second, some component specific values can be monitored as events (i.e. the connection state of particular client ports). Finally, some algorithm specific values - for example coming out of an algorithm in a component - can be also monitored as events. Examples for this kind of events are current calculation values, passing a certain checkpoint in an algorithm, catching up an exception in a component, etc. A core use-case for this problem cluster is described in 1.4.3.

The second problem is that the history of events must be without gaps. This means that the traceability of single events must be guaranteed and that none of the events are allowed to be lost. This issue is directly related to the *completeness* dimension, where at least the current snap-shot with a

history of events is needed. The projection in the future is not necessarily needed but can be generated online in specialized simulation components.

Finally the history of events must be in a chronologically correct order which is directly related to the *temporal* dimension. Although the order of events must be always correct - in most cases it is appropriate if this information is available (for example in a Monitor component) with a little delay. Little means in this case that for example for a system developer it is fully acceptable to see this information in a Monitor component with a delay of one or two seconds after the real occurrence of the event. This restricts the *locality* dimension to the value of monitor components. There are also cases where the execution of control in the system must obey a correct order of events. For example some actions are only allowed if some states in the real world are reached (which results in appropriate sensor values). In these cases the appropriate components must block till the particular event occurs. Finally there is a wide range of hard real-time applications, where the events must be always in time, which is however beyond the scope of this work.

The *distribution* dimension shows that the events can be fired from both a particular component or from an arbitrary component in the system (without direct information about its real origin).

Finally, the *autonomy* dimension shows the full range of values starting from customized messages which are analyzed by component developer to fully autonomous generated information, that is analyzed in a Monitor or a scenario control component.

#### 4.1.2.4. Influence on the system

**Problem statement:** *The **influence** from Monitoring on the running system must be kept low.*

Monitoring as understood in this work is a tool for component and system developers and for special components in the system (like scenario control component). Thus, there are parts of Monitoring that are only needed during development of components or of a whole system. On the other hand, Monitoring is certainly never for free. It will always require additional communication bandwidth, computational time, memory and other resources. This means that Monitoring will always influence the system in more or less negative way. There are two possible approaches to relax this problem. The first approach is to optimize the Monitoring functionality such that it requires as less resources as possible and therefore has as less influence on the system as possible. As a consequence this means that the Monitoring concept must be designed with care. A similar approach is fairly to choose a hardware that has enough power (which is not always possible in robotics). One another approach is to design the Monitoring functionality such that it can be easily switched on and off. This approach can be refined further such that the Monitoring system can be parametrized and thus single functions of Monitoring can be switched on and off.

In fact it seems that both approaches must be used in combination to reduce the influence of Monitoring on the running system. This problem cluster affects the use-cases 1.4.3 and 1.4.4.

The *locality* dimension leads to further problems. The influence of Monitoring on the system additionally depends on where the resources are needed. For example components that run on the target platform - which is often scantily scaled due to autonomy requirements - are not allowed to require much more resources (due to Monitoring) than are needed for the services of this component. On the other hand Monitoring components can often run on development systems (such as desktop computers) which allow to use resources more wastefully. This issue must be considered in the design of Monitoring functionality. Further, it is to be considered that the two requirements - to shift as much calculation effort into Monitor components and to reduce communication overhead between the system and the Monitor component - are often conflicting. If for example Monitoring is designed

to take over as much calculation work as possible this means that less pre-calculation and filtering of data is done inside of components which in turn means that more data have to be transmitted to the Monitor component and vice versa. Hence, a trade-off between these two requirements must be chosen.

This problem cluster is mainly independent of the *stakeholder*, *autonomy*, *completeness* and *generality* dimensions. The *temporal* dimension shows that the more recent the data must be in a Monitor component the more communication overhead can be expected (resp. the less room for optimization of the communication is left). The *distribution* dimension shows that the more the data is distributed the more communication overhead can be expected.

#### 4.1.2.5. Eavesdrop communication

**Problem statement:** *The communication between components must be **eavesdropped**.*

Till now the problem clusters focused mainly on the internal aspects of components. A second area of interest for Monitoring is the communication between components as also shown in section 1.3. The main idea is that the communication between components must be eavesdropped and a copy of the data must be redirected to a Monitor component. However, one of the main challenges as presented in the use-case 1.4.4 is that it is difficult to eavesdrop the data on the communication level (resp. on the protocol level). Again, the reason is that the communication level (resp. the communication protocol) does not provide enough meta-information that would be necessary to reconstruct the communication objects that are transferred between components. Therefore, the main challenge is to identify the right level (resp. location in the system) where all the necessary data and information is accessible.

If this challenge is solved and the data is available in a Monitor component another problem arises. Although component developers are interested in this data, they are often even more interested on specific aspects that can be derived from this data. One of such aspects is the *plausibility* of data (resp. the values in the data sets). For example a developer sometimes wants to know if some particular values stay inside of a predefined range. This can be solved easily by defining a plausibility check function that takes an upper and a lower bound and checks the value in each data-set whether it is in between the boundaries. The design question here is where this plausibility checks are best implemented - inside of components or in a Monitor component (as defined in the *location* dimension).

This information affects both *stakeholders* namely component- and system developers. A component developer wants to know whether the data that is generated in the component is correctly passed to the communication level (resp. communication ports in SMARTSOFT). A system developer regards components in a system as black-boxes and is interested in the communication between them. For example he is interested in whether input data that is received by a particular component leads to expected output values that are available on the output ports of this component.

A further issue in this cluster is the communication overhead (as also described in the foregoing problem cluster). As the experience shows the network in a robotic system is a delicate resource. Additional communication overhead in the network impacts directly on the timing issues in the system. For this reason it is often not sufficient to transfer the whole intercepted data on the same medium. One of the possible solutions for this is to run the analysis checks locally in the component where the data is generated and to transfer only aggregated (resp. compressed) results.

The resulting fife dimensions are not directly relevant for this problem cluster. Indirectly there are some relations. For example the more bandwidth is required by the regular communication, the less bandwidth is left for Monitoring data and the more efficient the communication must be (as described above). Further, the more distributed the data is, the more difficult it is to eavesdrop the

communication. The reason for this is that the data must be synchronized, which is a difficult problem in distributed frameworks.

#### 4.1.2.6. Simple Statistics

**Problem statement:** *Simple **statistical values** must be derived from the communication between components.*

A component typically provides generic and customized data that is directly accessible within this component at run-time. Some examples for this are presented in the foregoing subsections. Making this kind of information available in a centralized Monitor component is the very first step towards automatic component- and system diagnosis. A second step is to interpret this data and to aggregate it to simple statistics. There are a lot of different statistics possible. For example some components in typical robotic applications provide a service which offers data from a sensor in this component. The sensor generates periodically new sensor values. In other words the corresponding service provides the sensor values with a certain frequency. This frequency is one of such *simple statistics*. Other examples are minimum, maximum and average values of particular variables in a component or in communication. These statistics are part of the QoS aspects as also mentioned in the use-case 1.4.4.

The *locality* dimension provides the following considerations. First it is important to identify the origin of data and its semantic information that is necessary to run appropriate statistics. For example it is important to denote whether data is available in a specific component in the system and whether this component provides further information to be able to run specific statistics.

The *temporal* and *completeness* dimensions are less important in this case. Although statistics often aggregate some data-sets over time it is not necessary to know the whole history of all data-sets at a specific time. Rather, it is necessary not to loose (or to oversee) single data-sets. Thus, the data can be considered to be independent of the time and a single snap-shot is sufficient.

The *autonomy* and *generality* dimensions show one further aspect. The more generic the data, the easier it is to develop particular statistics. For example, the liveliness status of a component can be easily (and automatically) analysed, because the set of possible values is perfectly known. One possible analysis for this is to analyse whether a component goes into a *FatalError* state and to react on this information. On the other hand, the less generic the data, the more intelligence is necessary in the analysis algorithm. For example, in a laser-scan or in a point-cloud it is not instantly clear how to develop generic statistics. In such cases additional semantic information and customized calculation algorithms are necessary to calculate statistics with a certain semantic.

Finally, the *distribution* dimension has effects on the synchronization requirements of data (as already mentioned in the previous subsections).

### 4.1.3. Conclusion

This section provided a broad view on the research problem clustering some problems that address similar questions into islands of problems. However, these problems are still quite generic. As shown in chapter 2 there is a wide range of approaches that can be found in the field of robotic and other domains. These approaches are analysed in the subsequent section using the problem-space from this section as a basis.

## 4.2. Discussion of potential approaches

This section lists some potential approaches which are derived either from the applications as presented in "*related work*" in chapter 2 or directly derived from the "*use-cases*" in section 1.4. Individual approaches address one or several problem clusters as presented in the previous section. For each of these approaches different possible solutions are presented and evaluated. The main goal is to decide on a sub-set of solutions that are mostly valuable and viable for the concept, which is introduced in the subsequent section. Thus, the approaches represent clusters (or islands) of solutions which belong together.

### 4.2.1. Logging

Logging is a very common approach in various domains to store different kinds of data from a system or from a software component (resp. module).

For example, as shown in section 2.1.2 and 2.1.3, logging can be used to store information - from a black-box (resp. from a diagnose module) - about the errors in the observed system. A developer can afterwards use this information to reconstruct the cause of an error. This approach shows some limitations. As the data is captured in advance the only general solution is to capture always as much data and information from a system as possible (or to run the same scenario multiple times with different logging parameters, which is in some cases not possible). The filtering is done later in appropriate analysis tools. The data is typically stored in the file system. High data load additionally leads to negative effects on the running system as described in 4.1.2.4.

#### 4.2.1.1. Examples

Some robotic middlewares provide further examples for Logging, namely RXBAG in ROS (as shown in 2.2.2.2) and DSS Log Analyser in Microsoft Robotic Studio (as shown in 2.2.4). As mentioned there both tools have their restrictions. RXBAG captures the data from topics ignoring all other communication ports which are based on services. RXBAG stamps each incoming data-set with a high precision time-stamp which however does not have much in common with the time in other components that receive the same data-sets, because of the latencies in the communication. In the following another solution is presented that addresses these problems.

The DSS Log Analyzer on the other hand presents a fairly general approach. All communication in the system can be captured in the DSS Log Analyzer. This is possible because of the standardized communication protocol (the SOAP), which includes all necessary information to reconstruct the data-sets on socket level. This flexibility comes however with a high price of lost efficiency on communication level, which is still an issue on autonomous platforms with limited resources. Additionally the time when data-set arrive in the receiver (or whether the data is received at all) is still not considered by this approach.

#### 4.2.1.2. Resulting solution for Logging

A possible solution for Logging - that addresses some problems from the examples above - is illustrated in Figure 4.1 and is described in the following. This solution can be implemented in most of the middlewares including SMARTSOFT.

1. Each component in the system must provide a clock, which has to be synchronized with a global system-time.

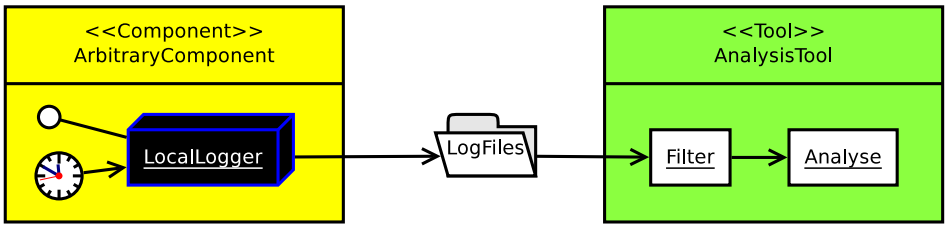


Figure 4.1.: Simplified approach for Logging

2. Each component further provides a local logger.
3. This logger should be configurable such that it can log data from different sources in the component.
4. The data is logged at run-time locally (for example on a file-system).
5. Each logging entry is labeled with a time-stamp that should be made instantly when the data occur in the component.
6. Afterwards data from all component-loggers is collected and analysed offline in appropriate tools (which are often specialized on a specific aspect).

Points ① and ⑤ are the consequence for the problem that each data-set must be back-traceable. This means that a data-set - which is generated in one particular component - is the same data-set in other components which receive it. An analysis tool can trace one particular data-set (which for example caused a problem) from somewhere in the system back to its origin. Points ② and ④ are consequences of high data volumes. Using a file-system reduces the influences from Logging on the running system. Additionally it is not necessary to transfer the data over a network because the analysis is performed afterwards and it is acceptable to collect the log-files prior to the analysis. Thus, a local-logger in each component can store the log-entries locally in a folder on the file system. Point ③ is necessary to further reduce the data load, because even with a local file system it is still not always feasible to log all available data from each component in the system. In many cases it is possible to configure a coarse grained pre-filtering in the local-logger. For example it can be configured to log only internal values of the component or additionally all data-sets which are received from other components. Finally, point ⑥ is the typical encapsulation between logging of data at run-time and analysing these data afterwards off-line with several (customized) analysis-tools.

#### 4.2.1.3. Pros and cons

Logging in this way provides some advantages. First, it is possible to store data from different sources like communication ports and other resources in a component. This kind of logging is independent of communication protocols and communication ports. The data synchronization can be guaranteed to the limits of the time synchronization algorithm. Different aspects can be analysed on the logging data:

- Analysis of single variables over time.
- Analysis of the execution order of single commands/values/states/etc.

- Plausibility of single component-states at discrete times.
- Timing analysis is only partly possible due to the uncertainties of the time.

However some issues are still open:

- Offline analysis after completing a run is sometimes not feasible because logging saves only the robots view on the world. It is sometimes additionally necessary to consider real states of the environment and the interactions with it. A direct interaction with a robot and its instant reaction in a monitor component can help to find the real cause of a problem faster.
- Logging must save as much as possible to get most information out of logging data. This requires high filtering effort in analysis tools. Again, it is still difficult to find the origin of a problem detected through logging.
- It is not possible to change a scenario sequence at runtime according to the Logging results (to provoke a certain error). The reason is the static loop: first to run the scenario and to store the data, and afterwards to analyze this logging data.
- A replay back into the system often leads to different system behavior, because the system depends on many uncontrollable aspects like current system load, thread scheduling, communication bandwidth, etc.

#### 4.2.1.4. Conclusion

Logging on the first view has many similarities with Monitoring. The goal is the same namely to get more insight into the system behavior. But the purpose is different. Logging in most cases means to collect data at run-time in advance and to analyse it afterwards offline. This is sometimes not feasible because errors in a system occur under certain circumstances which have to be reverse engineered out of logging data, which is an ambitious task. Monitoring on the other hand allows to detect errors at run-time. This can improve and speed up the development process. For these reasons it is considered that although logging is a valuable feature during development of components (resp. of the system) it can provide more value by using it as an add-on to the Monitoring concept.

### 4.2.2. Hardware-state Monitoring

Each robotic system consists of hardware and software parts. As shown at the beginning, software components can fail due to errors, failures and faults in the system. However, hardware parts are also prone to errors and must be considered as a further source of failures and faults in the system. For example, sensors and actors can fail either due to errors in their firmware or due to abrasion of their mechanical parts. In both cases it is often not possible to repair faulting hardware at run-time. The reason is, that in most cases hardware parts are closed systems which are only accessible through restricted interfaces (namely the hardware drivers). In a few cases it is possible to restart the faulty hardware device, which may unblock the hardware. If the cause for the error was temporary the hardware may run again otherwise the error remains. But in most cases the only way is to accept the error on the system level and to reduce (resp. to avoid) negative effects on the system with a certain strategy like to choose another sensor or actor (if redundancy is available).

#### 4.2.2.1. Examples

To be able to react on hardware errors one aspect is essential. It is necessary to recognize (resp. to detect) that an error in a hardware device occurred. First, this feature can be provided by the hardware itself. For example, appropriate test functions or return values can be provided at the interface to this hardware. Additionally a specialized software can monitor locally the correctness of the hardware system. This is a matter of local responsibility inside of this component. The information about a hardware error can be now made accessible on the system level. Specialized monitor components in the system can collect the data and react on the results. An example for this approach is presented in ROS, namely the *Diagnostics* stack as mentioned in 2.2.2.6. In this example a Toolchain is provided for hardware parts of the *PR2* platform (which is one of the main platforms for public ROS examples).

Another example for hardware Monitoring can be easily implemented using the *Event Pattern* of SMARTSOFT (see references in 2.2.1). For that a base component - which controls the robot base platform - can provide an Event Server that fires events if the battery voltage falls below a certain threshold which can be further parametrized at run-time on the Event Server. This approach makes it possible to implement a scenario control component such that it reacts on this event with an interruption of the current task and the navigation to a charging station. In this case the Monitoring of the base platform is directly included into the regular behavior of the system.

#### 4.2.2.2. Analysis

As shown in 4.1.2.1 one particular *vital part* of a component is the hardware (resp. hardware drivers) that this component uses internally. The information about the errors (or problems in general) of a hardware can be generalized in the component to the current state of a hardware device. One common semantic for the general states of a hardware device is based on the idea of a traffic lights (with its three colors). According to the traffic lights there are three states *green*, *yellow* and *red*. Green often signals that a hardware is ready to be used and no errors are detected. Yellow often means that the hardware is still running, but a minor problem occurred (which is often correctable). Finally, red means in most cases that a critical error occurred and the hardware is unable to continue its work. This kind of information can be used in different use-cases. For example at start-up of components in the system (as described in 1.4.1) the hardware states can be used as a precondition during the initialization procedure of components. Of course the traffic lights semantic is not limited to the states of the hardware, but can be used more general in a corresponding way for any kind of states in a component (for example states in StatePattern, Task states, user-defined states, etc.).

Additionally the information about the current state of a hardware in a component can contribute to the current state of the component (which uses this hardware) as described in 4.1.2.2. Further, in some scenarios it can happen that a hardware is erroneous only for a short period of time - depending on a temporary condition in a real environment that the hardware device can not handle. This can lead to an error in the system that is then difficult to be back-traced to its origin, because the responsible hardware device may be in a regular state again. For these cases it is additionally necessary to store the history of events as shown in 4.1.2.3. Finally, one particular type of devices are sensors (and other input devices). One characteristic property of these devices is that they generate data which can be further analysed. The analysis of communication data is described in detail in 4.2.3 below.

#### 4.2.2.3. Conclusion

From the examples and the analysis above three main aspects can be identified. First, it is important to detect (resp. to recognize) the current state of a hardware inside of a component and to deduce a

generic state value out of this information. Second, the states of all hardware parts in the whole system must be visualized in one Monitor component. Finally, the information about the hardware-state must be generic enough to be used as a further source of information in the system.

The *Diagnostics* stack in ROS (as shown above in 4.2.2.1) provides a valuable approach for the first and the second parts of hardware Monitoring (as described above). Unfortunately, the documentation of the stack and its implementation leave some open questions on the third part. For example it is not clear how to use the information about the current state of the hardware within the running system (similar to the battery voltage example as also shown above in 4.2.2.1). Additionally the documentation provides only examples for the PR2 platform. This makes it difficult to test the examples and to identify the core ideas behind the implementation. For these reasons, the *Diagnostics* stack in ROS is considered to be a work in progress that must be analyzed again in future work.

### 4.2.3. Data-flow Monitoring

As shown in 4.1.2.5 the communication between components in the system is one of the important aspects in a robotic system that must be monitored in different ways. One of the approaches in this area is to intercept the communication between service requestors and service providers. The base idea is quite simple, a tool similar to *wireshark* intercept the communication and show the data, that is transmitted on the line. However, there are some challenges that have to be solved, which makes this idea not that simple at all (as shown below). The base idea is illustrated in Figure 4.2. There are different levels of interception possible (like the protocol level, the middleware level, etc.) which are not shown in the figure, but are discussed in the following.

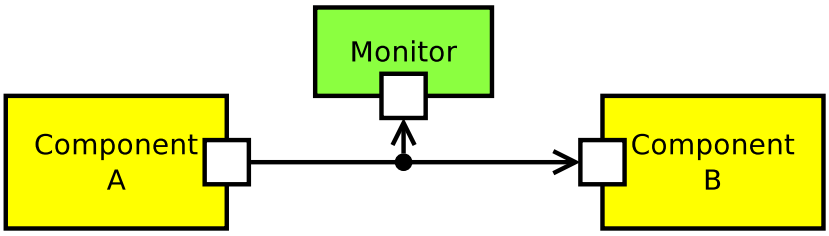


Figure 4.2.: Intercept communication

The communication between components can be sub-divided into two categories, namely the communication that is mandatory for the execution of components and communication that is additionally caused by Monitoring (resp. debugging) during development. As shown in 4.1.2.4 the latter communication category must be detachable. This influence further on the following aspect. Most of the robotic middlewares as described in section 2.2 provide at least two different *communication patterns*. First, there is always some kind of publish-subscriber communication. Second, a request-response, a query or a remote procedure call related communication is provided. The first communication pattern allows to intercept the communication on a high level, simply by attaching a further subscriber (i.e. the Monitor component) to the publisher. This is however in some cases not sufficient, because only the data from the publisher can be observed. It is not clear if a client really receives this data at all and for example the delay of the communication can not be measured. The second communication pattern provides a further problem. Service oriented communication is typically not connection oriented. This means that there is no permanent connection between components which could be intercepted. Some possible solutions are discussed further in detail in 4.2.3.2.

#### 4.2.3.1. Examples

There are two main reasons to intercept the communication as can be found in several robotic middlewares (see section 2.2). First, the data-sets that are communicated between the components can be visualised in a specific component. Such components typically know a predefined set of data structures (like images, laser-scans, position, etc.) which can be visualized in appropriate ways. Second, some characteristics from the communication can be observed. For example the plausibility of data can be checked for each data-set. There are several possibilities of doing so like to define thresholds for data values or to observe the continuity of data changes between several data-sets (for example if it is assumed that the robot moves, the communicated position of the robot should change from one data-set to the next one). The interception of data provides one additional method, namely to calculate simple statistics on data as described in 4.1.2.6.

The visualization of data-sets can be found in at least the following three robotic middlewares. ROS provides a set of tools for this purpose (see 2.2.2). The tool RXPLOT provides a limited possibility to visualize primitive data types (like integers) over time in a 2D plot. The tool RXBAG provides a feature to show logged data sets. There are also a very limited number of data structures possible. A more sophisticated approach is provided in the RViz tool of ROS (see 2.2.2.5). Here is a set of 15 different data structures available that can be visualized in a 3D window. Further, Microsoft Robotics Studio provides the *Dashboard* tool that can connect to different services (like a laser range finder, a webcam, a sonar device, a robot platform, etc.) and visualize the data in appropriate GUI frames. Finally, SMARTSOFT provides two components that can visualize different communication objects in a 2D GUI window. For that, the *smartVisualizationServer* component provides a general service to visualize simple graphical items (like circles, rectangles, points, etc.) in a 2D GUI frame. The *smartVisualizationMultiClient* on the other hand provides a converter for different communication objects, which transform these communication objects into sets of simple graphical items that can be drawn in the server. This implementation is quite similar to the XServer principle on Linux.

#### 4.2.3.2. Analysis

The main question for all of these approaches is how to intercept the communication between components. In principle there are two different approaches for this problem.

The first approach is to use the special characteristics (resp. properties) of the underlying communication mechanism. For example as shown above the publisher-subscriber allows to add (and to remove) a further subscriber with little influence on the other subscribers in the system. This approach has the advantage that with very little implementation overhead quite a powerful feature is implemented. The major disadvantage is however the limited generality of this solution. As most of the middlewares provide at least two different communication semantics (as shown above) it is often not sufficient to support only one of them for Monitoring issues as is the case in ROS (using topics without services). A further drawback of this approach is that it is tightly connected to the underlying communication mechanism and can be barely reused in other robotic middlewares. For these reasons this type of approaches is considered to be less valuable for this work.

For the first approach some exceptions can be found in the field of robotics, where this kind of interception can be implemented generic enough. One of these examples is based on service oriented communication in Microsoft Robotics Studio and the other is based on connection oriented communication patterns in SMARTSOFT. The former offer enough meta-information on the communication level (using SOAP) and uses a standardised communication mechanism, namely HTTP. The advantages brought by SOAP lead however to one considerable disadvantage, namely the extensive

communication overhead which is a critical criteria for many robotic applications. The latter example in SMARTSOFT presents sophisticated communication patterns that are easily portable to many kinds of communication mechanisms and offer an abstract communication semantic, which in principle allows to intercept the communication on the protocol level. This approach is further analysed in the subsequent subsection (see 4.2.3.3).

The second approach is to intercept the communication not on the *wire*[1] but directly inside of the component, where the data is either generated or received from the communication channel. Therefore, a branch must be redirected from the data - that is reached from the user-code in the component to the communication mechanism on the component hull - to a local logger which in turn saves the data-sets on the file system for example. A possible solution based on SMARTSOFT is discussed in 4.2.3.4.

#### 4.2.3.3. Intercept communication in SMARTSOFT on protocol level

As shown above one potential approach to intercept the communication in SMARTSOFT is to intercept it on the protocol level. Before the solution for this approach is discussed two preconditions must be considered. First the communication in SMARTSOFT is abstracted and is independent of the implementation basis. For this reason the interception solution must implement all types of protocols that correlate with different implementations of SMARTSOFT. Second in most of the implementations the protocol does not have a simple readable data (like it would be in services with HTTP protocol for example). Instead on communication channel only raw data plus some internal identifiers are transmitted. The communication is very efficient, because no overhead (that is not directly necessary for communication) is transmitted. This is mandatory in robotics, because of the high variability of requirements on communication aspects (like net-load, message frequency, etc.). The information about the structure of the data is hold inside of communication objects. Thus, if a message is intercepted on the communication level and some assumptions[2] about the underlying communication objects are taken one could reconstruct the message.

For ACE/SMARTSOFT the solution would be as follows. The communication protocol is described in more detail in the PhD Thesis [Schlegel2004]. In principle the interception of communication as described above would lead to the following steps:

1. The interception tool must know the address of the naming-service. This is usually not a problem because the same is true for all other components in the system.

2. The naming-service provides the server-references. This is composed out of a *server-name* and a *server-address-value*. A *server-address-value* is composed out of a *TCP socket address* and a *service-identifier*.

3. A server-name is composed of a *component-name*, a *pattern-type*, a *service-name* and one or several *communication-object-name(s)* (depending on the communication pattern).

4. The last part of the server-name gives the hints required for proper interpretation of the data stream representing the very communication objects.

5. Now, with significant effort, one could eavesdrop communication between the server and potential clients. However, because each client create its own connection, this leads to the following preconditions:

[1] Wire is used as an abstract term including wireless communication (i.e. with WLAN).
[2] This information can be taken for example from the naming-service.

a) The interception tool must run before any clients could connect to the server, otherwise the creation of the communication would be missed and there is no chance any more to find the proper socket port-number.

b) The connection procedure that is based on the ACE_Connector and ACE_Acceptor patterns must be intercepted and interpreted as well.

6. If one now knows the right socket port-number some further problems occur:

   a) Some communication patterns use up to three communication objects at the same time (like the Event pattern for example). Thus it is necessary to distinguish between communication objects on the socket level. Further it is necessary to identify which communication objects belong together. For example which activation resulted in the observed event message.

   b) Additionally, depending on the current internal state of the client and server ports different service handlers with different parameters are called. In principle, one must implement all possibilities which are also implemented inside of communication ports. In particular, ports interact either to communicate data as communication objects or to transfer control commands (completely without user data) for internal port coordination like the subscribtion in PushNewest pattern.

Taking all these challenges into account leads to a very artificial solution. Although this approach seems to be convenient at the first glance there are too many problems to be solved compared to the gained value. Overall, it seems further that the level of interception is still not appropriate. Additionally some aspects like the latency of the communication are not observable with this approach. For these reasons this approach is considered to be less feasible in this thesis.

#### 4.2.3.4. Intercept communication in SMARTSOFT directly inside of components

As introduced in 4.2.3.2 the second approach to intercept the communication between components is to access the data inside of components. The first advantage of this is that in most cases the information that is necessary to reconstruct the data-sets is available in the component. For example it is known which communication ports are used and which communication objects are configured in these ports. An overview of all interfaces that must be considered is shown in Figure 4.3.

As shown in the figure there are seven communication patterns, each configurable to use one to three communication objects. The interception - as illustrated with a dashed rectangle in the figure - allows to copy each communication object that is reached from the user-code inside of the component to the communication ports on the component hull or vice versa (depending on the direction of the communication). One concrete solution for this could be to implement a wrapper around the implementation of the communication port classes. This wrapper must copy the currently communicated communication object and pass this copy to a local logger. Because it is known which communication pattern is used it is possible to associate between different communication objects in one communication pattern. For example it is possible to store one particular request of a *query* together with its answer (by using the query id). The communication objects can be further stamped with a local time-stamp in the component, which makes it possible to bring all data-sets in the component in chronological order. If the component additionally synchronises its time with a global time in the system it is further possible to bring all communication objects in the whole system in chronological order (with a certain precision as described in 4.1.2.3).

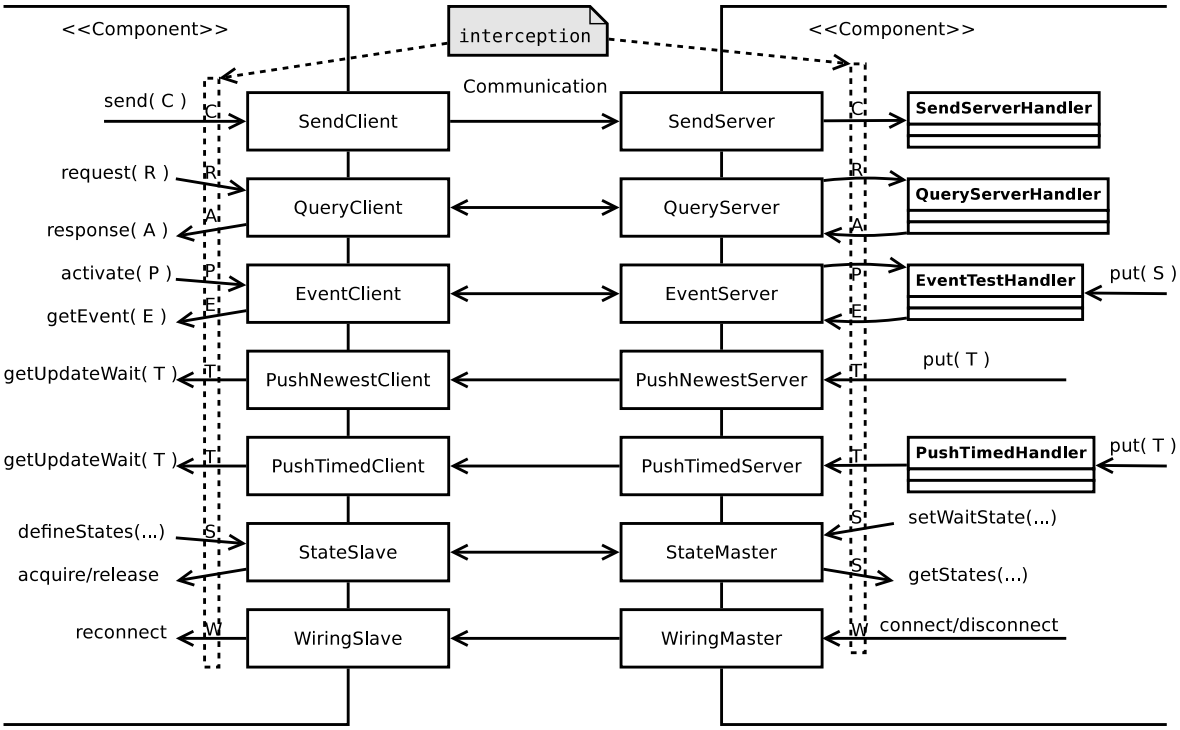


Figure 4.3.: Overview of the communication patterns in SMARTSOFT and their template parameters

By comparing this approach with the previous ones shows an additional advantage. If the communication are intercepted on both sides (the server- and the client side) some further characteristics can be monitored. For example it is possible to monitor the delay (resp. the time that it takes to transmit one particular communication object) of the communication.

#### 4.2.3.5. Conclusion

From all the presented approaches the last one leads to the best ratio between the advantages and the implementation effort. In principle, this conclusion can be taken in most of the middlewares like ROS, RT-Middleware and SMARTSOFT. As shown at the beginning of this section MSRS is an exception due to SOAP. However, even in MSRS it is not possible to observe such characteristics like the communication delay. Additionally, the communication overhead in MSRS is not acceptable in many cases. For these reasons the last approach (as presented in 4.2.3.4) is considered to combine the most of the advantages and to be the most generic one. This makes it valuable for the concept that is presented in section 4.3.

### 4.2.4. Heartbeat Monitoring

One aspect of Monitoring as introduced in 1.3 is the communication between the system composed out of components and the monitor components. As mentioned there it can be distinguished between query like communication (request-response) from monitor component to the components in the system and report like communication from components in the system to the monitor component. The latter approach is quite similar to the idea of a heartbeat as presented in subsection 2.1.1. With this

approach a Monitor component is able to observe the components in the system and to determine their current *liveliness* state. A liveliness state can be defined with different semantics as shown in the following.

#### 4.2.4.1. An Example

One possible design concept for heartbeat-monitoring is illustrated in Figure 4.4. In principle the heartbeat works as follows. An arbitrary component in a system provides a generic functionality which consists of the two parts: a Periodic-Task ② and a Liveliness-Checks ① classes. A liveliness check is a function that can validate the current state of vital parts of this component as described in 4.1.2.1. Validation can be done with different semantics as shown below. The periodic task ② executes such checks with a configured rate and pulses a signal on the specialized heartbeat port ③. The kind of the signal can depend on the result from the last check and can be defined with different semantics that are further analysed below. A Monitor component can collect the heartbeat from different components in the system and can for example react only if one of the heartbeat pulses delay (using a watch-dog timer ④) or a certain heartbeat value is received (using some kind of an observer ⑤).

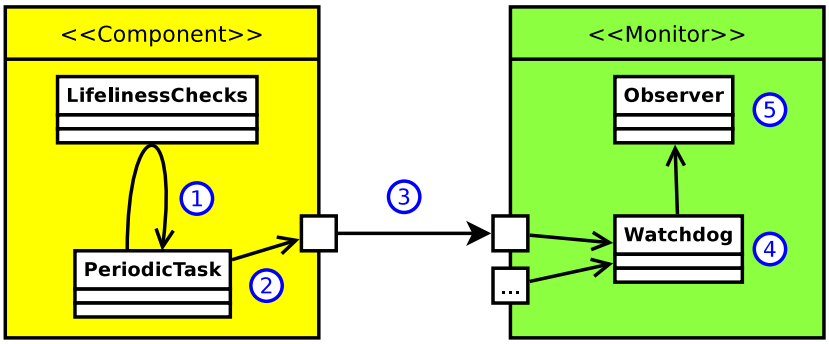


Figure 4.4.: Example for a heartbeat concept

#### 4.2.4.2. Analysis

**Granularity** As mentioned above, two parts in the figure - which are labeled with ① and ③ - must be further analysed. The granularity and semantic in one of them depend on the granularity and semantic in the other part. For example a liveliness check can be defined such that the result is a boolean ③. A "true" value could be returned if all liveliness checks are successful (i.e. all vital parts are ready and running). A "false" value could be returned if one of the checks fails. One possible semantic on the communication level ③ is that a signal is either transmitted or not (i.e. with an empty communication object) depending on the result of the check. An alternative way is to send the boolean value within the communication object. This allows to indicate in the Monitor component if a component is running at all and if all parts in this component are running too.

An alternative semantic for the result of the liveliness checks is that a component defines different state-values. For example a component can distinguish between essential parts that are really crucial for the execution of this component and parts that reduce the quality of a service in this component. Thus, if one of the less critical parts fail the component can continue its execution with reduced services. The semantic on the communication level (resp. in the Monitor component) can be now

customized with different strategies. According to the current situation a Monitor (or a scenario control component respectively) can decide to choose one alternative component in the system or to accept the reduction of the quality of the service.

Finally the most complex semantic is reached if further characteristics like component states as described in 4.1.2.2 and simple statistics as described in 4.1.2.6 are also taken into account inside of the liveliness checks. Although it offers the most possible flexibility and interpretation possibilities, it also leads to further challenges. For example, the communication overhead (resp. the influence on the system as described in 4.1.2.4) is a problem that has to be solved. A complex semantic on the communication level makes it difficult to analyse both the heartbeat and the values in the Monitor component.

**Heartbeat Port** A heartbeat as described above leads to a further problem. At the core a heartbeat must be a mechanism that is independent of the rest of the component. This is necessary because a heartbeat must not be affected by possible errors in a component. The consequence is that a heartbeat must be an additional communication port. This however leads to a design problem. On the one hand a Monitor (or a specialised) component can analyse the heartbeat from several components in the system. On the other hand this information is hard to integrate in the system such that other regular components can use the heartbeat as a mean for problem detection. The reason is that in this case a component that must observe a heartbeat from another component must locally run some kind of heartbeat monitoring mechanism. If this component additionally observes a heartbeat from several components the number of ports and the local monitoring effort increases. This means that such a heartbeat scales quite badly which decreases the value of the heartbeat concept.

#### 4.2.4.3. Conclusion

On the first view the idea and the simple concept of a heartbeat in each component is attractive. However, a simple semantic as shown above is not sufficient for many scenarios. A complex semantic on the other hand reduces the value from the simple heartbeat base idea. Thus the consequence is to design the concept for a heartbeat somewhere in between of the two extremes.

Further the extra communication port and the calculation effort on the receiver is not feasible in many cases where the resources of a component are limited.

For all these reasons the heartbeat concept does not justify the need to use it as the core concept with an extra communication port in each component. However parts of this concept can be integrated the Monitoring concept as described in 4.3.

### 4.2.5. Introspection

Introspection is a general view on the Monitoring of particular parts of components in the system. As introduced in 1.3 the question on the internal structures of a component is twofold. First, there are aspects that concentrate on static structures and properties of a component in the system. Such aspects are described in the following. Second there are aspects that concentrate on parts of a component that are dynamic and are more likely to change during the execution of this component. These aspects are discussed in detail in the subsequent subsection.

#### 4.2.5.1. Examples

Static structures and properties of a component are those which are stable during the whole life-cycle of this component. In particular this means that these structures and properties are initialized (resp. set-up) during start-up (resp. initialization) of the component and are stable till the component is stopped (resp. destroyed).

Typical examples for such structures are the communication ports (resp. the public interface) of a component, internal tasks and hardware devices (resp. their drivers) in the component. Certainly these structures could change during the execution of the component - most of the robotic middlewares provides such features - however, many scenarios show that these structures are in most cases stable. For example a component that inherits a particular sensor (like a laser range finder, a camera, the odometry, etc.) initialize the drivers for this sensor at the beginning and it simply does not make sense to delete and to create this driver again. In some cases it is necessary to restart the sensor because of a malfunction or to halt it to save resources. However, in most cases this is done by using a particular method of the interface of the driver. The same goes true for the tasks in the component. Although the task can block due to synchronisation mechanisms like condition variables, semaphores, etc. the task self remains available. Finally the communication ports of a component represent their public interface for other components. It is not reasonable to destroy one particular communication port at runtime of the component because other components in the system may rely on the availability of this interface.

One particular example for such structures can be found in the Toolchain of RT-Middleware as shown in 2.2.3.3. The Toolchain provides a *Properties* window which lists the properties of the currently selected component. However the Properties window not only shows the static properties but also the dynamic ones. This is a reasonable approach but has some particular aspects that must be designed carefully as shown in the following.

#### 4.2.5.2. Analysis

One reason to consider *introspection* and *state and status Monitoring* separately, is their different requirements on the communication. Introspection as shown above requires flexible and complete information about the structures of a component. The reason for this is that there must be enough meta information to be able to analyse this information in a general Monitor component. Additionally, if such structures change at all this can be assumed to be seldom (as shown above). Thus, the influence on the system (as presented in 4.1.2.4) can be assumed to be low. For these reasons, one feasible approach is to use XML for data representation. XML provides a standardized structure that is human readable and that can be parsed with general XML parsers (that are available in different programming languages and many libraries). Additionally a base structure for allowed tags can be defined using schema definition language (SDL).

To be able to systematically request for further information about a component the data from introspection must provide one additional information type. Particular parts (or properties) of the component must be marked as stable (resp. static) or dynamic in terms that the state of this part can change. For example in a XML structure some tags can be marked with a particular attribute that represent a dynamic property. A monitor component can screen for such tags and generate selective requests for the current state of these parts in a component. An example for a static property of a component is the initialization parameters for a particular device in it (i.e. COM1, LPT1, etc.) that is typically stable at runtime in this component. An example for a dynamic part is the current MainState of this component which is available in the StatePattern (as shown in 3.2.3).

The structures and properties as described above are mainly based on the middleware capabilities (like communication ports, tasks, etc.), which are assumed to be available in the system. There are also structures and properties that are related to the customized usage of libraries and algorithms in the component. One example for this is presented above, namely the drivers. Further examples are vital parts of the component as shown in 4.1.2.1. Again, XML presents a feasible approach for such structures. Due to the completeness of XML the information can be generally visualized in a Monitor component and for example a developer can choose the dynamic parts, which are currently of interest and can show their current state.

#### 4.2.5.3. Conclusion

On the first hand the introspection approach as presented above provides a functionality that is general enough to visualize some information about the internal structures in a component. Additionally, this functionality is the basis for the Monitoring of dynamic parts of a component (see next subsection). On the other hand a trade off must be chosen between the completeness of the information about the component and the generality of the information. A complete information which for example presents the possibility to screen the current value of each variable in a component would be not feasible because of the inefficient communication and tight coupling. However, a highly general information (like the component name) is in many use-cases like 1.4.2 not enough for monitoring issues.

On the whole, this approach is considered to be a core feature of the Monitoring concept as presented in 4.3 that must be however designed with care.

### 4.2.6. State and Status Monitoring

In the previous subsection static structures and properties of a component are presented. Contrary thereto, this subsection concentrates on the dynamic issues of these structures and properties and on how they can be monitored. Two different types of dynamic issues can be distinguished. First, the current state of the component self and of the parts in it can be monitored. Second, the current values of particular variables in a component can be monitored. Some examples for both types are presented in the following.

#### 4.2.6.1. Examples

**State Monitoring** Some examples can be found that show what a state of a component can be. In SMARTSOFT for example the life-cycle of the component (as described in 3.2.3) represents the high level state of this component. Similar to that RT-Tableware provides a state automaton for each component (see 2.2.3.2). In both cases the information about the current state is of high interest for monitoring. For example during development of components a component developer can observe the current state of a component that runs in the target system. As shown in the use-case 1.4.2 this is the core information to increase robustness of the system.

The state automaton in RT-Middleware provides the current state for the RT-Component and its ExecutionContext. This states can be accessed through generic interfaces, namely the *RTC Interface* and *RTCEx Interface* which are available in every RT-Component. However, this is also a limitation, because besides of these statically predefined states no other information (like customized states or other state values) can be transmitted.

SMARTSOFT provides further examples for the state of particular parts in a component. For example the StatePattern allows additionally to the Life-cycle States to define customized MainState. This

kind of states indicates the functionality of one or several services of this component. This information can be used similar to that as described above and can be also considered as a core information type for the use-case 1.4.2.

In the previous subsection 4.2.5 some static structures and properties of a component are presented. These parts often consists of dynamic parameters. For example, all communication ports in SMARTSOFT are connection oriented. In particular the client ports can be either connected or not. The connection state can be changed at runtime using the DynamicWiring pattern in SMARTSOFT. If this connection information is accessible on the system level, a monitor tool like RXPLOT (see 2.2.2.3) can be easily implemented.

Further, a Task in SMARTSOFT have different states (namely *Init*, *Alive*, *FatalError* and *Shutdown*) similar to the Life-cycle States in StatePattern. These states can be monitored in a central Monitor component. This allows to identify a potential problem in a component more quickly.

Finally most of the devices (resp. drivers) provide information on their interfaces about the current state of this hardware (for example the hardware is either running or not). Problems in a device (i.e. a sensor or an actor) can often lead to chained errors in several components. Thus, knowing the current state of this hardware can simplify the analysis of potential problems in the system.

**Monitoring the course of events** Additionally to the current state of a component and its parts another information type is common. As introduced in the use-case 1.4.3 it is needed to redirect some user-messages - that are printed out on the standard output - to a generic communication port. This allows to transfer such messages to a monitor component and to visualise these messages in a GUI for example. This approach faces some additional timing related problems which are discussed in the problem cluster 4.1.2.3.

For this problem ROS provides the RXCONSOLE tool as shown in 2.2.2.1. Although this tool seems to provide a feasible solution for message monitoring as described above, it has some limitations which have to be solved first. In the following a solution is presented that addresses these and other problems (as mentioned above).

#### 4.2.6.2. Analysis

Both categories the *state monitoring* and *monitoring the course of events* lead to similar base problems. There is a particular sequence that is typical for this problem which is illustrated in Figure 4.5 and is described in the enumeration below.

1. The source of information must be defined in the component. This can be either done automatically in case the information is generally available in the component (i.e. Life-cycle State of the component) or the component developer provides some messages with particular information types that are of interest for monitoring.

2. This data must be collected in a central place in the component. A generic interface for this storage helps to collect the data.

3. A generic and public communication interface (i.e. communication port in SMARTSOFT) at the component must provide the access to the data storage (previous point).

4. A Monitor component must use this public interface and collect the state and status information.

5. A Monitor component can now visualize the data in a GUI window according to the type of information (i.e. the current state or a list of messages).

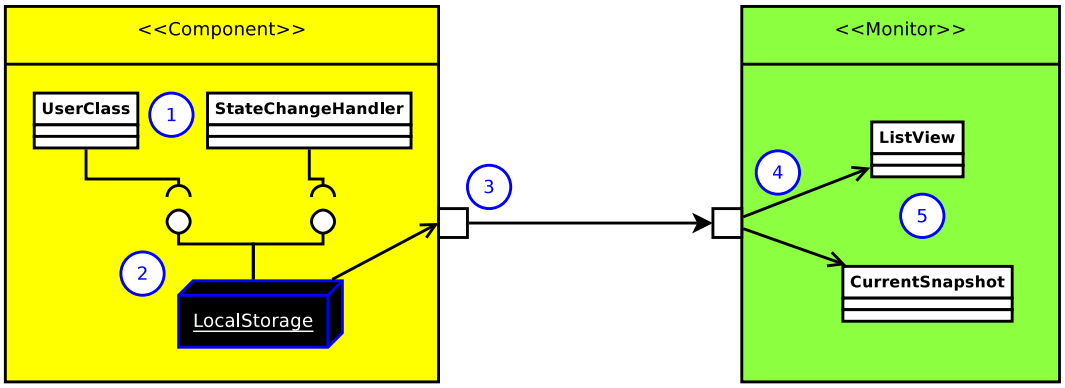


Figure 4.5.: Monitoring of the current state and the status messages

As shown in ① there are two types of information. First, generic information can be collected by using appropriate handlers. In most of the middleware such handlers must be defined at the beginning. In RT-Middleware and in SMARTSOFT such handler are directly provided by the middleware. RT-Middleware provides call-back functions in the RT-Component base class. SMARTSOFT provides a state-change-handler in the StatePattern.

The state-change handler in SMARTSOFT provides the ability to be informed about life-cycle state changes and about customized state changes. These two types of states are described in 3.2.3.

Customized messages are typically printed on standard output using a `printf` like interface. Such messages can be redirected to the local storage as mentioned in ②. For this, the Adaptive Communication Library (ACE) for example provides a `ACE_DEBUG` macro. This macro has a very similar interface to the *printf* method in standard C library. This macro uses the `ACE_Log_Msg` class. This class allows to easily redirect the message stream from standard output to any other customized stream (without changes on the interface).

The storage self ② can be seen as a black-box in the component which can be defined in the component during its development.

A generic interface and public interface at the component ③ provides the information that is in the local storage on the system level. This interface must be designed with care. As mentioned above in 4.2.6.1 the tool RXCONSOLE in ROS has some drawbacks related to this issue. The communication overhead highly depends on the parametrization capabilities of this public interface. This problem can be easily solved by using a more sophisticated communication mechanism. One predestined mechanism for this can be found in SMARTSOFT, namely the Event communication pattern. This pattern offer all features to be able to customize (resp. parametrize) the communication of messages according to the configuration (for example in the GUI). For this purpose the parameter communication object in the Event pattern can be used to activate only that type of events, that are currently of interest.

A Monitor component can use this interface to collect events from several components ④ according to the currently selected configuration in its GUI window ⑤.

According to the type of messages some additional issues must be considered in the Monitor component. For example if the chronological order of events is important (as described in problem cluster 4.1.2.3) the data-sets must be additionally marked with a time-stamp from a synchronized clock in ①. This allows to order the messages correctly in the GUI of the Monitor component ⑤.

#### 4.2.6.3. Conclusion

As shown above the monitoring of the current state and the course of events is one of the basic features that can be used in various use-cases like 1.4.2 and 1.4.3.

Although some promising approaches are presented in frameworks like ROS and RT-Middleware they leak on generality and flexibility. An approach that combines the features from these two frameworks and addresses their weak points, is presented above. This approach is seen to be one of the core functionality for the concept as presented in the following section.

## 4.3. Concept for Monitoring in Robotic Systems

In the previous section 4.2 several approaches are presented and potential solutions for them are discussed. Each of those approaches (resp. their solutions) lead to a set of requirements which are collected in the first part of the subsection 4.3.1 (in the following). As will be shown some of the requirements from different approaches overlap and lead to similar structures. Thus, the second part of this subsection collects similar requirements from these approaches and combines them to more general ones. The resulting requirements-list is the basis for the concept that is introduced in 4.3.2. That followed, the details of the concept are described in 4.3.3.

### 4.3.1. Resulting requirements

This subsection lists the requirements from the approaches as presented in 4.2. For that purpose, first the requirements from each of these approaches are collected and listed in 4.3.1.1 separately. After that similar requirements are combined and generalized in 4.3.1.2. The value and importance of each of them for the concept is rated as a priority value.

#### 4.3.1.1. Collection of the main requirements in different approaches

Requirements from the approaches as presented in 4.2 are collected in the following and are listed in separate Tables. An overview of the tables is given in the following:

- Requirements for **Logging** are shown in Table 4.3
- Requirements for **Hardware Monitoring** are shown in Table 4.4
- Requirements for **Intercepting communication** are shown in Table 4.5
- Requirements for **Heartbeat Monitoring** are shown in Table 4.6
- Requirements for **Introspection** are shown in Table 4.7
- Requirements for **State and Status Monitoring** are shown in Table 4.8

| ID | Requirement |
|---|---|
| 001 | Each component in a system needs a local clock that is synchronized with a global system-time. |

Table 4.3.: Requirements for Logging

| *Requirements for Logging (continued)* | |
|---|---|
| **ID** | **Requirement** |
| 002 | Each component in a system needs a local logger that stores (dumps) all incoming data on a file system. |
| 003 | The local logger must provide a generic interface for the usage inside of the component (i.e. XML based interface). |
| 004 | At regular shutdown of a component the local logger must ensure the integrity of the log-files. |
| 005 | Each incoming data-set in the logger must be enriched with a current time-stamp. |
| 006 | The log-files must contain enough information such, that an analysis tool can filter and diagnose the data. |

Table 4.3.: Requirements for Logging

| ID | Requirement |
|---|---|
| 010 | State changes of devices in a component must be detected and stored in a local (central) place in this component. |
| 011 | This states must be combined to a general value with a simple semantic (like the traffic lights semantic). |
| 012 | Additionally the state changes must be stored with a history without gaps. |
| 013 | The history must be in a chronological order. |
| 014 | A Monitor component on the other hand must be able to visualise the current states in a corresponding view. |
| 015 | A Monitor component must be able to visualise the history of state-changes in a corresponding view. |

Table 4.4.: Requirements for Hardware Monitoring

| ID | Requirement |
|---|---|
| 020 | Communication Objects must be intercepted between user-code and the communication ports inside of the component on the producer side. |
| 021 | Communication Objects must be additionally intercepted between user-code and the communication ports inside of the component on the receiver side. |
| 022 | All intercepted communication objects must be enhanced with a local time-stamp. |
| 023 | Communication objects must be translated into a general readable representation (i.e. into a standardized language like XML). |
| 024 | After interception the communication objects must be passed to a local logger inside of the component. |
| 025 | The local logger can save the data on a file system. |

Table 4.5.: Requirements for Intercepting communication

| *Requirements for Intercepting communication (continued)* | |
|---|---|
| **ID** | **Requirement** |
| 026 | The local logger can pass the data further to a generic port in the component. In this case a generic port is needed. |

Table 4.5.: Requirements for Intercepting communication

| ID | Requirement |
|---|---|
| 030 | Each component in the system must provide a continuous pulse that indicate on its liveliness. |
| 031 | The data transmitted with this pulse must contain information on the current status of the originating component. |
| 032 | The current status of the component can be determined by a local liveliness check in the component. |
| 033 | The liveliness check can use the states of vital parts in a component to determine the current status. |
| 034 | The pulse must be accessible on a public interface in the component. |
| 035 | A Monitor component must measure the delay between single signals and evaluate the data that is transmitted with this pulse. |
| 036 | Each pulse must be enriched with a local time-stamp before transmitting. |

Table 4.6.: Requirements for Heartbeat Monitoring

| ID | Requirement |
|---|---|
| 040 | It is necessary to describe customized structures (like used devices) in a component in a general way (i.e. with a standardized language like XML). |
| 041 | It is necessary to describe generic structures (like used communication ports) in a component in a general way (i.e. with a standardized language like XML). |
| 042 | These descriptions must be self explaining (resp. contain enough meta information) to be interpreted in a general Monitor component. |
| 043 | Those parts of the descriptions that represent component-parts which can change their state at runtime (i.e. connection state of communication ports) must be labeled with a corresponding attribute (i.e an XML attribute). |

Table 4.7.: Requirements for Introspection

| ID | Requirement |
|---|---|
| 050 | Each component must provide its life-cycle state on the system level. |

Table 4.8.: Requirements for State and Status Monitoring

| *Requirements for State and Status Monitoring (continued)* | |
|---|---|
| **ID** | **Requirement** |
| 051 | State changes of vital parts (like devices, communication ports, tasks, etc.) in a component must be detected and stored in a local (central) place in this component. |
| 052 | This state information must be provided on the system level through a generic interface. |
| 053 | Debugging messages (that are usually printed on standard output) must be redirected to a local (central) place in the component. |
| 054 | Each data-set can be enhanced with a current time stamp from a synchronized local clock. |
| 055 | A Monitor component collects this data and visualise it in a snap-shot view. |
| 056 | A Monitor component collects this data and visualise it with a history of events. |

Table 4.8.: Requirements for State and Status Monitoring

#### 4.3.1.2. Generalized Requirements

This subsection provides functional requirements (Table 4.9) and nonfunctional requirements (Table 4.10) for the concept that is introduced in 4.3.2. Functional requirements are derived from the approach-specific requirements in 4.3.1.1. For each requirement a priority is chosen (Low, Middle or High). The priority indicates the value and the importance of a particular requirement for the concept. The origin of each functional requirement in Table 4.9 is marked either with IDs from approach-specific requirements or with a reference to the location in this document where this requirement is described. This references and IDs are shown in the right column of the table. Additionally some new terms are introduced in the table. These terms are described in the following.

**Black-Box:** A black-box is a central place in the component where different information about the component can be stored and can be provided for other components in the system through different interfaces.

**Profile:** A profile (for example component profile) describes the structure of particular parts in the component in a general way (e.g. by using XML).

**Schema definition:** The structure of a particular profile can be defined by creating a schema definition (for example by using Schema Definition Language (SDL) in XML).

| ID | Priority | Requirement | Origin |
|---|---|---|---|
| 100 | high | Each component in the system must consist of a local **black-box** that stores different types of information. | 002, 010, 024, 051, 053 |
| 101 | middle | Each component in the system must provide a local clock. | 001, 022, 054 |
| 102 | high | The local clock in the component must be synchronized with a global time in the system. | 001, 054 |
| 103 | middle | The local black-box must provide the configuration option to store (dump) all internal data directly to the file-system. | 002, 006, 025 |
| 104 | middle | In case data from a local black-box is stored on the file-system, the integrity of the files must be ensured. | 004 |

Table 4.9.: General Functional Requirements for the Monitoring Concept

| *General Functional Requirements (continued)* | | | |
|---|---|---|---|
| **ID** | **Priority** | **Requirement** | **Origin** |
| 105 | high | Each data-set that is received in the black-box from the user-code in the component must be enriched with a time-stamp. | 005, 022, 036, 054 |
| 106 | high | The local black-box must provide a local interface to receive different *profiles* from the component. | 040, 041 |
| 107 | high | Schema definitions of generic profiles must be defined for generic (static) structures, for service definitions and for Tasks in a component. | 040, 041 |
| 108 | low | Examples for schema-definitions of profiles for customized structures in a component (i.e. hardware) must be defined. | 003, 040, 051 |
| 109 | high | Parts of the profiles that represent attributes in a component that can change its state at runtime must be marked with a certain attribute (for example XML attribute) | 010, 043, 050 |
| 110 | high | These parts in the profile (see 109) must be enriched with an ID that is unique inside of the black-box. | 010, 051 |
| 111 | high | The black-box must provide an interface for state updates of such dynamic parts (see 109) | 010, 051 |
| 112 | middle | Profiles must contain enough meta-information to be interpreted in a general Monitor component. | 042 |
| 113 | middle | A black-box must provide an interface (similar to printf or ACE_DEBUG) to receive customized messages. | 053 |
| 114 | high | Each component in the system needs one generic interface that is accessible from other components in the system. This interface must receive data from the black-box. | 052 |
| 115 | middle | This generic interface must be configured at runtime for each connected client separately such that the client receives only those data he is interested in. | 4.1.2.4 |
| 116 | high | A Monitor component must provide a properties windows that shows all profiles with current states for one particular (selected) component with a snap-shot semantic. | 014, 055 |
| 117 | middle | A monitor component must provide a list of components with simplified liveliness semantic (i.e. a traffic lights color for each component) | 010, 050 |
| 118 | high | A Monitor component must provide a window that shows a list of state-changes and customized messages in a chronological order for several (selected) components. | 015, 056 |
| 119 | low | A Monitor component must provide a window that shows all connections in the system in a graph representation. | 2.2.2.4 |
| 120 | low | The depth of the history of events must be configurable in the black-boxes (in each component) and in Monitor component(s). | 4.1.2.4 |
| 121 | low | A Monitor component can calculate simple statistics and provide them as a service for further diagnostics. | 4.1.2.6 |

Table 4.9.: General Functional Requirements for the Monitoring Concept

Additionally to the functional requirements, Table 4.10 presents non-functional requirements with a corresponding priority. Nonfunctional requirements describe any required constraints by which the Monitoring system must abide. This can contain external interfaces (or systems) as well as interoperability and performance issues or other requirements which do not fit into functional requirements.

| ID | Priority | Requirement |
|---|---|---|
| 200 | low | A guideline for a *Component Developer* is required to show what he has to implement in a component to use the Monitoring features. |
| 201 | low | The new components that are created for the Monitoring features must be interoperable in terms of independent of a specific platform (OS and/or Hardware). |
| 202 | middle | The communication overhead caused by Monitoring components must be kept low. |
| 203 | high | It is required to switch all Monitoring capabilities on and off without recompiling the components in the system (e.g. with an ini file). |
| 204 | high | A crash of any Monitoring component must **not** affect other components (and their services) in any kind. |

Table 4.10.: General Nonfunctional Requirements

### 4.3.2. The core idea of the concept

At the beginning of this thesis a bird's eye view of the Monitoring concept is given in section 1.5. The concept is presented there from a high-level view, leaving many details open. In contrast thereto, this section presents the concept with the knowledge in mind which is established in the foregoing sections. For this purpose the same high level view is chosen at first. Figure 4.6 presents an overview similar to that as presented at the beginning of this work. However, this time the developed terminology from the requirements above is used.

Figure 4.6 presents a high level view on the holistic concept for Monitoring in robotic systems. Again, the core idea is based on a local black-box in each component. This black-box collects information from the component (where the black-box is included) and to provide this information for other components in the system. Other components are either regular components that use the results from Monitoring in a particular component to improve robustness, fault-tolerance, etc., or specialized components (like the Monitor component) which can analyse the Monitoring data at runtime with different focus.

The communication between a Monitor component and any of the regular components in the system is based on a *Master/Slave* relationship. This relationship makes more sense than Client/Server for the following reasons. A Monitor component, which is the Master, is the active part for activation/deactivation, parametrization and the control of the communication. On the other hand, any component in the system is the passive part, that collects its local Monitoring data and make it available for each other component that requests for one of the particular information types. However, if for example the Monitor component requests to be informed about one particular type of information from one particular component, this particular component gets active and sends updates (resp.

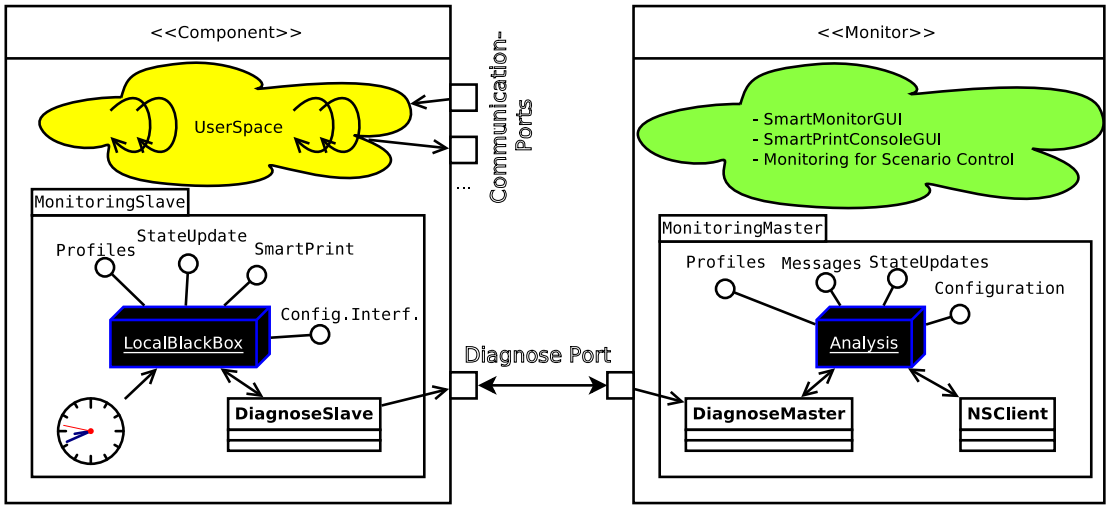


Figure 4.6.: Overview of the Monitoring concept

state changes) continuously and autonomously to the Monitor component. Because there is a particular separation of concerns in the communication, this relationship is difficult to describe with the Client/Server principle. Thus, the Master/Slave is chosen.

In the Monitor component an *Analysis* black-box uses Monitoring information from several components in the system for example to run different diagnoses or to visualize it in its GUI. More details on this and other parts are described in the following subsection 4.3.3.

### 4.3.3. Concept details

The main goal here is to link single requirements from 4.3.1.2 to corresponding parts in the concept. For this purpose corresponding parts of the concept are described with an abstraction level that allows to identify these requirements. The subsections in the following zoom in on these parts of the concept.

#### 4.3.3.1. Monitoring Slave

This subsection focuses on the MonitoringSlave which is used in each regular component in the system. This part is highlighted in the overview in figure 4.7. However, this figure does not provide enough details. Thus, figure 4.8 zoom in on the highlighted part of 4.7 and provide additional details. In the following the interfaces and the constituent parts are described.

As defined in requirement 100 each component in the system must consist of a local black-box. The black-box represents a central place in the component where this component can store different information for Monitoring purposes. On the other hand, the black-box is responsible to receive this information from the component, to convert it into general representation and to provide it on a public interface, the *DiagnosePort* (which is accessible from other components in the system). The first prominent differences (compared to figure 1.5) are the interfaces of the local black-box. The formerly presented *GenericInterface* is now called *Profiles* as defined in requirement 106. This is an important interface that allows to provide a generic description of the internal structures in the component. For that purpose, the following profiles (requirement 107) are defined:

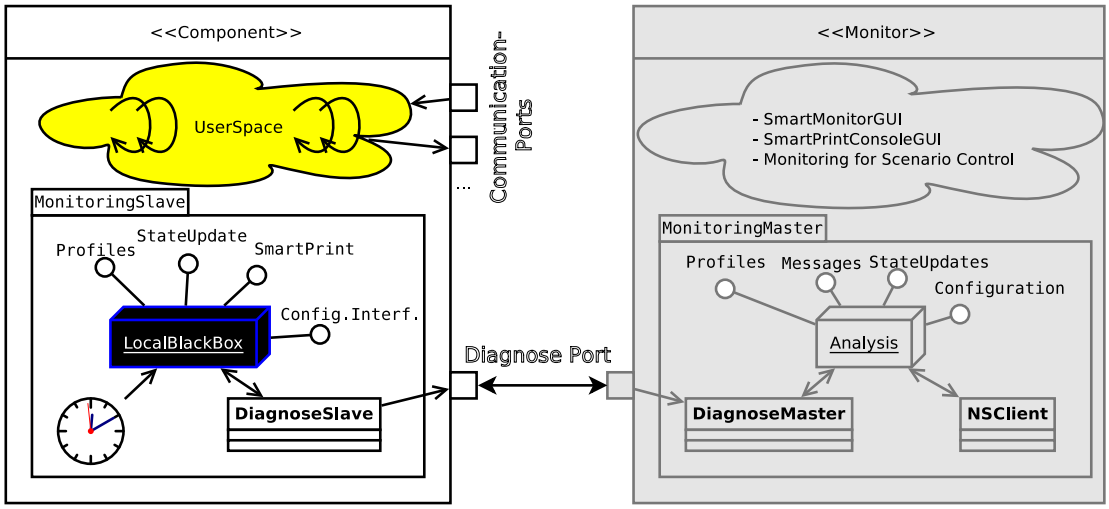


Figure 4.7.: Overview of the Monitoring concept - Monitoring Slave

**GenericProfile:** describes generic parts of the component which can be automatically generated without further information from user-code. Such parts are for example the component name and the version of the Middleware that is used.

**ComponentProfile:** describes common information about the component. There are two categories. The first represents the current liveliness MainState of the component which is in fact the first item that represents a dynamic part which changes its state at runtime (as defined in requirement 109). The second category represents tags for documentation purposes (like *short component description*, *vendor name*, *component version*, etc.)

**PortsProfile:** is one of the most interesting profiles, because it allows to define a description of all communication ports which are initialized with this component. Therefore this profile consists of a list of single port definitions with dynamic parts (like the definition of the connection state of a particular client port).

**TasksProfile:** describes the tasks that a component initialises. One part of this profile is the current state of the tasks.

**UserProfile:** defines a description for customized structures (as defined in the requirement 108). This gives the freedom to define every kind of structure that might be of interest in a particular component (for example a description of a hardware device). In this concept such profiles can be visualized in the monitor component and the dynamic parts of them can be additionally visualized with a history as customized messages. A possibility for a more general analysis is currently work in progress. However, as the structure is very flexible, it is a matter of definition, not a matter of concept redesign.

One of the most reasonable approaches to define the structure for such profiles is to use the *Schema Definition Language* (SDL). A predefined structure allows to parse XML based profiles in the Monitor component by using any kind of XML parser library (which fits perfectly to the requirement 112). Additionally, it is possible to write the XML profiles using standard XML editors or even to generate

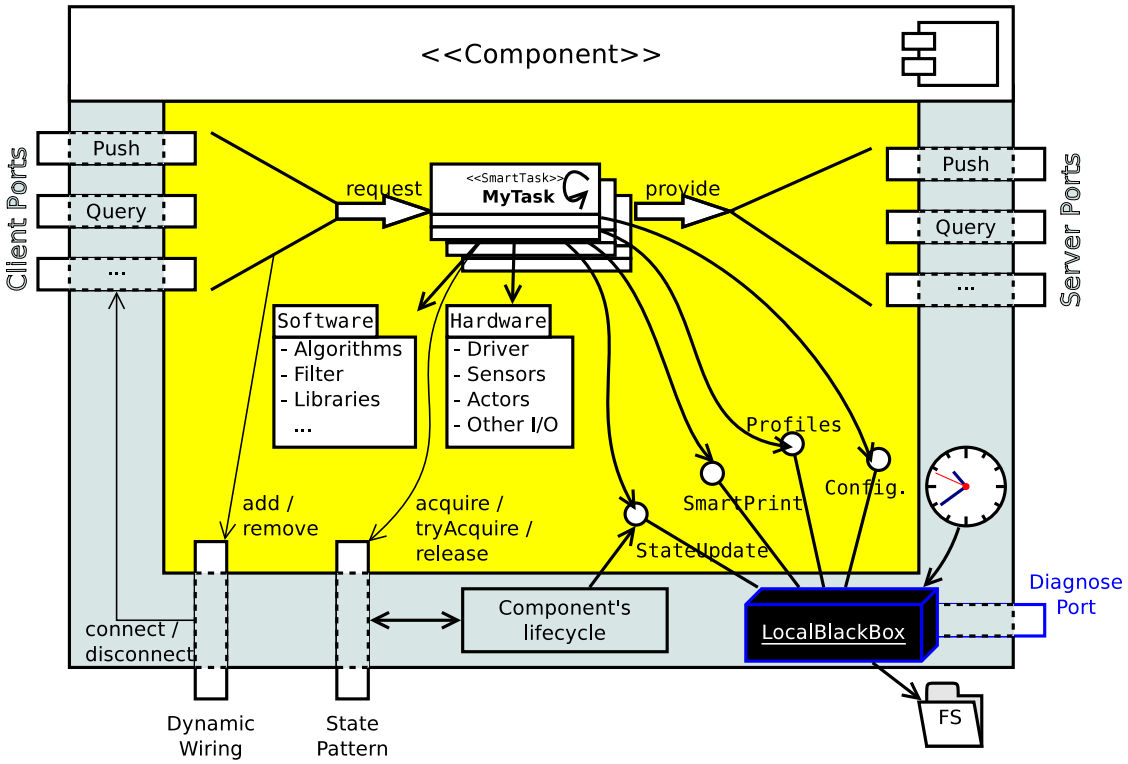


Figure 4.8.: Internals of a SMARTSOFT Component including Monitoring aspects

them in a MDSD Toolchain similar to that as presented in [Steck2010]. For these reasons the interface "Profiles" of the black-box must receive the profiles either as a URL to the profile file or as a pointer to the locally constructed XML object.

The next interface is formerly presented as the *UserInterface*. This interface is now subdivided into two more specific user-interfaces, namely *StateUpdate* (requirement 111) and *SmartPrint* (requirement 113). As defined in requirement 110 the dynamic parts in the profiles must be enriched by a unique ID. This ID can be used in the StateUpdate interface to update the state of a particular part in a profile without to update the whole profile. This is a general solution if using other languages than XML. However, if using XML it is also possible to use XPath to address one particular part of the profile and to update its value (resp. state). Some of the generic examples for such states are the current *Life-cycle state* of the component, the current state of particular Tasks, the current connection state of client ports, etc.

Further, *SmartPrint* provides a very common interface similar to the `printf` function from standard C library. This interface is very familiar for many C/C++ developers.

The last interface of the black-box is the *ConfigurationInterface*. As defined in requirements 103 and 120 the black-box must provide an interface to change the behavior of this black-box. This configuration is set-up at initialization of the black-box. For this purpose an *ini* file for example can be used.

Each data-set - that comes from any of these interfaces of a black-box - must be enriched with a current time-stamp (as defined in requirement 105). This time-stamp is requested from the local clock in the component (as defined in requirement 101). To be able to compare these data-sets on the system

level the local clock must be additionally synchronized with a global time in the system (as defined in requirement 102).

Finally, each component provides different public communication interfaces (i.e. ClientPorts, ServerPorts, DynamicWiring and StatePattern) to communicate with other components in the system (as shown in figure 4.8. The Monitoring concept provides two additional interfaces. The DiagnosePort (as defined in requirement 114) is a public interface of the component that is based on the *EventPattern* in SMARTSOFT. The EventPattern provides the necessary flexibility (as defined in requirement 115). The usage of the EventPattern does not restrict this concept to SMARTSOFT. However, the patterns implements much of the low level and error prone synchronization and parametrization mechanisms, which must be reimplemented in other frameworks. More details for implementation of the DiagnosePort and the internals of the black-box are given in the section 4.4.

Additionally to the DiagnosePort - which allows to provide data from Monitoring results on the network for other components - it is also possible to store this data locally on the file-system (illustrated as FS in figure 4.8). As defined in requirement 103 this option must be configurable on the ConfigurationInterface. To store the data on the file-system make sense in cases where the data is too clumsy to be transmitted on the network. If the data is stored as XML in the black-box this XML structure can be dumped directly to the file-system. In this case the integrity of the files must be guaranteed (requirement 104), at least for the cases where the component is able to shut down correctly (for example in error case).

#### 4.3.3.2. Monitoring Master

This subsection focuses on the *MonitoringMaster* which is used in specialized Monitor components in the system. This part is highlighted in the overview in figure 4.9. As shown in this figure, there are two parts to be described. The first one is the MonitoringMaster itself, which is illustrated as a package in the figure. The second part represents the visualization capabilities of the Monitor component. This part is illustrated as a green cloud in the figure.

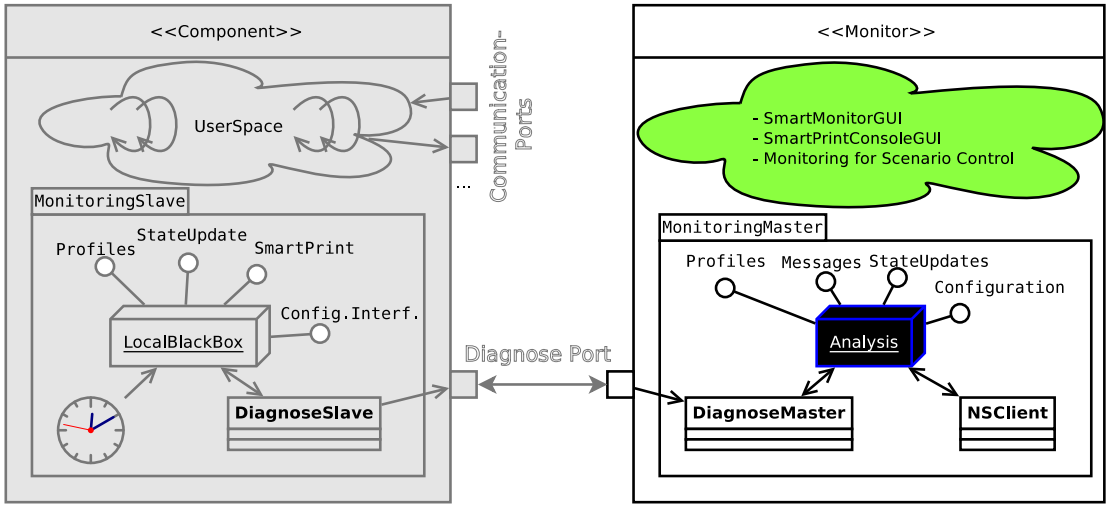


Figure 4.9.: Overview of the Monitoring concept - Monitoring Master

Again, the overview in figure 4.9 shows a representation similar to figure 1.5. However, the interface of the local *Analysis* black-box is adopted to the terminology as defined with the requirements in

4.3.1.2. The formerly presented *Introspection* is now called *Profiles*. Corresponding to the definition in the MonitoringMaster the interface Profiles provides information about structures of particular components in the system. This is a consequence of the requirement 116. Again, the data in the Profiles must contain enough meta-information to be analysable (as defined in requirement 112) in the *Analysis* black-box. This means in particular that each profile can be parsed in such a way that the contents can be visualized in a properties window (which will be presented later below). Additionally it is necessary to screen the profiles for dynamic parts (as defined in 109) and to find the information for one particular dynamic part. For the latter the ID (or the XPath URL) can be used (as defined in 110). This is a fundamental feature for the requirements 116-119.

The next interface is called *Messages*. This interface provides the messages that are created in the MonitoringSlave through the *SmartPrint* interface. These messages are mainly used in terms of *Retrospection* (as introduced in 1.5) to be visualized in a list view with a history of messages (as defined in requirement 118). A corresponding GUI representation will be introduced later (below).

The third data interface is called *StateUpdate*. Again, this interface provides information that is collected in the MonitoringSlave through its StateUpdate interface. In contrast to the interfaces *Profiles* and *Messages*, the *StateUpdate* interface can be used as a hybrid for different purposes. First it can be used to update the dynamic parts of particular profiles (resp. particular contents in the properties view) by using the IDs (as described above). This feature is defined in requirement 116. Second, the current state of particular components can be updated in the component list view as defined in requirement 117. Due to the fact that additional information to the current state-update can be locally requested from a corresponding profile, both of these information is combined and converted into a particular message. This message in tern can be visualized similar to a message from the interface *Messages* (as described above). This also a part of the requirement 118. Finally, a stat-update can represent a changing connection of a particular client port. This in tern, can be used in the connection graph windows (as will be shown below). This feature is defined in the requirement 119.

The last interface of the *Analysis* black-box is called *Configuration*. This interface is additionally necessary to parametrize the behavior of *Analysis* as defined in requirement 120.

The next part of the MonitoringMaster is the NamingService-Client (illustrated as the NSClient class in figure 4.9). This class provides the interface to communicate with the NamingService in the system. This is an important base feature of the Monitor component. All components in the system register their services as name-value pairs in the NamingService. One of these services is the DiagnosePort in each component (as introduced in 4.3.3.1). By getting a list of all entries in the NamingService, that contains the service name *DiagnosePort* the Monitor component can get a full list of all components that currently run in the system. This is essential for the requirements 115-119.

For each component in the system the Monitor component must create the Master side of the DiagnosePort. This communication port must be configurable such that only those data is communicated, which the Monitor component is currently interested in. Again, the EventPattern of SMARTSOFT provides this feature.

#### 4.3.3.3. Prototypes

This subsection provides some examples for the visualization capabilities in a Monitor component (this part is highlighted in the overview in figure 4.10 as a green cloud). The description of the base functionality and the data types that can be used for the visualization are described in 4.3.3.2. As the description there is quite abstract, this subsection here provides three tangible GUI prototypes which help to understand the ideas of the concept. These prototypes are described in the following.

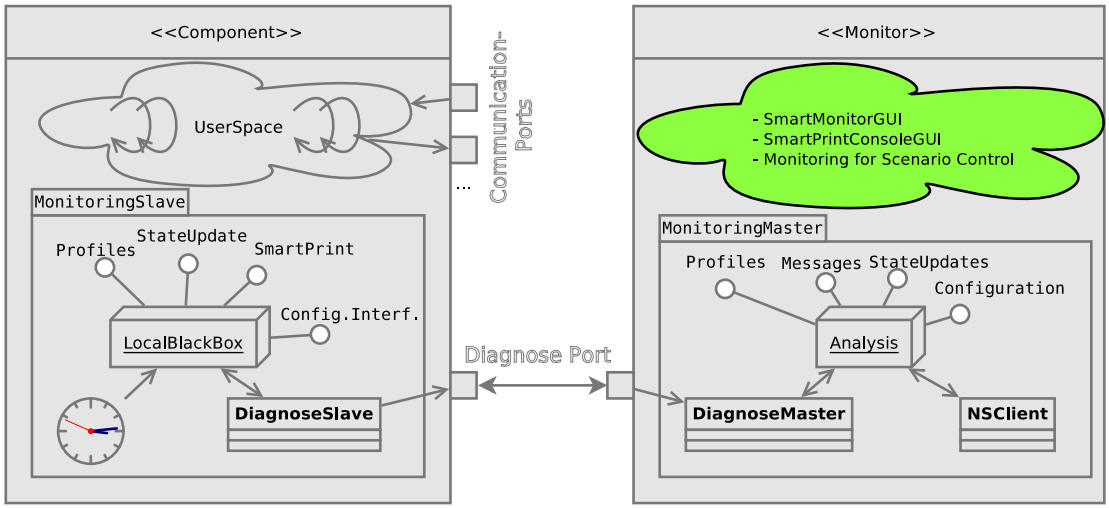


Figure 4.10.: Overview of the Monitoring concept - GUI Prototypes

First, it is important to notice that these prototypes do not represent the current state of the implementation. Implementation details are shown in section 4.4. In fact, the prototypes are mockups that are only used for explanation purposes. The final GUI representations deviates from these prototypes for particular reasons (which are described in 4.4).

**MonitorGUI prototype** A screen-shot of the first prototype, the *MonitorGUI*, is presented in figure 4.11. This prototype consists of the two parts, namely a list of components (that currently run in the system) and a *Properties* window that shows all profiles for the selected component in the list view. This prototype is the visualization that is defined in requirement 116 and 117. The refresh button in the prototype GUI can refresh the list of component, including newly started components and removing terminated components.

**PrintConsole prototype** A screen-shot of the second prototype, the *PrintConsole*, is presented in figure 4.12.

This prototype consists of three parts. On the upper left a list of components (that currently run in the system) is provided. In contrast to the component-list (as presented in the previous prototype), each component item in the list consists of a check-box. An arbitrary number of items (null to all) can be activated at the same time. This has the following advantage. For each component in the list - whose check-box is selected - the DiagnosePort is configured such that the *Messages* (see 4.3.3.2) are communicated and displayed in the *ConsoleOutput* list (on the upper right in the figure). A selection of all components produces a list of messages (sorted according to the time-stamp) from all the selected components. In addition thereto, the *ConsoleOutput* frame consists of a set of check-boxes, which allow to activate (resp. filter) particular types of information. An important difference of this functionality compared to the tool RXCONSOLE in ROS (as shown in 2.2.2.1) is that the configuration of these check-boxes directly result in appropriate parametrization of the DiagnosePort(s). This issue is defined in the nonfunctional requirement 202. Additionally, the prototype provides some helpful features. A further filter can be defined by using a regular expression which is applied to the content such, that only those messages are shown that pass the regular expression. Further, some but-

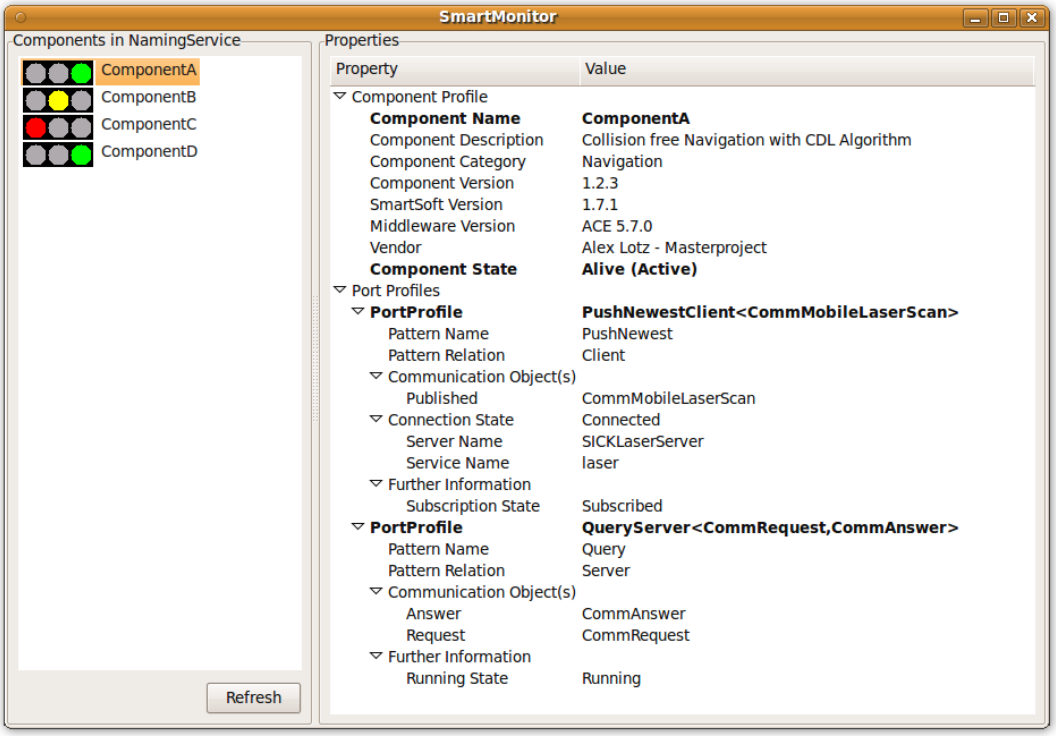


Figure 4.11.: Prototype for Introspection Data and Component State Monitoring

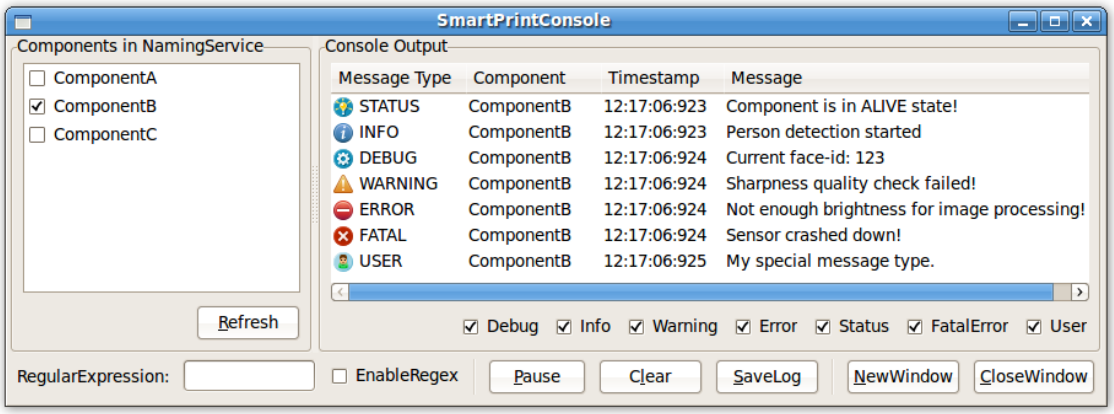


Figure 4.12.: Prototype for global PrintConsole

tons (on the bottom in the figure) control the behavior of the *ConsoleOutput*. The list of messages can be frozen with the *Pause* button or cleared with the *Clear* button. Additionally, the currently visible list can be dumped to a log-file with the *SaveLog* button. Finally, with the buttons *CloseWindow* and *NewWindow* a various number of main-frames can run at the same time. This feature gives the freedom to visualize some sub-sets of components in individual windows (which can be a helpful feature in big systems that contain a great number of different components).

**WiringGraph prototype** A screen-shot of the last prototype, the *WiringGraph*, is presented in figure 4.13.

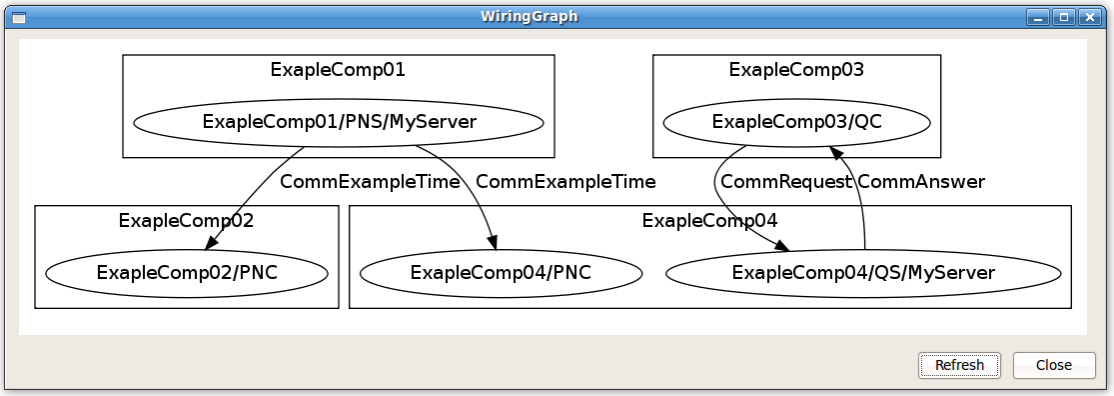


Figure 4.13.: Prototype for a WiringGraph

This prototype provides a specialized visualization type, namely to show all connections between components in the system in a graph representation. For this purpose, the *StateUpdate* interface from the MonitoringMaster (as described in 4.3.3.2) is used. Among other information bout state-changes this interface provides the information about connection changes between client and server ports of several components. This data can be enriched with further information coming from corresponding profiles (which gives the information about the name and type of the communication ports). With that information a graph can be generated. There are several different tools and libraries available (like *GraphViz*[3] for example) that can generate a graphical representation for the generated graph (similar to that as presented in the figure). Although this prototype provides the button *Refresh*, it is actually not necessary. As the connection changes are typically rare (compared to other state updates) the graph can be regenerated each time a connection update is received.

### 4.3.3.4. Optional add-ons for the core concept

The concept as presented above combines and includes most of the approaches which are described in section 4.2. However, two of those approaches are excluded till now (for particular reasons as described below). These approaches are considered as add-ons to the core concept. This means that the core concept is designed to be extended by additional functionality as described in the following. There are the following add-ons possible:

[3]GraphViz is online available at http://www.graphviz.org/

- As described in 4.2.3 the communication between component in the system can be intercepted. This is also perfectly possible to be integrated into the core concept. However, it is not clear whether this feature gains enough value to justify the additional complexity in the core concept. This issue is a work in progress.

- The second add-on is the heartbeat. Although it seems to be valuable on the first view, it turned out that the integration of this approach does not lead to additional value. There is also a use-case missing for this feature. For this reasons this approach is not included in the core concept.

### 4.3.4. Conclusion

The core concept as presented in this section addresses all the approaches with their requirements. Two of these approaches are considered to be optional and can be easily integrated into the core concept as add-ons. The core concept provides a trade-off between valuable features, efficiency and complexity of the concept. The core concept represents a recommendation for how to integrate Monitoring issues into robotic applications which consist of various components.

The core concepts leads some low level details open. These details are related to implementation issues and are described in the next section.

## 4.4. Implementation details

In the previous section the core concept for Monitoring in robotic applications is presented. This concept is presented on a certain abstraction level, which leads some implementation details open. These details are the main focus in the following. This section is structured as follows. First, the contents of the *LocalBlackBox* are shown as a white-box in 4.4.1. After that, the structures of the implementation and other details are presented in 4.4.2.

### 4.4.1. White box

The core idea of the concept as presented in the previous section is to include a *LocalBlackBox* into each component in the system. The interfaces and the semantics of this black-box are described within the concept. The structures inside of the black-box are still open. These structures are related to implementation details which are described in this subsection. The content of this black-box are illustrated in figure 4.14 as a white-box.

The white-box consists of the following two main parts. First, there is the implementation of the interfaces of the black-box. The second part describes the implementation of the *DiagnosePort*.

The central part of the white-box is the class `LocalLogger`. This class receives internally the data from all interfaces and hand over it to one of the output types. The *ConfigurationInterface* provides a generic interface of the white-box to parametrize (resp. to customize) it according to the local needs in the component during the initialization of the white-box (resp. black-box). This interface can be used either during initialization of the component in the user-code, or by changing the *ini* file, which contains the same parameter set. The latter allows to change the parameters of a black-box (of a particular component) without changes in the source code of the component (resp. without to recompile the component). This issue is defined in the nonfunctional requirement 203.

The *ConfigurationInterface* is illustrated as a package in the white-box (figure 4.14). There are the following configuration options possible:

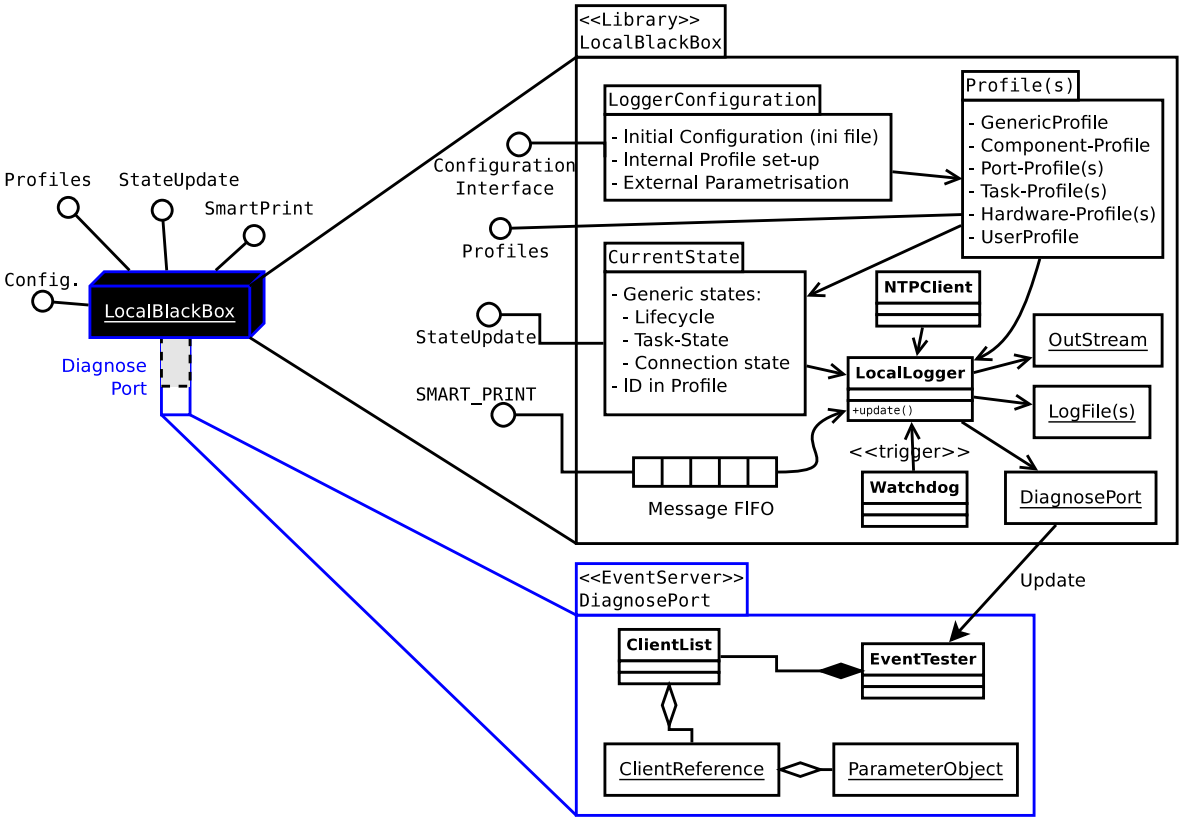


Figure 4.14.: White-Box

**Activation:** A boolean value that defines whether Monitoring is to be switched on or off at initialisation as defined in nonfunctional requirement 203.

**OutputType:** A list of output types can be defined. Currently three output types are possible (in combination):

- Print all data to standard output (resp. on the console)
- Store all data in log-files.
- Use the DiagnosePort to provide the data to all components in the system.

**MessageFIFO:** The maximum size of the message FIFO can be configured. This allows to buffer some messages till the Monitor component is able to connect and to activate its DiagnosePorts. This is only at start-up of the system necessary. If the Monitor component is initialized and ready, no messages can be lost any more.

**WatchDog:** The timeout time of the watchdog can be parametrized (see below).

The *ConfigurationInterface* consists of one further feature. During initialization of the white-box, the generic profile is generated. This can be done because this profile is completely independent of any structures in the component and independent of any user-defined information. This profile consists of the *ComponentName*, the *SmartSoftVersion* and the *MiddlewareVersion*.

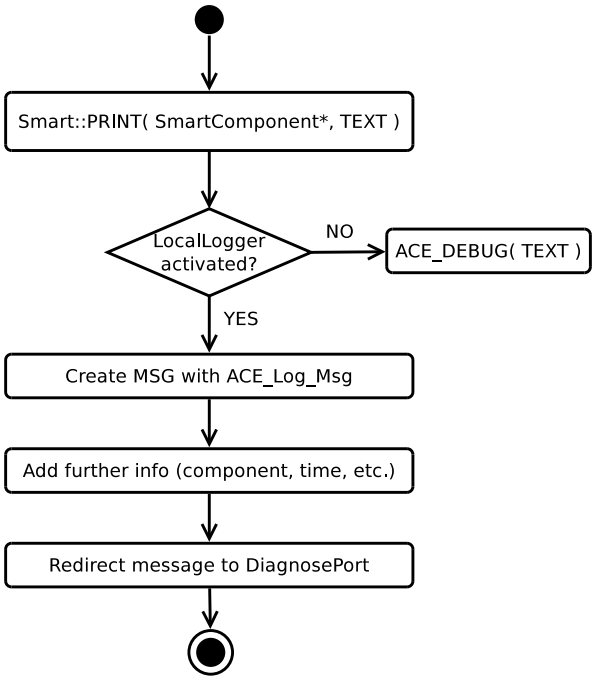


Figure 4.15.: The implementation of the SmartPrint interface in ACE/SMARTSOFT

The next interface is called *Profiles*. With this interface it is provide descriptions about the structures in the component to the black-box. For this purpose the structures are described in the form of profiles. These profiles are written in XML. The general content of particular profiles is described in 4.3.3.1. As mentioned there the structure of these profiles can be defined by using the Schema Definition Language (SDL). The SDL files for the *GenericProfile*, the *ComponentProfile*, the *PortsProfile* and the *TasksProfile* are given in the appendix in A.1. As can be seen there the Profiles consist of mandatory and optional parts. Additionally, those parts that represent dynamic structures (which can change its state at runtime) are enriched with the attribute *Dynamic*, which is fixed to the value *true*. This allows to screen a profile for dynamic parts. Thus, such SDL files allow to describe the structures of a component in XML. The address (URL) of the resulting XML files can be passed to the *Profiles* interface.

As mentioned above the *GenericProfile* is generated automatically in the *ConfigurationInterface*. The reason for this is, that this profile is the minimum information about the component. A Monitor component that connects to one or several components in the system can expect at least this profile.

The next interface is *StateUpdate*. This interface receives current states of particular parts (which are defined to be dynamic in one of the profiles). A plausibility function inside of this interface ensures, that the received state update is valid in terms of that it could be found in one of the profiles. If additionally a profile contains an ID for the current tag, the state-update is enriched with this ID before it is passed to the `LocalLogger`. Otherwise the XPath URL is used and attached to the state-update.

The Last interface is called *SmartPrint*. This interface receives all kind of messages from the user-code (of a component) and hand over a generalized representation of them to the `LocalLogger`. The `LocalLogger` on the other hand delegates this message to one (or several) configured outputs.

For this purpose the class ACE_Log_Msg of the ACE library can be used for example. This class provides the ACE_DEBUG macro with an interface similar to the printf method in the standard C library. The advantage of this class is, that it provides the possibility to redirect the messages from standard output to any customized stream. This issue is illustrated in figure 4.15. By deriving from the ACE_Log_Msg class it is possible to redirect the Messages for example to the *DiagnosePort* or to *LogFiles*.

The next part of the white-box is the NTPClient. As described in the core concept each incoming data-set in the LocalLogger must be enriched with a time-stamp. This is done inside of the LocalLogger each time it receives a new data from one of its interfaces. The current time is available in the NTPClient, which synchronizes the time with a NTP server.

Finally, the white-box consists of the Watchdog timer. It is used for the DiagnosePort as described below.

#### 4.4.1.1. DiagnosePort

As mentioned in the core concept the *DiagnosePort* is based on the EventPattern of SMARTSOFT. For this purpose the EventServer is implemented inside of the black-box. Thus, if a component includes the black-box it automatically includes the DiagnosePort. A Monitor component on the other hand implements a list of EventClient ports (one for each component in the system).

Figure 4.14 provides an internal view on of the DiagnosePort, which is initialized and used inside of the black-box (resp. white-box). As the DiagnosePort is based on the EventServer it consists of the following parts. The most important part is the class EventTester. This class receives all updates from the LocalLogger. The EventServer implements internally (actually not visible on the interface) a list of connected clients. Clients on this (low) level are connection(s) with Monitor component(s). For each client (resp. Monitor component) a parameter object is managed. This parameter object provides the information which data-sets a Monitor component wants to receive. This parameter object is adjusted during activation of particular events in the Monitor component[4]. According to this parameter object, the EventTester class decides - individually for each Monitor component - whether data is transferred to the Monitor component. This parametrization can be changed freely at runtime, according to the current needs inside of the Monitor component. This feature is important for the nonfunctional requirement 202.

Finally, the last element inside of the white-box is the *Watchdog* timer. This timer is necessary due to the combination of the LocalLogger and the EventServer. Each current message or state-update inside of the LocalLogger is instantly transferred to the EventServer (resp. the DiagnosePort) which in tern transfers this information further to all Monitor components, which are interested on this information. Thus, the update is triggered from the user-code inside of the component. If however the Monitor component activates an event which is not actively triggered from the user-code (this is for example the case for Profiles) another trigger is necessary. This trigger is realized in form of a Watchdog timer. This means, if a Monitor component activates the event for the profiles (and/or other events) the EventTester is either triggered by the next active update from the user-code or at the latest when the watchdog times-out. The timer (of the Watchdog) must be parametrized as a continuous (not one-shot) timer. A rate of 10 Hz is empirically evolved to be acceptable.

[4]For more information see Doxygen description of the EventPattern at http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/index.php

### 4.4.2. Experimental implementation

The implementation of the core concept is based on SMARTSOFT as motivated in 2.2.1. In particular the ACE/SMARTSOFT is chosen because of the StatePattern (see 3.2.3). An overview of the structure of the implementation is presented in figure 4.16.

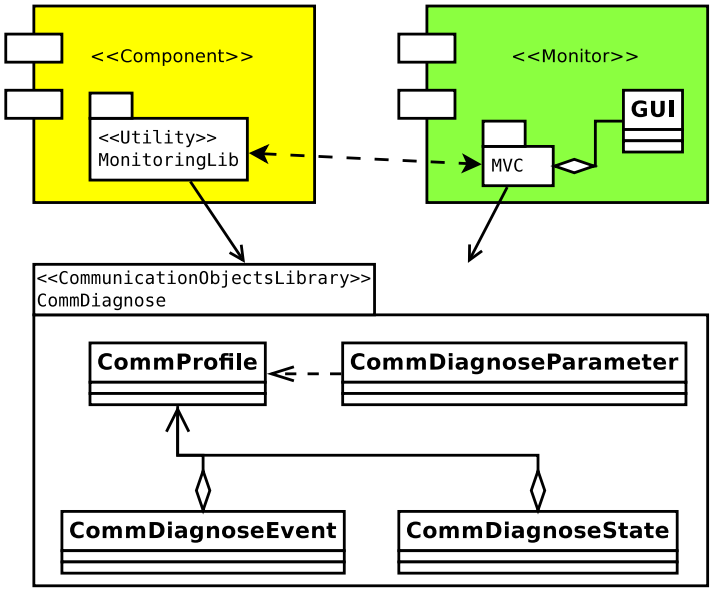


Figure 4.16.: Overview of the packages of the concept implementation

The implementation of the Monitoring concept is structured as follows. First, a set of communication objects is defined. They represent interoperable data structures that are communicated between the Monitor component and other components in the system. For that purpose, this set of communication objects is implemented as a stand-alone library which is included in both, the MonitoringMaster and the MonitoringSlave. The only dependency of this library is the *ADAPTIVE Communication Environment* (short ACE)[5]. Thus, this library can be used on any operating system (including Windows, Linux, etc.) that is supported by ACE.

The MonitoringSlave (which is used in regular components in the system) is also implemented as a stand-alone library. This library is stored in SMARTSOFT as an utility. This library is called `MonitoringLib` on the CD. This library has the following dependencies:

**ACE:** Because ACE is used by the *CommunicationObjects* and ACE/SMARTSOFT it is automatically used by the MonitoringLib.

**ACE/SMARTSOFT:** The MonitoringLib is completely based on the SMARTSOFT interface (resp. ACE/SMARTSOFT).

**CommDiagnose:** Is the library for the *CommunicationObjects* as described above.

**Xerces-C:** The only external library is the Xerces-C library which provides the XML parser for Profiles.

[5]Online available at http://www.cs.wustl.edu/~schmidt/ACE.html

The MonitoringMastr is currently directly implemented inside of the Monitor component. This component is stored together with other components (like those for navigation) in the folder *Smart-Components* on the CD. This component has the same dependencies as MonitoringSlave. Additionally the *WxWidgets* GUI C++ library is used for the GUI implementation.

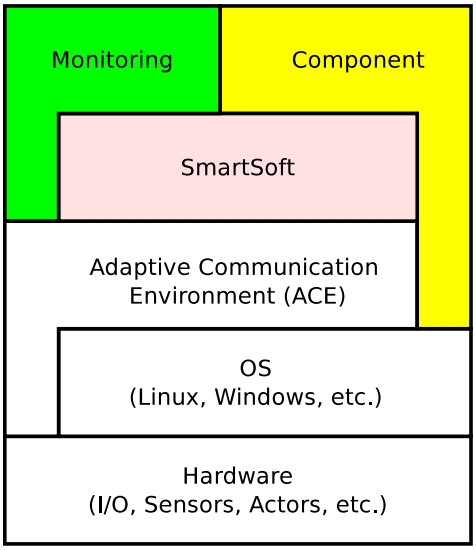


Figure 4.17.: Layered Architecture with ACE/SMARTSOFT

As shown in figure 4.17 the complete implementation of all Monitoring parts is independent of any operating system or communication middleware. This issue is defined in the requirement 201. As shown in the figure the Monitoring libraries only rely on the interface of SMARTSOFT and on the ACE library. The ACE library is needed for the *SmartPrint* implementation as described in 4.4.1. Each component in the system can include the MontoringLib and to use its interface.

### 4.4.3. Current state of the implementation

As already mentioned the implementation is not completed yet. Currently the implementation of the Profiles, the DiagnosePort, and parts of the GUI (resp. the *Model View Controller* pattern) are implemented. Currently it is possible to define a Profile XML by using one of the SDL files as presented in the appendix. This XML file can be passed to the MonitoringLib. Inside of the MonitoringLib this XML file is parsed by using Xerces-C library and is transferred into a string-stream which is then stored inside of the *DiagnoseState* communication object. This communication object is passed to the EventTester as described in 4.4.1. The EventTester checks the *DiagnoseParameter* object (which is passed at the activation of the Event from the Monitor component). If appropriate flags are set in the *DiagnoseParameter* the EventTester copies the content of the *DiagnoseState* object to the *DiagnoseEvent* communication object which is then transferred to the Monitor component. Unfortunately, the transformation from the *DiagnoseEvent* into the internal model representation of the MVC pattern is not implemented yet. The view part is however, already implemented. The watchdog timer is also implemented to trigger the EventTester. The internal implementation of the SmartPrint interface is implemented and all messages are currently printed on the console output. In the Monitor component the NamingService client is also implemented and delivers a list of all components in the system,

which consist of a DiagnosePort.

Not implemented are the following features:

- NTPClient inside of the MonitoringSlave
- StateUpdate interface and the transformation of state-updates into the *DiagnoseState* and the *DiagnoseEvent* objects
- Managing of state-updates and messages inside of the EventTester
- The redirection of messages from SmartPrint to the DiagnosePort is also not fully implemented
- In the Monitor component the the control er of the MVC pattern and the Model are only partly implemented.

Altogether, the current state of the implementations shows that the structures are reasonable and that the rest of the implementation is a matter of time. Therefore it is highly desirable to fulfill the implementation.

# 5. Results

The results in this chapter are structured in two parts. First a scenarios is presented in detail in 5.1. It is used to evaluate some particular features of Monitoring to demonstrate the strength and weaknesses of this approach. After that the concept is observed from a more general view in section 5.2 and is compared to other approaches as presented in related work.

## 5.1. Scenarios

One of the crucial tasks in robotics is the navigation. A mobile robot is mobile first if it is able to navigate. Additionally thereto a robot must navigate in a reliable and secure way. For example it is not acceptable to collide with obstacles, because these obstacles could be humans who are not pleased to be injured. Thus, there is a high demand on reliable and secure navigation components. This means that these components must be always aware of possible problems in the system to be able to react in a predictive and secure way. A very first step towards this requirement is the awareness of problems and errors in the system - where the Monitoring as presented in this work - comes in. On the one hand, in service robotics it is not acceptable to observe every single part of the whole system (resp. of each component), because this would mean to reimplement the complete logic of the system inside of the observer. Additionally, even an observer can fail which means in turn that the observer must be also observed and so on. On the other hand, a generic (resp. universal) observer can at least detect problems in the system. Such a system typically contains of scenario control component(s) on the sequencing layer. These components can use both specialized information from particular components in the system and generic information about the system from a Monitor. This leads to a clear separation of concerns. A Monitor component is able to detect particular problems and a scenario control can decide on a strategy to react on these problems.

A practical example for navigation can be found on the SMARTSOFT homepage as *Advanced Scenario*[1]. Single components of this scenario are described at *Components*[2]. Although, the descriptions are used for components available with CORBA/SMARTSOFT they are equally valid for the ACE/SMARTSOFT based implementation, which can be found on the CD (attached to this thesis).

The Monitoring capabilities as described above can be explained by using the *Advanced Scenario*. However, the *RobotConsole* component can be replaced by a simple scenario control component which automatically starts the scenario to drive reactive. The test case is that all components from this scenario are started all at once. The scenario control component must monitor the current life-cycle state of all navigation components and set the parameter *MoveRobot* of the CDL component at the time when all other components are in their *Alive* state. This scenario demonstrates the usecase *Start up a system*.

If the scenario is running, the Monitor component can be started in addition and the information from selected navigation components can be displayed in the GUI (i.e. the component-list view, the message view and the properties view). This demonstrates the ability to visualizes state information

[1] http://smart-robotics.sourceforge.net/corbaSmartSoft/using-advancedDemo.php
[2] http://smart-robotics.sourceforge.net/corbaSmartSoft/components.php

of one or several components. This is a test case for the usecases *Monitor the current state of a component* and *Monitor the course of events for the system states*.

As the monitoring of the communication between components is not included in the core concept it is currently not possible to monitor QoS parameters. However, if this feature is included as described in 4.3.3.4 the testcase is as follows. A monitor component calculates the minimum, maximum and average delay between providing a current laser-scan in the *LaserServer* component and receiving a velocity command in the PioneerBase component. This is an interesting value for designing the navigation parameters in the system.

## 5.2. Comparison of the current state of this work with related-work

Monitoring as presented in this thesis represents a generic solution to observe particular values of a running system. Compared to other approaches as presented in "related work" this is a more general solution, because it can be applied to different robotic frameworks in a similar way. Of course such a solution is always a trade off between functionality and generality. Thus, it would be valuable to gain more experience of working with this tool and to evaluate its real powers and weaknesses. The structures of Monitoring are not static and can be extended by further features. The reason is, that it can be expected that the improvements in the robotic middlewares will arise new requirements which must be addressed by Monitoring in the future.

# 6. Conclusion

This thesis addressed the problem of how to Monitoring particular aspects of a running application which is composed out of components in a service-robotic system. For this purpose, first some approaches are evaluated, which can be found in the field of robotics. After that the main concepts and ideas behind this examples are analysed. Finally these approaches and own ideas are combined to one concept. This concept includes all requirements, that are identified to be important in this work. The experimental implementation shows the value of some crucial parts of the concept. The general benefits are evaluated in the previous chapter. The result is that the concept present a generic solution for Monitoring that can be implemented in different robotic middlewares (resp. frameworks). SMARTSOFT is considered to provide the most capable basis for this concept. The current state of the implementation is that parts of the concept are implemented, but the implementation is to be finished in future work.

## 6.1. Future work

The very next step for this work is to fully implement the whole concept, which is presented in section 4.3. For this purpose each detail of the concept must be implemented and tested. A fully implemented solution could be then used in some robotic applications to gain additional experience which can be used to further improve the ideas in the Monitoring concept.

One of the possible enhancements is to advance the Monitor component by a service interface which provides the results from Monitoring on the system level. A component on the sequencing layer (see 3.2.1) could use this information to improve the quality of its scenario execution.

An interesting question is whether the Monitoring concept can be easily integrated into the MDSD Toolchain [Steck2010]. Answers to this question can lead to approaches which combine the benefits of both the MDSD approach and the Monitoring approach. Possible entry question is whether the Profiles (as described in the concept) can be easily generated out of the Model using *Open Architecture Ware* (OAW). A further question can be whether the results of a Monitor can be used in the deployment process of the MDSD Toolchain as an additional source of information. For example it is imaginable to use a live view in the Toolchain that shows current connections and states of the components.

Two particular robotic middlewares - which are further of interest - are OROCOS and GenoM. Some online forums show interesting ideas related to monitoring. However, they are currently under development and do not yet allow an evaluation. GenoM also shows some interesting aspects related to monitoring but the documentation and publications are not informative enough to allow an evaluation. Deeper investigation on code basis is required as a basis for evaluation.

## 6.2. Summary

This thesis showed that there *is* a general solution possible for Monitoring in a service-robotic application. This solution fulfill the use-cases that are described at the beginning. As showed in chapter "related work" there is a high demand to monitor component based robotic applications. Additionally

thereto the scope for Monitoring and the interfaces from Monitoring to other systems and vice versa are defined in chapter “fundamentals”. To evolve a general concept some sub-problems are analysed and solved in the first part of the chapter “method”. The second part provides a holistic solution for Monitoring in form of a concept. This solution combines the sub-solutions that were evolved in the problem analysis. The main part of this work concludes with an example implementation for one particular thread in the concept. This shows the realisability of the concept. After this, chapter “results” shows the capabilities of the Monitoring concept, by using some tangible real-world scenarios. Finally, the next steps that are reasonable and valuable from the author’s point of view are presented in “future work” (above).

Altogether, this thesis presented a concept that is reasonable according to evolved requirements. This concept must be still implemented in full detail, which is however a straight-forward task.

# A. Appendix

## A.1. XML Schema Definitions for different Profiles

This section provides the XML Schema Definitions for the Profiles as presented in 4.3.

### A.1.1. Generic Profile

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/GenericProfile"
xmlns:tns="http://www.example.org/GenericProfile"
elementFormDefault="qualified">

    <complexType name="GenericProfile">
     <sequence>
     <element name="ComponentName" type="string"></element>
     <element name="SmartSoftVersion" type="string"></element>
     <element name="MiddlewareVersion" type="string"></element>
     </sequence>
    </complexType>

    <element name="GenericProfile" type="tns:GenericProfile">
    </element>
</schema>
```

### A.1.2. Component Profile

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/ComponentProfile"
xmlns:tns="http://www.example.org/ComponentProfile"
elementFormDefault="qualified">

    <complexType name="ComponentProfile">
     <sequence>
     <element name="State" type="tns:ComponentState"
         maxOccurs="1" minOccurs="1">
     </element>
     <element name="Description" type="string"
     maxOccurs="1" minOccurs="0">
     </element>
```

```
     <element name="Category" type="string"
     maxOccurs="1" minOccurs="0">
     </element>
     <element name="Vendor" type="string" maxOccurs="1"
     minOccurs="0">
     </element>
     <element name="Version" type="string" maxOccurs="1"
     minOccurs="0">
     </element>
     <element name="FurtherParameter"
        type="tns:NameValuePair"
        maxOccurs="unbounded" minOccurs="0"></element>
     </sequence>
    </complexType>

    <element name="ComponentProfile" type="tns:ComponentProfile">
    </element>

    <complexType name="NameValuePair">
     <sequence>
     <element name="Name" type="string"></element>
     <element name="Value" type="string"></element>
     </sequence>
    </complexType>

    <complexType name="ComponentState">
     <sequence>
     <element name="InitialState" type="string"></element>
     </sequence>
     <attribute name="Dynamic" type="boolean"
        use="required" fixed="true"></attribute>
    </complexType>
</schema>
```

### A.1.3. Ports Profile

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.org/PortProfile"
    xmlns:tns="http://www.example.org/PortProfile"
    elementFormDefault="qualified">

    <complexType name="PortProfile">
     <sequence>
     <element name="ServiceName" type="string"
         maxOccurs="1" minOccurs="0"></element>
     <element name="PatternName" maxOccurs="1" minOccurs="1">
```

```
 <simpleType>
 <restriction base="string">
 <enumeration value="PushNewest"></enumeration>
 <enumeration value="PushTimed"></enumeration>
 <enumeration value="Query"></enumeration>
 <enumeration value="Send"></enumeration>
 <enumeration value="Event"></enumeration>
 </restriction>
 </simpleType>
 </element>
 <element name="CommunicationObject"
 type="tns:CommunicationObject" maxOccurs="unbounded"
 minOccurs="1">
 </element>
 <element name="ConnectionState" type="tns:ConnectionState"
 maxOccurs="1" minOccurs="0">
 </element>
 <element name="FurtherInformation"
 type="tns:FurtherInformation" maxOccurs="unbounded"
 minOccurs="0">
 </element>
 </sequence>
 <attribute name="IsClient" type="boolean" use="required">
 </attribute>
</complexType>

<element name="PortProfiles" type="tns:PortProfiles">
</element>

<complexType name="CommunicationObject">
 <sequence>
 <element name="Relation" type="string"
     maxOccurs="1" minOccurs="1"></element>
 <element name="Name" type="string"
     maxOccurs="1" minOccurs="1"></element>
 </sequence>
</complexType>


<complexType name="ConnectionState">
 <sequence>
 <element name="ServerName" type="string"
     maxOccurs="1" minOccurs="1"></element>
 <element name="ServiceName" type="string"
     maxOccurs="1" minOccurs="1"></element>
 </sequence>
 <attribute name="IsConnected" type="boolean">
```

```
     <attribute name="Dynamic" type="boolean"
        use="required" fixed="true"></attribute>
     </attribute>
    </complexType>

    <complexType name="FurtherInformation">
     <sequence>
     <element name="Name" type="string"
         maxOccurs="1" minOccurs="1"></element>
     <element name="Value" type="string"
         maxOccurs="1" minOccurs="1"></element>
     </sequence>
    </complexType>


    <complexType name="PortProfiles">
     <sequence>
     <element name="PortProfile"
         type="tns:PortProfile"></element>
     </sequence>
    </complexType>
</schema>
```

### A.1.4. Tasks Profile

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.org/TaskProfile"
    xmlns:tns="http://www.example.org/TaskProfile"
    elementFormDefault="qualified">

    <complexType name="TaskProfile">
     <sequence>
     <element name="TaskName" type="string">
     </element>
     </sequence>
    </complexType>

    <element name="TaskProfiles" type="tns:TaskProfile">
    </element>
</schema>
```

# List of Tables

# List of Figures

# Bibliography


[Ando2005] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon. RT-Component Object Model in RT-Middleware - Distributed Component Middleware for RT (Robot Technology). *Computational Intelligence in Robotics and Automation, 2005. CIRA 2005. Proceedings. 2005 IEEE International Symposium on*, pages 457 –462, jun. 2005.

[Avizienis2004] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11 – 33, January 2004.

[Bensalem2009] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and Nguyen Thanh-Hung. Designing autonomous robots. *Robotics Automation Magazine, IEEE*, 16(1):67–77, March 2009.

[Brooks2005] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback. Towards component-based robotics. *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 163 – 168, aug. 2005.

[Brugali2010] D. Brugali and A. Shakhimardanov. Component-Based Robotic Engineering (Part II). *IEEE Robotics & Automation Magazine*, 17(1):100–112, March 2010.

[Bruyninckx2001] H. Bruyninckx. Open robot control software: the OROCOS project. *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation*, 3:2523 – 2528 vol.3, 2001.

[Delgado2004] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *Software Engineering, IEEE Transactions on Software Engineering*, 30(12):859 – 872, dec. 2004.

[Elbaum2000] S. Elbaum and J.C. Munson. Investigating software failures with a software black box. *Aerospace Conference Proceedings, 2000 IEEE*, 4:547 –566 vol.4, 2000.

[Gat1998] Erann Gat. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, chapter On Three-Layer Architectures, pages 195–210. AAAI Press, 1st edition, March 1998.

[ISO/DIS-15031] ISO/DIS 15031-5 - Road vehicles – Communication between vehicle and external equipment for emissions-related diagnostics – Part 5: Emissions-related diagnostic services. International Organization for Standardization.

[ISO14229] ISO 14229-1:2006 - Road vehicles – Unified diagnostic services (UDS) – Part 1: Specification and requirements. International Organization for Standardization.

[Jackson2007] J. Jackson. Microsoft robotics studio: A technical introduction. *IEEE Robotics & Automation Magazine*, 14(December):82–87, 2007.

[Kortenkamp2008] D. Kortenkamp and R. Simmons. Robotic Systems Architectures and Programming. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, chapter 8, pages 187–206. Springer Berlin Heidelberg, 2008.

[Lotz2010] A. Lotz and C. Schlegel. ACE/SmartSoft on Sourceforge. online available, 2010. http://sourceforge.net/projects/smartsoft-ace/.

[Lussier2004] Benjamin Lussier, Raja Chatila, Felix Ingrand, Marc-olivier Killijian, and David Powell. On Fault Tolerance and Robustness in Autonomous Systems. *Computing Systems*, pages 1–7, 2004.

[Mallet2010] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. GenoM3: Building middleware-independent robotic components. *Robotics and Automation (ICRA), 2010 IEEE International Conference on Robotics and Automation*, pages 4627 –4632, May 2010.

[Quigley2009] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. *ICRA Workshop on Open Source Software*, 2009.

[Schlegel1998] C. Schlegel. Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot. *Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).*, 1(October):594–599, 1998.

[Schlegel1999] C. Schlegel and R. Worz. The software framework SMARTSOFT for implementing sensorimotor systems. *Intelligent Robots and Systems, 1999. IROS '99. Proceedings. 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 3:1610 – 1616 vol.3, 1999.

[Schlegel2004] C. Schlegel. Navigation and execution for mobile robots in dynamic environments: An integrated approach. *PhD thesis, University of Ulm*, 2004.

[Schlegel2006] C. Schlegel. Communication Patterns as Key Towards Component-Based Robotics. *International Journal of Advanced Robotic Systems*, 3(1), 2006.

[Schlegel2009] C. Schlegel, T. Hassler, A. Lotz, and A. Steck. Robotic software systems: From code-driven to model-driven designs. *Advanced Robotics, 2009. ICAR 2009. International Conference on Advanced Robotics*, pages 1 –8, June 2009.

[Schlegel2010] C. Schlegel, A. Steck, D. Brugali, and A. Knoll. Design Abstraction and Processes in Robotics: From Code-Diven to Model-Driven Engineering. In *2nd International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, Darmstadt, Germany, 2010. (accepted / to appear).

[Staples1997] G. Staples, M. Ross, and I. Court. Software black box mechanism: a pragmatic method for software crash diagnosis and usage maintenance testing. *Proceedings International Conference on Software Maintenance*, pages 142–149, 1997.

[Steck2010] A. Steck and C. Schlegel. Towards Quality of Service and Resource Aware Robotic Systems through Model-Driven Software Development. In *1st International Workshop on Domain-Specific Languages and models for ROBotic systems (IROS - DSLRob)*, Taipei, Taiwan, 2010.