University of Zurich^UZH

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a Module Framework for Sensor Data Management

Christian Ott
Zurich, Switzerland
Student ID: 12-723-896

Supervisor: Dr. Corinna Schmitt, Dipl. Ing. (FH) Jan Schirrmacher
Date of Submission: January 24, 2018

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

This thesis contributes to a development project at General Acoustics e.K. (GA), a company in the environmental data measurement field. First, the design and implementation of a module framework for sensor data management is shown. A second part presents the development of a LOG_aLevel 2.0 prototype, a product in development at GA. The module framework should result in a library that assists in the development of modular, configurable, and extendable applications. The framework should provide a middleware layer that simplifies lifetime management, dependency handling, persistence problems, and state control for the user. The goal of the prototype part was the implementation of LOG_aLevel functionality in a modular architecture by using the module framework. A focus was put on the design of a structured data storage solution. The implementation was done in C++ with SQLite as database technology. The prototype was developed for a low-power Linux device.

The framework was designed based on related work and covers the requirements of GA. It allows the creation of modular applications that can be composed at runtime over a configuration file. The resulted prototype application provides three core functionalities. It has a preprocessing interface to create human understandable measurement values from raw sensor data. This data is stored in a SQLite-based structured data storage solution. Advanced interval based algorithms can be integrated to postprocess the stored data. The evaluation of the structured data storage solution has shown that a SQLite-based implementation is suitable for the use in a LOG_aLevel system. The development project at GA will use the contributions of this thesis as groundwork. From there on, the framework and the prototype implementation will be developed further until the new LOG_aLevel version reaches market maturity.

Zusammenfassung

Diese Arbeit ist Teil eines Entwicklungsprojekts bei General Acoustics e.K. (GA), einer Firma welche sich mit der Messung von Umweltmessdaten auseinandersetzt. Design und Implementation eines Modulframeworks werden in einem ersten Teil beschrieben. Ein zweiter Abschnitt beschreibt die Entwicklung eines LOG_aLevel 2.0 Prototypen, ein zur Zeit in der Entwicklung steckendes Produkt von GA. Das entwickelte Framework soll bei der Entstehung modularer, konfigurierbarer und erweiterbarer Anwendungen dienen. Es soll eine Middleware Schicht zur Vereinfachung von Lebensdauer- und Statuskontrolle sowie Abhängigkeits- und Persistenzmanagement bereitstellen. Das Ziel des Prototypen war die Umsetzung von LOG_aLevel Funktionalität in einer modularen Anwendung unter Verwendung des Modulframeworks. Eine Kernaufgabe war die Integration einer strukturierten Sensordatenspeicherlösung in den Prototypen. Die Implementation sollte mit C++ und SQLite als Datenbanklösung umgesetzt werden. Zielplattform des zu entwickelnden Prototypen war ein stromsparendes Linux Gerät. Es wird der Kern der neuen LOG_aLevel Version sein.

Das Framework entstand unter Einbezug in Zusammenhang stehender Arbeiten und deckt die Anforderungen von GA ab. Es ermöglicht die Erstellung von Anwendungen, welche zur Laufzeit mit Hilfe einer Konfigurationsdatei zusammengesetzt werden. Die Prototyp Anwendung verfügt über eine Schnittstelle zur Datenvorverarbeitung, einer Möglichkeit zur strukturierten Datenhaltung, und erlaubt die Integration erweiterter Algorithmen zur Datenaufbereitung. Die Evaluation der strukturierten Datenhaltung zeigte, dass eine SQLite basierte Lösung für ein LOG_aLevel System brauchbar ist. Das Entwicklungsprojekt bei GA wird basierend auf den Resultaten dieser Arbeit fortgesetzt. Framework und Prototyp werden weiterentwickelt bis die neue LOG_aLevel Version Marktreife erreicht.

Acknowledgments

First, I would like to thank my supervisor Dr. Corinna Schmitt for her support, guidance, patience and her valuable inputs and comments during the last months. Second, I would like to thank Dipl.-Ing. (FH) Jan Schirrmacher and Dipl.-Ing. Jens Hensse for the opportunity to realize this thesis in cooperation with General Acoustics e.K. Special thanks to Jan for his continuous support on architectural and technical decisions.

I would also like to thank Prof. Dr. Burkhard Stiller for the possibility to complete this bachelor thesis at the Department of Informatics of the University of Zurich.

Furthermore, I would like to thank Jörg, Rahel, and all other numerous friends that have spent a lot of their time reading and correcting this thesis. Finally, I would like to thank my dear Anne for her continuous support during the last months.

Contents

Chapter 1

Introduction

With the rapid emergence and spread of the Internet of Things (IoT), traditional businesses face new challenges to stay competitive and adapt to the new technologies and market trends in a quickly changing technological environment. The customers' expectations rise, new products must use the newest available technology, be connected to everything and reachable from everywhere in a mobile first way. Massive amounts of data are generated, enabled through the interconnected 'Things', and has to be analyzed and presented to the customer in real-time. The pressure on the businesses forces them to reinvent themselves to stay in the market.

An area where the IoT has entered, is the remote measuring of environmental data. The remote sensing stations have gained connectivity and can now be pooled into remote sensing networks. The ability to interconnect environmental data from various locations allows advanced analysis of meteorological events that would not have been possible in the past. This capability comes with a price. The massive amount of data from various locations has to be collected, processed, stored, and in the end presented. Various research efforts exist that evaluate best ways to handle the vast amounts of measured data [30][78].

One company in the environmental data measuring field that has to adapt itself to comply with these new market trends is General Acoustics e.K. (GA) [25], located in Kiel, Germany. GA is a leading-edge technology producer of special echo-sounders, water level, and wave sensors as well as flow measuring systems. Initially, GA was mainly a hardware developing company, and therefore, focused on the data generation. Based on project requirements, the measuring systems could be enhanced by custom data processing and presentation solutions. With the rise of the IoT, more and more additional hardware functionality and data management services were expected by the customers. With the current hardware generation at its limits and no unified strategy to handle the data, the additional requirements became harder to satisfy. To counteract this situation, GA elaborated a plan on how to handle not only data acquisition, but also data processing, data storage, and data presentation in an equal way. A data handling workflow concept was created, on which the new product generation should be based on. The development process was started with the redesign of one of the core products from GA, the LOG_aLevel, an ultrasonic level measuring station that can be extended by a wide range of other meteorological sensors. During the prototyping of the LOG_aLevel software pieces, it became clear that a monolithic implementation approach interferes with the wish for advanced flexibility. Therefore, it was decided to implement a software framework. It allows the implementation of the LOG_aLevel software in a modular and flexible way.

This thesis, written at the University of Zurich, will be done in cooperation with GA. Two contributions to the current development process at GA will be the content of this thesis. First and foremost, the mentioned software module framework will be designed and implemented. Second, a LOG_aLevel 2.0 prototype application will be built as an additional contribution. It will use the module framework as a basis for the implementation.

1.1 Description of Work

The work of this thesis can be divided into two parts. The first part is the design and the implementation of a software module framework appropriate for the GA ecosystem. The framework shall allow the implementation of modular applications that are simple to develop, easily extendable, and can be configured without the need for recompilation. Related work and requirements from GA should be used to design the framework.

As second part of this thesis, a LOG_aLevel 2.0 software prototype should be implemented. The design should follow the decisions made and elaborated in the sensor data management workflow concept that was created by GA. The implementation should be done in C++ and with the help of the developed software module framework. The focus of this prototype should be the implementation of the structured data storage and advanced data processing functionality, as both are new to the LOG_aLevel ecosystem.

1.2 Thesis Outline

The rest of this thesis is structured as follows. Chapter 2 presents the background information on GA and their core products. Chapter 3 introduces the related work that influenced the realization of this thesis. Chapter 4 presents the design of the software module framework as well as the design decisions on the LOG_aLevel 2.0 prototype software. In Chapter 5 the detailed implementation process is shown. The thesis ends with Chapter 6, in which the evaluation of the implementation is discussed, before the thesis is concluded with Chapter 7.

Chapter 2

Background

This chapter provides background information on GA. It will be the foundation for the work done in this thesis. First, the company is presented, current products are introduced together with the ongoing development project that ultimately led to this thesis. Second, the two central product lines at GA, the LOG_aLevel and the UltraLab ULS, are presented in detail. Further on, the General Acoustics Property Protocol (GAPP), which is heavily used at GA, is introduced. Lastly, the sensor data management workflow concept and its design process is shown. The concept was created before the start of this thesis at GA.

2.1 General Acoustics e.K.

GA is a leading-edge technology producer of special echo-sounders, water level, and wave sensors as well as flow measuring systems. The company was founded in 1996 as an offshoot of a university research team specialized in acoustic and sensor technology, and is now located in Kiel, in the north of Germany. Throughout the years, GA has established a global network of distributors for its products in more than 60 countries. Numerous GA systems and applications are installed and operating throughout all continents. GA is a producer of unique ultrasonic level gauge and wave measurement systems for laboratory and outdoor applications. The capability to measure dynamic water surfaces with high resolution enables different applications for ship model basins, simulation facilities at hydraulic laboratories, water resource management, coastal defense, hydrography, and the oil- and gas industry. [25]

Core product lines at GA are the LOG_aLevel system described in Section 2.1.1 and the UltraLab ULS system described in Section 2.1.2. Both exist in variants and can be adapted to customer wishes. The LOG_aLevel is an ultrasonic water level measuring and logging system, that can be equipped with various hydrological and meteorological sensors to grow into a full environmental measurement station. The UltraLab ULS series is a laboratory system to measure water levels or waves at highest precision and resolution simultaneously on different locations. Another product at GA is a Sub-Bottom Profiler, it can be used to display different ground layers and identify objects underground. Further on, GA provides hydro-acoustic cut verification systems to detect cut-throughs in water jet cutting scenarios deep under water [2].

The core sensing technologies used in these products were developed in the early stages of the firm. Throughout the years, the sensing hardware was optimized. The sensors gained precision, much higher resolution, and could measure higher ranges. The power consumption was cut down as much as possible. The development was driven by customer projects that introduced new requirements, such as a long-range level measuring sensor for offshore applications. This project-oriented development approach was primarily focused on the sensing hardware. The hardware to control these sensors remained basically unchanged. The controlling hardware is a combination of various microcontrollers with additional analog or digital components placed on one or more printable circuit boards (PCB). Each microcontroller is equipped with its own firmware. A significant change to one element would result in a complete redesign of the PCB's as well as a rewrite of the firmware. The development workload, as well as the risk of potential development issues, would be too high to be carried by a single customer project. The hardware used in the sensor controllers is getting old and slowly reaches its end of live state. Newer alternatives exist on today's market that provide much better performance at much lower power consumption, which put in strong arguments for an upcoming redesign.

Not only the hardware situation strives for change, the IoT movement introduced in Chapter 1 does too. As described, the customers' expectations grew with the IoT and new functionality for the 'Things' is required. The functionality requirements emerging in the GA ecosystem include the use of sophisticated communication solutions for the provided products as well as the possibility to offer advanced data handling services to the customer. The core task of the current product generation is the data acquisition. Besides raw data preprocessing, the hardware is designed to be a data source with the ability to log data without additional functionality. All data processing and presentation requirements are satisfied by a configurable and modular desktop application running on a dedicated machine. The new requirements introduce a shift of functional requirements to different locations. On one end, some data processing and presentation functionality should now be done on the measuring hardware, e.g., a LOG_aLevel should be able to process the incoming data fully, log the raw data, and send only aggregated values to a receiver to minimize the amount of transferred data. On the other end, a shift of functionality towards the Internet can be seen. Live data should not only be visible on one desktop computer in one software but as a dashboard all over the network.

The integration of these new functionalities into the current hardware and software architecture turned into more and more complex tasks. The current controlling hardware did not allow straightforward implementations. Therefore, to satisfy all requirements for a customer project, unique solutions were found to 'somehow' integrate these new requirements. The results of the specialized developments were a range of modified product variants and data handling workflows. As most more significant customer projects had a long-term character with support from GA, the simultaneous handling of different versions and workflows resulted in management overhead and was error-prone. Even with the customization effort to adopt the products, some requirements could not be met due to the lack of hardware capabilities. Especially data handling functionality, e.g., advanced on-site data processing and presentation was impossible to integrate.

To adapt the products to the new requirements, and to simplify the implementation of new functionality, a dedicated development project was started at GA. The long-term goal of this project is a hardware refresh of the product lineup based on state of the art technologies. On short-term, a new LOG_aLevel version is targeted, it is primarily affected by quickly changing requirements and the problems they create. To achieve a uniform handling of data acquisition, data storage, data processing, and data presentation throughout the product lineup, the development of the LOG_aLevel should follow a data management workflow. The creation of this data management workflow concept was done prior to the start of this thesis and is documented in Section 2.3. During the concept development, prototypes for various LOG_aLevel functionalities were implemented. At the end of the concept creation, it became clear that a monolithic software design approach could not satisfy the flexibility needs of the concept. Thereof, the decision was made to implement a module framework. The design and implementation of this module framework is the core part of this thesis and shown in Section 4.2. As an addition, the LOG_aLevel functionalities that were prototyped to some extent during the concept creation will be combined and integrated into a modular application as second part of this thesis.

In the next section, the LOG_aLevel system as it is today is introduced. That section points out the parts of the system that can be improved for the next generation. In Section 2.1.2 the second important product at GA, the UltraLab ULS is described.

2.1.1 LOG_aLevel System

The LOG_aLevel system is a modular environmental data measuring system. As indicated by its name, the core functionality is the logging of measured water levels. Traditionally it does so, by using specialized ultrasonic sensing hardware. Appendix A describes the ultrasonic technology in detail. The core equipment of a LOG_aLevel system consists of three parts. A level sensing unit (green arrow in Figure 2.1), a reference unit (red arrow), and a LOG_aLevel controller (blue arrow) with a power supply and a data logger.

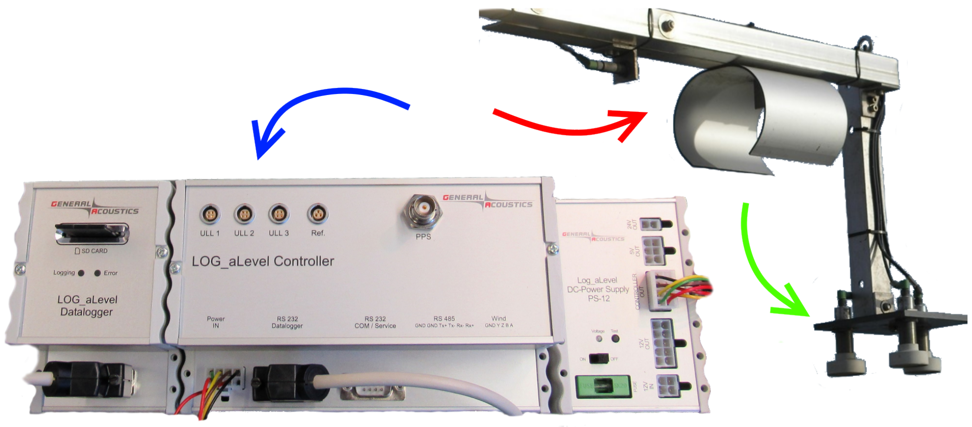

Figure 2.1: LOG_aLevel sensing unit with reference lane.

The reference is used to determine the current velocity of the ultrasonic sound waves. The LOG_aLevel system auto-calibrates itself with this reference velocity to current weather conditions. The calibration mitigates environmental influences, like temperature or humidity, and ensures highest precision. In Figure 2.1, the sensing unit is an array of three ultrasonic sensors. The use of multiple sensors strengthens the acoustic pressure and reduces the spreading of the soundwave cone. The result is a smaller measurement footprint. A single sensor may be used for smaller systems. The LOG_aLevel system can measure up to 30-meter distance and samples with up to 5 Hz. In rare cases, larger distances have to be covered. Radar sensing technology is used in such situations. The LOG_aLevel system can be used for various applications. The core applications of such a system are water level monitoring, wave measurement, and wave parameter analysis. Other applications include event alerting systems for fast detection of storm tides, floods or tsunamis. The LOG_aLevel system can be extended by a wide range of hydrological and meteorological sensors to provide a complete environmental monitoring station. Examples of addable sensing hardware are temperature sensors for water and air. Others are air pressure, wind, humidity or rain sensors. More advanced additions include visibility range sensors, cloud height sensors or flow measuring systems like an acoustic Doppler current profiler (ADCP) [53].

The complexity of the system varies depending on the configuration. Table 2.1 presents sales per year for different LOG_aLevel system configurations. Higher complexity systems include the features of the lower ones. It is visible that more complex systems are sold far less than pure level measurement stations. Yet, in recent years the revenue generated by 'Complexity 3' sales exceed the revenue of the other sales by far. Consequently, GA is anxious to offer more 'Complexity 3' systems with the corresponding environmental information management and project-wide long-term support. The focus on more complex systems with project character was troublesome. The requirements on the sensors that were used as extensions changed rapidly over the years. The initial design was intended for a small fixed set of additional sensors. The inclusion of more complex sensing technology, as required by customer projects, is always accompanied by substantial development stunts. Missing input-output (IO) lines and limited processing power of the current LOG_aLevel interfered most with the development.

Table 2.1: LOG_aLevel system complexity compared to sales per year.

LOG_aLevel Complexity	**Complexity 1** · level measurement · LOG_aLevel software	**Complexity 2** · data logging · simple sensors (wind, temperature)	**Complexity 3** · long-term support · custom solutions · advanced sensors (ADCP, cloud height)
Sales per Year	10	4	2

The numbers are of theoretical nature but reflect the current situation at GA.

As shown in Section 2.1, various accumulating problems lead to a pure development project at GA. The core of the development project is the refresh of the product lineup based on a consistent data handling strategy. For the LOG_aLevel, which is the first refreshed product, the goal of the development is the adaption to an extended range of possible data sources. The well-established modularity and scalability of the system should thereby be preserved. Not only the hardware should be scalable. The corresponding data management workflow should as well be structured into sub-functionalities. The modular structure would be configurable to different configurations similar to the hardware. The core component of such a redesigned LOG_aLevel system is a new controlling unit. It handles data from various data sources and serves the data over multiple interfaces. The functionality of the controller should be simply extendable to store and process the data. The design of the mentioned data management workflow concept was heavily influenced by the characteristics of the LOG_aLevel . The required LOG_aLevel functionality exceeds the requirements of other GA products. Other data controllers have a simpler structure, and have to cope with one data source, or a range of identical data sources. For that reason, the design of the concept focused on a well-designed solution for the LOG_aLevel. A modular design may then allow simple derivation to other products.

2.1.2 UltraLab ULS System

The UltraLab ULS System is the second crucial product for GA. It is a high speed and calibration-free system based on ultrasonic technology. Various different applications exist. Base application for all devices is the measurement of water levels and waves. Application sites vary from hydraulic laboratories, ship model basins, to towing tanks. Depending on the product version, different configurations are possible. The number of measuring channels ranges from 4 to 16 with one or three ultrasonic sensors per channel. The sample rate ranges from 10 Hz up to 250 Hz. The maximal measuring distance ranges from 30 mm to 600 cm. The channels are synchronized and can be used with an external trigger. A reference line can be used to calibrate the ultrasonic sensors. Figure 2.2 shows an UltraLab ULS Advanced. It has four channels with three sensing units per channel. A switchable sampling rate of 50 Hz or 100 Hz can be used. The device is meant for high precision measurements at very turbulent water with steep waves and maximum relative wave speeds of 15 m/s.

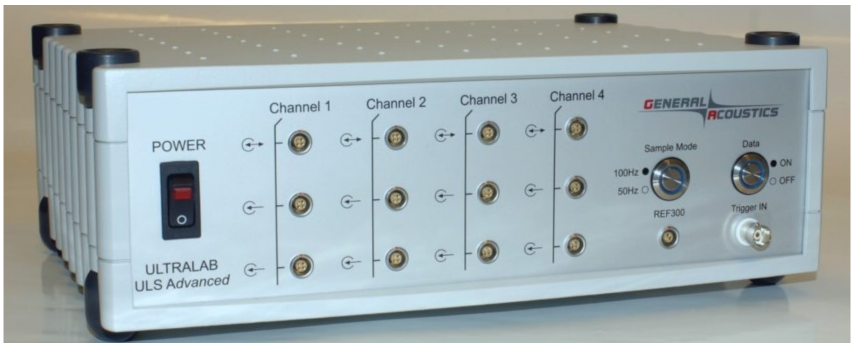

Figure 2.2: UltraLab ULS Advanced.

In contrast to the LOG_aLevel system, the sensor configuration remains stable. UltraLab ULS systems must only deal with a set of ultrasonic sensing hardware. The limiting factor of the current design is a slow connection of the processing microcontrollers of each channel to the master microcontroller over a Serial Peripheral Interface (SPI) bus. The amount of data generated at each channel (up to 9 MB/s) reaches the limits of SPI at 16 channels with one sensor each. The real-time processing of this amount of data can be currently done, but also reaches the limit of the possible.

2.2 GAPP Protocol

The **G**eneral **A**coustics **P**roperty **P**rotocol is a stateless, message-based protocol. It was derived from the National Marine Electronics Association (NMEA) protocol [52] and adapted to the use cases of GA. As the name indicates, GAPP is centered around 'properties'. In the GA environment, a property represents the state of an entity. This broad definition of the term allows almost everything that has some state information to be treated as a property. The main use case for the GAPP protocol is the communication between GA products with external systems. Various properties exist in those systems. For example, the measured values from the sensors are properties. But also configuration settings of the systems like output rate or offset and calibration coefficients are handled as properties. Two types of properties exist, active and passive properties. The value of active properties can change internally without interaction with the GAPP protocol. Examples are the measurement values of sensors. Passive properties, e.g., configuration values have to be changed over a message.

The protocol follows the Representational State Transfer (REST) or the RESTful way to communicate between systems closely. The term was introduced by Roy Fielding to design Hypertext Transfer Protocol (HTTP) 1.1 and Uniform Resource Identifiers (URI). Since then, if one refers to REST, habitually HTTP is implicitly meant as protocol. GAPP follows the principles of a RESTful communication but has nothing to do with HTTP. Roy Fielding defined the following principles of REST in his doctoral thesis [21]:

- **Client-Server** – Separation of user interface concerns and data storage concerns.
- **Stateless** – A request must contain all information needed for the server to handle the request. It shall not depend on client state information on the server.
- **Cache** – Responses should be cashed on a client to improve network efficiency.
- **Uniform Interface** – a fundamental constraint in the design of a RESTful service. It enables independence of the involved components by simplifying and decoupling them. The interface should allow the identification of the resource in each request. A request should allow the manipulation of resources through the requests meta-data. Each message should be self-descriptive.
- **Layered System** – A client does not know if it is connected directly to the 'final' server or to an intermediate server.

The GAPP Protocol mostly follows the REST principles. However, due to the nature of active properties, some modifications had to be made. The core change introduced, is the ability to push messages to the clients without a request. If a value is measured by a sensor, the value shall be actively sent to all clients. Contrary to the REST way, where the state of the sensor property has to be pulled actively. The client-server roles were slightly modified and renamed to:

- **Observer** – It caches the incoming values from the source and is able to get or set the values of a property held by a source.
- **Source** – The proprietor of the properties, it sends messages to all observers when a property changes.

The statelessness of the communication is guaranteed through the self-contained messages in the GAPP protocol. Two kinds of messages exist. A '**query**' message, which corresponds to a HTTP GET message and allows to ask a source for a property value. And a '**definition**' message, corresponding to a HTTP SET message, that can be used to set the values of a property. A query can only be sent by an observer. An observer does not hold any values and is not allowed to respond to a query. The response to a query is sent as a definition message from the source back to the observer. If a definition is sent to the source, the source acknowledges the definition only if the values for that property have changed. In which case it sends a definition back to the observer. The cacheability principle of rest is satisfied by an observer. The observer holds the current values of the properties. In fact, contrary to REST, it always holds the currently correct value of the properties due to the characteristics of the source that always broadcasts changing properties to the observers.

Each message starts with the name of the property. The name can be prefixed by dot-separated scopes to logically group the properties. `SYS.SERIALNUMBER`, `LEVEL.OFFSET` or `W` (wind) are examples of property names. The property name can be extended with an index when a range of identical properties exists. An UltraLab system may have a range of channels, and each channel can be calibrated via coefficients. The names for this property would be `CHANNEL.COEFF[n]`, where `n` is the channel number.

A query message is formed only by the property name. A definition message is formed by the property name, an equal sign (`=`), and a comma-separated list of values that are needed to define the property. The value types may either be strings, integers, or floating-point numbers. This message structure follows the RESTful way. It allows to manipulate properties with a single definition request. At the same time, it violates one REST characteristic. It is not self-descriptive and does not provide information over the transferred data. The GAPP protocol was designed for the use cases of GA. Thus, a global dictionary with available properties and their type structure can be created. This removes the need of type encoding in the messages. All in all, the GAPP protocol allows a RESTful pull communication between observer and source with the additional push functionality of the source to the observer. A layered approach can be achieved by adding the role of a proxy. The proxy presents itself as a single source. It uses internal observers to forward messages to multiple sources located deeper in the hierarchies.

Figure 2.3 shows an exemplary communication flow between an observer and a source. Green arrows indicate the automatic pushes of the source. Some properties return more than one line long messages. Such a situation is the querying of the system configuration. In this case, one line with the definition of this property is returned followed by a range of comment messages. A comment message is formed with // and a text string. In the figure, an example is shown and marked with dark yellow. The properties that are handled by GAPP may have access restrictions. If a violation of these restrictions is detected, the source answers with an error message (blue arrow). The same behavior can be seen when a wrongly formatted or semantically incorrect definition is sent to a source.

The GAPP engine used at GA is decoupled from the communication interfaces. Additional layers can be included between the GAPP processing and the communication layer. Layers for encryption or compression may be added. GA uses a checksum layer that adds a XOR checksum to each message, which enhances the data integrity significantly.

Until now, the protocol is used for the communication between the measurement devices and the end-user or a software solution by GA. The aim is to widen the use cases of the protocol in the context of the developed data management workflow concept of Section 2.3. The idea is the use of the protocol as inter-modular or inter-process communication to unify the used interfaces, make the modules exchangeable and enhance the testability of each part.

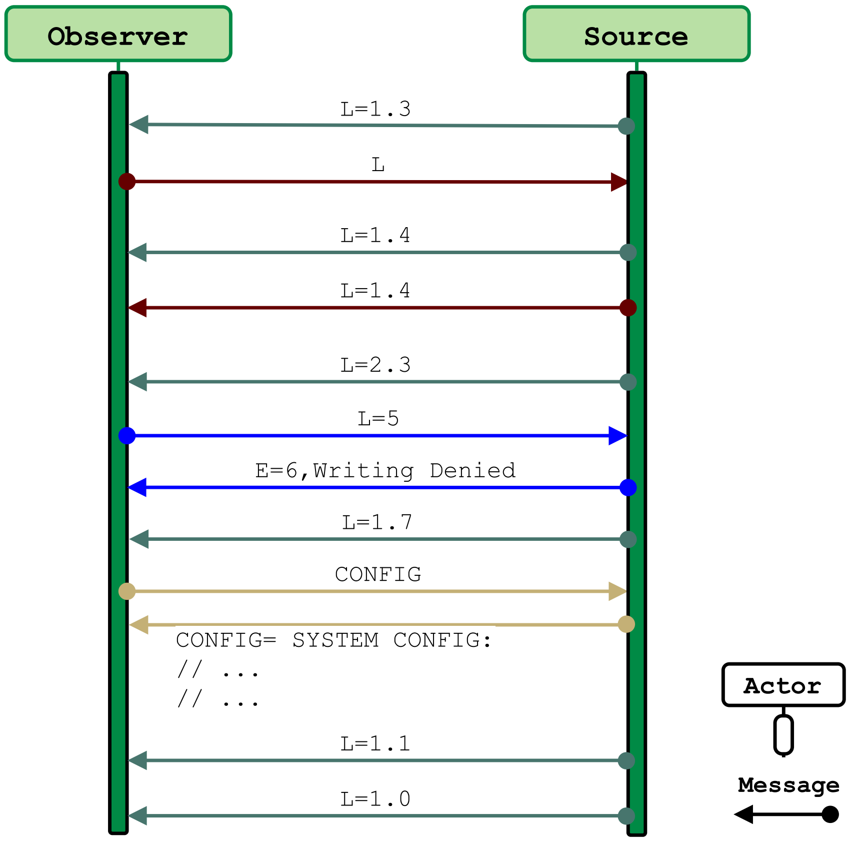

Figure 2.3: GAPP communication examples between source and observer.

2.3 Sensor Data Management Workflow Concept

This section describes the sensor data management workflow concept of GA. It was developed before the start of this thesis. The concept was created in an attempt to organize and structure the handling of measured environmental data. It should guide the development of new hard- or software products to deal with the ever-changing requirements of IoT. The workflow concept should cover the core requirements of the GA environment and consider the project-oriented development process. The design process is described in Section 2.3.1. A top-down approach was taken to design the workflow. Related work in academia and the commercial sector was considered in the design, as well as requirements and experiences by GA. Use case scenarios were crafted, to be prepared for possible customer project in the future. The resulted workflow concept is described in four sections (2.3.2 - 2.3.5). The degree of abstraction varied. Uncertainty on the hardware that will be used for the next generation of products at GA hindered the design process. For example, the data acquisition step of the workflow is only described on a more abstract level. It shows the algorithms and protocols used, but it hides implementation specific details. Data storage and data processing, on the other hand, were not depending on hardware decisions and could be designed in a more detailed way.

2.3.1 Design Process

The design process of the workflow concept followed a top-down approach. In a first step, current research in the environmental data measuring field was analyzed. Two different approaches to the management of environmental sensor data were found. One approach from Horsburgh et al. [30] was concerned with the handling of continuous time-series data, where a real-time aspect is not as important. Their focus was on collaboration between universities and a consistent data handling. An advanced database schema, the Observational Data Model (ODM) was developed to support all kinds of environmental data. The second approach from Wong and Kerkez [78] is focused on the real-time handling of sensor data. They perform an *adaptive sampling* [5] [6] of water quality based on current environmental values. Both environmental data handling workflows did not cover the needs of GA. Horsburgh et al. lacked required real-time processing and presentation capabilities of the data. Wong and Kerkez included external cloud services as data storage facility, which is not suitable for GA from a security perspective. Other sensor data handling workflows were searched in the commercial environment. Providers of such workflows, e.g., Campbell Scientific Inc. [9] or OceanWise [54], try to deliver a complete data handling experience for their systems. Some provide a relational database to handle environmental data, mostly wrapped in closed source applications. Complete solutions can be costly and may result in a lock-in effect.

The considered workflows in academia were research-oriented and present generalized solutions applicable to a wide range of use cases [30] [78]. In contrast, the workflow designed for GA had to be focused on their products, use cases, and requirements. A data management workflow similar to the Campbell Scientific, Inc. [9] closed source data management solution to support own products was intended. Albeit reduced and simplified to the handful of use case scenarios that exist at GA. It was not the intent of GA to implement full-fletched scenarios that could handle all kind of environmental data, as e.g., envisioned by the ODM database scheme [32].

After the current research was considered, the needed requirements to design a suitable workflow for GA were engineered. The GA ecosystem had to be narrowed down, broken into core pieces and analyzed in detail. A broader use case overview of the GA ecosystem provides the basis. It is shown in Figure 2.4 in an Unified Modeling Language (UML) use case diagram. On the highest abstraction level, a single use case exists – A GA customer wants a GA solution **to measure** some form of **environmental data**. This data measurement use case is composed out of four data management tasks. In the figure, the four tasks are modeled as sub use cases. **Data Acquisition** and **Data Processing** are always included in a GA system. **Data Acquisition** contains the physical data measurement. This includes the timestamp management of the measured signals. **Data Processing** at least contains the preprocessing of the raw data into human understandable values. In more complex systems, the **Data Processing** use case can contain advanced processing algorithms. They perform aggregate computations or calculate more complex values from the measured data.

Depending on the customer requirements, the data measurement use case can be extended by a **Data Storage** and a **Data Presentation** use case. Data storage adds the possibility to store the measured data. The data is stored either as text-based log files or in a structured storage solution with interval-based access to old data. Data presentation adds new ways for the customer to access the data.

The architecture of the GA system influences the order and the structure of these sub use cases in a provided customer solution. Some use cases may exist at multiple locations in systems with spatially distinct components. For example, an offshore station with a server station onshore or in a measuring network. Some use cases overlap. Some processing has to be done before the data can be stored. The data presentation can happen at different locations with the need to access the stored and processed data. Thus, a clear spatial and chronological separation between the use cases is not possible. Five scenarios were created in an attempt to structure the relationships between the four use cases. The scenarios describe the main use case configurations in customer projects. They are based on experiences with previous projects and expected market trends. The next paragraphs describe the scenarios.

Figure 2.5[1] shows the simplest and most common use case scenario in GA systems. A self-contained device, that measures data with one or more sensors, processes the measured values in some way and outputs the data as a GAPP data stream to the customer system. The customer system is tasked with the management of the data such as data logging, advanced processing or presentation. An environmental data measuring system at GA is called a `station`, in the figure shown as a grey box. A `station` can be a single physical device or a group of directly connected devices. Examples of this scenario are laboratory devices (UltraLab ULS) or simple measuring stations without data management needs (basic LOG_aLevel configuration). The red shaded data acquisition component in the figure is the core component in all GA system scenarios. It contains physical data measurement and timestamp management functionality. Both are real-time critical.

[1]The four sub use cases can be seen as functional distinct components and are shown in the figures as UML component diagram symbols

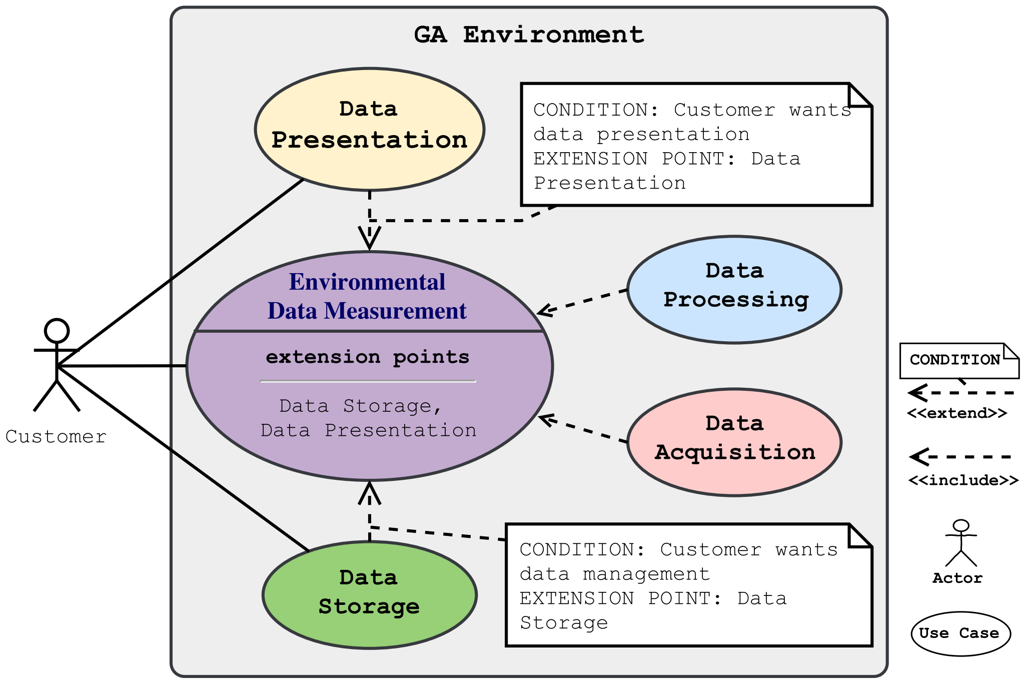
GA Environment
Data Presentation
CONDITION: Customer wants data presentation EXTENSION POINT: Data Presentation
Environmental Data Measurement
extension points
Data Storage, Data Presentation
Data Processing
Data Acquisition
Customer
Data Storage
CONDITION: Customer wants data management EXTENSION POINT: Data Storage
CONDITION
<<extend>>
<<include>>
Actor
Use Case

Figure 2.4: GA core use cases.

To cope with changing sensing hardware requirements over time (see Section 2.1), it was decided to implement an abstraction interface between the data acquisition and the remaining workflow components. The key idea of this interface is the decoupling of the hardware-oriented real-time data acquisition part from the software-oriented data management part. The connection between the red acquisition component box and the blue box represents this interface.

The second use case that is always present in a GA system in some form is data processing, in the figure shown as a blue component box. Preprocessing may be the conversion of a range of ultrasonic echoes to the correct level value, offset calibration of measured values, or outlier fixer algorithms. The processing component presents the processed data over a GAPP data stream where it can be accessed by the customer system. The components described in this scenario are the foundation for all other scenarios and, therefore, are not described again when their intent is clear.

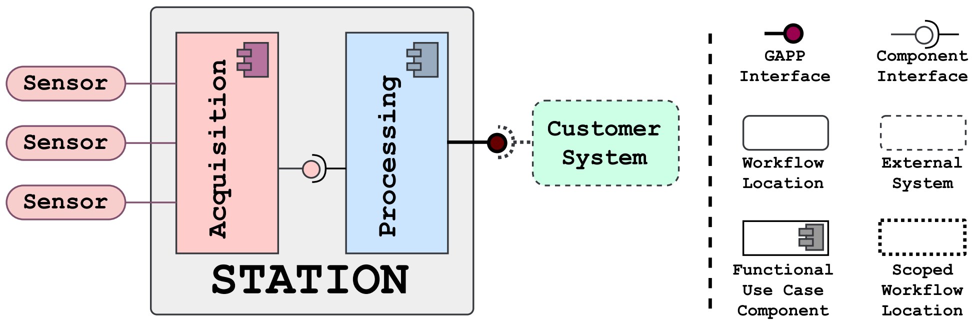

Figure 2.5: Simple use case scenario.

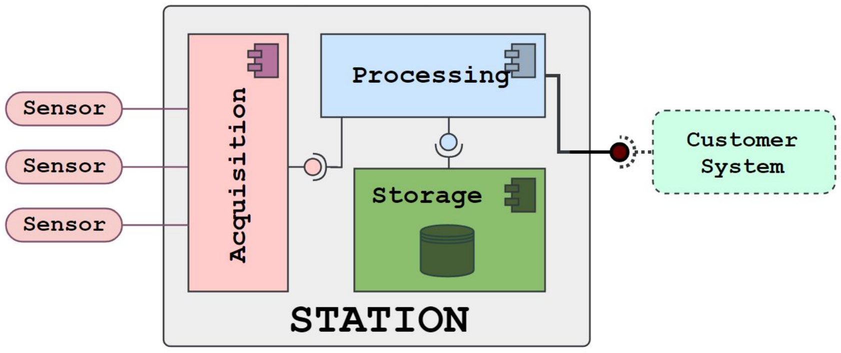

Figure 2.6: Use case scenario with data storage.

The second very common use case scenario is almost identical to the first scenario. Shown in Figure 2.6, the green data storage component is the only addition. Self-contained measurement stations with limited power supply and, therefore, no or limited communication possibilities are typical stations where such a scenario is used. This scenario, with the added data logging, is more and more common in GA solutions, even when the station has a full connection to the customer system and a stable power supply. It serves as a backup when technical problems arise.

The two introduced scenarios, moreover the stations of the scenarios cover what the current product generation at GA can provide. Specialized functional additions are possible, but introduce colossal development overhead as described in Section 2.1. The next scenario, shown in Figure 2.7 shows the functionality blocks that should be covered by the next hardware refresh at GA. Scenario three enhances the functionality of the measurement station to comply with current customer requirements. The data acquisition stays the same as in the previous scenarios, so does the interface to the processing as well as the preprocessing step. From there on the data is not only logged to files, it can also be stored in a structured data storage component, which allows an interval-based access to old data. The green storage box was moved inside the processing component as it interacts deeply with the added processing functionality: An interface has to exist that allows the inclusion of advanced postprocessing algorithms into the station to generate aggregated values. An example of a postprocessing algorithm would be the calculation of wave characteristics from raw level values. Newly introduced in Figure 2.7 is the yellow presentation component. Data presentation contains every way to serve the data to the customer or the next system. The presentation functionality is optional and can vary depending on requirements.

The three presented scenarios have focused on the measurement station and assumed an integration into a customer system. Some station configurations, the ones with data logging functionality, may even run without a connection to an external system. The borders between the scenarios are not clear, mixes between scenario two and three emerged in the last years. The goal of the developed scenarios was to present a minimal, a standard, and a maximal equipped station scenario. The next two scenarios introduce situations where GA provides the surrounding infrastructure for one or more stations. Which station variant is used for the next scenarios is not relevant, but to cover all possible functionality, a scenario three station is assumed.

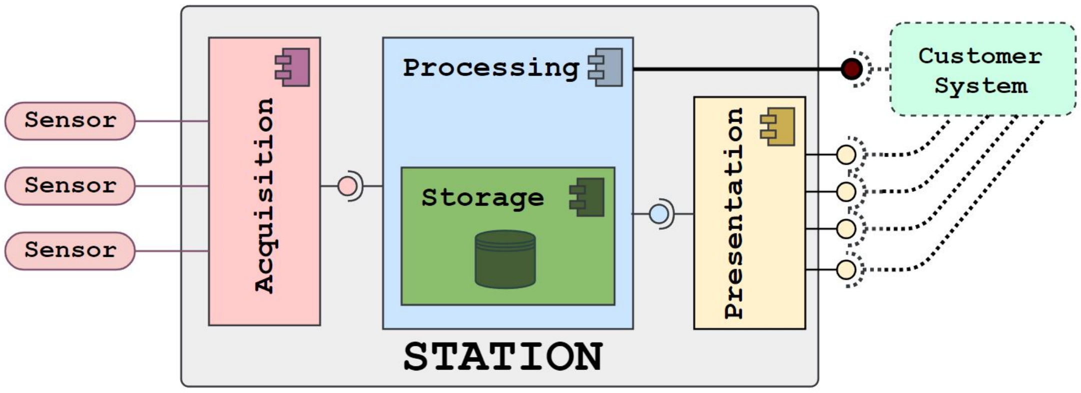

Figure 2.7: Advanced use case scenario with processing and structured data storage.

Figure 2.8 shows scenario four and introduces the concept of a `measurement site`. In GA terms, a measurement site is a location where one or more stations are connected to a `station server`, in the figure shown as a light blue box. A station server is used to collect data from a station where it is processed, stored, and then presented to the customer. Currently, the LOG_aLevel (or UltraLab) desktop software fulfills all those tasks in a modular and quickly adaptable way. It can be seen in Figure 2.8 that the components in the station server are equal to the components in the station of Figure 2.7 apart from the data acquisition. The goal of the future data management workflow, as described in Section 2.1.1, is the structuring of the overall functionality into smaller sub-functionalities, which can be better spread around in a system. Therefore, the components were designed in a compatible way.

The last scenario is shown in Figure 2.9. It shows an environmental data measurement network with multiple measurement sites and a datacenter. Projects with this structure are targeted by GA but could not be done until now. The goal is a replication of the data storage components in the station servers into the datacenter. Therefore, the presentation component can present site overlapping data. The blue processing component is only needed to manage the different storage components and provide unified access.

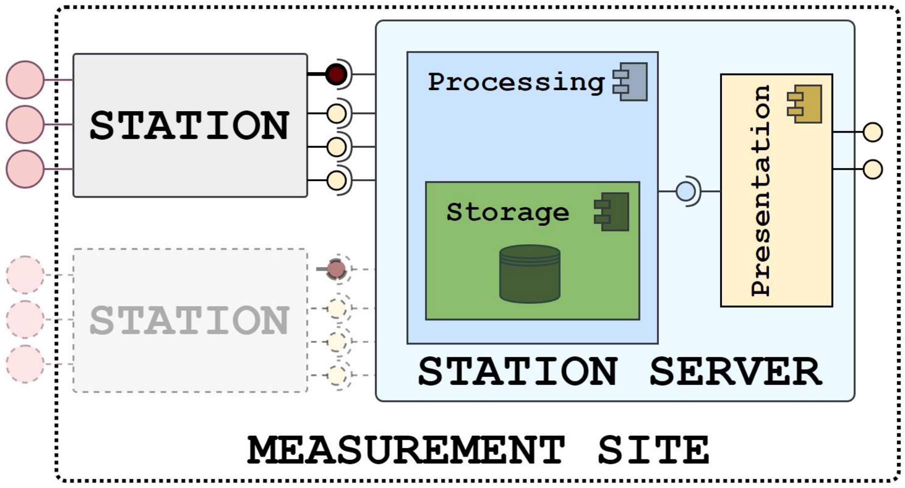

Figure 2.8: Use case of stations with a station server.

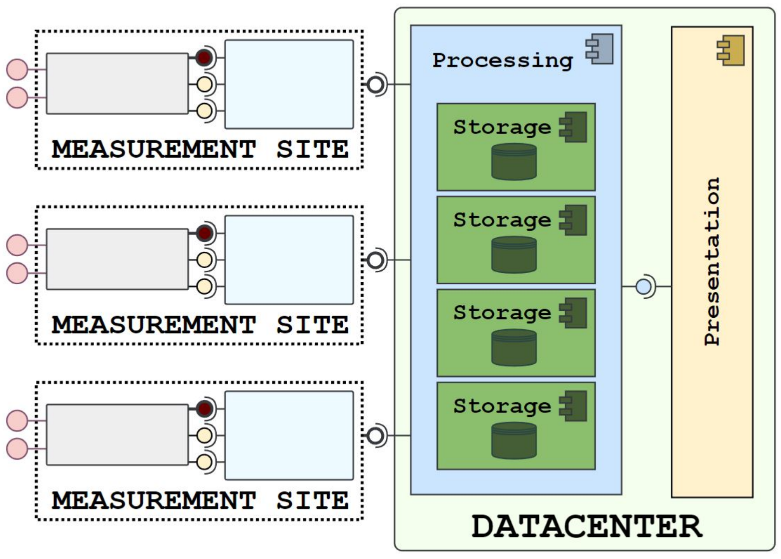

Figure 2.9: Measurement network scenario with multiple measurement sites.

The five introduced scenarios cover the structure and distribution of the use cases shown in Figure 2.4 based on real-world scenarios. A range of functional requirements for the four use cases were presented in the introduction process.

Besides creating the use cases and use case scenarios, a Linux driven **D**ata **H**andling **U**nit (DHU) was added as a requirement in the station controllers. It introduces more processing power and simplifies the software development with newer high-level technologies. Mapped to the scenarios, the DHU is responsible for everything behind the data acquisition part, which has to be real-time driven. The inclusion of a Linux DHU implies some architectural problems that have to be solved. The data acquired by the real-time part, further called **R**eal **T**ime **U**nit (RTU), has to be transferred to the non-real-time Linux system. Therefore, the abstraction interface was added as shown above. It hides the implementation specific hardware details of the RTU from the DHU and decouples both. The area of responsibility of the RTU is reduced to controlling the measurements and timestamping the results. The DHU will perform all other data handling tasks. The intention of this abstraction and the clear cut of responsibility division between the real-time and the Linux component is a decoupling from the sensor hardware development and the development of the data handling process afterward. The integration of new hardware should not force a rewrite of all software components, only new hardware specific classes should be added. Figure 2.10 shows a station with the added DHU (yellow) and RTU (light red). Depending on the scenario, the DHU may contain functionality in different complexity.

After the use cases and scenarios were elaborated, concrete requirements were created. The design process of the workflow concept is not part of this thesis. Therefore, the decision making on the requirements is not described[2]. One important fact that had to be considered in the design of the workflow was the limited available development resources at GA. They call for a pragmatic approach regarding implementability. GA should be able to implement the workflow into their ecosystem with reasonable effort.

[2] Further information on the requirement design can be requested at GA

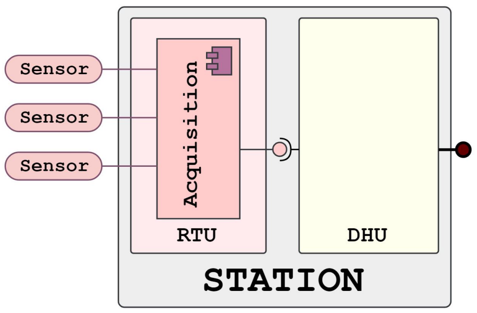

Figure 2.10: Separation of RTU and DHU in a station.

The final environmental sensor data management workflow concept is presented in the next four sections. It serves as a basis for the work in this thesis. The resulting structure of the software components to store and process the sensor data were prototyped at GA . It became clear that a modular approach to develop the components is needed, whose development is the core of this thesis. Further on, the LOG_aLevel 2.0 prototype, developed as the second contribution in this thesis, is also based on the workflow concept shown in the next sections.

2.3.2 Data Acquisition

For GA as a measurement device producer, the data acquisition process belongs to the core tasks of a data management workflow. This section introduces the data acquisition part of the data management workflow concept. Compared to the following sections, this section depends heavily on the hardware development department at GA. During the concept creation, the hardware design changed multiple times, without resulting in a concrete concept. Therefore, the data acquisition functionality is described from a higher abstracted view, detailing only the concrete interface decisions. The data acquisition step happens only in the stations, but it is present in every scenario.

As introduced in Section 2.3.1, a Linux DHU was added to the station. The result is a functionality and responsibility separation between the real-time part and the DHU. The inclusion of a DHU, which runs with a scheduler-based Linux operating system (OS), requires data buffering on the RTU side as it is unsure when the Linux scheduler fetches the queueing data. The key tasks of the RTU are the controlling of the sensor measurement as well as the timestamp management. In most cases a Coordinated Universal Time (UTC) synchronized pulse per second (PPS) signal is available for the RTU to set and correct the internal master real-time clock (RTC). If no PPS signal is available, the correct time has to be set by hand. A low-power central processing unit (CPU), ARM Cortex-M0 [4] or similar, will be used to control the RTU. The sensors will be attachable to the RTU in a configurable way. A unique sensor ID (SID) identifies each sensor. A physically identical sensor at a different hardware port has a different SID. This unique identifier of the sensor hardware together with a concise software interface is critical for the envisioned decoupling of the sensing hardware and the following data management workflow. As mentioned, a buffering mechanism is needed to transfer the measured data from the RTU to the DHU. It was decided to use a ping-pong buffer system, where one buffer is read by the DHU and the second is written by the RTU.

Besides the hardware interface between the DHU and the RTU, the software interface is critical for the decoupling of the RTU and the DHU. Apart from the globally unique SID for each sensor entity, each data source has a two-byte long channel number assigned. A sensor may have one or more channels assigned, for example a temperature sensor may have a single channel assigned for its values while a level sensor with three ultrasonic transducers may have at least three or more channels assigned. Even a time signal or system status information can have a channel assigned, such channels can belong to a virtual sensor. A contract is defined for each unique channel in the GA ecosystem. The contract states what data belongs to the channel and the way the data is formatted. Based on this contract, the RTU prepares the measured data and creates chunks that contain the two channel bytes, a four-byte payload size, and a byte payload. As the buffer mechanism has limited memory available, a memory optimized binary representation of the data is preferred. The RTU writes incoming chunks to the buffer and keeps track about the number of written bytes. When the write buffer is full, a four-byte number, containing the complete number of written bytes, is written to the beginning of the buffer. Therefore, the DHU knows how much data is available in the buffer. The concrete channel configuration of the current system for each RTU is stored on the DHU. On every system start, the RTU fetches the configuration from the DHU and adapt the firmware at runtime to the current architecture.

The next section describes the data processing functionality of the workflow, which includes the handling and preprocessing of the chunks that were created by the RTU and are accessible by the DHU over the buffer.

2.3.3 Data Processing

The data processing functionality is twofold. First, the chunk data that is acquired by the RTU and passed to the DHU has to be preprocessed to create human understandable values. This process is always present on a station. Second, interval-based postprocessing of gathered data has to be possible on the station server and the station over clear defined interface. After some general terms in the GA environment are introduced, the concept for both, preprocessing and postprocessing, is described below.

In order to enable the understanding of the following data management workflow concept steps, some GA specific terms have to be described. These terms emerged in the past years as the number of data management oriented projects rose. Three terms are used to classify the environmental data:

- **Raw Data** – Data that comes directly from physical sensors without any transformation. The chunks produced by the RTU is classified as raw data.
- **Primary Data** – Data that was preprocessed and brought into human understandable form.
- **Secondary Data** – Data that is based on primary data and was created by postprocessing algorithms.

The concept of secondary data is context depending. The postprocessing of level values on a station to create wave characteristics may, from a station viewpoint, be the creation of secondary data. Viewed from an external system, the wave characteristics data can be seen as primary data. Therefore, entities of an external scope may decide how they treat data that is classified as secondary data in a local scope.

Further on, as described in Section 2.2, the communication with a station is done over the GAPP protocol. Coupled with GAPP comes the concept of a **Property**. A property may be anything that has a name with an associated value. Essential for the processing functionality are properties that influence the processing algorithms, but also the properties that directly model the measured and processed values. Good experiences with GAPP in the past was a decision maker to use it more often in the new data management workflow, and not only for the communication between station and station server. In compliance with the idea to create a more structured, modular, and, therefore, configurable software architecture, GAPP should also be used as intermodular communication between software modules on the station and the station server. The benefits are a unified interface and simpler testing of separated modules. The cost of an ASCII-based communication between software modules has to be evaluated when the prototyping phase has started.

The first part of the processing functionality is the preprocessing of raw data to primary data. The data acquisition is done by the RTU. Therefore, the preprocessing functionality can be seen as the first module of the DHU, which is present in every configuration. Figure 2.11 shows a component diagram of the preprocessing module. A station with minimal configuration may be complete with this module.

The module is called **Primary Data Unit** (PDU). The black shadowed driver component in the PDU handles the connection to the RTU and is able to process the chunks. As described above, each sensor entity, physical or virtual, has a unique SID assigned. These sensor entities are represented in the figure as multiple violet shaded components. The sensor components contain the preprocessing functionality and output the primary data. As input serve the chunks generated by the RTU, which contain the channel number and the payload. Therefore, a sensor entity in the GA environment has the following characteristic:

- **Chunk Producer** – Each physical or virtual sensor has at least one data source that generates chunks. Additionally, the sensor objects in the PDU may produce new chunks that may be consumed by other sensors.
- **Chunk Consumer** – Each sensor in the PDU consumes at least its own chunks to generate primary data, it may use chunks that are produced by different sensors for its calculation.
- **Primary Data Producer** – Each sensor in the PDU outputs primary data, which is forwarded over GAPP.

An assigned channel ID (CID) to a specific SID as producer may not change under any circumstance, whereas additional consumers of a chunk can be added. The driver distributes the chunks to the appropriate sensor object. The cardinality between CID and SID mapping is N-to-N, a chunk may be distributed to many sensors, and a sensor may need a range of different chunks to create its primary data.

Each sensor object has preprocessing functionality, in the figure shown in blue. The algorithm has the knowledge of the chunk format and keeps track of the arriving chunks. The processing algorithm has access to the GAPP properties, which are handled by the GAPP source colored in green. The dashed green arrows symbolize this access informally. The properties may be used for the preprocessing algorithms. When the processing component has gathered all necessary chunks for its algorithm to work, the algorithm processes the chunks and creates a primary data value. Each sensor object is associated directly with a property. The generated value is then handed to this property, from whereon the GAPP engine is responsible for the further distribution. The primary data generator together with the RTU covers the core functionality of the current hardware generation. The new architecture allows a simple modification and addition of the hardware and the software. The interface for the inclusion of new preprocessing algorithms for newly added sensors is thin and clear defined. A developer has to implement a processing class based on an abstract processor interface that only contains a single procedure.

The second processing functionality part is concerned with the postprocessing of primary data to create secondary data. In contrast to the preprocessing, which operates element-wise, the postprocessing algorithms generally work with intervals of measured data. These algorithms could not be implemented on a station until now, as the limited hardware had prevented the buffering of such an amount of data. The postprocessing was done on the station server with the help of the LOG_aLevel software.

In one project in Iraq, GA used a database solution to store the environmental data for the first time. In this project, the postprocessing ran as a separate process and received notifications from the database when enough data for the processing of an interval was available. When a notification arrived, the process selected the interval of data from the appropriate table, made its calculations, and inserted the created secondary data into a different table. The workflow was working great, and the database provided more flexibility to present the data to the customer than the file-based workflow. Therefore, it was decided to integrate such a data storage solution into the workflow concept.

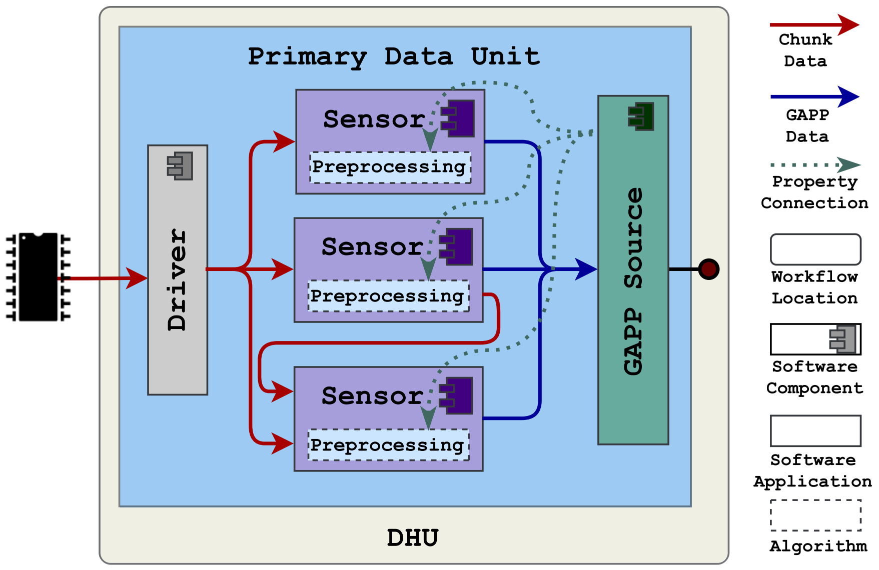

Figure 2.11: PDU component overview.

Figure 2.12 shows the data storage unit (DSU) on the DHU. It is connected to the PDU described above. Even if the postprocessing concept is shown for the station module, the same functionality is intended for the station server with a slightly modified implementation. A GAPP observer handles the incoming GAPP data stream. Whenever a property changes, the new value is inserted into the structured data storage solution. The solution has dedicated storage for primary and secondary data. The primary data, coming from the PDU, is inserted into the appropriate primary storage (marked with P). The storage module knows the intervals of the postprocessing and checks at each insert if postprocessing is needed. If it is, a notify event is sent to the postprocessing manager (violet). The postprocessing manager fetches the needed data over the correct interval from the primary data store. The processing algorithm processes the fetched data and inserts the calculated data into the data store.

This concept is already in production on the station server in the Iraq project and has proven to work very well. The notification concept allows triggering the secondary data generation manually, which is vital for service purposes. The actual processing algorithms expect a range of timestamp-value tuples and return a secondary data value. These algorithms are independent of the actual implementation of the notification and data access mechanisms and can be used on the station as well as on the station server.

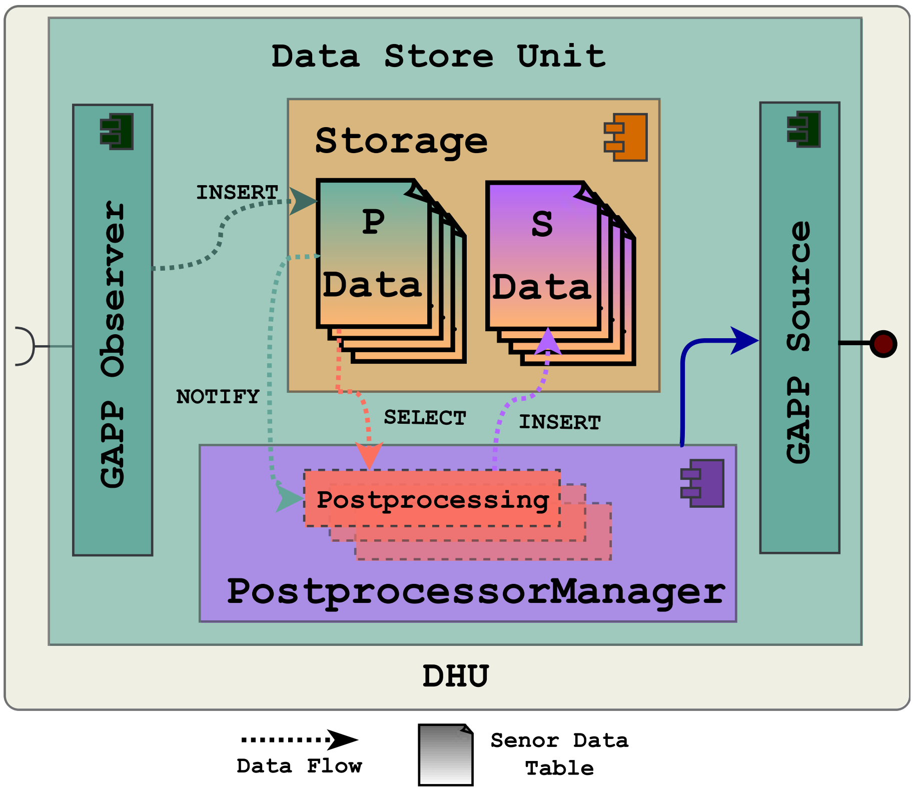

Figure 2.12: DSU component overview with postprocessing mechanism.

2.3.4 Data Storage

Storing the measured environmental data is an essential step in an environmental data management workflow. This section presents a data storage concept for the workflow in the GA environment. In contrary to the data acquisition functionality, which is strongly dependent on the hardware development department of GA, the core decisions for the storage functionality concept could be taken. The following paragraphs, first, introduce the simple GAPP data stream logging functionality, as it is present in current stations. Second, a solution for a structured data storage component, which can be used on a station and a station server, is introduced. Lastly, the possibility to combine the structured environmental data from a range of measurement sites into a datacenter is shown.

Each GAPP component has an associated IO manager, which coordinates the dispatch and receiving of GAPP messages over the available hardware interfaces. This IO manager can easily be extended to log the incoming and outgoing traffic to a file. To couple the raw data logging to the GAPP interfaces opens up new possibilities in the data management. With the intensified use of GAPP in the station and the station server, it is possible to log the data at various locations. The outgoing traffic of a station, as well as the incoming traffic of a station server, could be logged and compared to find issues in the communication. Logging of output from various components inside the station could be used for testing purposes and to detect bugs early in the workflow.

Two core decisions on the design of the structured storage component were taken during the concept creation. First, the data storage technology had to be decided. Second, the modeling of the data had to be specified. In the context of databases, the way how the data is modeled is called a database schema.

Both decisions were made based on research, experience in past projects, and prototype implementations. The decision on the data storage technology, which should be used for the first implementation of the concept in the second part of this thesis, fell on a Structured Query Language (SQL) solution. GA has built up knowledge in the SQL database area in the past two years, which can ease the development process. Further on, a SQL database implementation on a station server exists in the mentioned Iraq project, from where some functionality can be reused. SQLite was used as an embedded library in the data storage component, as it is the de-facto standard for embedded databases and has a lot of available resources to simplify the development.

A conservative approach was chosen for data schema – it was decided to use a sensor data schema that can be adapted to different data storage technologies. This results in a flat, decentralized data schema for sensor data. Beside a decentralized schema for the actual sensor data, which can be implemented with various data technologies, the relational meta-data should be implemented with a SQL technology to allow a highly normalized data schema. As the meta-data information rarely changes, eventual performance problems of a SQL solution are not an issue.

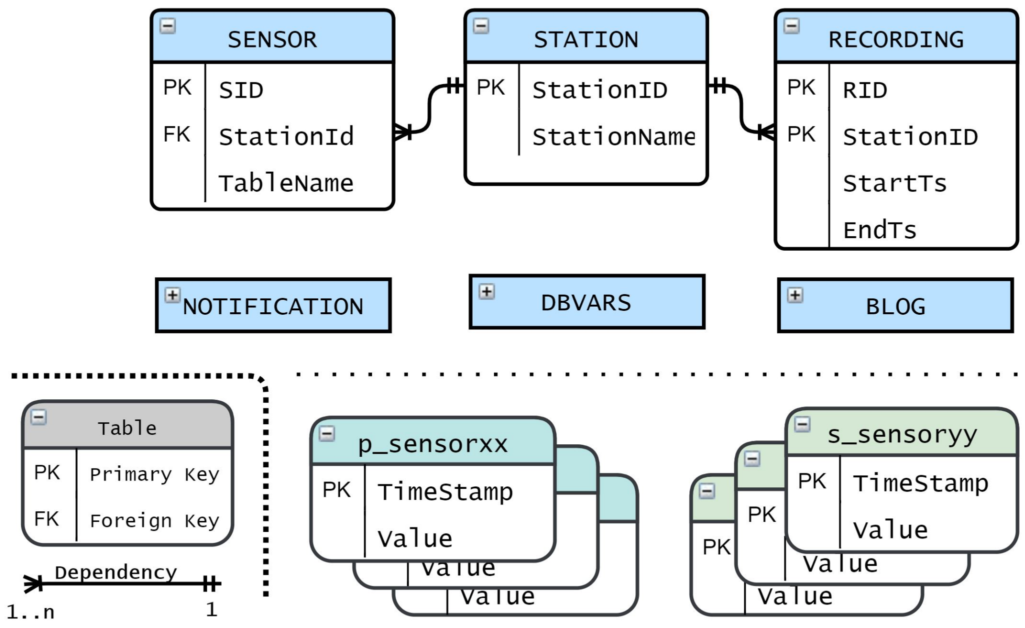

Figure 2.13: Concept of a database to store sensor data.

Figure 2.13 shows the chosen data model. The data schema is divided into two parts, a relational and normalized meta-data section (blue colored tables) and a decentralized non-relational sensor data storage section. As introduced in the previous section, different tables exist for primary and secondary data. The light green colored tables with a `p_` prefix show examples of primary data tables, whereas a prefix `s_` indicates secondary data tables. The meta-data section includes information about the sensors, the station, and information about recordings. A recording is essentially an interval with a start- and an end-timestamp where measurements happened in between. Further tables exist, which are necessary for notification purposes of the postprocessing or other data storage relevant mechanisms. A blog table lists every significant event that happened on a station, e.g., re-calibration of sensors, critical voltage supply, or other station state information.

A thin wrapper over the data storage solution has to be implemented on both, station and station server to fulfill the data storage functionality requirements of the data storage components. It provides an abstraction layer and allows for a unified interface on station and station server despite the used technology. The following three interfaces have to be abstracted from the used technology.

- **Insertion** – Single element- or bulk insertion on a single sensor data table has to be available.
- **Selection** – Selection of a single element or an interval-based group of elements for a single sensor data table has to exist.
- **Notification** – A conceptually equal interface for the postprocessing entities has to exist.

With the use of a SQL solution on the station and the station server, presentation processes or other components that need access to the measured data can access the database directly without the need of a wrapped interface.

Data replication between the station and station server is the last part that has to be considered for the storage component. In cases of deliberate or unintentional connection loss, the station server storage component must be synchronized. The regular data flow from the station to the station server happens over GAPP. Behind the GAPP observer on the server is the data storage component, which wraps the database. The wrapper keeps track of the inserted data. It knows when a connection loss has happened, which may indicate an inconsistent database. In such a scenario, the database wrapper can query the station over a second protocol to get the missing data. The protocol allows querying single sensor tables over a specified interval. The implementation of this protocol was not planned during the concept creation and is also not planned for the thesis. For the sake of documentation, this interface is preliminarily called **G**eneral **A**coustics **T**able **P**rotocol (GATP).

To complete the concept of the data storage functionality, some remarks had to be made for the storage component on the station server and the datacenter. First, due to the station servers processing resources, the need for a custom data storage solution does not exist. If the currently used PostgreSQL [63] solution proves to be not powerful enough, sophisticated NoSQL solution exist that are optimized for write heavy data, e.g., Apache Cassandra [3], Riak TS [67], and others. Such solutions could scale horizontally if needed and could replace a SQL-based solution. Second, the data center is intended to serve as a backup and centralization of measured data at various measurement sites. No processing is performed on a datacenter. Therefore, a read-only database replication to the data center in a specified interval suffices for this purpose. The functionality can be reached by using replication tools of the used database technology.

The next section is concerned with the data presentation to customers that are enabled by the use of structured data storage components throughout the GA environment.

2.3.5 Data Presentation

The last step in a sensor data management workflow is the presentation of the measured data to the end-user. The inclusion of a Linux DHU with a structured data storage component into the products enables advanced presentation solutions. Functionality, like the use of a small webserver on a station to visualize current data, could not be provided with the current hardware generation. With the use of a Linux system, the possibilities to present the measured data to the customers are significantly enhanced. A concrete design of such presentation components was not the focus of the concept. After a core system has been developed for production, additional presentation requirements by customers can be implemented. Therefore, rather than designing presentation solutions for the workflow, the new options to implement such solutions are listed.

- **Network Stack** – Through the use of a full Linux OS, an implemented and well-tested network stack is available. Therefore, the focus can be on application-, and high-level protocol development instead of rebuilding lower level networking layers.

- **File System** – The use of a file system makes not only data logging to files simpler. Configuration of the stations can be extracted from the application code, e.g., into an Extensible Markup Language (XML) file, which reduces the need for different binaries for different configurations. Persistent database files can be used with a filesystem, whereas persistence solutions have to be found on systems without a filesystem.

- **Structured Data Access** – The use of a structured data storage component to store the measured sensor data on the station serves as a foundation for most advanced presentation functionality. Simple interval-based access to previously measured data, with optional aggregation, can be used for live visualizations on a display or a web server. The creation of daily or monthly reports also benefits from the availability of such a storage solution.

- **High-level Scripting** – Various high-level programming- and scripting languages can be used on the Linux OS. Those tools enable quick development of new functionality that was not possible before. Regular tasks, e.g., daily email status notifications can be implemented reasonably simple. Additional functionality for a specific project can be added to the system without the need to modify the core station applications as an additional process.

All the presented points, to enable the implementation of advanced functionality, exist based on the decision to include a Linux driven DHU. The added presentation functionality comes with the cost of losing absolute real-time characteristics of the data, as it has to be buffered between RTU and DHU. In practice, the real-time characteristic to the millisecond was not an important criterion. Critical was the correct time stamping of the data and an accurate triggering of the measurements. Those tasks are performed by the RTU and are not influenced by the inclusion of a DHU.

Not mentioned up to now in this section are the presentation solutions on the station server and the database. Compared to the current workflow at GA, presentation of the data on the station is new. Solutions for the station server or a database already exist (LOG_aLevel software) or can be developed if needed without architectural restrictions. The use of a similar structured data storage component on the station server and the station allows future implementation of presentation solutions for both locations.

Chapter 3

Related Work

This chapter presents the related work on which this thesis builds. First, essential terms used in this thesis are described. Later on, general related information on frameworks to develop component-based software is shown. Afterwards, related work on component-based application solutions in the embedded world is introduced. In the same context, related work at GA is presented, which influenced the design of this thesis. Finally, Section 3.5 provides background information to embedded C++ development, which is used for the implementation part of this thesis.

3.1 Introduction

Chapter 2 introduced the current situation at GA and provided background information. The sensor data management workflow concept, shown in Section 2.3, structured the needed functionality into sub-functionalities. After the implementation of prototypes for this functionality, it was concluded that a monolithic object-oriented (OO) approach does not suffice to project the proposed flexibility and modularity into the software. The search for a more modular design approach led to this thesis.

In a classic OO programming approach, a *class* is a collection of properties and methods [23]. Instances of such a class, called *objects*, build the smallest composition unit of an OO application. Programming with objects enables the use of design principles like encapsulation and polymorphism [19]. The design suggestions presented by those principles simplify well-structured development. Despite that, each OO application has to solve the same problems. The objects, more often a swarm of objects, have to be **initialized** in some way. Afterwards, the **lifetime** of the objects has to be controlled. In most applications, the objects have **dependencies** between each other, which must be established and managed by some instance. Finally, the **persistence** of objects has to be considered. All of these common obstacles have to be solved in the development of an application. Besides classes, programming languages provide additional facilities to simplify the handling of those problems. Interfaces are used to decouple dependencies. Depending on the programming language, a garbage collector controls the lifetime of the objects. Even with these facilities, the monolithic design of well-structured and flexible applications remains difficult. Whenever the core logic of an application has to be adapted, the surrounding application management infrastructure has to be changed as well. When a software reaches a critical size, the needed effort to design the surrounding infrastructure exceeds the effort needed to design the core logic. To remain efficient, such software is typically split into smaller, self-contained parts. These parts can be maintained independently from other parts or the main application. The application gains **modularity** and remains maintainable. Modularity is the degree to which a system or computer program is composed of discrete components, such that a change to one component has minimal impact on other components [27]. Prof. Bertrand Meyer [38] describes five requirements of modularity as follows:

- **Decomposability** – "A software construction method satisfies Modular Decomposability if it helps in the task of decomposing a software problem into a small number of less complex sub-problems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them."
- **Composability** – "A method satisfies Modular Composability if it favors the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed."
- **Understandability** – "A method favors Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others."
- **Continuity** – "A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules."
- **Protection** – "A method satisfies Modular Protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules."

Various terms exist to describe modularity. Most commonly, the terms *module* and *component* are used. Other terms that are used in the same context are *assembly* or *package*. Clear definitions to differentiate the terms do not exist. The understanding of both notions is context depending. All terms are closely related and mean the structuring of software into smaller pieces. This thesis uses module and component interchangeably.

A change from a monolithic to a modular OO application design solves parts of the problems of OO development, which were introduced above. Typically, the separate modules solve dependency chaos and control their lifetime independent form the whole application. Nevertheless, a development focus only on independent components does not solve all problems. The self-contained group of modules has again to be "glued" together, similar as the objects have to be managed in an OO oriented approach. Bruce Wallace states that components without this glue are useless [76]. With a sufficient amount of modules, the same problems emerge as with the objects in a monolithic development process. A solution is the use of module or component frameworks. Wallace highlights the importance of such frameworks as glue for modular applications.

Component frameworks provide a middleware layer, which simplifies the composition of components into an application. They provide a unified way to create, connect, and manage the components. Various frameworks exist. Section 3.2 introduces classical frameworks, which simplify the creation of modular software. More specialized component frameworks, aimed for embedded development, are introduced in Section 3.3. Finally, the solutions at GA to build component-based applications are shown in Section 3.4. All presented frameworks influenced the design of a custom module framework in this thesis, which targets the requirements of the sensor data management workflow shown in Section 2.3.

3.2 Development Frameworks for Modular Software

As shown in Section 3.1, component frameworks are crucial to develop modular software. A multitude of them exists. The Wikipedia page to component-based programming [77] has a big list of frameworks. This section presents three of the most common component ones. Section 3.2.1 introduces the **C**ommon **O**bject **R**equest **B**roker **A**rchitecture (CORBA) [14]. Section 3.2.2 shows the **C**ommon **O**bject **M**odel (COM) from Microsoft [45]. Finally, Section 3.2.3 introduces Sun Microsystem's [70] JavaBeans and Enterprise JavaBeans. A recapitulation over the presented frameworks is made in Section 3.3, where the usefulness of the frameworks for embedded systems is evaluated.

3.2.1 CORBA

CORBA was created by the Object Management Group (OMG) and is an open standard for application interoperability. The OMG is a group of over 400 software vendors and object technology user companies [55]. Simplified, CORBA allows applications to communicate despite different execution locations or application creators. CORBA, therefore, handles the component interoperability [8]. CORBA uses an Interface Definition Language (IDL), which presents interfaces of components to the outer system. CORBA components communicate only through these interfaces. The IDL is mapped to a wide range of programming languages.

A core part of a CORBA system is the Object Request Broker (ORB). It is a middleware layer that establishes the connection between client and server. A client can invoke a method on the server, whose location is abstracted by the ORB. The ORB intercepts the method call, finds the needed object, invokes the method, and returns the result to the client. The client does not know where the object is located, on which OS it is running, which programming language was used, or any other aspects that are not related to the IDL. Therefore, the ORB introduces interoperability between applications in heterogeneous distributed environments.

CORBA is widely used in OO based distributed systems [80]. Even though it targets mainly desktop- and server applications, there exist component-based embedded systems that are built on CORBA specification. The OMG defined two standards for embedded applications: *Minimum CORBA* and *Real-Time CORBA*. Minimum CORBA defines a fully interoperable subset of CORBA. It is appropriate for applications with limited resources. Real-time CORBA extends CORBA so that it can be used in deterministic applications.

3.2.2 COM

Microsoft introduced COM 1993 as a general architecture for component-based software [8]. Microsoft states, that "COM is a platform-independent, distributed, object-oriented system for creating binary software components that can interact" [46]. COM builds the foundation higher-level software services like those provided by Microsoft's OLE (compound documents) [47] and ActiveX (Internet-enabled components) [48].

COM defines the fundamental concepts how the components interact. It prescribes a binary standard for function calling between components. COM gives a provision how to group functions into interfaces in a strongly-typed way. A base interface (`IUnknown`) allows components to discover the interfaces that were implemented by other components. Additionally, the base interface provides reference counting facilities, which allow components to track their lifetime and delete themselves when necessary. A COM component can implement one or more interfaces. An entity that implements multiple COM interfaces is called a *COM class*. The interaction between a COM client with another COM component only happens over an interface pointer. To identify the components and their interfaces uniquely, COM uses globally unique identifiers (GUIDs), which are guaranteed to be unique across space and time. Finally, COM provides a component loader that creates component instances from a deployment. [45]

COM depends on a number of parts that need to work together to created component-based applications. The host system has to provide a COM-specification conform runtime environment. A registry that keeps track where the components are installed needs to be available. So does a service control manager, which locates the components and connects servers with the clients. Windows provides these parts, in contrary to other OS. Therefore, it is the main platform for the COM technology. Any programming language can create COM components, as long as the language can create structures of pointers, and call functions through pointers. OO based languages, like C++, provide programming mechanisms that simplify the implementation of COM objects. In recent years, COM has been superseded by the .NET technology [49], some of it is even deprecated in favor of .NET.

An extension to COM is distributed COM (DCOM). It enables COM components to communicate directly and secure over a network. It was designed to be used with various network transport protocols, including HTTP. DCOM replaces local inter-process communication with remote communication over the network. The change of physical connection is transparent to the components, neither client nor server is aware of it. The use of COM in embedded applications is discussed in [37]. Other literature could not be found on that subject, which indicates a rather low usage of COM in that field.

3.2.3 JavaBeans

Sun Microsystem introduced a Java-based component model [70], which consists of two parts. The **JavaBeans** are used for client-sided component development. Enterprise JavaBeans (EJB) are the counterpart for server-side component development. The used Java platform offers portability due to bytecode usage. Security is given through the concept of trusted and untrusted Java applets. Therefore, it provides the technology that enables the development of embedded enterprise applications.

Three types of components exist for EJB's: EntityBeans, SessionBeans, and MessageDriven-Beans. All beans are deployed over an EJB container. The container manages the components at runtime, which includes state handling (start, stop, pause) and the handling of performance, security, and reliability. EJB's are tightly related to the Java programming language, which requires a Java virtual machine (JVM) to execute the code. Therefore, EJB's offer high portability at the cost of a needed JVM, which restricts the embedded use cases. For example, real-time use cases cannot be served with EJB because of the needed virtual environment.

3.3 Module Frameworks in Embedded Environments

Section 3.2 has introduced three frameworks that are commonly used to create modular software. Typically, these frameworks are used to build huge, possibly spatial distributed applications. Even though the frameworks have subsets, which target the embedded development, their primary use case targets desktop- and server development.

From GA's view, the requirement for distributed module frameworks was not necessary. The workflow concept of Section 2.3 imposed modularity, but did not require location overlapping applications. Therefore, more suitable frameworks for embedded modular development were searched.

3.3.1 Open Platform for Robotic Service (OPRoS)

A component framework, proposed by Jang et al., matches the expectations of GA best. Supported by the Ministry of Knowledge Economy of Korea, they created OPRoS, a platform for network-based intelligent robots [34]. OPRoS aims to simplify the development of sophisticated robot software by introducing a robot software component model.

An OPRoS application is contained out of components. Two types of components exist, an atomic component and a composite component. The composite component contains a group of atomic components and presents itself as a single component to the outside. An atomic component supports three mechanisms to interact with other components, a remote procedure mechanism, a data flow mechanism, and an event mechanism. A combination of components either type builds an OPRoS system application.

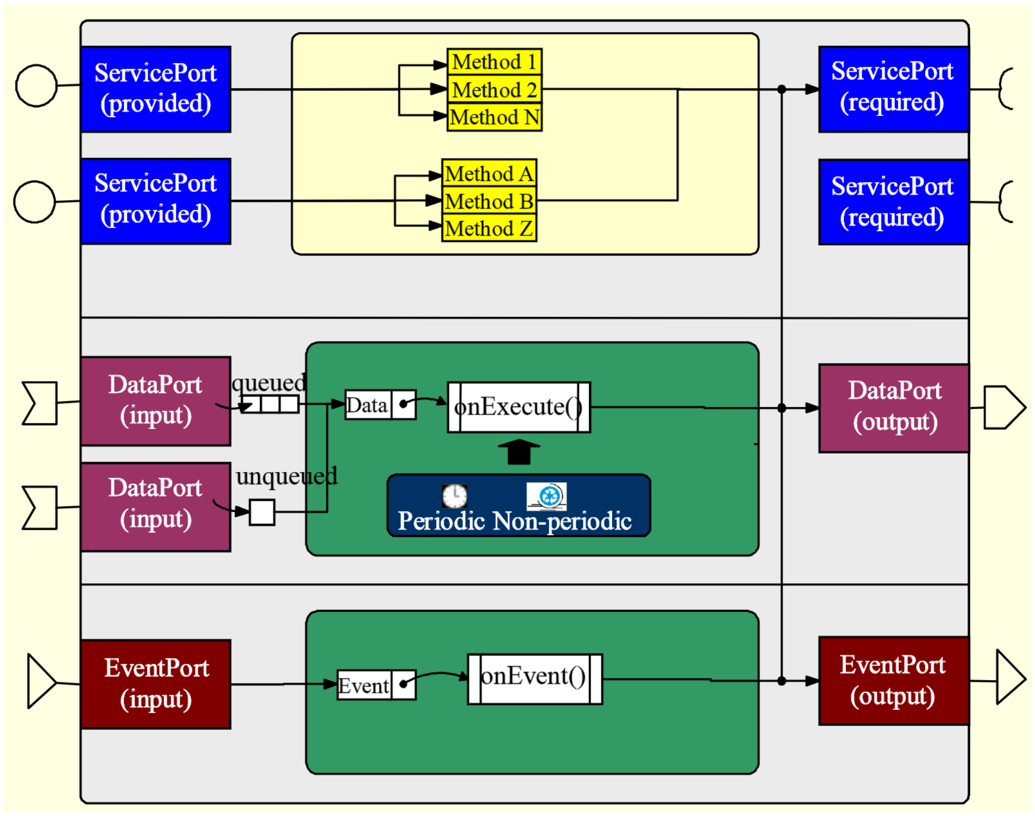

Figure 3.1: OPRoS atomic component model

Figure 3.1 shows a high-level model of the atomic component from OPRoS. Song et al. abstracted the interfaces to the components and called them **ports**. Three types of ports exist:

Service ports, which expose the required and provided method interfaces,

Data ports to pass data between components, and

Event ports to transmit events.

A provided service port allows other components to execute one of a set of available methods at the port. A required service port is a proxy, which let the component call methods of another connected component. Data ports are either for data input or data output. Both, input- and output port of two connected components must have the same signature to enable data exchange. A data port may be queued or unqueued. The data is then handled in the `onExecute()` method of a function. The event ports are similar to the data ports, given that they both transfer some structured data. In contrast to data ports, events on an event port are always processed immediately. Ports for data or event transfer do not block when invoked, whereas service ports support both blocking and non-blocking calls.

The execution of OPRoS components can happen in three modes: periodic, non-periodic, or passive. The `onExecute()` method of components in periodic mode is called repeatedly in a defined interval. The non-periodic mode of components is used whenever the duration of an `onExecute()` call is long or unpredictable. In that case, a dedicated thread is used to run the component iteratively until it is destructed. Components in passive mode neither have an own thread or an `onExecute()` callback. They are only active when triggered by an event or a method invocation from other components.

Every custom component in the OPRoS framework inherits from a base-class called `component`. This base class inherits a set of interfaces that abstract lifetime management, port management, and property handling of the components. The created components are held in a container object. The container has an executer object, which handles the execution of the components. The available components are registered to the executor by the container. Each component runs through a series of states during its lifetime. Every time when a component transits to a new state, an appropriate callback method is called. This lifetime management allows error handling and a recovering from errors. Composite components are used to incorporate other components. They can be composed out of either atomic or other composite components. The composite components abstract the functionality of its inner components and presents only a single interface to the outside. Calls to the interface of the composite component may be redirected to inner components. The characteristics of a component are described in an XML file, which is called the 'component profile'. It stores information about port types, execution semantics or properties. The XML file is interpreted by the component execution engine, which manages and executes the components. It takes the responsibility from the developers to handle problems like thread and state management or resource handling.

OPRoS was designed with the users of the framework in mind. The users would be developers that want to build robotic software based on components. Therefore, the creators of the framework implemented a set of development tools that simplify not only the implementation of components, but also the composition of components into a robotic application. Two plug-ins for the Eclipse IDE [20] were built. The *component authoring tool* let the users specify port interfaces, callback functions, and the component profile. The tool produces C++ skeleton code for the selected component parts. The skeleton can then be extended by user specific code. The component binary is compiled into a shared library. The library can be used by their second tool, the *component composer*. The component composer allows the creation of robot applications, by composing components. The components imported from the authoring tool are stored in a local repository. The user of the component composer can graphically compose applications by using drag and drop of components in the main diagram, and connect the different ports. Further on, it allows the creation of composite components. Finally, the application is deployed to a component execution engine on a robot over the network.

The OPRoS framework, as described in the last few paragraphs, solves many of the requirements targets as shown in Section 4.1.1. That section also shows that OPRoS lacks some of the flexibility requirements proposed by GA. Further on, the level of abstraction provided by OPRoS and its development tools is not needed for the framework at GA. Nevertheless, the OPRoS framework had significant influence on the design process of the framework developed in this thesis, as shown in Section 4.2.

3.3.2 Compound Object Model

A publication from Stepan Orlov and Natalia Melnikova [56] does not introduce a component or module framework, but instead an object model to develop scalable systems in C++. They introduce a model that allows the design of software packages with many components to achieve a high level of code reuse. At the time this thesis was written, the implementation of the model was closed source, albeit with the intent to open source it after some refactoring work. Therefore, their paper was the only available information about the model. It was written at a theoretical level and did not go into implementation specific details. A description of the theoretical reasoning process is left out and can be read in their paper ([56]), as it would exceed the scope of this thesis. Only the core design decisions of their model are introduced in the next few paragraphs.

Orlov and Melnikova introduced the concept of a *compound object.* it is composed out of one or more components that are connected to each other in a tree-like structure. Figure 3.2 shows such a tree. The root is called a *primary object.* All other components are called *tear-off's.* An instance of a C++ class that follows the following conventions is called a **component**: The class must inherit from either the primary object class or from a tear-off class. The component class must also declare a numeric identifier, which is unique for the entire system. Further on, component classes typically implement interfaces. The compound model supports a set of interfaces, which are C++ classes with pure virtual- or inline methods. All interfaces inherit from the same base class and have a unique identifier. Tear-off components vary in their lifetime. They may be loaded together with the primary object, or created on demand when an interface pointer is requested. Further on, components implement a form of reference counter, which is a non-negative integer, incremented whenever the user needs a component. When the counter reaches zero, the component is deleted from the memory. The reference counting is made possible by using smart pointers.

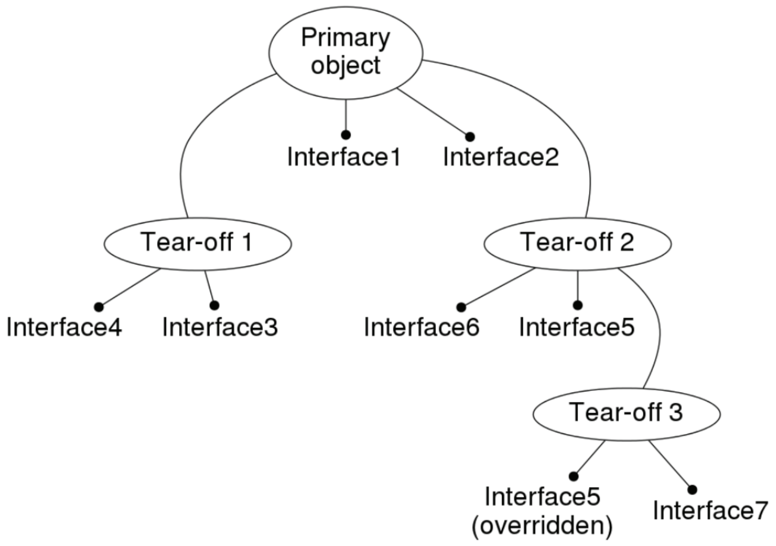

Figure 3.2: A compound object with interfaces.

The object model specifies a configuration of all modules, components in each module, as well as interfaces supported by each component in a file. The configuration file additionally contains the configuration of the compound objects, which states the tree of tear-offs for each primary object. After the object model is loaded, the objects can be created with the help of an *object factory*. When a module is loaded, it performs **initialization of static variables** at which stage all components of a module are registered in the class registry. Afterwards, the factory is able to create components.

The object model uses three interfaces to create communication between the components. A *message receiver* is a component-wide interface that allows exposing message handler methods. Those methods are regular methods of components from a compound object. A *message target* interface allows compound objects to receive messages, which are forwarded to receivers in the compound that are interested. The *message delivery* interface allows to send messages and to subscribe to targets for certain messages.

The introduced characteristics of the object model show the core functionalities that influenced the design of the framework in this thesis. Other mechanisms on how to handle document data or how to tackle serialization issues are present in the object model, but are less relevant in the context of this thesis. To summarize this section, it can be said that the compound object model targets a different goal than OPRoS. Its purpose is to enable scalable component-based software written in C++ of any kind rather than focus on problems of embedded robotic application like execution management, state management or event handling. Nonetheless, both approaches were considered in the design of the module framework in this thesis, shown in Section 4.2. The next section introduces the component framework that is currently in use at GA for graphical applications.

3.4 Component Frameworks at GA

Section 2.1 introduced the current situation at GA. It was shown that, until now, most data processing and presentation functionality was covered by modular and configurable GUI applications. The foundation of these applications is a custom module framework, called `ModKit`. ModKit was developed by Jan Schirrmacher to counter the ever-changing customer requirements. The framework is developed in Delphi and proved crucial to tackling the requirements.

The foundation of the ModKit framework is built by two characteristics. First, the modules are organized in a tree, similar to the compound object model introduced in Section 3.3.2. A running ModKit application has a tree of modules similar to a Document Object Model (DOM). The second characteristic is a meta-programming of the application with the help of XML. It is possible to describe data flows, elements in the UI and overall architecture structure in a single XML file. Therefore, the application can be adapted by editing the XML file and without the need for recompilation. It is even possible to load or unload parts of the module tree, and replace it with different modules based on another XML file at runtime. This approach has the following benefits:

- **Object Relation -** The modules hang in an object tree, similar to an XML DOM. Therefore, a module knows its parent and its children. Path iteration allows a module to find any of the other modules in the tree.
- **Serializability -** The object tree can be described in an XML document. With the help of an XML library, it is possible to de-serialize or serialize the complete tree or sub trees at runtime.
- **Runtime Flexibility -** A part of the application could be reloaded based on a different XML file, which could fundamentally change the internal structure of the modules without the need of recompilation or even a restart.
- **Error Reporting -** Controlled phases of XML loading, object creation, and object instantiation allow a better error reporting. Rollbacks to previous phases and distinctive error messages at each phase allow a quicker error localization.

The loading phase of a ModKit application not only considers the XML file, it considers an INI file and registry entries in the process. Further on, the framework is used in a model view controller (MVC) GUI development and relies on signal-slot mechanisms to pass events around. Recent requirements to store and load different configurations of the GUI added more functionality to the framework. The XML file can now contain template code, which is expanded at runtime into a part of the configuration depending on the loaded profile. ModKit uses high-level language features from Delphi to implement the runtime flexibility. Meta-classes that provide runtime information over a class can serve as a feature, which is not available in, e.g., C++. The latest beta version of ModKit introduced interfaces and interfaced implementation delegation to attach functionality to modules. Before that, the problem had to be solved by inheritance, which was limited due to the single inheritance model of Delphi.

The ModKit framework had the most significant conceptual impact on the design of the software module framework in Section 4.2 compared to the previous sections. Nevertheless, the context of this thesis' framework differs, as it is not about GUI's and is focused only on business logic implementation. It must be portable and run at least on Linux, which is not a requirement of ModKit.

3.5 Embedded C++ Development

The framework and the prototyping done in this thesis will be implemented in modern C++. The decision was set by GA in order to create portable and efficient code. An internal coding style guide will be used, which is oriented after current programming habits at GA and Herb Sutter's coding standard [71]. Scott Meyers 'Effective Modern C++' book [39] is taken as a reference on how to use the modern C++ features in a correct way. The knowledge of C++ at GA is based on the older C++03 standard, which has evolved significantly in the last decade.

Embedded development of the product firmware at GA was currently done in C. Therefore, information had to be gathered, which describes how C++ can be used efficiently in an embedded environment. For newer hardware generations, the intention is to move to C++ for all firmware written at GA. In that sense, the prototyping done in this thesis serves as a pioneer in order to gather experience in the embedded C++ field. It has to be said, that the prototyping will be done on the low-power Linux driven ARM CPU. It provides enough resources, strict housekeeping of the resources is not needed. Nevertheless, it is a direction into the embedded world. Scott Meyers teaches seminars on modern C++ [40] and embedded C++ [41], the lecture notes from the embedded C++ seminar serve as guidelines what can be done, and what shouldn't be done in embedded C++ programming.

The slides provide insights on C++ feature implementations like vtables or offer information about features that are 'no-cost' compared to C. Such features include:

· All C functionality, classes, namespaces

· Static functions and non-virtual member functions

· Function and operator overloading

· Constructors, single inheritance, virtual inheritance

On the contrary, temporary objects and templates can have a huge impact on embedded C++ programs.

By implementing the software module framework, some principles of embedded programming had to be sacrificed in order to achieve the required level of runtime flexibility. Further decisions on the C++ programming were taken in Section 5.

Chapter 4

Design Decisions

This chapter describes the design processes of this thesis. First, the requirements for the module framework and the LOG_aLevel 2.0 prototype are shown. Later, the design of the software module framework is described in detail. Finally, the design decisions taken for the prototype implementation of the LOG_aLevel 2.0 are introduced.

4.1 Requirements

Chapter 3 introduced the related work on which the work in this thesis builds on. A range of frameworks to create component-based applications were presented. This section introduces the requirements of a component framework in the context of GA and their sensor data management workflow concept shown in Section 2.3. Further on, this section introduces the requirements of the LOG_aLevel prototype, which will be based on the software module framework. The requirements serve as a foundation for the design decisions taken in Section 4.2 and 4.3.

4.1.1 Requirements: Module Framework

All frameworks introduced in Section 3, target the development of component-based software. Each one tries to cover their specific requirements. These requirements differ in various degrees from the requirement of GA for a framework to construct embedded and modular applications. For example, the 'big' frameworks introduced in Section 3.2, like CORBA or COM, are focused on distributed solutions in bigger environments. CORBA even needs a dedicated middleware layer (the ORB) to accomplish object communication. The OPRoS framework and the compound object approach, introduced in Section 3.3, came closer to what GA envisioned. Both frameworks influenced the framework of this thesis. Nonetheless, their target on usability to create new components and the high abstraction of the component model was not needed for the framework at GA. The compound object model [56] provided valuable input on the aggregation of interfaces to composite objects but lacked the broader view to create module-based applications in the embedded field. Finally, the ModKit framework from GA itself was developed with GUI applications in mind and works only on the Windows platform. However, the XML solution to configure the application was one of the core factors that lead to the design shown in Section 4.2. To be able to design a framework that is appropriate for the use cases at GA, the requirements have to be known. The next paragraphs introduce the requirements from the related work, the requirements implicated by the sensor data management workflow concept, and requirements based on experience from GA.

The description of the GA ecosystem in Section 2 has shown that **flexibility** is one of the core requirements for the hardware as well as for the software. Two different ways of the flexibility requirement can be seen. On one end, the LOG_aLevel system is configurable based on customer wishes. Thus, the available software functionality should be simply adaptable to different system configurations. On the other end, the software has to be flexible enough to allow the integration of entirely new functionality requirements without the need to re-develop the already available solutions. The introduction of a module framework should provide this flexibility and take the responsibility of the software developer. Further on, the framework should act as a middleware layer. It should free the developer of modules from problems that appear in application development (see Section 3.1). The following core requirements for the software module framework at GA can be stated:

R01 – The framework should provide modularity, which allows better extensibility by reducing coupling and dependencies.

R02 – The framework should provide meta-programming functionality, which allows applications to be configured and adapted without the need of recompilation.

R03 – The framework should act as middleware for the application development. It should handle...

- · instantiation,
- · lifetime management,
- · dependency management,
- · persistence problems, and
- · state control.

Aside from these core requirements, the design of the framework has to account for the limited resources available at GA. This includes reasoning on what the framework tries to accomplish and what not. The core goal of the framework is assistance in the creation of applications in the GA environment. Therefore, only the software developers at GA have to work with it. This implies that there is no need to create supportive tools, e.g., GUIs to generate and compose components as it was done for OPRoS. As one or two developers will implement all modules that use the framework, high-level abstraction and introspection mechanisms are not needed for a single module. Therefore, (**R04**) the framework should focus on the core functionality and prioritize implementability.

4.1.2 Requirements: LOG_aLevel Prototype

Section 4.1.1 introduced the requirements of GA for a software module framework. This section introduces the requirements of the LOG_aLevel software prototype that will be developed in a second implementation part of this thesis. In general, the prototype should follow the sensor data management workflow concept of Section 2.3 closely. It should and implement the functionality of a 'station'. Due to the occupied hardware development division of GA, parallel development of a LOG_aLevel 2.0 hardware prototype could not be accomplished. Therefore, the work surrounding the prototype software is focused on the DHU functionality, which was shown in the workflow description. The data acquisition part, including the RTU, will be ignored and abstracted.

Table 4.1 introduces a list of core functionality requirements. Those requirements were extracted from the workflow concept presented in Section 2.3. The red shaded acquisition requirements serve merely as context for the actual requirements of the PDU and the DSU, which show the required functionality to process (blue) respectively store (green) data. The LOG_aLevel application prototype has to be designed around the module framework. The workflow concept introduced the PDU and the DSU as separate units that are connected over a GAPP interface. This separation is not a critical requirement for the implementation of the prototype. It is more important to implement a working system based on the module framework that covers the required functionality in distinct modules.

The next sections present the design process. First, the design decisions and the architecture of the module framework are shown in Section 4.2. Afterwards, the design of the LOG_aLevel prototype software is shown in Section 4.3.

Table 4.1: Prototype requirements from the workflow concept.

ACQ-01	Implementation of an abstraction interface to decouple the real-time hardware part form the following workflow.
ACQ-02	Use of a Linux DHU in the station controllers.
PRO-01	The data, which is collected by the acquisition part, has to be preprocessed and brought into human understandable form.
PRO-02	An interface has to exist that allows the inclusion of advanced postprocessing algorithms into the station to generate aggregated values.
PRO-03	The interface to the postprocessing has to be compatible with both station and station server.
STO-01	Logging of raw data in a file based way is required.
STO-02	The sensor data has to be stored in a structured data storage component that provides interval based access.
STO-03	The interface to the data storage component has to be compatible with both station and station server.

4.2 Design: Module Framework

This section introduces the design of the module framework. In a first part, general design decisions, which were taken during the framework design phase are introduced and reasoned. Later, the architecture of the framework with an overview of its core components is presented.

Except for ModKit from GA, none of the introduced frameworks in Section 3 provided the needed meta-programming flexibility (R02 of Section 4.1.1). Due to the available knowledge and experience with the XML solution from ModKit at GA, it was decided to integrate a similar meta-programming solution in the software module framework design. The XML solution has been proven to work well with the data flow oriented data processing. Which is needed at GA through the handling of real-time environmental data. This knowledge of usability of the solution was a key driver to the decision to adopt this solution. Further on, an available implementation of the framework and a range of software products that build upon it provided insight to the critical parts of such a solution.

Section 3.4 shows that the XML solution is used to describe the application architecture with module dependencies and properties. This requires tight integration into the framework. Therefore, the decision had a significant impact on the design of the framework. The adoption of the XML solution implied architectural similarities to ModKit. Some ideas of the module framework in this thesis may have been taken over from ModKit, but the design and implementation were done with C++ in mind, utilizing its strengths and minimizing its weaknesses. Developed in Delphi, ModKit can rely on advanced language features for meta-programming and reflection, which simplify the implementation of the compile- and runtime flexibility. With C++ being a low-level programming language and only recently getting some generic functionality (C++11 and later), even simple tasks, as the creation of a type based on its name, comes with significant effort and the need to know every corner case of the language. Therefore, the focus was set on the core functionality, stripped down to the necessary parts. Most of the runtime introspection functionality from ModKit, which is needed for the GUI development, was omitted.

Despite ModKit, the other component-based frameworks shown in Section 3 had an impact on the framework design as well. The compound object model gave input how the module classes can be registered to the factory. The chosen implementation is shown in Section 5. OPRoS had a range of interesting functionalities. The abstraction of the ports was considered, as well as the concept of composite modules. In the end, it was decided not to implement these functionalities in favor of the more important event loop mechanism. The port abstraction may be useful for advanced introspection, but it complicates the design and is not needed. Composite modules can be built by hand without significant effort as is shown in the implementation section where sensor modules are grouped together in a `SensorContainer`. A core addition, driven by OPRoS' design, is the integration of execution modes for modules, which are handled by an execution engine. Similarly, a lifetime control instance was integrated into the framework, again influenced by OPRoS. It keeps track of the current state of the application and handles state changing events. The three heavyweight frameworks introduced in Section 3.2 helped in the understanding of modular software but did not contribute directly to the design process of the module framework.

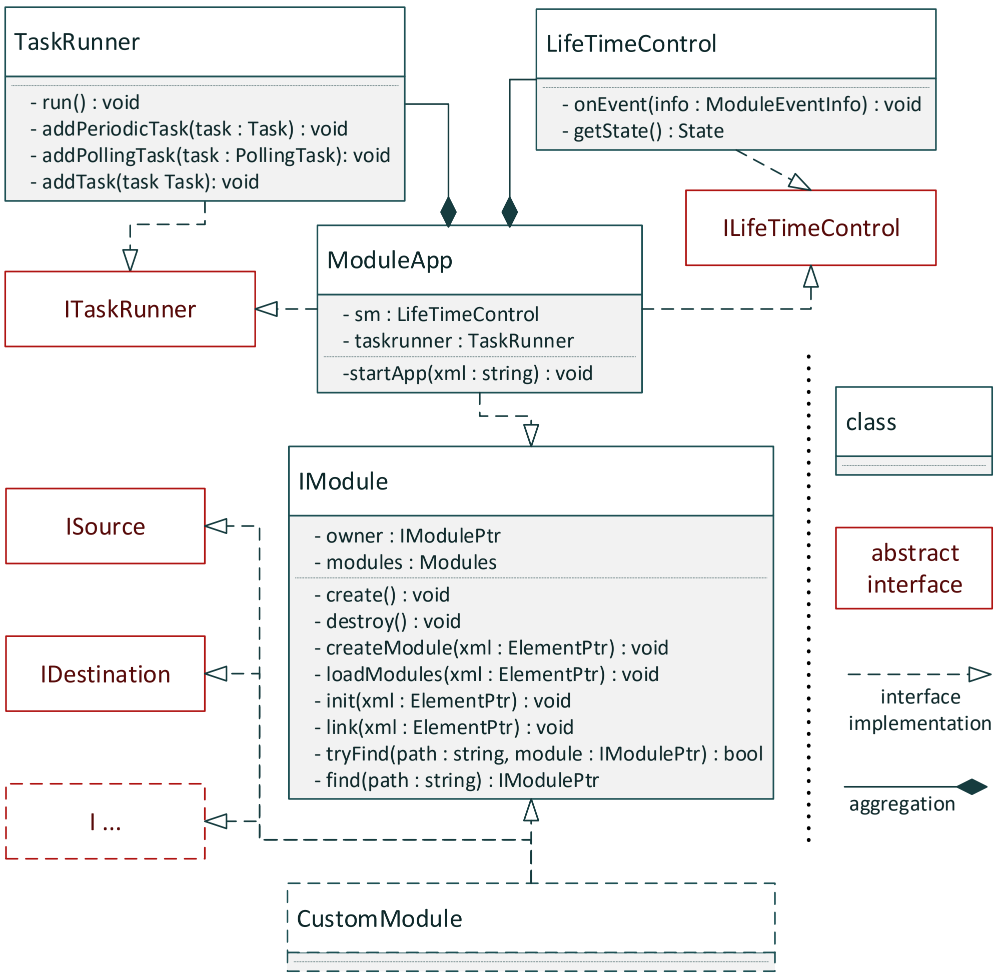
TaskRunner
- run() : void
- addPeriodicTask(task : Task) : void
- addPollingTask(task : PollingTask): void
- addTask(task Task): void
LifeTimeControl
- onEvent(info : ModuleEventInfo) : void
- getState() : State
ILifeTimeControl
ModuleApp
- sm : LifeTimeControl
- taskrunner : TaskRunner
-startApp(xml : string) : void
ITaskRunner
IModule
- owner : IModulePtr
- modules : Modules
- create() : void
- destroy() : void
- createModule(xml : ElementPtr) : void
- loadModules(xml : ElementPtr) : void
- init(xml : ElementPtr) : void
- link(xml : ElementPtr) : void
- tryFind(path : string, module : IModulePtr) : bool
- find(path : string) : IModulePtr
ISource
IDestination
I ...
CustomModule
class
abstract interface
interface implementation
aggregation

Figure 4.1: Class diagram of the module framework.

Figure 4.1 shows the core components of the module framework in an UML class diagram. The diagram is simplified and presents only the members and objects of interest for the understanding. More information about details of the architecture are given in the implementation section. The framework was designed around the concept of a module. The modules are located in a tree, where each module holds its child modules. The `IModule` class is the base class of all modules. It provides convenience functions to find specific modules in a module tree. Virtual functions to initialize, load, and unload a module are meant to be overwritten by derived modules. A custom module always implements `IModule` and can add functionality by implementing from a range of interfaces. For example, `ISource` or `ISink` should be used to unify data flow between modules.

Each application in development needs a controlling entity that coordinates the execution of the application. In a simple solution, the `main()` routine of a C++ program is sufficient for this task. A special module `ModuleApp` was added to the framework to encapsulate the controlling functionality. The `ModuleApp` is intended as the root of the module tree. It provides procedures for the start, execution, and termination of an application. The `ModuleApp` aggregates two classes. A lifetime control manager and a task runner. The

`LifeTimeControl` contains a state machine, which shows the state of an application and handles transitions based on events. The `TaskRunner` enables different execution modes for the modules. It provides an event loop, which serves as primary execution loop. Both components are crucial for the controlling of modular applications. Thus, they are presented in more depth further down. Not shown in Figure 4.1 is the factory mechanism to register new module classes. The module factory presents a simple way to statically register new module classes without the need to change any code of the framework and recompile the library. The implementation section (5.2) introduces the used solution with more technical details.

That said, the design choices described in the last two paragraphs were taken during the implementation phase and have been proven to work as needed. Other application structures could have been chosen. For example, a trunk module, which contains the module tree with the root and includes the needed application controlling functionality over aggregation. Due to the modularity, the actual structure is not too important. It would be possible to refactor the framework to use a different structure without huge efforts. The presented structure was chosen, as it worked best during the development.

Before the details of the execution engine and the application state manager are introduced, the XML solution is explained. Listing 1 shows an exemplary configuration of a module framework application. It must always contain a `ModuleApp` node. This node may contain properties to configure the `TaskRunner` or the lifetime controller. Inside, the configurations of the modules are stored as sub-nodes of the `ModuleApp` node. The path to this configuration file is given to a newly created `ModuleApp`. When the `startApp()` method of the `ModuleApp` is called, the XML configuration is loaded. The configuration is then passed to the static methods of the `IModule` base class to load the module tree (`loadModules()` and `createModule()`). A callback function is given to both methods, which should be used to emit state changing events to the `LifeTimeControl` object. The rest of the XML loading process is tightly coupled to the state machine of the `ModuleApp`. It is shown in the next paragraph, where the lifetime management is described.

```xml
<? xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ModuleApp Property1="xyz">
    <DatabaseModule Class="SQLiteDatabaseModule" >
        <SensorStore Name="p_levelus" >
            <Notification Name="SLEVEL" Interval="60"/>
            <Notification Name="LEVELAVG" Interval="300"/>
        </SensorStore>
        ...
    </DatabaseModule>
    ...
</ModuleApp>
```

Listing 1: General XML configuration of a module framework application.

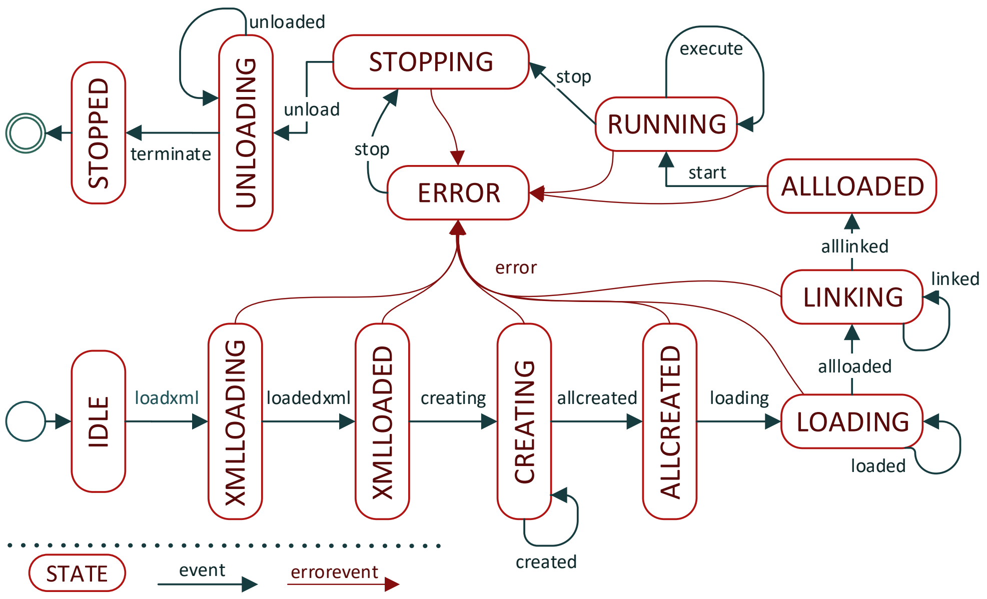

Figure 4.2: Default state diagram of a module framework application

The decision to design a dedicated `LifeTimeControl` class was influenced by the OPRoS solution, which also aggregates a lifecycle manager into their application. In contrary to OPRoS, the modules in the module framework do not contain an internal state machine. This decision was made in an attempt to keep the base `IModule` class lean. Instead, it is assumed that all modules are connected to the `LifeTimeControl` and communicate state changes over events. This connection to the `LifeTimeControl` can only happen when the modules already exist. This means that a different way of state management has to be found during the time the modules do not yet exist. There, the previously mentioned static `IModule` methods come into play.

Figure 4.2 shows a diagram of the state machine that a `ModuleApp` follows. When a `ModuleApp` instance is created, the `LifeTimeControl` is instantiated immediately. The application is then in an `IDLE` state. First, the `ModuleApp` connects itself to the `Life-TimeControl` (`LTC`) to be able to emit events. When `startApp()` is called, a `loadxml` event is emitted to the `LTC`, which transitions to the state `XMLLOADING`. A callback, which was attached to the transition as transition action, is called. This callback initiates the loading of the XML file into a XML DOM object. When the XML is loaded, and no error appears, a `xmlloaded` event is emitted, which starts a transition into the `XMLLOADED` state. The associated transition action calls the static functions to load and create the module objects from the XML DOM. Due to their static property, they are not aware of their context and are not informed about the `LTC` or event connections to it. Therefore, a callback is passed to them as argument, which is intended to be used as event emitter. First, a `creating` event is sent out to put the `LTC` into a `CREATING` state. Then, whenever a module is constructed, a `created` event is thrown. When all modules were constructed, an `allcreated` event puts the application into an `ALLCREATED` state. The construction follows a postfix depth-first traversal of the XML DOM. Besides the construction, two additional tree traversals happen. All modules are initialized in the same order. The `LTC` transitions to `LOADING` over a `loading` event. A `loaded` event is thrown for each initialized module. When all modules are loaded, the applications transitions to `LINKING` over the event `allloaded`. Afterwards, the connection between the modules are constructed in a third linking traversal. For each linked module, `linked` is emitted. For example, each module is linked to the `LTC`, which allows them to emit events directly. After all modules are loaded, a final event `allloaded` transits the `LTC` to `ALLLOADED`. The static loading methods are exited at this stage. From now on, all modules communicate directly with the `LTC`. A `start` event puts the application in a `RUNNING` state. The modules can emit `onexecution` events when they execute, if needed. A `stop` event, thrown from any module, initiates the shutdown sequence. In the `UNLOADING` state, all modules of the tree are unloaded, where each module that gets unloaded emits an `unloaded` event. After a final `terminate` event, the application goes to `STOPPED` and destroys itself. Whenever an `error` event is thrown, the application changes to an `ERROR` state. The `LTC` calls then a callback, which was offered by the `ModuleApp` module. The callback handles the error and initiates the shutdown of the application.

Besides an event sink and an internal state machine, the `LTC` has a signal where every event is outputted again. Interested modules can attach to this signal, which allows them to react on events. An example may be a log module, which logs all events. The benefits of such a lifetime control come to shine in debugging situations. Each event is associated with its creator module, the current path of the module in the tree, and other information that signals the origin of the event. Based on this, and with a history of previous events, the location of errors can be tracked accurately. Other designs of a lifetime control facility were considered, especially a version where each module contains an own state machine. Nonetheless, such a design would also need a centralized instance where the states of each module are monitored. Therefore, this design was abandoned to keep the overhead of each module low. The implementation specific details of the `LTC` are shown in Section 5.2.

Another important part of the module framework is the execution engine called TaskRunner. The OPRoS approach, with different execution modes of the modules and an execution engine, inspired the design of this functionality. The decision to include a TaskRunner tackled two situations that could be troublesome for users of the framework. First, it provides a central place to register polling functionality of modules. Second, it allows breaking up deep call stacks that could appear by chaining passive modules that react on signals. Without a place where modules can request polling, an external entity needs to know all modules that need polling, and it has to call them in a loop. This could be accomplished by using a custom derived `ModuleApp` with a loop in the `run()` method, by the cost of introducing dependencies and therefore tight coupling. The TaskRunner solves this overhead and allows each module to register itself for polling. Deep function call stacks are the second problem that can be circumvented by using the TaskRunner. It allows adding one-time tasks to the loop that is executing the polling. Therefore, modules that would signal other modules in a call chain could pass the signaling as a task to the TaskRunner, and break the call chain. These breaks in the call chains reduce the threat of long blocking calls and distribute the processing power more evenly over all modules.

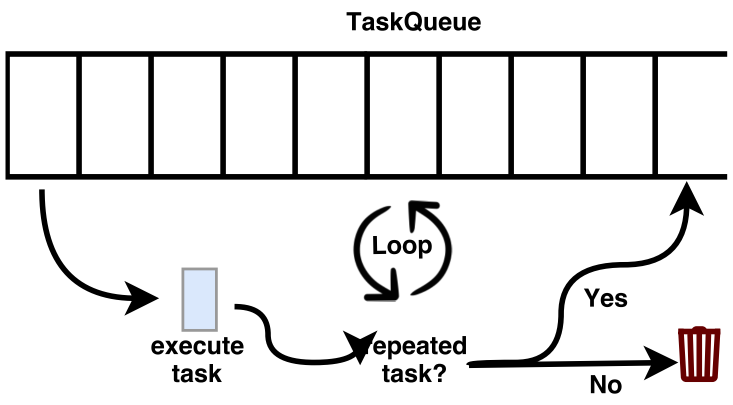

Figure 4.3: Task queue with task loop functionality.

Figure 4.3 shows an overview of the core functionality of the TaskRunner. A core piece is the task queue, in the figure shown as a horizontal ladder. It contains callback functions that should be called when a task is executed. A task loop, called with `run()` iterates over the current items in the queue. Each task is popped and executed. If it is a repetitive task, it is pushed back into the queue again, otherwise it is destroyed. A module can insert one-time tasks at any time. Because the insertion operations are made thread save, threaded modules can insert a task independent from the main application flow. Some modules need to be called periodically. For that case, the TaskRunner allows to add tasks with an interval. Whenever an interval has passed, it pushes the task into the queue.

The TaskRunner simplifies event-driven development and can be used to create asynchronicity in the application. Execution modes of the modules similar to OPRoS are possible. Together with the `LTC` and the XML configuration solution, the requirements of GA are satisfied. The design decisions can be summarized in the following list:

- **D01** – Use of a XML meta-programming solution to configure the modules. It is an adapted and stripped down version of the ModKit solution used by GA.
- **D02** – Use of a factory mechanism that allows the addition of new modules without the need to recompile the core library.
- **D03** – Functionality abstraction and decoupling by using a set of abstract interfaces.
- **D04** – Introduction of a lifetime control entity, which overviews the state of the application based on events emitted from the modules.
- **D05** – Addition of a task runner, that controls the polling of modules and allows asynchronous execution flows over a callback loop.

Section 5 shows the implementation of the module framework. It goes into architectural details and introduces the core functionality based on code snippets. Before that, the next section introduces the design of the LOG_aLevel 2.0 prototype, which is based on the module framework introduced above.

4.3 Design: LOG_aLevel Prototype

This section introduces the design of the LOG_aLevel 2.0 prototype software. It follows the design guidelines given by the workflow concept of Section 2.3 and should be built with the help of the module framework, which was presented in Section 4.2. The scope of this prototype implementation is confined by the requirement description in Section 4.1.2. It was shown that an emphasis was put on the required core functionality and its integration into a modular application that is driven by the module framework. Hence, other work that needs to be done for a complete LOG_aLevel, was not part of this thesis. Examples of additional work may be a deep hardware integration, the accounting of sleeping phases, or optimizing the implementation for an embedded environment.

This focus on functionality allowed some degree of freedom in the realization of the design. As is shown in Section 4.1.2, the functionality of the PDU and DSU (see Section 2.3) should be implemented. It was decided to implement the functionality of both, in a single application. The inter process communication (IPC) over GAPP between PDU and DSU was left out, to simplify design and development. Only a simple, not really performant GAPP serial driver was implemented, which was another reason for this decision as it would have meant additional effort to implement a better IPC driver.

The core design decisions for the prototype were already made during the workflow concept development and can be found in Section 2.3. The use of a module framework was not planned during the creation of the workflow concept. Despite not developed for it, the designed architecture could be ported to a modular architecture without too much effort. One can get a visual overview of the functionality by combining Figure 2.11 and 2.12 on page 20 and 21. The GAPP data that would leave the PDU over the source goes directly into the structured data storage and the GAPP source of the DSU. Listing 2 presents the XML module configuration of the prototype. It captures the intent of the modular architecture better than an additional figure. The listing is described on page 50. Before, the following list summarizes the general design decisions that were made during the workflow concept development:

· **RTU Abstraction Interface** – A chunk data (see page 17) source abstraction should make the prototype independent form the current data source.

· **Data Preprocessing** – Sensor objects shall be added, that perform the conversion of chunk data to human readable data.

· **Data Storage** – Structured data storage functionality with an interface to insert and select data should be used. Interval based access has to be included.

· **Database Design and Abstraction** – SQLite shall be used as database technology with an abstraction interface to allow different database technologies later on. A database schema as shown in Figure 2.13 should be implemented.

· **Data Postprocessing** – Interval based postprocessing functionality has to exist. A notification mechanism should be used to trigger the postprocessing.

```xml
<? xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ModuleApp>
    <!-- Interface to the raw data -->
    <ChunkDriver Class="FileChunkDriver" />

    <!-- Preprocessing Sensor container -->
    <SensorContainer Class="SensorContainer">
        <Sensor Class="LevelSensor" SID="1"
                DispatcherModule="../../ChunkDispatcher"/>
        <Sensor Class="XYZSensor" SID="99"
                DispatcherModule="../../ChunkDispatcher"/>
            ...
    </SensorContainer>

    <!-- Primary data insertion into DatabaseModule -->
    <DataInserterModule Class="DataInserter"
                        DatabaseModule="../DatabaseModule">
        <InsertionTask ID="15" Sensorstore ="p_levelrd"/>
        ...
        <InsertionTask ID="10" Sensorstore ="p_levelus" Highspeed="true"/>
    </DataInserterModule>

    <!-- Structured data storage -->
    <DatabaseModule Class="SQLiteDatabaseModule" >
        <!-- Sensor stores -->
        <SensorStore  Name="p_levelus" >
            <Notification Name="SLEVEL" Interval="60"/>
            <Notification Name="LEVELAVG" Interval="300"/>
        </SensorStore>

        <SensorStore Name="p_wind" />
        ...
    </DatabaseModule>

    <!-- Postprocessing Sensor container -->
    <SensorContainer Class="SensorContainer" Name="SecondaryGenerators">
        <Sensor Class="DefaultAVGSensor" SID="50" />
        ...
    </SensorContainer>

    <!-- Property modules that contains properties from the GAPP module -->
    <PropertyModule Class="PropertyModule" Name="SourceProps" >
        <Property ID="10" Name="L" />
        <Property ID="100" Name="XYZ" />
        <Property ID="50" Name="L.10MAVG"/>
        ...
    </PropertyModule>

    <!-- GAPP source to output data -->
    <GAPPModule Class="GAPPModule" Kind="Source" />
</ModuleApp>
```

Listing 2: XML configuration of the prototype.

Listing 2 introduces the module-based architecture of the prototype as an XML snippet that can be loaded by the module framework. To keep the code snippet at a reasonable size, recurring modules omitted by `"..."`. For the same reason, only key XML attributes are included. The module structure is similar to the theoretical component structure presented in Figure 2.11 and 2.12 of the workflow concept, albeit as single application without intermediate GAPP IPC: A driver exists, denoted as `ChunkDriver` XML element at line four, so does a group of `Sensor` nodes at line seven for the preprocessing. Contrary to the concept diagram, the sensor modules are wrapped in a `SensorContainer` module. It provides helper functionality to access the different sensor modules and hides base pointer casting. Instead of emitting the data only to the GAPP engine as in Figure 2.11, the data is also inserted into a data storage module (line 24). This insertion is shown in Figure 2.12 as an arrow, but instead from a GAPP observer, the data comes from a `DataInserter` module. A second group of sensor modules is held by a `SensorContainer` at line 36. The sensors listen on notifications and perform the postprocessing functionality to generate secondary data, which is then emitted over the GAPP engine. A graphical representation of the mechanism can be seen in Figure 2.12. Using the module framework, the sensor manager component in the figure got obsolete as each sensor module is self-responsible for the appropriate interaction with the data storage module and the source properties. At line 50, a GAPP engine can be found. The use of the module framework, therefore, a different way to manage the different application components forced certain changes compared to the component structure proposed in Figures 2.11 and 2.12. The separation of the GAPP source into a `GAPPModule` and a `PropertyModule` is one of the bigger differences. The reason behind this decision is the ubiquitous use of the properties in the business logic of most sensor modules. The separation causes a decoupling of the properties from the GAPP engine. This allows the handling of the property objects based on the requirements of the application without possible impact on the GAPP engine. Other `PropertyModule` implementations that handle properties differently are possible with this separation.

The presented modular architecture works as follows: The raw data is read by a module called `ChunkDriver`. The `FileChunkDriver` specialization was implemented to read simulated data. Later on a `RTUChunkDriver` may handle the connection to the RTU. The chunks handled by the driver are then passed to the `SensorContainer` module. It passes the chunks over an `IDispatcher` interface to the sensor modules based on the chunk ID. Each preprocessing sensor module (after line seven) connects to the `SensorContainer` and registers the needed chunk IDs for dispatching. The sensors may implement `ISource` or `ISink` interfaces, which allows them to connect to each other. The algorithms inside the sensor classes analyze the arriving chunks and, if possible, calculate a primary data sample. The according property for that sample is updated. The `PropertyModule` at line 42 is then informed over the change of a property. An `onChanged` event on the property signals to the `GAPPModule` at line 50 that a message with the changed values has to be emitted. The `GAPPModule` can receive messages to update property values from the outside, which in return signals a value change to all interested parties. Besides signaling the GAPP engine of new values, the preprocessing sensors insert the GAPP values to the `DataInserter`, who knows the mapping to the data storage tables and inserts them there. The structured storage solution is one of the core additions to the LOG_aLevel prototype, and therefore, described in more detail.

As mentioned in the requirement list above, SQLite shall be used as database technology for the implementation of the prototype and an abstraction layer should decouple the prototype implementation to allow the implementation of different technology later on. Figure 2.13 of the workflow concept presents the database schema that shall be implemented. It is two-fold, one part describes the storing of sensor data in a decentralized way, the second part is concerned with the relational representation of meta-data. The meta-data is particularly important when a workflow with multiple stations or even station servers is realized. For the prototype implementation, where a single station is considered, the importance of meta-data is not as high. Therefore, and to focus the implementation efforts, it was decided to implement only the functionality needed for the data storage and the postprocessing functionality. Advanced meta-data functionality can be added later to the prototype or as a separate component in the DHU. The same goes for the GATP interface that was introduced in Section 2.3.4, as it only needs read access to the database, it can be implemented later on as an external module.

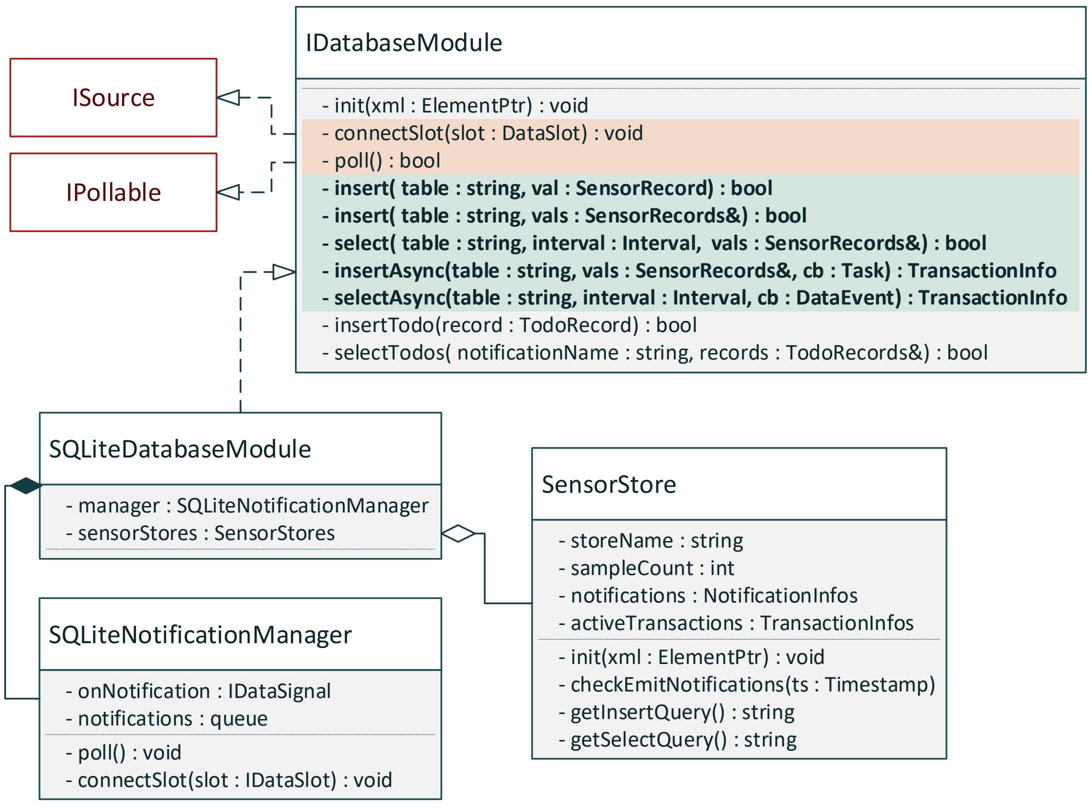
IDatabaseModule
- init(xml : ElementPtr) : void
- connectSlot(slot : DataSlot) : void
- poll() : bool
- insert(table : string, val : SensorRecord) : bool
- insert(table : string, vals : SensorRecords&) : bool
- select(table : string, interval : Interval, vals : SensorRecords&) : bool
- insertAsync(table : string, vals : SensorRecords&, cb : Task) : TransactionInfo
- selectAsync(table : string, interval : Interval, cb : DataEvent) : TransactionInfo
- insertTodo(record : TodoRecord) : bool
- selectTodos(notificationName : string, records : TodoRecords&) : bool
ISource
IPollable
SQLiteDatabaseModule
- manager : SQLiteNotificationManager
- sensorStores : SensorStores
SensorStore
- storeName : string
- sampleCount : int
- notifications : NotificationInfos
- activeTransactions : TransactionInfos
- init(xml : ElementPtr) : void
- checkEmitNotifications(ts : Timestamp)
- getInsertQuery() : string
- getSelectQuery() : string
SQLiteNotificationManager
- onNotification : IDataSignal
- notifications : queue
- poll() : void
- connectSlot(slot : IDataSlot) : void

Figure 4.4: Data storage component structure as a class diagram.

The XML module structure of the data storage module can be seen after line 24 in Listing 2. The class diagram in Figure 4.4 extends the listing and introduces the class structure of data storage functionality. On line 24, a `DatabaseModule` can be found. It is an instance of a descendant of the `IDatabaseModule` class, which specifies the postprocessing notification interface (orange) and the interface to insert and select sensor data (green). To take full advantage of the module framework design, asynchronous insertion and select operations were added. Sensor data is held in records of the type `SensorRecord`, which is a `struct` of a timestamp represented as double and a range of float values that contain the measured data. The XML description of the database module contains `SensorStore` nodes, e.g., at line 26. They specify the sensor 'table' name and the number of sample columns a data store has. The implementation of the `IDatabaseModule` has to handle the sensor store description.

A sensor store may have notifications as shown on line 27. A notification consists of a name and an interval in seconds. The notifications are used to trigger postprocessing sensors to start their work. The insertion functionality of a database module checks if a notification has to be issued. It is up to the realization of the interface to implement the notification mechanism. Different database technologies have fundamentally different ways how such a mechanism would be implemented. For the serverless SQLite, the whole mechanism has to be implemented by hand, whereas PostgreSQL already features a complete notification mechanism for external processes.

Figure 4.4 shows the `SQLiteDatabaseModule`, an implementation of the `IDatabaseModule` for SQLite. It uses helper classes to implement the interface functionalities. A `SQLiteDatabaseModule` contains a `SensorStore` object for each sensor table. The `SensorStore` class contains information over the specific table, its postprocessing notification entries, and status information over running asynchronous operations. The `NotificationManager` offers the functionality to connect a notification slot. The manager is held by the database module. A more detailed description of the notification mechanism is given in Section 5.3, where the implementation for SQLite is discussed. The `IDatabaseModule` interface was kept thin and covers the needs of the workflow described in Section 2.3.4, where the core functionality of the storage component was listed.

With this modular approach, the architecture of the prototype is very flexible to different use case scenarios. To add a new kind of sensor, be it a new physical sensor in the GA ecosystem or simply an optimized preprocessor, the addition of a single class that derives from `ISensor` is enough. Everything else can be configured over the XML file. In a productive environment, it may be possible to change the internal structure on the fly by automatically detecting the addition of physical sensors. With the current hardware generation at GA such a hardware change always comes with the modification of the firmware. The implementation of the prototype is shown in Section 5.3. There, the focus will shift from the architecture to the implementation specific details. A focus will be on the integration of SQLite as the database technology for the data storage components.

Chapter 5

Implementation

The previous section introduced the architectural design of the module framework and the LOG_aLevel 2.0 software prototype. This chapter describes the implementation details of said components. Additionally, this section introduces helper libraries, which were developed during the implementation period. The libraries provide basic functionality to simplify the development of the other components in the GA environment.

The chapter concentrates on the transmission of the core ideas of the implemented solutions to the reader. Selected implementation parts are introduced in more detail whenever it is helpful for the overall understanding. Knowledge of recent C++ versions is assumed. Various code snippets are shown in the chapter to illustrate the text. To keep the snippets short, constructors, getter, and setters, but also lengthy error handling was omitted. The snippets reflecte the state of the implementation at the this chapter was written. The final implementation may vary from the shown snippets.

5.1 `GALIB` Libraries

The application development in Delphi over the past years at GA has led to a collection of small libraries that encapsulate recurring tasks in the sensor data management field. The libraries provide additional functionality to the standard libraries distributed with Delphi. Grouped into a single library, the `GALIB`, those libraries provide the foundation for more abstracted development. No dependencies to libraries outside of the `GALIB` exist.

Throughout the implementation phase of this thesis, the low-level orientation of C++ compared to Delphi became apparent. Even simple tasks, e.g., the trimming of a string, resulted in significant code blow. With the number of such situations rising, the decision was taken to create a `GALIB_CPP` library collection. Some libraries were ported from Delphi, at least regarding functionality. Other libraries fill basic functionality needs that are already covered by the Delphi standard library. It was neither the time nor the idea to port the full `GALIB` to C++, only the required parts for the work of this thesis were implemented. Later on, the implementation of the `GALIB_CPP` can be extended and gradually adapted to the functionality of the Delphi libraries.

Albeit some libraries needed significant effort to implement and have grown big and complex, they were not the core work of this thesis. Additionally, a detailed implementation description of all the libraries would be too much for the scope of this thesis. Therefore, interested readers are referred to the source code of the libraries where the core ideas and important implementation parts are described. The following libraries were implemented throughout the work of this thesis:

- `byteFns` – A library to handle the conversion of numerical values to bytes and back. This library is particularly important for binary communication, e.g., between the RTU and the DHU.
- `propFns` – It provides the classes that are used for the GAPP communication. The GAPP engine, property classes, and various value classes are the most prominent parts of this library. The library was developed in parallel in Delphi and C++.
- `sysFns` – This library provides functionality that is too small to be grouped into a single library. Currently, only overloaded `enum class` bit shift operators are implemented, which allows the convenient use of an `enum class` as bit flags.
- `textFns` – The text functions library provides helper functions to split or trim a string. Additionally, type to string and string to type conversion was implemented for trivial types.
- `timeFns` – Timestamp management is essential for the management of sensor data. Therefore, this library provides functions to convert a timestamp between `string`, `double`, or internal `TimeStamp` representation. The library uses the C++ `<chrono>` library and sets the internally used clock with precision to specify the `TimeStamp` and `Duration` types.
- `xmlFns` – With the implementation of the module framework that relies on XML files, a XML library was needed. The Delphi `GALIB` contained a well-tested XML library that was specially built to work with the `ModKit` library. The library was ported to C++ to ensure maximal compatibility between the C++ and the Delphi library.

5.2 Implementation: Module Framework

The design choices for the module framework were introduced in Section 4.2. This section presents details on its implementation. First, the final decisions on the XML configuration file mechanism are shown. Later on, the implemented factory mechanism is introduced, it is one of the core pieces of the framework. Afterwards, the `ModuleApp` module is presented. There, the loading mechanisms to create an application from a configuration file is shown together with an introduction on the `LTC`. Finally, the description of the `TaskRunner` implementation ends this section.

5.2.1 Signal-Slot Mechanism and Data Transport

As can be seen in Section 4.3, a lot of data communication between the modules exists. The data flows from the input to the processing, into the data storage, and finally to the output. The implemented framework relies on a signal-slot mechanism. With C++11, `std::function<>` can be used as function pointer, which in turn can be used for callback implementations. This functionality provides the most basic signal (the callable function pointer) and slot (a function that binds to the pointer) mechanism. A disadvantage of this mechanism is the 1-to-1 cardinality between the signal and the slot. Only one slot can be bound to a pointer. For convenience, `Boost.Signals2` [7] was used instead for the signal and slots. Besides allowing N-to-N bindings, automatic live time management between the signals and the slots simplifies the development significantly. The consequent use of type aliases like

```
using DataSignal = boost::signals2::signal<void(IDataPtr)>;
```

allows exchanging of the used technology if needed. The Boost library introduces some overhead. The implementation of a simplified signal-slot library, which fits to the GA use cases, is intended later on, after the work of this thesis has finished.

Coupled to the used signal-slot mechanism is the use of a data base class `IData`. With the help of shared pointers, arbitrary data can be passed through an application. Listing 3 shows an example of such a data class. It is a struct that inherits from `IData`, therefore, `std::shared_ptr<IData>` can be created from it. In the case, the snippet shows the implementation of the chunk data format, containing the CID and a byte payload.

```
struct  ChunkData : public IData
{
  public:
    ChunkData(){};
    int CID;
    std::vector<std::byte> data;
};
```

Listing 3: `ChunkData` implementation.

Other data classes similar to `ChunkData` exist in the prototype implementation. The data packages are created on the heap and referenced by using smart pointers. This allows safe package transport by passing the pointers around in the application. A reference count keeps track on the number of used pointers. If all pointers go out of scope, the data pointed to will be freed.

5.2.2 Module Framework: XML Configuration

During the design phase, it was decided to use an adapted XML meta-programming solution from ModKit. This section introduces the structuring of such a configuration file and highlights the core decisions of the integration into the framework.

Listing 4 presents a dummy XML configuration to show the functionality. At the top, the XML declaration gives information on the XML version and the encoding. It was decided to integrate a versioning of the used syntax early on. For that purpose, the configuration is wrapped in a `ModKitLite` element, which defines this version (see line two). As said in the design section, the root of a module framework application is the `ModuleApp` module. Therefore, a XML element with the tag `ModuleApp`, as on line three, is always present. Each node in the configuration file can have an arbitrary number of properties as XML attributes. Nodes of custom modules need to include a `Class` attribute, which specifies the C++ class that should be used to instantiate it. An example is the `CustomModule` on line four, which specifies `CustomModuleImpl` as target class. There may exist additional XML nodes anywhere in the configuration, e.g. the `NotAModule` node at line nine. These nodes are ignored in the module creation phase. It is up to the parent node to handle them. Typically, a path through the module tree is built by having the root (ModuleApp) as `/`, followed by the tags of the appropriate child modules. If multiple modules with the same tag exist, the first one from top to bottom is chosen. To allow unique paths, a `Name` attribute can be added, as it was done on line five and six. Only the name will now be considered for path resolution.

```
1  <? xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <ModKitLite version="100">
3      <ModuleApp Property1="xyz" ... >
4          <CusotmModule Class="CustomModuleImpl" >
5              <EqualModule  Class="ModClass1" Name="mod1" />
6              <EqualModule  Class="ModClass2" Name="mod2" />
7          </CusotmModule>
8          ...
9          <NotAModule>
10             <Content/>
11         </NotAModule>
12     </ModuleApp>
13 </ModuleApp>
```

Listing 4: Dummy XML configuration of a framework application.

5.2.3 Module Framework: Module Registration

A core functionality requirement of the module framework is the possibility to add new modules to an application without the need to change or recompile the framework library. Advanced language features in Delphi, for example, class references (meta-classes), virtual constructors, and a unit based initialization section, would simplify the implementation of such a class factory mechanism. Because C++ misses these language features, another way had to be found that the framework can provide an elegant interface to add new modules.

Listing 5 shows the code for the factory mechanism. The first class is the factory itself, called `ModuleFactory`. At line seven, the `factoryFunctionStore` map stores factory functions for all registered classes. The static `instance()` function holds a static factory instance and returns a pointer to that factory, which can be used to register a class or to

```
 1 using FactoryFunction =  std::function<IModulePtr(void)> ;
 2
 3 class ModuleFactory{
 4   private:
 5     ModuleFactory(){};
 6     std::map<std::string, FactoryFunction> factoryFunctionStore;
 7   public:
 8     static ModuleFactory *instance(){
 9         static ModuleFactory factory;
10         return &factory;
11     }
12
13     void registerClass(const std::string &className, FactoryFunction factoryFunction){
14         factoryFunctionStore[className] = factoryFunction;
15     }
16
17     IModulePtr create(const std::string &className){
18         IModulePtr instance = nullptr;
19         auto it = factoryFunctionStore.find(className);
20         if (it != factoryFunctionStore.end()) {
21             instance = it->second();
22         } else {
23             throw "Module: " + className + " not registered!";
24         }
25         return instance;
26 }   };
27
28 template <typename T>
29 class Registrar{
30   public:
31     Registrar(const std::string &className) {
32         ModuleFactory::instance()->registerClass(className, [](void) -> IModulePtr {
33                                                   return std::make_shared<T>(); });
34 }   };
```

Listing 5: Module factory with registration functionality.

get an instance of that class. The `registerClass()` function at line 15 stores a factory function in the `factoryFunctionStore` to register the class. The last method in the `ModuleFactory` class at line 19 provides the functionality to create an instance of one or the registered classes. The `factoryFunctionStore` is searched after the class name, if the requested class is registered, an instantiation of it is returned. If the class is not registered, an appropriate error message is returned. After the factory at line 32, a templated helper class `Registrar` is presented. It can be used by module implementations to register a new module class. The `Registrar` constructor call gets a pointer to the factory over the `instance()` method and calls `registerClass()`. Through the template type, the registrar is able to create a lambda function that has the signature of a `FactoryFunction`, creates the template class and returns a shared `IModulePtr`. This lambda function is stored through the `registerClass()` function, and called when someone calls `ModuleFactory::instance()->create("classname")`.

The user of the framework has to link to the framework library and implement a module that derives from `IModule`. Afterwards, a single line in the source code is enough to register the class. `static Registrar<Sensor> registrar("Sensor");` would be the registration of a sensor class. Albeit not specified by the standard, current compiler initialize the static variables at namespace scope before the program enters the `main()` routine. Thus, it is guaranteed that all needed classes are registered in the factory before the execution starts. One thing to be cautious of is the linking phase of the module framework library. An aggressive optimizer may not export symbols that are not used at link time, e.g., the `registerClass()` function is not in use at that time, as no modules are registered. At least the GCC [26] linker needs a special `-whole-archive` flag to link everything properly.

This module registration solution was based on an article by John Cumming [16]. Numerous previous attempts to implement such a factory were not as elegant or needed to revert to macro usage.

5.2.4 Module Framework: Base Class Functionality

The `IModule` class is the base class of each module. It contains convenience functionality to find and access modules in the module tree. A crucial functionality of the base class is the loading and instantiating of the modules from an XML configuration file. This functionality is implemented in static class methods. Therefore, any entity that either is a module or contains a module, is able to instantiate the module tree or parts of it.

The class method in Listing 6 shows this loading phase. The function receives a pointer to a XML element. This XML element comes from the `xmlFns` library and contains the deserialized XML file. The element contains a tree of nodes that have an `INode` base class. Each node can be a `Comment`, `Instruction`, `Text`, or an `Element` node. The element node passed to the `IModule::loadModules()` method contains the configuration of the modules that should be loaded during the method call. In the chosen architecture (see 4.2), a `ModuleApp` root module parses the XML file in its `loadApp()` function into the `Element`. Afterwards `IModule::loadModules()` is called.

In line two, a temporary struct is created to hold the loaded modules and the corresponding XML subtree. A vector is used to store the module information object for each created module. The core of the load function is the `loadChilderen` named lambda function at line nine. For each node from type `Element` of the `xml` parameter, a second static method `createModule()` is called. In case the element contains a `Class` attribute, the module is created and returned. If the class does not exist in the factory, an exception is thrown. If the element does not have a `Class` attribute, an empty `std::optional` is returned at line 13. On successful module creation, the lambda calls itself for every child of the `xml` parameter. Additionally to the recursive call, the resulted module with its XML subtree is stored in a `ModuleInfo` object.

The lambda is first called at line 25, which performs a postfix depth-first iteration of the XML file and creates every module. Afterwards, a double iteration over the created modules initializes them. First, all necessary initialization of a module is done on line 28. In a second iteration, the interconnection between modules, which requires already initialized modules, is done on line 32. Emitted events over the `onStateChanged` callback were omitted to keep the code short.

```
 1 void IModule::loadModules(IModulePtr owner, ElementPtr xml, ModuleEvent onStateChanged){
 2     struct ModuleInfo {
 3         IModulePtr module;
 4         ElementPtr element;
 5     };
 6     std::vector<ModuleInfo> infos;
 7 
 8     auto loadChildren =
 9         [&] (IModulePtr owner, ElementPtr xml, ModuleEvent onStateChanged) -> void {
10         for(auto node : *xml){
11             if(typeid(*node) != typeid(Element)) continue;
12             auto element = std::static_pointer_cast<Element>(node);
13             auto module = createModule(owner, element, onStateChanged);
14             if(module){
15                 ModuleInfo info;
16                 info.module = module.value();
17                 info.element = element;
18 
19                 loadChildren(module.value(), element, onStateChanged);
20                 infos.push_back(info);
21     }   }   };
22 
23     loadChildren(owner, xml, onStateChanged);
24 
25     for (auto info : infos) {               // Loading Stage One -> Initialization
26         info.module->init(info.element);
27     }
28 
29     for (auto info : infos) {               // Loading Stage Two -> Interconnection
30         info.module->link(info.element);
31 }   }
```

Listing 6: Loading method to create the module tree from an XML file.

```
 1 bool IModule::tryFind(const std::string &path, IModulePtr &module) {
 2     ModulePathIterator iterator(path);
 3     IModulePtr current = shared_from_this();
 4     bool result = true;
 5     std::string item;
 6     while (result && iterator.next(item)) {
 7         if (item == "/") {
 8             if (iterator.position == 1) {
 9                 current = getRoot();
10             }
11         } else if (item == "..") {
12             current = current->getOwner();
13         } else {
14             result = false;
15             for (auto m : *current) {
16                 if (m->name == item) {
17                     current = m;
18                     result = true;
19                     break;
20     }   }   }   }
21 
22     result ? module = current : module = nullptr;
23     return result;
24 }
```

Listing 7: Function to find a module in a module tree.

Another functionality of the module base class is the finding of a function in an existing module tree. Listing 7 shows a stripped down code snippet of the module finding function, error handling, and appropriate formatting is omitted to safe space. A relative path to the current module, something like `"../../PropertyModule"`, is expected as an input parameter. An `IModulePtr` reference is passed as an output parameter. A `ModulePathIterator` object is then created from the path at line two. This iterator provides a method to iterate over the separate parts of the path based on the`"/"` path separator. The while loop at line six iterates over the path parts. If the part equals a path separator and the current iterator position is at the second character in the path, the root module is chosen for further iteration. If a double-dot is the next part, the owner of the currently selected module is selected as `current`. When the path part is something different, the modules of the `current` module are iterated, if a module name equals the path part, it is selected as current. After the path has been iterated, the result is returned. This search algorithm only finds the first module of modules that have the same name. The option to set a `Name` attribute in the XML instead of relying on the by default chosen XML tag, circumvents this problem.

The implemented functionality of the module base class provides the foundation to implement modular applications. Nonetheless, additional functionality is needed to be able to create a working application. Therefore, a root module `ModuleApp` was added. The next section introduces the implementation of it.

5.2.5 Module Framework: `ModuleApp`

The `ModuleApp` module is the glue of a module framework application. As shown in Section 4.2, it is always the root of a module framework application. The `ModuleApp` is the only module that is instantiated by hand in the main function of a module application. The main function calls `startApp()`, which initiates the booting of the application. A path to the XML configuration is passed to the `ModuleApp` in this function call.

The class diagram in Figure 4.1 shows that the `ModuleApp` contains two core components. The `LTC` and the `TaskRunner`. The implementation of both components is presented in the following paragraphs. As introduced in the design section, the `LTC` oversees the state of the application based on events. Its possibility to add callbacks on state transitions allows the `ModuleApp` to control the application flow as needed. During the `ModuleApp` creation, the state transitions are added to the LTC to model a finite state machine that looks like the state diagram of Figure 4.2. A callback is attached to some of the transitions. On the `loadxml` event, a call to `loadApp()` of the `ModuleApp` is added to launch the module creation process. Similarly, a `run()` call to the `TaskRunner` is attached to the transition from `LINKING` to `ALLLOADED`.

The `ModuleApp` adds a callback to an `onStateChanged` signal. The callback is shown in Listing 8. It outputs event information to the standard output as seen on line three. This callback makes the `LTC` useful, especially in the case of an error. Later on, a dedicated logging module may be connected to that signal, which could provide advanced logging mechanisms. The implemented callback is suited for debugging purposes, but it may not be ideal for a productive environment. Other modules can act on state changes by connecting to the same signal. Further on, the other modules can add states, events, transitions, and even callbacks on transitions if they want to. This means that they can modify the state machine to their need. For example, a module could add states and events that form a separate state machine that is independent of the main state machine and can be used for their purposes. Entry and exit actions on states were implemented in an attempt to stay close to the UML state machine specification. Until now, a use case for them could not be found. Future improvements, which should result in deterministic and bug-free finite state machine for the use in embedded scenarios, may need a state machine that is more compliant with the UML standard.

```
1 void onStateChanged(IDataPtr data){
2     auto eventinfo = std::static_pointer_cast<EventInfo>(data);
3     std::cout << "EVENT: " << eventinfo.getEvent() << std::endl \
4               << "\temitter: " << eventinfo.getSender() << std::endl \
5               << "\tpath: " << eventinfo.getSenderPath() << std::endl \
6               << "\tmessage: " << eventinfo.getMessage() << std::endl;
7 }
```

Listing 8: Callback function to output state changes.

The second component that is held by the `ModuleApp` is the `TaskRunner`. Its main task is the provisioning of a main loop for the application. At the heart of the `TaskRunner` is a task queue. The queue holds tasks, which are callback functions of the type `std::function<bool(void)>;`. The `TaskRunner` provides a `run()` method and three ways to add different task types to the queue, as can be seen in Figure 4.1 on page 43. The `TaskRunner` allows adding polling tasks. Such tasks would typically be hard-coded in an endless loop within the main routine. Further on, it is possible to add single tasks or tasks that should be executed periodically in a specified interval.

```cpp
bool TaskRunner::run_once(){
    bool result = true;
    for(auto i = 0; i < tasks.size(); i++){
        mutex.lock();
        auto [task, repeated] = tasks.front();
        tasks.pop();
        mutex.unlock();
        auto res = task();
        if(repeated){
            insertReccurringTask(std::move(task));
            result = result && res;
        }
    }
    return result;
}

void TaskRunner::addPollingTask(Task task){
    insertReccurringTask(std::move(task));
}

void TaskRunner::addPeriodicTask(Task task, Milliseconds interval){
    std::thread t([=](){
        while(true){
            std::this_thread::sleep_for(interval);
            insertTask(task);
        }});
    t.detach();
}

void TaskRunner::addTask(Task task, Milliseconds timeout){
    if(timeout > Milliseconds(0)){
        std::thread t([=](){
            std::this_thread::sleep_for(timeout);
            insertTask(task);
        });
        t.detach();
    } else{
        insertTask(task);
    }
}
```

Listing 9: Core functionality of the `TaskRunner`

Listing 9 shows the implementation of the core functionality. The method `run_once()` at line one is called endlessly in the `run()` method of the `TaskRunner`. The `run()` loop sleeps for a few milliseconds if it `run_once()` returns `false`, as no work has to be done. Inside the `run_once()` function, the task queue is iterated. At line five, the task at the front of the queue is accessed using structured binding. At line six, the front is then popped from the queue. The access operations to the queue are protected by a mutex, which enables other threads in the application to enqueue tasks to the `TaskRunner`. At line eight, the selected task is executed. If it is a repeated task, which means a polling operation, it is enqueued again into the task queue. A Boolean value is built over the return values from the polling tasks. It keeps track of work that has to be done or if the application can sleep for a short time.

The other methods in Listing 9 show the different ways to add tasks to the queue. At line 17, `addPollingTask()` ads a recurring task to the queue. It calls a thread-safe insertion function. The `addPeriodicTask()` method on line 21 does what its name inclines, and ads a periodic task. This is done over a thread to which a lambda is passed at line 22. The thread executes an endless loop, which sleeps for most of the time at line 24. Whenever the interval is run out, the thread wakes up and inserts the task into the queue of the main thread. The thread is detached at line 27. The last possibility is the insertion of a single task. As shown on line 30, an optional timeout can be passed, which delays the insertion of the task into the task queue. Similarly to before, a thread is used for this purpose, but instead of a loop, the thread goes to sleep only once at line 33. If no timeout is passed, line 38 inserts the task directly into the queue.

It must be said that this solution is a simple way to solve the needed functionality with C++ language features. One could even implement priority scheduling of the tasks in the future. There are downsides to this implementation, as will be assessed in Section 6.1.1. However, it is the simplest platform overlapping solution that could be found.

With the description of the `TaskRunner` implementation, the introduction of the module framework is complete. The framework provides the basic functionality to create a modular application. The addition of the `TaskRunner` gives the possibility of creating asynchronous applications to the user. The framework is evaluated in Section 6.1. First, its functionality is assessed, then it is compared to similar frameworks from related work. Finally, the usefulness of the framework for GA is determined. Before that, the implementation of the LOG_aLevel prototype is shown in the next section. The prototype is based on the module framework described above.

5.3 Implementation: LOG_aLevel 2.0 Prototype

This section introduces the implementation of the prototype, which was done as second part of this thesis. First, Section 5.3.1 provides general information on the implemented prototype and introduces general modules. Section 5.3.2 shows the parts that implement the preprocessing requirements. Finally, the postprocessing is shown in Section 5.3.3 together with the structured data storage component.

5.3.1 Prototype: Core Modules

Listing 2 on page 49 has introduced the XML configuration of the prototype application. This section presents the basic modules, which are needed to combine other, more on specific functionality focused modules into a working prototype. It was tried to highlight only the essential parts to keep the text at reasonable length in favor of a more detailed description of the data storage and postprocessing components.

`ChunkDriver` Module

The first module of the `ModuleApp` shown in Listing 2 is the chunk driver. The binary raw data has to be somehow passed to the processing modules as chunks. A chunk driver was implemented to fulfill this functionality. Due to the missing RTU part of the prototype, an `IChunkDriver` interface was introduced to keep the driver modules exchangeable. The driver modules assume a ping-pong buffering from the RTU as described in Section 2.3.2. Concrete details how this buffer mechanism will be implemented in the hardware were not available. A chunk data generator (CDG) was implemented that provides a file based buffer functionality. A `FileChunkDriver` that implements the `IChunkDriver` interface was created. It looks for a file on a specific file path. If the file exists it reads the chunk data into memory and deletes the file. The file deletion is recognized by the data creator, and a new file is created. The size of the raw data file determines the simulated delay through ‘buffering’ between the RTU and the DHU. A more sophisticated implementation will be needed as soon as the RTU is developed by GA. The functionality required for the prototype in this thesis is given by the implementation of the file based chunk driver. The chunk driver calls a signal each time it has a chunk available for distribution. Other modules can connect to this signal to receive the data.

`GAPPModule`

The last module in Listing 2 is the `GAPPModule`. It is a wrapper to make the `propFns` library usable in the context of the module architecture. The snippet in Listing 10 shows the thin wrapper class. It holds a `GAPP::IProcessor` object, which can either be a GAPP source or observer engine. The GAPP engine needs an IO manager to work correctly. The `GAPP::IIOManager` descendants are concerned with the sending and receiving of GAPP messages. A `SerialIOManager` is used for the prototype. It allows asynchronous message-based serial communication. Additional decoration layers can be implemented in the IO Manager. Currently, a XOR checksum is added to the GAPP messages. Further implementations may add compression or an encryption layer. The `poll()` method at line nine polls the IO manager for new incoming messages. Outgoing communication is made over the GAPP engine or the properties. The processor and the IO manager are both created in the creation of the module. The properties, which are held by the `PropertyModule` described below, are connected to the processor with the `connectProperties()` procedure. A map of properties (`IProps`) is passed where each property is associated with a unique integer ID. To get a mapping to the GAPP name, a second map is passed, which connects the name to the ID.

```
class GAPPModule : public IModule, public IPollable
{
  private:
    IProcessorPtr processor;
    IIOManagerPtr iomanager;
  public:
    void init(ElementPtr xml) override;

    bool poll();
    void connectProperties(IPropsPtr props, IPropIDsPtr propIds);
};
```

Listing 10: `GAPPModule` implementation.

`PropertyModule`

Coupled to the `GAPPModule` is the `PropertyModule`. It was introduced to create some decoupling between the business logic and the GAPP engine. The decoupling was not driven too far, property types of the GAPP library were used for convenience. The module manages `GAPP::IProp` properties and connects them to the `GAPPModule`. The combination of the `PropertyModule` with the `GAPPModule` in the implemented form is a compromise. A clean solution how to handle these properties could not be found. Even in the ModKit based LOG_aLevel applications at GA, the property handling is one of the structural problems. The addition of a dedicated value manager implies that the GAPP engine has to monitor this manager and synchronize itself with it. A lot of casting and copying of the values would be included. This was suppressed by keeping the connection between both modules.

Listing 11 shows the class diagram of the `PropertyModule`. Visible at line six is the connection to the GAPP source. This implementation is trimmed to having only one GAPP source. In cases where a GAPP observer may exist, a new solution would have to be found.

```
1  class PropertyModule : public IModule{
2    private:
3      GAPP::IPropIDs propertyids;
4      GAPP::IProps properties;
5      void addProp(const GAPP::PropInfo &info);
6      std::shared_ptr<GAPP::Source> source;
7
8    public:
9      virtual void init(ElementPtr xml) override;
10     virtual void link(ElementPtr xml) override;
11     bool getProperty(const std::string &propName, IPropPtr &prop);
12     IPropPtr getProperty(const std::string &propName);
13 };
```

Listing 11: Implementation of the `PropertyModule` class.

5.3.2 Prototype: Preprocessing Functionality

The last section introduced essential core modules of the prototype implementation. This section presents the modules that cover the preprocessing functionality requirements. Section 4.3 has already introduced the architectural design. It was shown that a sensor container exists, which contains a group of preprocessing sensors. The sensors take chunk data as input and produce human understandable primary data. The data is emitted over the GAPP module and inserted into the data storage module with the help of a data insertion module. Before the sensor container and the preprocessing sensors are introduced, the parent module of all sensor module is presented below.

`ISensor` Modules

Typical modules are the sensor modules that derive from an `ISensor` interface. The final implementation went through various redesigns and refactorings. The original idea was the use of a single `Sensor` class where only the internal algorithms were generic and could be changed. The deep nesting of modules and necessary module casting made this approach unusable.

The next implementation was the deviation directly from the `ISensor` base class. This approach worked, albeit different sensor kinds lead to repeated code in the `init()` methods. Two sensor categories stood up The primary data generating preprocessing sensors and the secondary data generating postprocessing sensors. Two subclasses to the `ISensor` class were made. The UML class diagram in Figure 5.1 shows said structure. The `ISensor` base class is concerned with the SID and other functionality that each sensor has. The `IPrimaryDataSensor` and the `ISecondaryDataSensor` extend the functionality for their respective domain. Most difference is in the `init()` method, where different attributes have to be handled depending on the sensor kind. The inheritance hierarchy is not visible to the module user, only the modules at the lowest layer will be registered into the module factory. For a developer of a new module it is important to know that `parent::init()` has to be called before the own loading mechanism is executed. The importance of the different base classes for the actual sensor modules may grow in the future when the implementation of specialized sensors modules is done. Only then, it is possible to extract overlapping functionality into the respective base classes. Figure 5.1 shows two default sensor modules, one for each sensor category. The modules cover the functionality needs of the prototype. They do not reflect the complexity of a productive LOG_aLevel system where advanced algorithms are involved. These algorithms would have to be integrated into sensor modules, which was not a requirement of this thesis' work. Both default sensors are described below after the `SensorContainer` module has been shortly introduced.

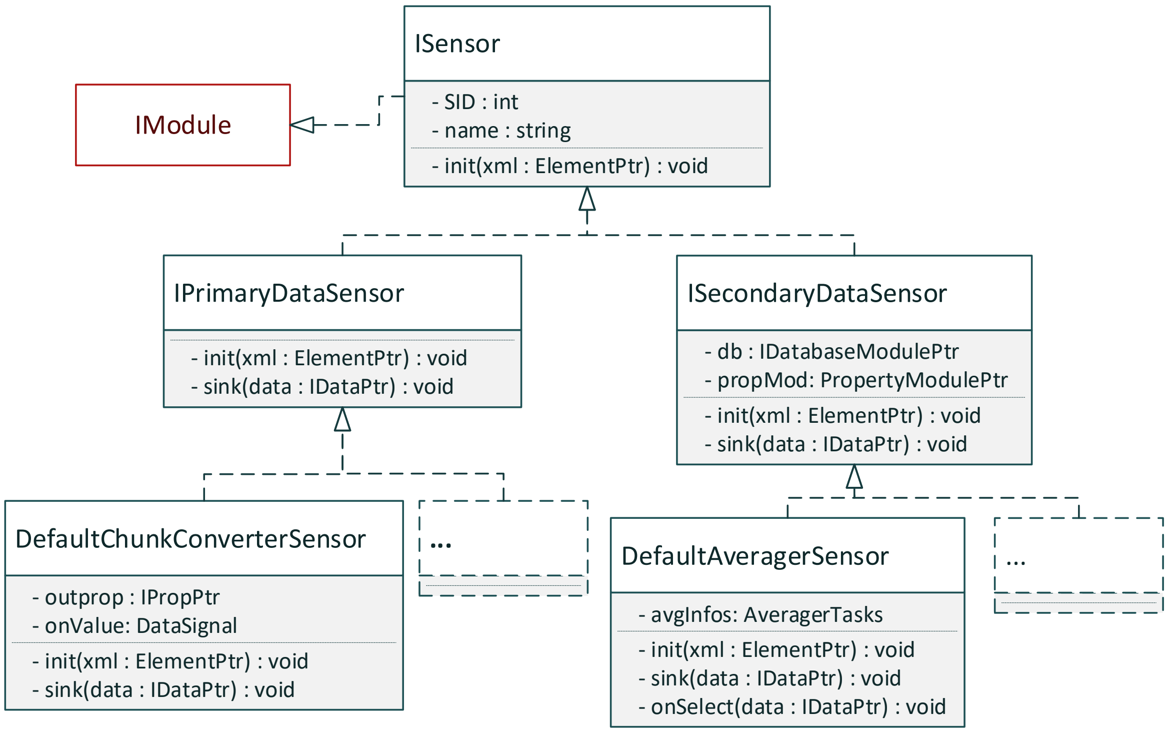
ISensor
- SID : int
- name : string
- init(xml : ElementPtr) : void
IModule
IPrimaryDataSensor
- init(xml : ElementPtr) : void
- sink(data : IDataPtr) : void
ISecondaryDataSensor
- db : IDatabaseModulePtr
- propMod: PropertyModulePtr
- init(xml : ElementPtr) : void
- sink(data : IDataPtr) : void
DefaultChunkConverterSensor
- outprop : IPropPtr
- onValue: DataSignal
- init(xml : ElementPtr) : void
- sink(data : IDataPtr) : void
...
DefaultAveragerSensor
- avgInfos: AveragerTasks
- init(xml : ElementPtr) : void
- sink(data : IDataPtr) : void
- onSelect(data : IDataPtr) : void
...

Figure 5.1: `ISensor` class diagram with the inheritance hierarchy.

`SensorContainer` Module

Section 4.3 has shown that the processing sensor modules are grouped into sensor container modules. The `SensorContainer` module realizes the `ISink` and the `IDispatcher` interface. In case of preprocessing functionality, the container takes `ChunkData` in over the sink and distributes them based on CID to the sensors that have connected to the dispatcher slot. Further on, the container module allows simple access to the sensors.

`DefaultChunkConverterSensor` Sensor Module

This paragraph describes the default primary data generating sensor that can be used to convert raw data chunks of typical timestamp-value sensors into primary data. The class can be seen above in Figure 5.1, it is called `DefaultChunkConverterSensor`. Listing 12 shows an example of a XML description for this sensor module. The first two attributes are handled by `ISensor`, they are present on each sensor object. The attributes at line five and six are handled by `IPrimaryDataSensor`, each primary sensor is connected to its container that implements the dispatcher interface. Only the attributes at line eight to ten are handled by the sensor implementation itself. Listing 13 shows the initialization methods of the sensor. As shown in Section 4.2, the initialization process contains two stages. One for the initialization and one for the interconnection of the modules. Line one shows the initialization function. Line five the linkage function of the `DefaultChunkConverterSensor`. Both functions call their parent implementation on line two and six. The linking phase after line five is meant to interconnect the modules, which are properly initialized after `init()` was called for all modules. In this case, the `OutputPropertyID`

```xml
<Sensor Class="DefaultChunkConverterSensor"
        Name="Watertemp"
        SID="10"

        DispatcherInputIDs="20"
        DispatcherModule="../../DispatcherModule"

        OutputPropertyID="15"
        PropertyModule="../../../PropertyModule"
        SinkModule="../../DataInserterModule"
        />
```

Listing 12: XML configuration of a `DefaultChunkConverterSensor` module.

attribute is read out at line eight and converted to an integer that describes the property ID. Further, at line 14 the property module path is read out from the `PropertyModule` attribute and searched in the module tree. If found, the property that corresponds to the property ID is taken from the property module and stored in the sensor module. Finally, the sink module is found at line 18 and connected to the output signal on line 23.

Further down in Listing 13, the core functionality of the `DefaultChunkConverterSensor` is shown with the `sink()` procedure. This function represents the slot that is connected to the dispatcher. Therefore, the function is called whenever the dispatcher executes the associated signal. First, at line 27, a new value object is created based on the value of the saved property. The base class value pointer is then cast to the appropriate timestamp-float value. The same is done for the `IDataPtr`, which is cast to a `ChunkDataPtr`. At line 31 and 32, the first eight bytes from the byte payload of the chunk are converted to a double that represents the timestamp. For each sample that follows, four Bytes are converted to a float, which represents a sampled sensor value. The timestamp and the sensor values are then inserted into the property value. At line 40, the newly created value is compared to the value that the output property is holding. In case they are different, the new value is assigned to the property. The `onChanged` signal of the property is then invoked. The signal leads to an emission of a GAPP message through the GAPP module. Line 44 calls the output signal, which initiates the insertion of the new value to the data storage module. Other, much more complex sensors may be implemented later on, but the basic workflow to create primary data form chunk data is shown here.

5.3.3 Prototype: Storage and Postprocessing Functionality

This section introduces the implementation details of the data storage and data storage functionality. The core design decisions were taken during the workflow concept creation and can be found in Section 2.3. The design was extended and adapted for this thesis' work. The design process can be found in Section 4.3 where the modular structure of the functionality was introduced. Modules like the `SensorContainer` or the `ISensor` descendants were introduced in the previous section. Similarly, the core mechanics of the modular applications, e.g., the signal-slot mechanism, was already shown. Therefore, this section is focused on the data storage modules and the postprocessing mechanism.

```cpp
void init(ElementPtr xml) override{
    IPrimaryDataSensor::init(xml);
}

void link(ElementPtr xml) override{
    IPrimaryDataSensor::init(xml);
    sampleCount = xml->attrDef("Samplecount", 1);
    int propID = xml->getAttrAsInteger("OutputPropertyID");

    IModulePtr m;
    if (!tryFind(xml->getAttrAsString(("PropertyModule"), m)) {
        throw ControllerErr{"Could not find: PropertyModule"};
    }
    auto propModule = std::dynamic_pointer_cast<PropertyModule>(m);
    if (!propModule->getProperty(propID, outprop)) {
        throw ControllerErr{"Could not find Property with ID: " + propID};
    }
    auto path = xml->getAttrAsString(std::string("SinkModule"));
    if (!tryFind(path, m)) {
        throw ControllerErr{"Could not find: SinkModule"};
    }
    auto sink = std::dynamic_pointer_cast<ISink>(m);
    onValue.connect(boost::bind(&ISink::sink, &*sink, _1));
}

void sink(IDataPtr data) override{
    auto value = outprop->getValue()->newValue();
    auto sensorvalue = static_cast<GAPP::TimestampFloatVectorValue *>(value);
    auto chunkdata = std::static_pointer_cast<ChunkData>(data);

    Bytes buf(chunkdata->data.begin(), chunkdata->data.begin() + 8);
    sensorvalue->ts = ByteFns::toDouble(buf);
    int offset = 8;
    for (int i = 0; i < sampleCount; i++) {
        Bytes buf(chunkdata->data.begin() + offset, chunkdata->data.begin() + offset + 4);
        auto f = ByteFns::toFloat(buf);
        sensorvalue->samples.push_back(f);
        offset += 4;
    }
    if (!outprop->getValue()->isEqual(value)) {
        outprop->getValue()->assign(value);
        outprop->doChanged();

        onValue(std::make_shared<ValueData>(value,outprop->getId()));
    }
    delete value;
}
```

Listing 13: Initialization and sink method of the `DefaultChunkConverterSensor` class.

`DataInserter` Module

The data, which is generated by the primary data generating sensors, has to be inserted into the structured database. This task is done with the help of a `DataInserter` module. It implements the `ISink` interface. All primary data sensors are connected to this module and call the data sink when a new value is available. Listing 14 shows the `init()` and `sink()` method of the inserter. At line one to seven, the database module is found and stored. Line nine loops through every insertion task node of the XML file (e.g., see line 18 on page 49). The tasks are inserted in a task map, which holds the GAPP property ID as key. Two types of tasks exist. The default insertion task inserts each arriving value directly into the database. The second insertion task was implemented to allow transaction encapsulated inserts of multiple sensor values. Instead of directly calling an insert if a value arrives, the task buffers the values until it has reached the predefined buffer size. The buffered values are then inserted into the data storage module at once. Each insertion task description with a `"Highspeed"` attribute uses buffered inserts. Later advancements of this sensor may introduce a more intelligent buffering that can react to different insertion frequencies and handles timeouts.

The use of buffered inserts was necessary as it improved the SQLite insertion performance significantly. Section 6.2.2 provides more insight on these performance issues. The use of the asynchronous insertion operation helped to improve the overall performance again as it does not lock the main thread on insertion. Line 19 and following, shows what happens if the sink function is called. The received `IData` is cast to `ValueData`. Finally, the correct task is extracted from the map and executed.

```
 1 void DataInserter::init(ElementPtr xml){
 2     std::string path = xml->getAttrAsString(std::string("DatabaseModule"));
 3     IModulePtr module;
 4     if (!shared_from_this()->tryFind(path, module)) {
 5         throw ControllerErr("Could not find: " + path);
 6     }
 7     this->dbmodule = std::static_pointer_cast<IDatabaseModule>(module);
 8
 9     for (auto e : *xml) {
10         if (typeid(*e) == typeid(Element)) {
11             auto element = std::static_pointer_cast<Element>(e);
12             if (element->getTag() == "InsertionTask") {
13                 bool isHighspeed = element->attrDef(std::string("Highspeed"),false);
14                 std::string name = element->getAttrAsString(std::string("SensorStore"));
15                 int id = element->getAttrAsInteger(std::string("ID"));
16                 addInsertionTask(id, name, isHighspeed);
17 }   }   }   }
18
19 void DataInserter::sink(IDataPtr data){
20     std::shared_ptr<ValueData> value = std::static_pointer_cast<ValueData>(data);
21     insertionTasks[value->propID](data);
22 }
```

Listing 14: Initialization and sink method of the `DataInserter` class.

Data Storage Abstraction

The design of the data storage abstraction interface was shown in Section 4.3. The class diagram in Figure 4.4 introduced the `IDatabaseModule` base class. The implementation of this abstract interface classes followed the class diagram closely. Listing 15 shows the implementation of it. The addition of asynchronous operation, as shown on line 19 and 23, became unexpectedly complicated due to the introduced multithreading. The user of the database module must have some handle to the started transaction, and it should be ensured that the data is returned to the caller. Therefore, a `TransactionInfo` class was created. It contains the type of the transaction (insertion or select), a unique transaction ID as well as an error code in case something unexpected happened. For the prototype, an error may be returned when a second asynchronous insertion call is made while the first has not finished yet. The return values of an asynchronous select can be handled over a DataEvent callback that can be passed to the function. It isinsured that the callbacks are called in the main thread with the help of the module frameworks task runner.

```
1  class IDatabaseModule : public IModule, public IPollable, public ISource
2  {
3    public:
4      virtual void connectSlot(DataSlot slot) = 0;
5      virtual bool poll() = 0;
6
7      virtual void init(ElementPtr xml) = 0;
8
9      virtual bool insert(const std::string &table,
10                         const std::vector<SensorRecord> &values) = 0;
11
12     virtual bool insert(const std::string &table,
13                         const SensorRecord &value) = 0;
14
15     virtual bool select(const std::string &table,
16                         const SensorInterval interval,
17                         std::vector<SensorRecord> &records) = 0;
18
19     virtual TransactionInfoPtr selectAsync(const std::string &table,
20                                            const SensorInterval interval,
21                                            DataEvent callback) = 0;
22
23     virtual TransactionInfoPtr insertAsync(const std::string &table,
24                                            const std::vector<SensorRecord> &values,
25                                            Task callback) = 0;
26
27     virtual void insertTodo(const TodoRecord& record) = 0;
28
29     virtual void selectRemoveTodos(const std::string &notification,
30                                    std::vector<TodoRecord> &records) = 0;
31 };
```

Listing 15: Abstract `IDatabaseModule` class.

SQLite Data Storage

Most development effort of the prototype went into the SQLite implementation of the data storage abstraction interface. As shown in the class diagram of Figure 4.4, the `SQLiteDatabaseModule` derives from the abstract interface. The serverless characteristics of SQLite require more database management in the implementation as necessary for an equal implementation with PostgreSQL as a database. Therefore, the `SensorStore` class and the `SQLiteNotificationManager` class was added as shown in Figure 4.4. The benefits thereof are that no external database preparation has to be made. If the database does not exist, it will be created with the database schema on application start. A wrapper around the C API from SQLite (SQLiteCpp [68]) was used to simplify the development. The wrapper provides a modern C++ API and hides the low-level SQLite handling.

Listing 16 shows the `init()` procedure of the `SQLiteDatabaseModule` class. Line two extracts the database path from the XML element. Line four creates the database if it does not exist and opens a connection. The flags at line five and six indicate that the database should be opened in read-write mode and that it should be created if it does not exist. Other database settings follow afterward to optimize the database. Currently, the write-ahead-log (WAL) is enabled for the database. It allows parallel reads while another connection inserts data. Overall, the use of a WAL improved the SQLite performance significantly at the cost of an additional temporary file. Line nine initializes the `SQLiteNotificationManager` by passing the database name.

```
 1 void SQLiteDatabaseModule::init(ElementPtr xml){
 2     dbName = xml->getAttrAsString("Database");
 3 
 4     defdb = std::make_shared<SQLite::Database>( dbName,
 5                                                 SQLite::OPEN_READWRITE |
 6                                                 SQLite::OPEN_CREATE);
 7     defdb->exec("pragma journal_mode = WAL");
 8 
 9     manager.init(dbName);
10     for (auto e : *xml) {
11         if (typeid(*e) == typeid(Element)) {
12             auto element = std::static_pointer_cast<Element>(e);
13             if (element->getTag() == "SensorStore") {
14                 auto store = std::make_shared<SensorStore>();
15                 store->init(element);
16                 defdb->exec(store->getCreateQuery());
17                 sensorStores[store->getName()] = store;
18                 store->onTodo.connect(boost::bind(&SQLiteNotifyManager::insertTodo,
19                                                   &manager,_1));
20     }   }   }
21     std::dynamic_pointer_cast<ITaskRunner>(
22         this->getRoot())->addPollingTask(
23             std::bind(&SQLiteDatabaseModule::poll, this));
24 }
```

Listing 16: Loading function of the `SQLiteDatabaseModule` class.

Line 10 to line 19 of Listing 16 create and initialize the sensor data tables. A `SensorStore` object is created for each XML node with attribute `"SensorStore"` at line 14. Line 15 initializes it with the sensor store element node. The sensor store table is created in the database on line 16 if it does not exist. Line 17 saves the store pointer in a map, with its name as the key. Finally, the `onTodo` signal is connected to the notification manager, which lets the store emit notifications. At line 21, a polling task is added to the task runner, which polls the database module for notifications.

Listing 17 shows methods that describe the core mechanics of the `SensorStore` class. The first procedure after line one shows the initialization of the sensor store. Line two to five extract the necessary information from the XML attributes like the name of the store or its number of columns. Line seven to fifteen handle eventual postprocessing notifications

```
 1  void SensorStore::init(ElementPtr xml){
 2      storeName = xml->getAttrAsString("Name");
 3      sampleCount = xml->attrDef("SampleCount", 1);
 4      sampleColumnNames = TextFns::SplitString(xml-
    ↪   >attrDef("SampleColumnNames",getDefaultSampleColStr()),',');
 5      allowOldData = xml->attrDef(std::string("AllowOldData"),false);
 6  
 7      for (auto e : *xml) {
 8          if (typeid(*e) == typeid(Element)) {
 9              auto elem = std::static_pointer_cast<Element>(e);
10              if (elem->getTag() == "Notification") {
11                  std::string intervalstr{"Interval"};
12                  std::string namestr{"Name"};
13                  int interval = elem->getAttrAsInteger(intervalstr);
14                  std::string name = elem->getAttrAsString(namestr);
15                  notifyInfos.push_back(NotificationInfo(name, interval));
16      }   }   }
17      insertquery = buildInsertQuery();
18      selectquery = buildSelectQuery();
19      updatequery = buildUpdateQuery();
20      createquery = buildCreateQuery();
21  }
22  
23  std::string SensorStore::buildInsertQuery() {
24      std::stringstream ss;
25      ss << "INSERT INTO ";
26      ss << storeName << " (ts";
27      for (auto colName : sampleColumnNames) {
28          ss << ", " << colName;
29      }
30      ss << ") VALUES (?";
31      for (int i = 0; i < sampleCount; i++) {
32          ss << ",?";
33      }
34      ss << ");";
35      return ss.str();
36  }
```

Listing 17: Core functions of the `SQLiteSensorStore` class.

that exist on a sensor store. For each notification, a `NotificationInfo` is created and pushed into a vector that contains all notifications for the sensor store. Line 17 to 20 build and cache SQL queries for insertion, update, selection, and creation. The method that creates the insertion query is shown after line 23. Question marks are inserted on line 21 and 23 as parameters for the values that will be later bound to a prepared statement.

SQLite provides no notification mechanisms similar to PostgreSQL's listen-notify system [64]. Therefore, a solution had to be implemented by hand. Section 2.3.3 has shown the core functionality of the notification interface. Postprocessing sensors register their notification slot to the database module. Internally, the slot will be connected to the `onNotification` signal of the notification manager. The database module can be polled for notifications, which, when available, trigger the signal of the notification manager. The implemented `SQLiteNotificationManager` is shown in Listing 18. It provides a queue to store notifications that are created by the sensor stores with the `checkEmitNotifications()`

```cpp
void SensorStore::checkEmitNotifications(double timestamp){
    TimeStamp ts = TimeFns::toTs(timestamp);
    for (auto &info : notifyInfos) {
        TimeStamp pts = TimeFns::paceOf(ts, info.interval);
        if (pts > info.lastTs) {
            if(info.lastTs == TimeStamp()){
                info.lastTs = pts;
                return;
            }
            emitNotification(info, ts);
}   }   }

class SQLiteNotifyManager
{
  private:
    std::string dbName;
    DataSignal onNotification;
    std::queue<Notification> notifications;
    void insertNotification(Notification note) { notifications.push(note); };
  public:
    void insertTodo(const TodoRecord &record);
    void selectRemoveTodos(const std::string &notification, std::vector<TodoRecord> &records);
    void init(const std::string& dbName);
    void connectSlot(DataSlot slot) { onNotification.connect(slot); };
    bool poll(){
        bool result{false};
        if (notifications.size() > 0) {
            onNotification(std::make_shared<NotificationData>(notifications.front()));
            notifications.pop();
            result = true;
        }
        return result;
    };
};
```

Listing 18: Notification mechanism for SQLite.

function shown at line one. `checkEmitNotifications()` tests inserted timestamps to each of these stored notifications. Line four of Listing 18 shows the `TimeFns::paceOf()` function. It returns the next lower timestamp that conforms to the notification interval. An input of `"17:22:05"` would result in `"17:00:00"` for an interval of ten minutes. This timestamp is then compared to the last issued notification timestamp at line five. If it is greater, a notification is issued at line ten. Two actions are triggered. A notification is inserted into the queue at line 18 using the `insertNotification()` function at line 19, which was bound to the sensor store signal. Additionally, a todo record with the information for this notification is inserted in the todo table. By polling the notifications with the `poll()` method, the signal for the postprocessing is executed, and the queue is reduced. The signaled postprocessing sensors can then select their todos by using `selectRemoveTodos()` from the `IDatabaseModule` interface and begin with their action.

Most parts of the data storage module have been introduced. Missing is the description of the actual data insertion and selection functionality. Listing 19 describes the asynchronous selection method. Functionality that is not shown, like the data insertion into the database, follows the same principals as the listed selection mechanism. Line five extracts the data store that matches the table name. The next line acquires a new transaction from the sensor store class. If the store does not exists, an error is set in the transaction on line nine, which is then returned on line ten. If the store exists, the value of the `std::optional` is extracted and stored as direct data store pointer on line twelve. The transaction type is set to the transaction on line 13, afterward, the transaction is marked as active in the sensor store on the next line. Line 16 creates the thread that executes the data selection. A lambda function with the source code for the thread is passed. The lambda captures the important values by copy, otherwise by reference. Line 17 creates a read-only database connection. The next line creates a prepared statement for the database connection and the select query of the sensor store. A transaction is started on line 19. The next two lines bind the time interval of the data selection to the prepared statement. The while loop on line 26 steps through the query result. A `SensorRecord` is created for each result record. The record is filled with the timestamp of line 28 and the sample values from the fore loop of line 30. The transaction is committed and marked as finished in the sensor store on line 37 and 38. Finally, on line 41, a task is added to the task runner. It consists of a lambda that calls the callback function by passing a `SensorRecordData` object with the result records. The thread starts with the instantiation of the thread object and is detached from the parent thread on line 47. If this call is not made, the main thread blocks when the function goes out of scope because it has to wait on the destruction of the thread. Line 48 ends the asynchronous selection function by returning the transaction information.

The description of the prototype implementation is almost complete after the data selection was shown. As said, the other database interaction methods are alike if not simpler to the method shown in Listing 19. The only missing piece are the postprocessing sensors that are shown in the next section.

```cpp
TransactionInfoPtr SQLiteDatabaseModule::selectAsync(const std::string &table,
                                                     const SensorInterval interval,
                                                     DataEvent callback) {

    auto store = getSensorStore(table);
    auto tr = SensorStore::getTransaction();

    if(!store){
        tr->setError(TransactionError::teTableUnknown);
        return tr;
    } else {
        auto storeptr= store.value();
        tr->setType(TransactionType::ttSelect);
        storeptr->startTransaction(tr);

        std::thread t([&,this, callback, tr, sp = storeptr, interval ](){
            SQLite::Database db(this->dbName, SQLite::OPEN_READONLY);
            SQLite::Statement select(db, sp->getSelectQuery());
            SQLite::Transaction transaction(db);
            select.bind(1, interval.start);
            select.bind(2, interval.stop);

            auto data = std::make_shared< SensorRecordData>();
            std::vector<float> floats;

            while (select.executeStep()) {
                floats.clear();
                double ts = select.getColumn(0);
                // one based for sqlite
                for (int i = 1; i < sp->getSampleCount()+1; i++) {
                    float sample = static_cast<float>(select.getColumn(i).getDouble());
                    floats.push_back(sample);
                }
                data->records->push_back(SensorRecord(ts, floats));
            }

            transaction.commit();
            sp->endTransaction(tr);

            std::dynamic_pointer_cast<ITaskRunner>(this->getRoot())->addTask(
                                                                    [=](){
                                                                        callback(data);
                                                                        return false;
                                                                    });
        });

        t.detach();
        return tr;
    }
}
```

Listing 19: Asynchronous selection mechanism implemented for SQLite.

Postprocessing Sensors

The last missing piece of the prototype is the postprocessing functionality. A `Default-AveragerSensor`, to create ten minutes averages, was implemented to showcase the postprocessing. It derives from the `ISecondaryDataSensor`, which handles the connection to the data storage module and the source property module. The sensor object connects to the database module for notifications. When the database module executes the notification signal, the `sink()` procedure from Listing 20 is called. First, the received `IDataPtr` is cast to extract the notification on line two. The postprocessing tasks, also called ‘todos’, are selected from the database on line four. Each selected todo is processed in a for-loop at line six. Line seven to nine collect information that is needed to extract the sensor data from the database module. Line ten initiates the asynchronous selection process. The sensor store name, the selection interval, and a callback is needed for the selection call. A lambda function is passed as a callback at line eleven. The lambda takes an `IData` object as input parameter and calls `onSelectData()` of the processing sensor. The lambda captures timestamp, transaction ID, and notification. This gives the necessary information to `onSelectData()` to average the selected values, insert them into the database module as secondary data, and output them over the GAPP engine.

```
 1 void DefaultAveragerSensor::sink(IDataPtr data) {
 2     auto note = std::static_pointer_cast<NotificationData>(data);
 3     std::vector<TodoRecord> recs;
 4     dbModule->selectRemoveTodos(note->notification, recs);
 5 
 6     for(auto r : recs){
 7         auto [inputTable, outputTable, propID] = avginfos[r.notification];
 8         double t2 =TimeFns::toDouble(TimeFns::toTs(r.utc)-TimeFns::toSeconds(r.period));
 9         std::vector<SensorRecord> srecs;
10         auto tr = dbModule->selectAsync(inputTable,{t2,r.utc},
11         [&,this, note = r.notification, ts = r.utc](IDataPtr data){
12             this->onSelectData(data, trid, note, ts);
13         });
14         if(tr->getError()!= TransactionError::teNone){
15             throw ControllerErr("Asynchronous selection failed");
16         }
17         activeTransactions[trid] = tr;
18         trid++;
19     }
20 }
```

Listing 20: Input slot of the `DefaultAveragerSensor` class.

Chapter 6

Evaluation

This chapter evaluates the work that was done throughout the course of this thesis. As described in Section 1.1, two contributions to a development project at GA were the goal of this thesis. The first contribution contained the design and implementation of a module framework. The second was the implementation of a LOG_aLevel 2.0 prototype that can store sensor data in a structured way. The prototype should be built based on the module framework. Both parts were designed around the requirements of GA. A focus was put on the LOG_aLevel product line due to the imminent product refresh. The implementation should result in a LOG_aLevel software prototype. Its core addition is the inclusion of a structured data storage functionality.

The next sections assess the results of the work that was done. First, the module framework is evaluated in Section 6.1 based on the functional requirements and the related work. Further on, the evaluation of the prototype is done in Section 6.2. In a first step, the prototype is evaluated regarding functionality, related work, and realization of the workflow concept. A second phase evaluates the performance of the prototype. A final passage at the end of each section shows the maintainability, usability, and extensibility of the thesis results from the perspective of GA.

6.1 Module Framework Evaluation

This section evaluates the module framework, which was designed in Section 4.2. First, the functional aspects are assessed in Section 6.1.1. The focus of that subsection is the requirement coverage checking of the resulted framework. Second, the framework is compared to the related work in Section 6.1.2. Finally, the framework is evaluated from the perspective of GA in Section 6.1.3.

6.1.1 Framework: Functional Evaluation

The idea of a module framework emerged as part of the sensor data management workflow concept creation, which was introduced in Section 2.3. The requirements of that concept have led to a flexible and extendable design. Prototypes of workflow parts, implemented in a monolithic OO approach, could not satisfy the flexibility needs of the workflow. GA decided that a framework should be used to implement the workflow applications in a modular way. The design and implementation of it was part of this thesis' work. The design process should incorporate related work, which was introduced in Section 3, and follow the requirements of Section 4.1.1. This section assesses the design decisions and the results of the implementation in the context of the requirements. The next section focuses on a comparison of the framework to the related work.

The framework was designed and implemented around the requirements shown in Section 4.1.1. The implemented solution works as expected. A user of the framework is able to create flexible and extendable modular software. As example serves the implementation of the LOG_aLevel prototype application in Section 5.3. The framework simplifies the implementation of such applications. It provides middleware functionality that covers the requirements of **R03**. The following list details the requirement (see page 40) coverage by stating the related design decisions (see page 47) and comparing both with the final implementation result.

· **R01** – This requirement is satisfied by providing an implementation of the module class architecture, shown in Figure 4.1 together with the implemented design decisions **D01** to **D03**, which were described in Section 5.2. The modules in a module framework are loosely coupled and can be compiled independently from other modules. Only the XML configuration file introduces the dependencies between the modules.

· **R02** – The realization of **D01** enables this functionality. The precondition is a careful module design of the framework user. The use of abstraction interfaces (**D03**) simplifies this task, as it allows modules to cast module base pointer to interface pointers and eliminates the need to know the actual module class.

· **R03** – The required middleware functionality is given with the implementation of the design decisions **D03** to **D05**. Additionally, the framework architecture, with the `ModuleApp` as an entry point, stipulates a certain way how a modular architecture has to be built. This way alleviates the application design for the framework user. The lifetime control can provide good state and error information if integrated with care. Similarly, the TaskRunner offers functionality that can be used to create sophisticated asynchronously operating applications.

Not mentioned in the list is the **R04**. It states that the implementation should focus on core functionality and implementability. The resulted implementation focused on the features discussed above. These were necessary to cover the core requirements and made an implementation of the LOG_aLevel prototype possible. Advanced interface abstraction functionality and runtime introspection of interfaces between modules were omitted due to time constraints and to keep the C++ implementation simple and less prone to obstacles during the development. From an implementability viewpoint, it can be said that the resulted source code was generally crafted using simple modern C++. Some functionality, e.g., the factory mechanism, required advanced template programming. The use of it was minimized as much as possible. This means that future contributors to the framework should find themselves comfortable in the code.

6.1.2 Framework: Related Work Comparison

Significant differences can be seen in a comparison of the implemented framework with the frameworks presented as related work in Section 3. The differences rise from varying intentions of the respective frameworks. Almost no relation can be found in comparison to the traditional component frameworks shown in Section 3.2 besides the core principle of modularity. OPRoS of Section 3.3.1 came close from the idea but was focused on a different use case with multiple separate framework users that collaborate and provide robotic modules. The ModKit framework of GA shows strong similarities with the implemented framework. This relatedness was a byproduct of design decision **D01**, where it was decided to use a similar XML configuration solution.

Table 6.1 compares the general characteristics of the different frameworks. It can be seen that only ModKit and the framework of this thesis contain an architecture description language (ADL). In this thesis, ADL connotes the meta-programming functionality of frameworks to describe modular applications. OPRoS does not contain such a mechanism. The lack of it is compensated by the component composer application. It allows a graphical aggregation of components into a modular application. COM, CORBA, and EJB target a more distributed approach. The components in these frameworks build self-contained entities and communicate loosely over interfaces. They rely on interface introspection mechanisms to connect to each other. Thus, an additional architecture description is not needed. ADL and interface introspection mechanisms require a runtime deployment of the components. Therefore, they provide the possibility to change the component composition at runtime, at the cost of longer boot times. A comparison of interface functionality is drawn after the description of the table is finished.

COM and CORBA components can be written in many languages. Languages for COM components must know the concepts of pointers and classes. Despite the use of eventual platform-independent programming languages, COM relies on Windows system features like a registry and is therefore not portable. CORBA is portable due to different implementations of the object broker. EJB applications are written in Java. Therefore, they require the JVM to interpret the bytecode and execute the application. EJBs are portable for every platform that is able to run a JVM. OPRoS and the compound object model are written in C++. They do not rely on external mechanisms and can be compiled to any platform that provides a compiler. ModKit written in Delphi is currently locked-in to Windows. Platform independence could be achieved by switching to the enterprise version of the compiler. Such a switch would need a lot of work, all components that use Windows specific optimizations and all GUI components would have to be rewritten. The framework from this thesis is implemented in modern C++, which means that it features full platform independence.

Table 6.1: Comparison of related work frameworks and the framework in this thesis.

	ADL	Language	Platform independence	Deployment	Interface introspection
COM	✗	OO oriented	✗	compile- and runtime	✓
CORBA	✗	Multi-language	✓	runtime	✓
EJB	✗	Java	✓ with JVM	runtime	✓
OPRoS	✗	C++	✓	compile-time	∼
Compound object Model	✗	C++	✓	compile-time	✓
Modkit	✓	Delphi	✗	runtime	∼
Thesis Framework	✓	C++	✓	runtime	✗

Interface introspection is an area where the framework of this thesis cannot compete with the related frameworks. It means that a module can ask other modules about their available interface functionality. COM, CORBA, and EJB rely on this functionality. OPRoS provides static introspection over the available port classes. The OPRoS component composer creates the interconnection between the components before application deployment. Dynamic interface introspection is not available. Each object of the compound object model is able to access the interface of other compound objects, but a real introspection does not exist. Again, due to its static deployment, such functionality is not needed. ModKit provides some introspection. A module is available that visualizes the module tree. The visualization includes the interconnected modules and, if available, their interface type. The framework of this thesis has deliberately omitted interface introspection. Thus, more development time for other essential functionality was available. Advanced interface introspection is not needed in the small environment of GA. It can be assumed that the hand full of developers know the interfaces of all modules.

Finally, lifetime control and task runner mechanisms are compared to the related frameworks. COM, CORBA, and EJB do not have a predefined central entity of a component-based application. The developer of such software has to create this by himself. Such an entity includes the handling of application states and execution loop. The benefit of this approach is the freedom to design any application architecture without restriction. OPRoS introduced an execution engine, which executes all components. Further, each component has its own state machine and reports state changes to the execution engine. The compound object model has no such facilities and does not impose any application design restrictions. Each ModKit application contains a processor module as an entry point. The processor module contains a state machine module, which represents the application states. The processor has to include the functionality to run the application. The framework of this thesis introduced a task runner, based on the OPRoS execution engine. It contains an event loop, which can be used by other modules to implement asynchronous functionality. The task runner has to be used for all tasks that would be done in the main function of an application. The mandatory root module of this thesis' framework contains the task runner and lifetime control. This design imposes some restrictions on the overall application design possibilities, yet it offers the functionality needed by GA.

6.1.3 Framework: Usefulness for GA

This section evaluates the usefulness of the implemented framework for GA. A modular approach to software is not unfamiliar to the GA environment. The quickly changing customer requirements enforce a flexible software development approach. The desktop software of the LOG_aLevel and the UltraLab systems is built in this way with the help of ModKit. A current restriction of GA's modular software is the platform dependency to Windows. Other software at GA, like the firmware of the devices, is written with C in a monolithic way. The design and implementation of the module framework in this thesis provide a series of contributions to the GA ecosystem. The design of the framework brought knowledge of current component architectures to GA. Reflections about the core requirements and the architecture were made during the design phase, free from any pressure of customer projects. The implementation in C++ provides GA with the opportunity to implement platform independent and highly flexible and modular console applications. Besides these benefits, the usefulness of the workflow concept is assessed based on three criteria in the following list:

- **Usability** – The implementation of the framework in this thesis provides the groundwork for future use at GA. Some implementation weaknesses, mainly the slow signal-slot mechanisms, have to be improved until a productive deployment can be made. Further adaption to the embedded hardware is suggested for the use in the new LOG_aLevel systems.
- **Maintainability** – The attempt to write clean and simple code was hindered by C++' own complexity. Even though, the class architecture is simple and understandable. Therefore, a developer with modern C++ knowledge should be comfortable in the maintenance of the framework.
- **Extensibility** – The modular approach, even in the class design, makes it possible to extend the framework without huge effort. Future implementation of modular software will show where functionality has to be added.

The next section evaluates the LOG_aLevel prototype. It was developed with the module framework as a basis. This section has looked at the usability of the framework alone for GA. Section 6.2.4 evaluates the usability of the prototype and draws a conclusion about the overall usability of this thesis' work for GA.

6.2 Prototype Evaluation

This section evaluates the prototype implementation. First, a functional evaluation of the software solution is performed in Section 6.2.1. It is based on the designed workflow concept and the prototype requirements. Section 6.2.2 assesses the performance of the prototype. A comparison to related workflow implementations and current solutions at GA is drawn in Section 6.2.3. Finally, the usability of the solution for GA is evaluated in Section 6.2.4.

6.2.1 Prototype: Functional Evaluation

The primary goal of the LOG_aLevel 2.0 prototype was the implementation of the core functionalities of a station as proposed by the workflow concept in Section 2.3. This includes the implementation of preprocessing functionality, a structured data storage component, and postprocessing functionality. The prototype should be built with the module framework that was created as part of this thesis. Section 4.1.2 shows the precise requirements. The results of the second thesis part were two software applications:

- **cdg.exe** – A data generator that simulates the RTU and produces chunk data. The data is written into a file. Together with the implemented `FileChunkDriver` of the prototype, the ping-pong buffering between the RTU and the DHU is simulated. It was built using the module framework and can be configured over an XML file. The generator is only a necessary tool to test the prototype. Therefore, it was not described in the implementation section of this thesis.
- **prototype.exe** – The prototype takes chunk data from the CDG and preprocesses the raw data into primary data. The data is outputted over GAPP and inserted into a SQLite-based structured data storage solution. The insertion triggers postprocessing algorithms that generate secondary data. The secondary data is inserted into the data storage component as well as outputted over GAPP.

Table 6.2 goes into details on the resulted prototype in comparison to the requirements. The requirements of each use case are listed, below, the workflow results are summarized in a 'result' row. It can be seen that all requirements were fulfilled. It can be remarked that the workflow design resulted in two application. One for the preprocessing functionality. The other for the structured data storage with postprocessing functionality. As motivated in Section 4.3, it was decided to merge the two applications for the prototype implementation. The use of the module framework made the application flexible. A separation into two applications could be done later on with minimal effort. A reasonable GAPP IPC module has to be implemented to make the separation usable.

Table 6.2: Results of the prototype functionality compared to the imposed requirements.

ACQ-01	Implementation of an abstraction interface to decouple the real-time hardware part form the rest of the workflow.
ACQ-02	Use of a Linux DHU in the station controllers.
Result	The use of the chunk protocol implements the decoupling. The described interface remains at an abstract level due to the missing RTU implementation. The prototype performance was evaluated additionally on a low-level Linux controller. Therefore, both requirements are satisfied.
PRO-01	The data, which is collected by the acquisition part, has to be preprocessed and brought into human understandable form.
PRO-02	An interface has to exist that allows the inclusion of advanced postprocessing algorithms into the station to generate aggregated values.
PRO-03	The interface to the postprocessing has to be compatible with both station and station server.
Result	The prototype implements this functionality as was introduced in Section 4.3.
STO-01	Logging of raw data in a file based way is required.
STO-02	The sensor data has to be stored in a structured data storage component that provides interval based access.
STO-03	The interface to the data storage component has to be compatible with both station and station server.
Result	STO-01 is completed by the ability of the GAPP engine to log input to files, and was not considered further. STO-02 and STO-03 were designed and implemented, as also shown in Section 4.3. The storage solution was implemented for the station prototype using SQLite. The abstract interface allows the implementation of a different technology for use on a station server.

The integration of the prototype into the low-power Linux board is not as deep as it might have been with available station RTU hardware. A replication of current LOG_aLevel functionality was not the focus of the implementation. Thus, only default algorithms that cover the basic workflow functionalities were implemented. Advanced processing algorithms can be implemented later on by GA. A data converter sensor was implemented to showcase the preprocessing. It can create GAPP data from binary timestamp-value chunks. A default averaging sensor demonstrates the postprocessing workflow. It creates 10 minutes average values based on data in the data storage component. These modules cover the needs of the prototype and demonstrate the processing interfaces. The prototype depends on GAPP for input and output functionality. The GAPP engine itself was developed together with GA at the beginning of this thesis. It was not relevant to the design and implementation process of the thesis. A serial GAPP driver was implemented to test the GAPP engine. This driver was used to get a working prototype implementation. Other driver implementations, e.g., for the IPC were postponed as they were not necessary for the prototype.

The last paragraph has shown some restrictions on the prototype implementation concerning LOG_aLevel functionality and system integration into the hardware. In return, the use of the module framework provides significant benefits for the prototype architecture, and, on a longer term, for the implementation of the workflow at GA in general. The following advantages can be seen:

- **Architecture** – The benefits of the module architecture, like object dependencies and serializability, were mentioned in Section 3.1. Additionally, the event-based architecture would allow simple integration into a GUI application.
- **ExtenSibility** – The presented module structure is flexible and can be extended. New modules can be added to integrate new preprocessing or postprocessing functionality. The unified data transfer mechanism allows the integration of new modules that do not follow the workflow concept directly. Therefore, individual solutions can be integrated without problems. An example could be ADCP data, which would not be handled by the RTU.
- **Configurability** – Given that all necessary modules have been implemented, the XML configuration provides a meta-programming layer to compose the applications. No compilation is needed for changes to the XML configuration.

The last part of the functional evaluation is concerned with more technical implementation details. As seen, the prototype was implemented using modern C++ features. CMake [12] was used as the build system for all libraries and executables. CMake allows to prepare the build process for different compilers in a single `CMakeLists.txt` file. Properly configured, CMake is able to detect the build environment and execute the appropriate build commands. The build targets were successfully built using the MSVC [50] compiler, GCC [26] on MinGW [36], and GCC on a Linux computer. A custom GCC derivate for ARM was used for the cross-compilation toolchain. Besides portability, CMake projects can be imported into various integrated development environments (IDE). Visual Studio [44] was used for its good profiling tools. Eclipse [20] to cross-debug the applications on the low-power Linux controller. The software components were developed in Visual Studio Code [51], a new editor from Microsoft [43] that provides basic debugging functionality.

6.2.2 Prototype: Performance Evaluation

This section evaluates the performance of the implemented prototype. First, the evaluation setup is presented. Later on, the CPU usage, as well as the memory of the prototype is assessed. A focus is put on the evaluation of the SQLite implementation of the data storage component. Finally, some power consumption evaluations are made.

Setup

The measurements of the developed software were executed on two different computing devices. A high-performance device was used to test the maximum performance of the implemented solution. A low-power Linux driven controller was used to test the developed solution for their usability in a GA station.

A custom-built computer was used as a high-performance testing station. It had an Intel Core 7th generation i7 CPU [33] running at 4.2GHz. One of the fastest currently available solid-state disks, claiming up to 3500 MB/s reading- and 2100 MB/s writing speed [69] was installed. The machine had 32 GB of double data rate fourth-generation (DDR4) memory [15]. The device was built to hold up with high CPU loads. It was chosen for performance measurements and for the profiling of the applications.

A Linux controller from Phytec [60] was used to test the station capabilities of the software. A socket-on-module (SOM) with an ultra-low-power phyCORE-i.MX6 UL-G2 [62] CPU was used. The CPU is based on the ARM Cortex-A7 architecture. The SOM has 512 MB DDR3L memory and 512 MB NAND storage. The SOM was placed on a phyBOARD-Segin [61] evaluation board from Phytec to make the it usable. The SOM will be used at GA for the refresh of the LOG_aLevel. Thus, performance tests with this CPU provide essential feedback on the capabilities of this SOM. The carrier board, which provides access to the peripherals and interfaces, will be custom-built by GA. It will contain the RTU part of the workflow. The default Phytec Yocto distribution was used to run the software. It is based on Poky [81], the default distribution from the Yocto project. A virtual machine, provided by Phytec, was used to compile the sources. It contains the necessary cross-compilation toolchain for the phyBOARD. Further interaction with the board was done over SSH connections.

Two scenarios were evaluated. The first simulates a real-time scenario with a data flow that has to be expected for the new LOG_aLevel system. The simulation contained three level sensors that sample with 20 Hz each. Seven other meteorological data sources were included. They produce samples in intervals between three and 20 seconds. The second scenario pushes the data flow to the limits. It tests the database throughput, CPU performance, and power consumption under heavy load. To do that, the frequency of the level sensors was raised up to 250 Hz each.

Performance

This thesis' work focused on the architecture and functionality rather than the performance of the developed solutions. Nevertheless, a performance analysis is essential for the development project at GA. Eventually, the performance decides over the usability of the prototype implementation and the workflow concept for GA. The focus of the performance evaluation is on the data storage solution. The performance of the SQLite solution is the biggest element of uncertainty in the workflow concept. There was no known related work that showed the performance of SQLite on embedded systems together with the use case of high-speed real-time sensor data. First, some impressions on the performance are shown below, which directed the development process. Later, more concrete performance measurements are made.

The first measurements were made on the high-performance machine. At that time, the prototype was split into two applications as shown in the workflow concept in Section 2.3. The CDG and the prototype parts were executed. The CDG was configured to output data as fast as possible. The prototype applications were not able to parse all generated data from the CDG. The applications filled their buffers until they crashed. A profiling showed two key weaknesses of the implementation:

- The implemented `SerialIOManager` was really slow and consumed around a third of the whole CPU usage time.
- The implemented GAPP engine used the regular expression library from the C++ standard library to parse GAPP messages. It turned out that the parsing takes again around 10 % of the CPU time.

The CDG data rate was adjusted that the prototype could consume, parse, and output the data without crashing. Then, the first SQLite measurements were made. Around 120 inserts per second could be reached on the high-performance device with constant disk writing speeds of around 2 MB/s. Both numbers caused some frowning. They were way below the expected or envisioned numbers and would not have been enough for the ideas of GA. Some research on SQLite performance brought a StackOverflow entry up. There, it was shown that the insertion performance can vary significantly from 80 to 96000 inserts per second [74]. The encapsulation of multiple insertions into transactions improved the performance instantly. The `IDatabaseModule` interface of the prototype was extended to allow transaction-based batch inserts. Another decision was made to minimize the impact of the GAPP IO manager. The two prototype parts were merged into a single application. The intermediate GAPP IPC was not necessary anymore and could be removed. New performance measurements were made with this setup. Insertion speeds of 1000 up to 1800 inserts per second were measured on the desktop computer. The activation of WAL on the SQLite database and code optimizations improved the performance even more. Pleasing insertion speeds of 20000 up to 30000 inserts per second could be reached. With the knowledge of sufficient SQLite performance, the prototype implementation was finalized. The next paragraphs present measurements of the final prototype implementation.

First, the prototype was evaluated in a real-time scenario. The CDG was configured to output three level values at 20 Hz. Seven other meteorological sensors were included. They sampled at a 20-second interval. Therefore, the total data rate was around 60 Hz. The applications were built in release mode with default optimization enabled. In a first setup, the prototype was configured to insert each sample directly into the SQLite database. The results can be seen in Table 6.3.All tables show average values over a minute. The desktop computer performed reasonably well. The high memory usage for the CDG comes from the raw data that was used to create the samples and was expected. The CPU usage of the CDG and the prototype always stayed under 1.5 %, which was also expected. The insertions per second into the database, in the table called throughput, were at 59. The missing Hz was lost due to the not very exact data generation of the CDG and the implemented file based buffering. The disk write speed was at 2 MB/s, which is already high. Each insert comes with a write to a SQLite journal file before the insertion is written to the database file. Different was the situation on the iMX6 controller.

Table 6.3: 60 Hz data, inserted with single inserts.

Desktop			**iMX6-UL-G2**		
CDG	CPU	0.1 %	**CDG**	CPU	3.3 %
	MEM	311 MB		MEM	47 MB
Prototype	CPU	1.2 %	**Prototype**	CPU	**47.2 %**
	MEM	1.4 MB		MEM	0.7 MB
	Throughput	59 p.s.		Throughput	59 p.s.

Both processes required more CPU time to do their work. The memory consumption of the prototype is lower, which can be put on compiler- and architecture differences. The throughput is equal to the desktop computer performance with 59. The CPU usage of the prototype, on the other hand, is way too high. It varied between 35 and 55 % and averaged at 47.2 %. Such performance is not unusable for a productive environment.

The single insertion measurements into the database showed the performance limits of SQLite. The 60Hz was barely doable and ate away a lot of CPU time. There is not much room for other processes in the system or even higher sample rates. The next measurements were made with the use of transactions. 200 samples were buffered before they were inserted all at once. Otherwise, the setup stayed the same as in the first measurement. The results can be seen in Table 6.4. On the desktop side, the CPU usage of the prototype came down to 0.1 % as did the disk write speed. A better performance could not be expected. Similarly, the CPU usage of the prototype on the iMX6 went down to a reasonable 5.72 % on average. Contrary to the performance of the previous measurement, this performance is usable in a productive environment.

In order to test the capabilities of the prototype, a third measurement was made. This time, the three level sensors were set to 250 Hz each, which results in a combined data rate of 750 Hz. Transactions were used again to improve the insertion performance. Table 6.5 shows the results of the measurements. The desktop computer performed very well, CPU usage was low, and the memory consumption remained constant. The final throughput with 684 inserts per second does not reach the expected 750. The difference can again be put on the CDG due to its inaccuracy in the data creation. A different picture can be seen on the iMX6 controller. The CPU usage of the CDG went up to 19 %. Similarly, the prototype needed an averaged 15.8 % of the available processing power. The memory consumption remained stable. This indicates that the prototype was performant enough to process all the data in time. A maximum throughput of 577 inserts per second was reached, which can be considered as good.

Table 6.4: 60 Hz data, inserted using bulk insertion of 200 samples.

Desktop			**iMX6-UL-G2**		
CDG	CPU	0.1%	**CDG**	CPU	3.2%
	MEM	311 MB		MEM	47 MB
Prototype	CPU	0.1%	**Prototype**	CPU	5.72%
	MEM	1.4 MB		MEM	0.7 MB
	Throughput	59 p.s.		Throughput	59 p.s.

Table 6.5: 750 Hz data, inserted using bulk insertion of 200 samples.

Desktop			iMX6-UL-G2		
CDG	CPU	0.1%	**CDG**	CPU	19%
	MEM	310 MB		MEM	47 MB
Prototype	CPU	0.2%	**Prototype**	CPU	15.8%
	MEM	1.6 MB		MEM	1.1 MB
	Throughput	684 p.s.		Throughput	577 p.s.

In all three measurements that were taken, the CDG introduced performance uncertainties due to the inaccurate data generation. To minimize the effects on the data storage measurements, it was decided to use a big chunk data File as data input. A test file was created. It contained the 60Hz data stream that was used for the first two measurements. The measurements led to the results shown in Table 6.6.

The resulting performance was fast. The desktop computer was capable of making 82000 inserts per second at 12 % CPU usage. The CPU has eight logical cores. Therefore, one core was used entirely. The writing speed was at 25 MB/s. The MSVC compiler was needed to get the maximum performance. GCC used other system calls and did only reach 5-7 MB/s write speed, which slowed the whole performance down. The iMX6 was able to insert around 5700 samples per second, but it needed to use 96 % of the computing power. The memory consumption is much higher because the prototype reads the raw data from the file into the main memory. The downward arrows indicate a sinking memory as time progressed and the data was processed. For some reason, MSVC reserves a lot more memory than GCC.

The performance measurements can be concluded as follows. A realistic sample frequency for the new LOG_aLevel at GA would be at around 30 Hz. The measurements in this section started with sample rates of 60 Hz. It was shown that the iMX6 CPU and the prototype can handle this data rate, thus a sample rate of 30 Hz is also achievable. Higher data rates are possible at the cost of CPU time. Profiling during the development showed that a lot of processing time is lost in deep call stacks. The Boost signal-slot mechanism introduced a lot of nested calls and had a negative impact on performance. This issue was reduced by switching to a signal-slot library that had less of an impact. Nonetheless, the architecture of the module framework itself introduces a lot of nested calls. The task runner with the queue of callable objects creates most of these encapsulated calls. It is a cost of the gained flexibility. The iMX6 seems to be more vulnerable to huge call stacks. Modern Intel CPUs seem to optimize function calls that directly call a nested function. Further optimization of the signal-slot mechanisms will free processing time. All in all, the section has proven that a SQLite solution provides the needed performance for the LOG_aLevel sensor data workflow.

Table 6.6: Unrestricted bulk insertion from a chunk data file.

Desktop				iMX6-UL-G2			
Prototype	Throughput	CPU	MEM	**Prototype**	Throughput	CPU	MEM
	82000 p.s.	13%	118 MB ↓		5700 p.s.	96%	22 MB ↓

Power Consumption

The missing piece of the prototype performance evaluation is an assessment of the power consumption. In the very first thoughts about this thesis, the power consumption optimization was one of the targets. A direction change was caused by first implementations of some workflow concept parts. The design of a modular software architecture was preferred at the cost of a deeper hardware integration. Nonetheless, some basic power consumption considerations were made. Table 6.7 shows the results. The iMX6 SOM should draw between 0.5 and 1 Watt. The phyBOARD around the SOM needs some power too. The results vary between 1.09 and 1.33 Watt. This result is in the range that had to be expected. A custom board with a specialized RTU may save some energy in the future.

Table 6.7: Power consumption of the iMX6 controller in idle mode and under full load.

	Idle	Full Load
voltage [V]	24 V	24 V
current [mA]	45.4 mA	55.37 mA
power [Watt]	1.09 Watt	1.33 Watt

6.2.3 Prototype: Related Work Comparison

This section compares the prototype to related work. Because the core design of the prototype architecture was made during the workflow concept design prior to this thesis, the related work was just fairly mentioned during the concept introduction in Section 2.3. Nonetheless, a comparison is made to the related workflows and current systems at GA. Appendix B adds the missing information for interested readers. The concept of a station is in the center of this comparison. Not all workflows or data collection systems of the related work contain a component similar to a GA station. Therefore, only the implemented workflow functionality is compared. It ranges from the data measuring up to a data-collecting server.

Wong and Kerkez [78] (see 118) measure data using various sensors. The sensor data is collected by a Cypress PSoC5LP based controller [17]. It uses a C program to put the data into Xively feeds. From there on the data is managed by the Xively cloud. The adaptive sampling is done over a Python script on a second server that can access the Xively data. Based on Xively's RESTful API, a web application was built to present the measured data. Only the workflow until the Xively connection can be compared. The cloud can be seen as external server entity. Compared to the prototype functionality, only the data transfer functionality can be seen in their workflow. No data processing or data storage functionality can be found. A benefit of that setup is the minimal power consumption of the system due to the minimal tasks of the data collector. Without a connection to Xively, the system from Wong and Kerkez does not work. Whereas a prototype station could run as a self-contained device.

Horsburgh et al. used a more comparable workflow setup as described in Section B.1. They use a Campbell Scientific, Inc. [9] data logger to collect the data from various sensors. The logger synchronizes CSV files to a LoggerNet server. The data is then taken by an hourly executed Python script, and put into an ODM database on a second server. Similar to Wong and Kerkez's workflow, the data is just transferred to the server and put into a database. The processing and presentation of the data do again not occur at the 'station' level. Their data logger is at that level. Compared to a prototype station, the functionality of the station is limited. The difference in this case is, the data logger provides far more functionality than actually used by Horsburgh et al. Campbell offers a custom BASIC dialect CRBasic [10]. It can be used to implement data processing functionality in the logger. Similarly, various data presentation modes can be programmed. The data storage on the logger, however, is implemented using CSV files. Compared to the prototype solution, the Campbell data logger lacks the fine-grained preprocessing and postprocessing functionality. But the main distinction point is the lack of the structured data storage solution. Additionally, the use of Campbell loggers is coupled with a lock-in to the CRBasic and the Campbell Scientific Inc. world, other tools and languages cannot be used.

The best comparison of the prototype solution can be made to current GA systems. The current LOG_aLevel was introduced in Section 2.1.1. Its functionality can be described as a mixture of the RTU functionality with the preprocessing functionality. Timestamp management with preprocessing of the data is provided. With a developed RTU and implemented preprocessing algorithms the prototype fulfills the functionality of a current LOG_aLevel in a well-structured way. It that can be extended without problems, contrary to the current LOG_aLevel solution. The big difference of the prototype to the current systems is the addition of the structured data storage solution. Based of it, the inclusion of postprocessing functionality was possible.

The comparison can be concluded by pointing out that the two compared workflows did not put a focus on the station and used them only as data relay. The data logger from Campbell Scientific Inc. from in Horsburgh et al.'s workflow would provide processing and presentation functionality with the cost of a lock-in effect. The implemented prototype is best compared to current GA systems. The prototype implements the functionality of these systems and extends it with the data storage and postprocessing functionality.

6.2.4 Prototype: Usefulness for GA

This section assesses the usefulness of the LOG_aLevel 2.0 prototype for GA. The design of the workflow concept prior to this thesis gave GA the opportunity to assess the current situation, recapitulate the past years, and research current workflow in the environmental data sensing field. No hard customer requirements had to be met at the time of the concept creation. Therefore, a more abstract, top-down approach could be followed. Thereof, theoretical benefits for GA were made. Guidelines were set on how to implement such a workflow in a way that it covers the core requirements and can be extended as needed. Abstraction interfaces and protocols were introduced and determined. The design and implementation of the module framework added more value to GA, as shown in Section 6.1.3. It provided the needed environment to actually implement parts of the workflow with the needed flexibility. The following practical benefits can be emphasized:

· **Current C++ Knowledge** – A big amount of C++ knowledge was acquired during the time of this thesis. The knowledge of the CMake build system for platform independent development and of the newer C++ standards was brought into GA. With the developed code base for station controllers, the first steps into C++ controller firmware development were made, and can be continued by GA.

· `GALIB_CPP` – Parts from the `GALIB`, written in Delphi, were ported to C++. The API's were kept equal to provide as much similarity as possible between the libraries. This provides the possibility for GA to implement applications in a similar way in C++ or Delphi.

· **Station Controller Prototype** – The prototype provides the groundwork for further station software at GA. The module framework introduces great flexibility and extensibility. Low coupling allows the exchange of implementation parts without too much effort.

· **SQLite Evaluation** – The biggest benefit for GA of this thesis' prototype is the performance evaluation of a SQLite data storage implementation. It has shown that such a solution is practical and provides enough performance reserves to be used in a productive low-power embedded environment.

The benefits of this thesis' work for GA can be concluded as follows: The sensor data management workflow concept has shown a structured way to handle IoT influenced requirements of the GA customers. Complete productive workflows can be built if GA works out the details of the hardware integration- and the data presentation functionality. The implemented module framework provides a development environment to create software applications that comply with the workflow requirements. The prototype of this thesis introduces the knowledge and the tool-set to implement the workflow concept. It provides the core libraries and prototypical implementations of the data processing and storage functionality from the workflow.

Chapter 7

Summary and Conclusions

GA, a hydro-acoustic measuring system producer, is refreshing its product lineup to follow the current market situation. Two contributions to this GA development project were made throughout the course of this thesis. In a first part, a module software framework was developed. The Framework design was based on related component frameworks and requirements from GA. As a result, the framework covers the required functionality. It provides flexibility, extensibility, and a middleware layer to develop modular application. A core feature is the XML configuration solution. The users of the framework can add or change modules in the configuration file without the need to recompile the application.

The second contribution of this thesis was the prototyping of a LOG_aLevel software. A prototype has been developed that implements a part of the workflow concept. The use of the before designed module framework guided the architecture of the prototype. As a result, the developed application was built in a modular and flexible way. A structured data storage component was implemented with SQLite as database technology. Other technologies can be added later on, over an implemented abstraction layer. Aside from the data storage functionality, the data processing functionality requirements of the workflow concept were implemented by the prototype.

The evaluation of the thesis work lead to the following results: The implemented framework follows the requirements of GA and the workflow concept. It can be used for platform independent development of modular software. Further on, the framework is simple and extendable, which is crucial for the ever-changing requirements at GA. The requirements of the second part, the development of a LOG_aLevel prototype with structured data storage and processing functionality, were fulfilled. It was built with the module framework of the first thesis part. The performance evaluation on the iMX6 SOM, which will be used as CPU for the new LOG_aLevel at GA, has shown that the implemented solution with SQLite provides enough performance for the use on a LOG_aLevel station.

Given the goal of the module framework design and the implementation of a LOG_aLevel prototype software, one can draw first conclusions: As it turned out, the design and implementation of flexible and dynamic functionality in a statically compiled language as C++, is a complex task. The implementation part of this thesis has shown that efficient C++ development needs a lot of detailed knowledge on the language features. Custom solutions to very simple tasks have to be found. Too often simple errors have slowed down the implementation immensely due to unclear error messages. On a positive note, it can be said that the implemented solutions cover all required functionality. The offered performance surpasses the needs of current workflows at GA. All in all, the framework and the prototype provide a good foundation for a productive usage.

Further on it can be concluded, that both thesis parts have limitations. The framework was built with high-level platform independent C++. A productive solution would have to implement specializations to utilize platform depending optimizations. Further on, some frameworks were used for convenience rather performance. Both need to be done in future work to create market-ready framework. Such future work can build upon the groundwork done in this thesis. Similar, the prototype needs improvement to reach market maturity. First, a better performing GAPP solution has to be implemented. The development project at GA will be continued. Thus, the realization of the future work will be done in a not too distant future, after this thesis has been finished.

Bibliography

[1] H. Akima: *A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedures*; Journal of the ACM, 17(4), 1970, ISSN: 0004-5411, pp. 589-602, doi: https://doi.org/10.1145/321607.321609, last visited: Jan. 24, 2018.

[2] ANT - Applied New Technologies AG: *Water Jet Cutting*; URL: https://ant-ag.com/en/water-jet-cutting.html, last visited: Jan. 24, 2018.

[3] Apache Cassandra: *Manage massive amounts of data, fast, without losing sleep*; URL: https://cassandra.apache.org/, last visited: Jan. 24, 2018.

[4] ARM Cortex-M0: *ARM Cortex-M0*; URL: https://developer.arm.com/products/processors/cortex-m/cortex-m0, last visited: Jan. 24, 2018.

[5] K. E. Arzén: *A simple event-based PID controller*; 14th IFAC World Congress, Beijing, China, 1999, pp. 423-428, URL: http://portal.research.lu.se/portal/files/6083100/8521976.pdf, last visited: Jan. 24, 2018.

[6] K. J. Aström: *Comparison of Periodic and Event Based Sampling for First-Order Stochastic Systems*; 14th IFAC World Congress, Beijing, China, 1999, URL: http://portal.research.lu.se/portal/files/6208782/8520116.pdf, last visited: Jan. 24, 2018.

[7] Boost: *Boost.Signals2*; URL: http://www.boost.org/doc/libs/1_64_0/doc/html/signals2.html, last visited: Jan. 24, 2018.

[8] X. Cai, M.R. Lyu, K.-F. Wong: *COMPONENT-BASED SOFTWARE ENGINEERING: DEVELOPMENT FRAMEWORK, QUALITY ASSURANCE AND A GENERIC ASSESSMENT ENVIRONMENT*; International Journal of Software Engineering and Knowledge Engineering, 12(2), 2002, ISSN: 0218-1940, pp. 107-133, URL: http://www.cse.cuhk.edu.hk/~lyu/paper_pdf/S0218194002000846.pdf, last visited: Jan. 24, 2018.

[9] CAMPBELL SCIENTIFIC, INC.: *Rugged Monitoring - Measurement and control instrumentation for any application*; URL: http://www.campbellsci.com/, last visited: Jan. 24, 2018.

[10] CAMPBELL SCIENTIFIC, INC.: *CRBasic Tips to Simplify Data Post-Processing*; URL: https://www.campbellsci.com/blog/crbasic-tips-simplify-data-post-processing/, last visited: Jan. 24, 2018.

[11] CAMPBELL SCIENTIFIC, INC.: *LOGGERNET - Datalogger Support Software*; URL: http://www.campbellsci.com/loggernet, last visited: Jan. 24, 2018.

[12] CMake: *Build, Test and Package Your Software With CMake*; URL: https://cmake.org/, last visited: Jan. 24, 2018.

[13] M. Compton et al.: *The SSN Ontology of the Semantic Sensor Networks Incubator Group*; Web semantics: science, services and agents on the World Wide Web, 17, 2012, ISSN: 1570-8268, pp. 25-32, URL: https://www.w3.org/2005/Incubator/ssn/wiki/images/f/f3/SSN-XG_SensorOntology.pdf, last visited: Jan. 24, 2018.

[14] CORBA: *CORBA*; URL: http://www.corba.org/, last visited: Jan. 24, 2018.

[15] Corsair: *Corsair Vengeance LPX*; URL: http://www.corsair.com/en-eu/memory/vengeance-lpx-series, last visited: Jan. 24, 2018.

[16] J. Cumming: *A C++ Object Factory*; URL: http://www.jsolutions.co.uk/C++/objectfactory.html, last visited: Jan. 24, 2018.

[17] Cypress - Embedded in Tomorrow: *PSoC 5*; URL: http://www.cypress.com/products/psoc-5, last visited: Jan. 24, 2018.

[18] D. Diamond, S. Coyle, S. Scarmagnani, J. Hayes: *Wireless sensor networks and chemo-biosensing*; Chemical Reviews, 108(2), 2008, ISSN: 0009-2665, pp. 652-679, doi: https://doi.org/10.1021/cr0681187, last visited: Jan. 24, 2018.

[19] A. Eliens: *Principles of Object-Oriented Software Development*; Addison-Wesley Longman Publishing Co., Inc., 2000, ISBN: 0-201-39856-7.

[20] The Eclipse Foundation: *Eclipse*; URL: https://www.eclipse.org/, last visited: Jan. 24, 2018.

[21] R. Fielding: *Architectural Styles and the Design of Network-based Software Architectures*; Doctoral dissertation: University of California, Irvine, 2000, URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/evaluation.htm, last visited: Jan. 24, 2018.

[22] E. Fleisch: *What is the Internet of Things*; Economics, Management, and Financial Markets, 5(2), 2010, ISSN: 1842-3191, pp. 125-157, URL: https://www.deepdyve.com/lp/addleton-academic-publishers/what-is-the-internet-of-things-an-economic-perspective-wPfDQh01at

[23] E. Gamma: *Design patterns: elements of reusable object-oriented software*; Pearson Education India, 1995, pp. 14, ISBN: 9780201633610

[24] M.R. Gartia et al.: *The microelectronic wireless nitrate sensor network for environmental water monitoring*; Journal of Environmental Monitoring, 14(12), 2012, ISSN: 0167-6369, pp.3068-3075, doi: https://doi.org/10.1039/C2EM30380A, last visited: Jan. 24, 2018.

[25] General Acoustics e.K.: *Sophisticated. Efficient Measurements*; URL: www.generalacoustics.com, last visited: Jan. 24, 2018.

[26] GCC: *GCC, the GNU Compiler Collection*; URL: https://gcc.gnu.org/, last visited: Jan. 24, 2018.

[27] A. Gercia et al.: *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*; IEEE Press Piscataway, NJ, USA, 1991, ISBN:1559370793

[28] T. Gruber: *What is an Ontology?*; URL: `http://www-ksl.stanford.edu/kst/what-is-an-ontology.html#1`, last visited: Jan. 24, 2018.

[29] J. Gubbi et al.: *Internet of Things: A Vision, Architectural Elements, and Future Directions*; Future Generation Computer Systems, 29(7), 2013, ISSN: 0167-739X, pp. 1645-1660, doi: `https://doi.org/10.1016/j.future.2013.01.010`, last visited: Jan. 24, 2018.

[30] J. Horsburgh, A.S. Jones et al.: *A data management and publication workflow for a large scale heterogeneous sensor network*; Environmental Monitoring and Assessment, 187(6), 2015, ISSN: 0167-6369, pp. 348, doi: `https://dx.doi.org/10.1007/s10661-015-4594-3`, last visited: Jan. 24, 2018.

[31] J. Horsburgh et al.: *An integrated system for publishing environmental observations data*; Environmental Modelling & Software, 24(8), 2009, ISSN: 1364-8152, pp.879-888, doi: `https://doi.org/10.1016/j.envsoft.2009.01.002`, last visited: Jan. 24, 2018.

[32] J. Horsburgh et al.: *A relational model for environmental and water resources data*; Water Resources Research, 44(5), 2008, ISSN: 1944-7973, doi: `https://doi.org/10.1029/2007WR006392`, last visited: Jan. 24, 2018.

[33] Intel: *Intel Core i7-7700K*; URL: `https://ark.intel.com/products/97129/Intel-Core-i7-7700K-Processor-8M-Cache-up-to-4_50-GHz`, last visited: Jan. 24, 2018.

[34] C. Jang et al.: *OPRoS: A new component-based robot software platform*; ETRI journal, 32(5), 2010, ISSN: 2233-7326, pp.646-656, doi: `https://doi.org/10.4218/etrij.10.1510.0138`, last visited: Jan. 24, 2018.

[35] L. Laurent, et al: *Semantic sensor network xg final report*; W3C Incubator Group Report, 28, 2011, URL: `https://www.w3.org/2005/Incubator/ssn/XGR-ssn-20110628/`, last visited: Jan. 24, 2018.

[36] S. T. Lavavej: *MinGW Distro - nuwen.net*; URL: `https://nuwen.net/mingw.html`, last visited: Jan. 24, 2018.

[37] S. Mailet: *Using COM for embedded systems*; Proc. Embedded Systems Conference (504), San Francisco, 2001.

[38] B. Meyer: *Object-Oriented Software Construction*; Prentice Hall, New York, USA, Vol. 2, 1988, ISBN: 0-13-629155-4

[39] S. Meyers: *Effective Modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14.*; O'Reilly Media, Inc., Sebastopol, Canada, 2014, ISBN: 978-1-491-90399-5

[40] S. Meyers: *Overview of the New C++ (C++11/14)*; URL: https://www.artima.com/shop/overview_of_the_new_cpp, last visited: Jan. 24, 2018.

[41] S. Meyers: *Effective C++ in an Embedded Environment*; URL: https://www.artima.com/shop/effective_cpp_in_an_embedded_environment, last visited: Jan. 24, 2018.

[42] W.K. Michener: *Meta-information concepts for ecological data management*; Ecological Informatics, 1(1), 2006, ISSN: 1574-9541, pp. 3-7, doi: https://doi.org/10.1016/j.ecoinf.2005.08.004, last visited: Jan. 24, 2018.

[43] Microsoft: *Microsoft*; URL: https://www.microsoft.com/de-ch/, last visited: Jan. 24, 2018.

[44] Microsoft: *Visual Studio*; URL: https://www.visualstudio.com, last visited: Jan. 24, 2018.

[45] Microsoft: *The Component Object Model: Technical Overview*; URL: https://www.cs.umd.edu/~pugh/com/, last visited: Jan. 24, 2018.

[46] Microsoft: *Component Object Model (COM)*; URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680573%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396, last visited: Jan. 24, 2018.

[47] Microsoft: *OLE Concepts and Requirements Overview*; URL: https://support.microsoft.com/en-us/help/86008/ole-concepts-and-requirements-overview, last visited: Jan. 24, 2018.

[48] Microsoft: *Introduction to ActiveX Controls*; URL: https://msdn.microsoft.com/en-us/library/aa751972(v=vs.85).aspx, last visited: Jan. 24, 2018.

[49] Microsoft Wikia: *Component Object Model*; URL: http://microsoft.wikia.com/wiki/Component_Object_Model, last visited: Jan. 24, 2018.

[50] Microsoft: *Microsoft Visual C++*; URL: https://www.visualstudio.com/de/thank-you-downloading-visual-studio/?sku=BuildTools&rel=15, last visited: Jan. 24, 2018.

[51] Microsoft: *Visual Studio Code*; URL: https://code.visualstudio.com/, last visited: Jan. 24, 2018.

[52] NMEA: *NMEA 0183 Standard*; URL: http://www.nmea.org/content/nmea_standards/nmea_0183_v_410.asp, last visited: Jan. 24, 2018.

[53] OceanInstruments: *Acoustic Doppler Current Profiler*; URL: https://www.whoi.edu/instruments/viewInstrument.do?id=819, last visited: Jan. 24, 2018.

[54] OceanWise: *OceanWise*; URL: https://www.oceanwise.eu/, last visited: Jan. 24, 2018.

[55] OMG: *Object Management Group*; URL: www.omg.org, last visited: Jan. 24, 2018.

[56] S. Orlov, N. Melneikova: *Compound object model for scalable system development in C++*; Procedia Computer Science, 66, 2015, ISSN: 1877-0509, pp. 651-660, doi: https://doi.org/10.1016/j.procs.2015.11.074, last visited: Jan. 24, 2018.

[57] ODMToolsPython: *ODMToolsPython*; URL: https://github.com/ODM2/ODMToolsPython, last visited: Jan. 24, 2018.

[58] Ontology: *Ontology*; URL: https://en.wikipedia.org/wiki/Ontology_(information_science), last visited: Jan. 24, 2018.

[59] OWL: *Web Ontology Language*; URL: https://www.w3.org/2001/sw/wiki/OWL, last visited: Jan. 24, 2018.

[60] Phytec: *Phytec*; URL: http://www.phytec.de, last visited: Jan. 24, 2018.

[61] Phytec phyBOARD: *phyBOARD-Segin*; URL: http://www.phytec.de/produkt/single-board-computer/phyboard-segin/, last visited: Jan. 24, 2018.

[62] Phytec phyCORE: *phyCORE-i.MX 6UL*; URL: http://www.phytec.de/produkt/system-on-modules/phycore-imx-6-ul/, last visited: Jan. 24, 2018.

[63] PostgreSQL: *The world's most advanced open source database*; URL: https://www.postgresql.org/, last visited: Jan. 24, 2018.

[64] PostgreSQL: *NOTIFY*; URL: https://www.postgresql.org/docs/9.0/static/sql-notify.html, last visited: Jan. 24, 2018.

[65] PT100: *Temperaturmessung mit Pt 100- Widerstandssensoren*; URL: http://www.pt100.de/, last visited: Jan. 24, 2018.

[66] RDF: *Resource Description Framework*; URL: https://www.w3.org/2001/sw/wiki/RDF, last visited: Jan. 24, 2018.

[67] Riak TS: *Ensuring High Performance for IoT Applications*; URL: http://basho.com/products/riak-ts/, last visited: Jan. 24, 2018.

[68] S. Rombauts: *SQLiteC++*; URL: https://github.com/SRombauts/SQLiteCpp, last visited: Jan. 24, 2018.

[69] Samsung: *Samsung SSD 960 Pro*; URL: http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/960pro.html, last visited: Jan. 24, 2018.

[70] Sun Microsystems: *Enterprise JavaBeans*; URL: http://www.inf.fu-berlin.de/lehre/WS99/project_vis/ejb.10.pdf, last visited: Jan. 24, 2018.

[71] H. Sutter, A. Alexandrescu: *C++ Coding Standards: 101 Rules*; Pearson Education, 2004, ISBN: 978-0321113580.

[72] H. Sutter, B. Stroustrup: *C++ Core Guidelines*; URL: https://github.com/isocpp/CppCoreGuidelines, last visited: Jan. 24, 2018.

[73] SQLite: *SQLite*; URL: https://sqlite.org/, last visited: Jan. 24, 2018.

[74] StackOverflow: *Improve INSERT-per-second performance of SQLite?*; URL: https://stackoverflow.com/questions/1711631/improve-insert-per-second-performance-of-sqlite, last visited: Jan. 24, 2018.

[75] TSA: *Time Series Analyst*; URL: http://data.iutahepscor.org/tsa/, last visited: Jan. 24, 2018.

[76] B. Wallace: *A hole for every component, and every component in its hole*; Existential Programming, 2010, URL: http://www.existentialprogramming.com/2010/05/hole-for-every-component-and-every.html, last visited: Jan. 24, 2018.

[77] Wikipedia: *Component-Based Software Engineering*; URL: https://en.wikipedia.org/wiki/Component-based_software_engineering, last visited: Jan. 24, 2018.

[78] B.P. Wong, B. Kerkez: *Real-time environmental sensor data: An application to water quality using web services*; Environmental Modelling & Software, 84, 2016, ISSN: 1364-8152, pp.505-517, doi: https://doi.org/10.1016/j.envsoft.2016.07.020, last visited: Jan. 24, 2018.

[79] Xively: *IoT Platform for Connected Devices*; URL: https://www.xively.com/, last visited: Jan. 24, 2018.

[80] S. S. Yau, B. Xia: *Object-oriented distributed component software development based on CORBA*; Proc. 22 Computer Software and Applications Conference (COMPSAC'98), Vienna, Austria, ISSN: 1998, 0730-3157, pp. 246 251. URL: https://doi.org/10.1109/CMPSAC.1998.716662, last visited: Jan. 24, 2018.

[81] Yocto Project: *Poky*; URL https://www.yoctoproject.org/tools-resources/projects/poky, last visited: Jan. 24, 2018.

Abbreviations

AD	analogue-digital
ADCP	Acoustic Doppler Current Profiler
API	Application Programming Interface
CDG	Chunk Data Generator
CID	Channel ID
COM	Common Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CUAHSI	Consortium of Universities for the Advancement of Hydrology Science, Inc.
DCOM	Distributed Common Object Model
DDR4	Double Data Rate Fourth-Generation
DHU	Data Handling Unit
DOM	Document Object Model
DSU	Data Store Unit
EJB	Enterprise JavaBeans
GA	General Acoustics e.K.
GAPP	General Acoustics Property Protocol
GATP	General Acoustics Table Protocol
GUID	Globally Unique Identifiers
HIS	Hydrologic Information Systen
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language
IO	input-output
IoT	Internet of Things
iUTAH	innovative Urban Transitions and Aridregion Hydrosustainability
JVM	Java Virtual Machine
MVC	Model View Controller
NMEA	National Marine Electronics Association
ODM	Observational Data Model
OMG	Object Management Group
OO	Object Oriented
OPRoS	Open Platform for Robotic Service
ORB	Object Request Broker
OS	Operating System
OWL	Web Ontology Language

PCB	Printable Circuit Board
PDU	Primary Data Unit
PPS	Pulse Per Second
QA	Quality Assessment
QC	Quality Control
RDF	Resource Description Framework
REST	Representational State Transfer
RTC	Real Time Clock
RTU	Real Time Unit
SID	Sensor ID
SOM	Socket on Module
SPI	Serial Peripheral Interface
SQL	Structured Query Language
SSN	Semantic Sensor Network
TCP/IP	Transmission Control Protocol/Internet Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifiers
UTC	Coordinated Universal Time
W3C	World Wide Web Consortium
WAL	Write Ahead Log
XML	Extensible Markup Language

Glossary

Datacenter A datacenter in the context of this thesis, is a location in the GA workflow, where databases from different measurement sites are brought together.

GAPP Definition A GAPP message that is used to set a property value. If the value has been set successfully, a definition message is sent back to the sender.

GAPP Observer A GAPP engine that observes a GAPP data stream from a GAPP source. It can send definition and query messages.

GAPP Property The values that are used by GAPP. The properties model sensor values or system settings in the workflow.

GAPP Query A GAPP message that is used to query a property value. A definition is returned to the sender.

GAPP Source A GAPP engine that is the proprietor of GAPP properties. It is not allowed to send query messages to the listening observers.

Measurement Site A measuring location that contains at least one station server and a station.

Primary Data Describes preprocessed data. Distinction between primary and secondary data is context depending. Secondary data on a station may be primary data on a station server.

Raw Data Raw sensor data values, typically counter values or analogue voltages that have to be preprocessed into primary data.

RESTful Something is RESTful if it conforms to the characteristics of Representational State Transfer.

Secondary Data Postprocessed primary data. Distinction between primary and secondary data is context depending.

Station A measuring location in the GA workflow. Consists of a group of sensors and a sensor controller.

Station Server A server, mostly onshore, to process, store, and visualize sensor data in a current GA workflow.

List of Figures

List of Tables

List of Listings

Appendix A

Ultrasonic Level Measurement

This section describes the ultrasonic level measurement technology. A large part of the GA product lineup depends on this technology, therefore, continuous efforts exist at GA to develop and improve the technology.

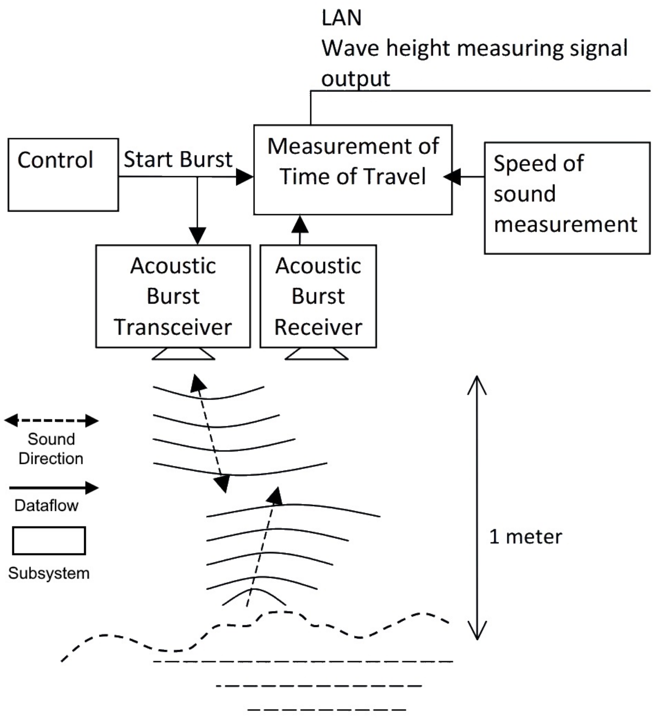
LAN
Wave height measuring signal output
Control
Start Burst
Measurement of Time of Travel
Speed of sound measurement
Acoustic Burst Transceiver
Acoustic Burst Receiver
Sound Direction
Dataflow
Subsystem
1 meter

Figure A.1: Ultrasonic wave measurement.

Figure A.1 shows a schematic drawing of the functional principle. A sound burst is emitted from an acoustic transceiver in direction of the water surface where it is reflected. The reflected sound waves are read by an acoustic receiver. The elapsed time of travel is measured between the start of the burst and its receiving. At the same time, a reference sensor measures the exact speed of sound. The distance between the burst transceiver and the water surface can then be calculated with the time of travel and the speed of sound. The UltraLab ULS Advanced is able to measure a range of 200 to 1000 millimeter at a resolution of 360 microns with an uncertainty smaller than one millimeter with the use of this technology. A relative wave speed of around 14 m/s can be measured with a frequency of 100 Hz. In an ideal scenario, one perfect echo will be returned from a ping. In reality,various echoes are returned, external factors lead to a series of echo types.

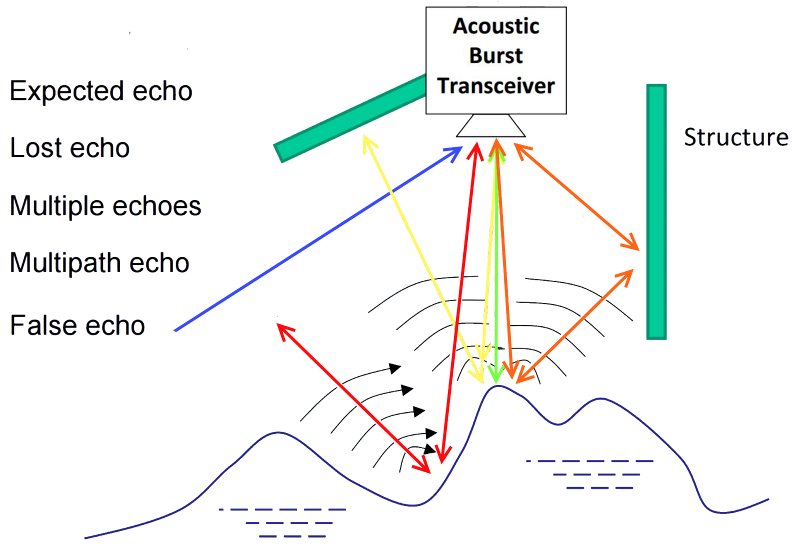

Figure A.2: Ultrasonic wave measurement echo types.

Figure A.2 shows a sketch of possible echoes. Green, the expected one-time reflected echo. Red shows a lost echo, due to the angle in which the sound burst hits the water surface, the waves are reflected away and cannot be received anymore. Sometimes the sound waves are split when they hit the water surface. One part returns as an expected echo, the other part can reflect somewhere in the environment, travel back to the water surface, and hit the receiver with the correct angle but with a much higher runtime. These are called multiple echoes and are drawn in yellow. Multipath echoes, shown in orange, exist if sound waves reflect in the wrong angle, hit other objects in the environment and finally reach the receiver with a wrong angle, but maybe with a reasonable running time. The last possible echoes are false echoes, drawn blue, where sound waves floating through the air are wrongly detected as echo by the receivers. The described echoes can exist under normal conditions. When special environmental conditions are present, other cases of error prone measurements exist. Heavy rain or extreme fume can lead to premature reflections without reaching the water surface. Such errors typically result in huge outliers in the final data.

The huge amount of probably erroneous echoes that can be received for a single measurement, demands for evolved preprocessing algorithms. Especially complex is the determination of the correct echo from a range of echoes. The use of an improved track finding algorithm (Akima Interpolation [1]) has helped significantly in rising the measurement accuracy. Besides the selection of the correct echo, an outlier fixer algorithm based on linear wave theory and statistical error calculation is used to minimize the effects of incorrect measurements.

With the use of multiple acoustic burst transducers at the same time, the quality of the measurements could be further improved. It allows to minimize the reflection area on the water surface and can generate a higher acoustic pressure for the measurements. Besides that, the use of multiple transducers enables the possibility to measure wave directions. By arranging three transducers in triangle form, each transducer measures the peak of a wave at a slightly different time. This time shift can be used to calculate the direction of the wave.

With the new developed and more powerful hardware, more advanced algorithms can be run in real-time. Therefore, the technology is further developed, and brought to its limits. With the processing of more parallel measurements, 3D wave or wave field sampling is a possible development direction.

Appendix B

Workflow Concept: Related Work

This Chapter presents the related work of the workflow concept introduced in Section 2.3. Currently relevant research, and solutions under the topic of environmental sensor data management are introduced in Section B.1 and Section B.2.

B.1 Environmental Sensor Data Management

The rise of reliable, low-cost wired or wireless communication technologies in recent years, permits devices to be connected [22] [29]. This technological progress was also applied to environmental sensor systems and led to "ubiquitous use of sensors and sensor networks in environmental monitoring" [30]. The enhanced connectivity of the sensing technologies enables use case scenarios with lot of spatially distinct measuring locations in rough terrain where regular on-site visits are not possible.

The reasons behind the usage of environmental sensing systems vary depending on the area of interest. Two environmental monitoring application domains can be noticed. In academia, the possibilities offered through complex spatial distributed environmental monitoring are leveraged for research efforts, e.g., to observe and understand environmental changes introduced by urbanizing areas or by climate changes. The second area where environmental monitoring is increasingly applied is in the commercial sector.

The heavy use of environmental observation systems to collect data, in case of sensor networks, data with spatial distribution, introduces challenges to handle the huge amount of data. Not only the data acquisition may be troublesome due to the distributed data sources and the high data volumes, but also challenges to preprocess, analyze, correct, and publish the incoming data have to be solved. In academia, the solutions to these challenges result as byproduct of the research where environmental observation systems were used. The solutions are published and describe the used workflows. Some publications present a generalized solution that can be applied to other environmental sensing applications [30] [78]. The research efforts to solve the difficulties of environmental sensor data is focused on the development of an automated data collection, processing, and presentation workflow.

Two different approaches to the management of environmental sensor data can be found. One approach is more concerned with the handling of continuous time-series data, where a real-time approach is not important. The second approach is focused on real-time handling of data to accomplish different needs, e.g. triggering actions based on measured values.

A range of publications by Horsburgh et al. describe the workflows and tools used to manage sensor data in the context of a case study from the innovative Urban Transitions and Aridregion Hydrosustainability (iUTAH) sensor network. The concrete workflow for the iUTAH network was published in 2015 [30]. Three river sites were observed by multiple stations, the Logan River, the Provo River, and the Red Butte Creek. On each observation site, a Campbell Scientific, Inc. [9] data logger was used to collect the data from the sensors. Each logger sends the data to a LoggerNet [11] base station where text based files of the data are generated. A python script loads the sensor data from the text files, converts them into a custom Observational Data Model (ODM) by the Consortium of Universities for the Advancement of Hydrology Science, Inc. (CUAHSI) Hydrologic Information System (HIS) [32] and inserts them into a relational ODM database for each river site. Additional tools in Python [57] provide quality control and processing capabilities that work directly on the databases. A web application [75] is used to present the data to the public. The core of this workflow is the ODM, designed by the CUAHSI HIS to "store observational data along with complete meta-data to facilitate unambiguous interpretation of data" [31]. The ODM is further described in Section B.2 where generic ways to model environmental data are introduced. The data loader, that fills the database, is run as a Windows Task every hour. Therefore, the target of this environmental data management solution lies on the storage and presentation of measured data and not on real-time measurements.

In contrary to Horsburgh et al.'s solution, Wong and Kerkez [78] present a workflow that relies on the real-time character of measured environmental data. The solution was developed around a use case to measure water quality in streams and rivers. Sensor technology, able to measure water quality like nutrients or bacteria on-site, is not available or expensive [24] [18], therefore, automated samplers are used to mechanically draw and store water samples. The number of samples, that can be collected by an automated sampler, is limited, and the power consumption to collect a sample due to the motorized mechanical components is huge [78]. The sample times and sample frequencies have to be optimized to be able to effectively capture events of interest. Wong and Kerkez used an adaptive sampling [5] [6] approach, where "a controller or algorithm persistently updates a model of a phenomen using realtime data and then samples only during event of interest" [78]. The environmental data workflow was centered around the Xively [79] IoT cloud platform. The Xively platform provides a comprehensive C library, which provides access to the online services over a RESTful application programming interface (API). The sensor nodes connect to the IoT platform over feeds. In the case of Wong and Kerkez, a CSV or JSON data format was used to send the sampled data over Transmission Control Protocol/Internet Protocol (TCP/IP) to Xively. An adaptive sampling controller, written in Python and also connected to the Xively cloud, analyzed the arriving data. Depending on weather forecasts from the Internet and the current sampling values, the sampling frequency and sleep phases were automatically adapted. A web application was implemented with the help of the Xively JavaScript API to visualize the data. The used workflow could react in seconds to a changing environment. When the water level rises spontaneously due to a thunderstorm, the collection of water samples could be initiated instantly.

Both examples of environmental data management workflows are built around existing data management components. Horsburgh et al. rely on advanced Campbell Scientific, Inc. data logger and built their workflow on top of that. Wong and Kerkez use Xively as cloud platform and attach custom components to it, to create their workflow. A complete development of a full workflow for a single project may not be economical.

In the commercial sector, the need for environmental data management solutions has been recognized. The efforts to satisfy the needs differ depending on the use case. Campbell Scientific, Inc. [9] is an international group of companies around the world that is specialized in scientific instrumentation. Their series of data collection and data logging hardware is highly advanced and easily configurable, extendable, and usable for a wide range of applications. Besides data loggers, they provide communication modules, power supply and management systems, a wide range of sensing hardware, cases and mounting hardware for the their products, as well as a data management and presentation system for the measured data [9]. It is possible to build a complete environmental observation workflow with Campbell Scientific, Inc. components. The only drawback comes from the price level of these advanced hardware. GA has used a range of their products for bigger projects, e.g. a data logger to collect data from a huge number of sensors in cases where the current LOG_aLevel could not satisfy all needs.

Another company that provides environmental data handling solutions is OceanWise [54]. OceanWise has specialized in handling marine environmental data. They present solutions that range from the acquisition, management up to publishing of environmental data. In contrary to Campbell Science, Inc. they are more focused on coastal and marine data, and not on environmental data in general. Therefore, their business field is similar to the ecosystem where the market niche of GA is located. Nonetheless, GA is focused on the development of specialized ultrasonic sensing hardware, which is complemented by custom workflows to provide a complete solution that fulfills customer requirements. Whereas OceanWise does not provide measuring hardware, and is only concerned with offering a complete workflow that starts with data logging. A proprietary database system is used to store the environmental data. The database can be used on-premise on a local server, or subscription based as a cloud, where it is offered as a service.

Further on, service provider like Xively emerge more frequently. Their use case is the management of a wide range of big data. A commercial example is Amazon AWS IoT [**?**], which is a managed cloud platform where connected devices can communicate securely with cloud applications or different devices. Wong and Kerkez analyze and compare a list of current IoT management providers [78]. Their main argument for the use of cloud based data management solution providers is the simplicity of use and, therefore, the ability to focus on the application development instead of system development and administration. A drawback of such solutions is their cloud dependency. In security relevant use case scenarios, such a solution may not be acceptable due to the loss of full data control. Further on, these solutions do not deliver a full data management workflow and can rather be seen as a possible data storage and presentation step in a workflow, as it was used by Wong and Kerkez. GA has had regular projects with customers from the government sector with high security requirements. Therefore, such cloud based solutions can not, and will not be considered further on.

In conclusion of the existing environmental data management workflow solutions, it can be said that commercial providers try to deliver a complete data handling experience for their systems. Some provide a relational database to handle environmental data, mostly wrapped in closed source applications [54]. Complete solutions from a single provider can be costly and result in a lock-in effect. Data management workflows in academia are more focused on an open source approach and emphasize an information sharing strategy. Therefore, the development process of these workflows were published and can be studied. Two main workflow types can be found, one is concerned with the structured management of environmental sensor data in the form of time-series. The other workflow type is focused on the handling of real-time data, with the ability to react and trigger events based on the measured real-time values.

Throughout most publications, an emphasis is put on the use of meta-data to bring measured data into context. The next section shows the reasons for the need of meta-data, and drives into detail how environmental data is modeled and stored in current research.

B.2 Environmental Sensor Data Modeling

Section B.1 introduced current research and workflows providing solutions to handle the difficulties that emerge from the ubiquitous use of environmental sensing technologies. The presented solutions focus on a workflow and data handling perspective of environmental observation technologies. This section is only concerned with the data that is produced by sensors or sensing systems in the environmental environment. First, the characteristics of environmental sensing data are introduced. Second, current ontologies that model environmental data are introduced. Finally, the importance of meta-data for environmental data is emphasized.

The environmental data produced by sensors and sensor networks follows certain characteristics. The vast majority of environmental sensor data are continuous time-series with changing values over time and a fixed sensing location. Sensors for temperature, humidity or level measurement typically emit series of `<timestamp, value>` pairs. Other sensors may emit time-series that encode spatial distribution of measured values. An ADCP [53], for example, measures the water current at different water levels and produces tuples containing a timestamp and a range of vectors with the flow direction and velocity. Even though these complex systems measure distinct locations, as long as the point of measurement is fixed (where the device is located), the data can be seen as a location independent series of tuples in the form of `<timestamp, v1, v2, ..., vN>`. Such time-series producing measurement systems are used to gain continuous long-term data and are heavily used in sensor networks.

Besides data in time-series form, time independent environmental observation data exists. Such data can be gathered e.g. by measuring or taking samples at various locations. In contrast to the measurement at a fixed location, location specific data has to be collected by actively moving the measurement devices. An example in the GA field is the Sub-Bottom Profiler. By moving the device around, a two- or three dimensional image of the layers under the earth is produced. An application is the bottom profiling of sea ports to detect bottom surface structures and the characteristics of the underlying layers. The generated data is bound to one, or maximal two timestamps, the start of the measurement and eventually the end of the measurement. The measured values change over distance, and not over time. A clear distinction between spatial- or temporal oriented data cannot be made, spatial discrete and time independent data may be brought into the context of a time depending scenario by adding additional meta information. In cases of physical samples, e.g. soil samples, the sampling time point may be important if the sample results are evaluated together with time-series data, e.g. temperature or rainfall statistics, to analyze the bio-chemical reaction of the soil after events like thunderstorms [78]. Samples, that have to be analyzed in a lab, do not only need their collection timestamp, but also the timestamp of the lab examination.

Environmental observation data are for the most part continuous long-term time-series with a fixed measurement location. Determined by the use case, environmental data may be location critical but not time critical or the data may be both, spatial- and time oriented but to various degrees. Despite most data follows the time-series 'category', the various spatial and temporal kinds of data introduce difficulties to uniformly handle environmental data.

Not only the different kinds of environmental sensor data impose problems to the handling of the data. Every sensor system outputs the measured values differently. Sensors like a PT100 temperature sensor [65], which is essentially a temperature depending platinum resistor, measure a voltage proportional to the resistor value. At some point this analogue voltage value has to be converted to a digital number. Such a conversion from the raw analogue values to digital values (AD) is common for most sensing systems. At GA, so called 'digital' sensors, are sensors that create digital values internally, add a timestamp, and output them over standardized interfaces like TCP/IP or serial in text or binary form. The 'analogue' sensors, e.g. a PT100, have to be embedded in a circuit that performs the AD conversion and the time-stamping. Analogue sensors are mostly passive and have to be actively triggered to measure a value, whereas digital sensors are generally active and measure in a certain interval. The heterogeneity of the sensors and the fact, that each sensing system implements a custom protocol to transfer the data, makes it difficult to manage sensor data in general, and not only in the environmental field.

Section B.1 introduced the problems of environmental sensing networks, e.g. the huge amount of data, sometimes a real-time data handling aspect. In the process of handling this data, the kind of the data and the heterogeneity of its sources, described above, have to be considered. But also, data quality considerations have to be made. Sensor data may be noisy or faulty due to measurement interference, e.g. rain that reflects ultrasonic waves before they hit the target. Measurement errors may happen, e.g. if the power supply can not deliver enough current, or errors in the communication may corrupt correctly measured values. All of these data uncertainty has to be considered and handled by an environmental data management workflow. Some solutions, presented in Section B.1, try to solve those challenges by looking at semantics of the data [30]. They created data models of the data to simplify the representation and management of the data. The rest of this section presents research, how environmental sensor data with all its characteristics can be modeled and represented.

A semantic model of environmental data is called an ontology. An ontology in the context of information science is "an explicit specification of a conceptualization" [28] or more generally, "a formal naming and definition of the types, properties, and interrelationships of the entities that really or fundamentally exist for a particular domain of discourse" [58]. In 2009 the World Wide Web Consortium (W3C) started an incubator group to study semantic sensor networks (SSN). They analyzed a series of existing ontologies and created a new one based on the gathered information [13]. They came up with a complex data model shown in Figure B.1 [35]. The data model is not only focused on the **measurement results** (marked blue), but mainly on additional information called meta-data. Platform sites or deployment events are modeled, information of the sensor device, its characteristics and the available processes are covered. A **property** entity (marked red) encapsulates information on the measurement properties like frequency, resolution, latency, battery lifetime or measurement range.

This, albeit complex system, allows to describe a concrete sensor network in a structured way. W3C uses the Web Ontology Language (OWL) [59], which is oriented at the Resource Description Framework (RDF) [66] to represent its ontology specification.

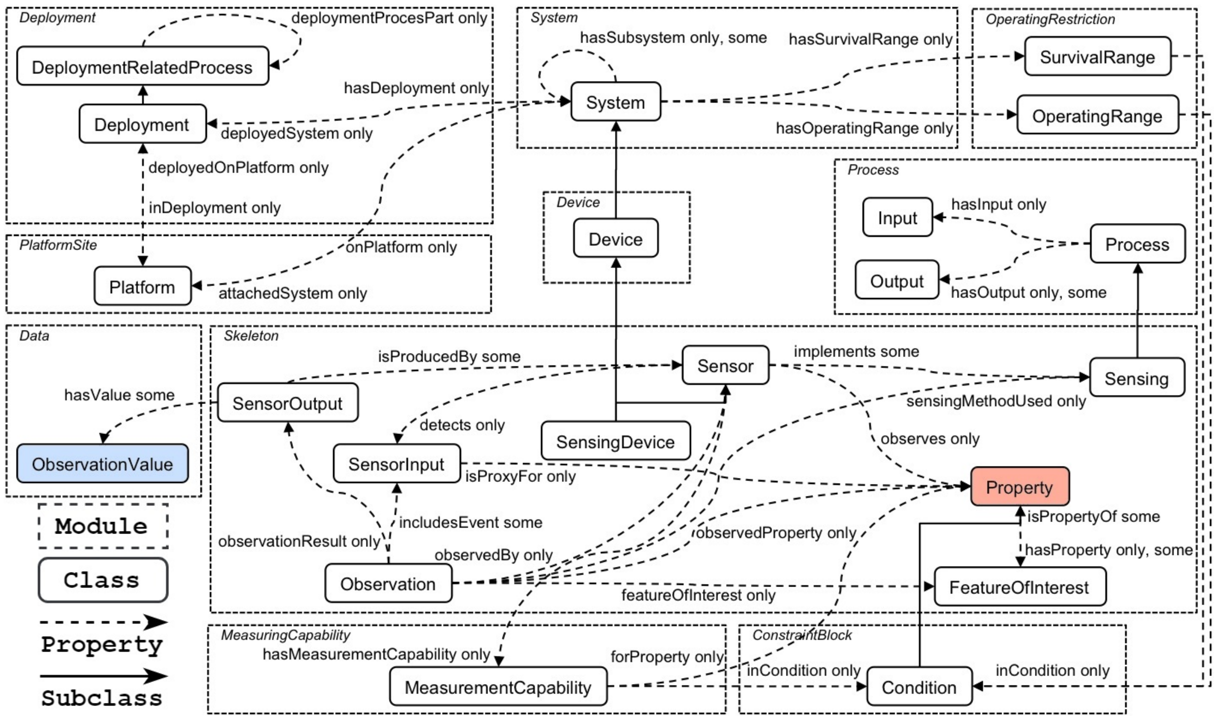

Figure B.1: SSN ontology database model.

In their report [35] every component is described in detail with examples how a sensor network can be described and how it can be used. The completeness of this ontology comes with the price of a huge description overhead, but allows for simple automated querying on the description files and the data. It would be simple to create a relational database schema out of this ontology to further facilitate the usefulness of such a semantic description system.

Horsburgh et al. developed a similar ontology with CUAHSI HIS and used it in the workflow described in Section B.1. It was called the Observational Data Model (ODM) [32] and is presented as a database schema. Similar to the SSN, a strong emphasis is put on the describing meta-data. The complex data schema was developed as a database schema that can be applied to most available database systems [30]. The core of the ODM are observations. An observation contains two elements: "an **Action** performed on or at a **SamplingFeature** that produces an observation **Result**, and a **Result** that is the outcome of that **Action**" [32]. Diverse schema extensions exist to widen the meta-data description possibilities. An extended SamplingFeature schema allows the addition of measurement cite information, e.g. location and name. The Result extension adds result types for general measurement results (single observed value), time-series results of a continuous measurement at a fixed site, section results (a series of values observed over varying X (horizontal) and Z (vertical/depth) offsets for a variable) or transect coverage result (a 2-dimensional transect line with varying x or y values).

The ODM was designed to be able to handle almost all possible kinds of environmental data. Even subsample structures can be modeled accordingly. The goal was a simple data management solution that can be used for a wide range of scenarios, and allows simple data sharing and exchange. The events can describe additional actions on the data, e.g. Quality Assessment (QA) or Quality Control (QC) where the data is tested for correctness, and possibly corrected.

In both shown data models, the importance of meta-data can be seen. Michener [42] highlighted the importance and use of meta-data. In research, the data discovery is simplified by the addition of meta information. Web-based data archives become searchable based on keywords. Added meta-data can also simplify the data gathering, data understanding and use of the data by humans, the additional information may provide research context, status of the data set or at least the physical structure of the data. And lastly, the added meta-information enables automated data discovery, processing and analysis. [42]

Figure B.2 introduces an end-to-end workflow that can be supported by the automated addition of meta-data at each step. Therefore, the history of the data is completely visible, which helps in the understanding of the final results. Through the meta-data history, it is possible to revert some steps in cases where errors were made, even years after the initial measurements.

To summarize this section, it can be said, that various kinds of environmental sensing data exist. Each kind of data comes with its own challenges that have to be dealt with. Various efforts exist to model environmental data in order to simplify the management of this data. Both described environmental data modeling solutions make extensive use of meta-data to structure the data. The SSN ontology is more focused on a semantic approach that allows RDF-like syntax to describe environmental measurement networks and the resulting data.

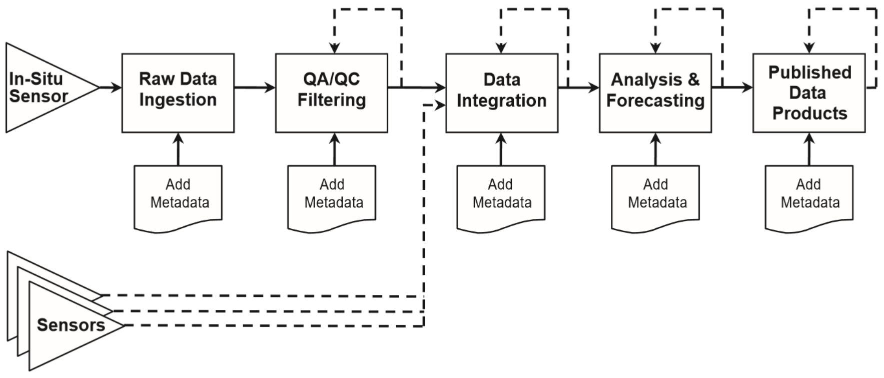

Figure B.2: Meta-data Workflow.

The ODM data scheme is focused on a database oriented approach and allows hierarchical sampling structures of all kinds of observations. Overall, the opportunities from the use of meta-data reach from enhanced understanding of the data, to optimized automated workflows. The benefit of meta-data, that may be of importance for the GA environment, is the fact, that the history of the data, including all modifications, is available after the sampling is completed and post processing operations have finished.

Appendix C

Contents of the CD

- **01_source_code** - Contains the final source code with built binaries.
- **02_tex_report** - Contains the source code of the report, including the images and the final printed PDF.
- **03_presentation** - Contains the pptx from the presentation.
- **04_other** - Contains various documents and files used for this project.
  - **cd_label** - The digital DC label used to print the CD's,
  - **measurements** - All relevant files that were used for the performance measurements.