Jarol - A Java Control Infrastructure

Magisterarbeit zur Erlangung des akademischen Grades
Diplom-Ingenieur der *Angewandten Informatik*
Angefertigt am
Institut für Computerwissenschaften und Systemanalyse
der Naturwissenschaftlichen Fakultät
der Paris-Londron-Universität Salzburg

Eingereicht von
Bernhard Kast, Bakk.techn.
Eingereicht bei
Univ. Prof. Dr. Ing. Dipl. Inform. Christoph Kirsch

for my family - Gertrude, Siegfried[1], Barbara and Waltfried[2].

für meine Familie - Gertrude, Siegfried[1], Barbara und Waltfried[2].

[1] ✝1992
[2] ✝2002

Abstract

We present *Jarol*, a control infrastructure written entirely in Java. It provides developers of control systems with facilities to abstract away hardware details and use of different language-supported concurrency models. Using Java, the development of the control code is supported by strong typing, language-based concurrency support, and dynamic memory management. To ensure the interfacing with different platforms and concurrency models, we derived the specific interface requirements of such platforms and models. This allows the explicit elaboration of the data and control flow at the interfaces, where platform is defined as a hardware configuration together with a real-time operating system. We show that Jarol is capable of interfacing different platforms together with different concurrency models for execution of control code. As platforms we used a model helicopter and a model submarine, for the concurrency models we used Java threads and Exotasks [27].

Contents

List of Figures

Personal Statement

This thesis marks the end of my education as computer scientist for now. I really enjoyed the research that lead to this thesis, and the work in the Computational Systems Group and its collaborating students and researchers. Farewell, I will carry on the work ethic and team spirit that guided me through the last months.

I want to thank Christoph for his advising, guidance, support and his commitment to science, students and the university. Rainer for his explanations of the JAviator, answering my questions and supporting me with my thesis. I am also grateful to Eduardo for his help with the implementation, design, thesis and various discussions. Harald for his immediate explanations of my questions, especially with the Gumstixs and Exotasks. I want to thank Ana for proof reading and useful tips. Ricardo for providing vital information on the Seascout and waving the flag of Heavy Metal high! Thanks to Josh for his help on the Exotasks. Silviu, Robert, Hannes, Horst and Leo for various discussions and conversations.
Outside the group I want to thank Thomas for rocking the world with me since the dark ages. And of course, Josef for numerous discussions about communications, social dynamics and leadership, resulting in a vast improvement of my well-being and thus scientific output.
My deepest thanks go to my mother and my sister who supported me my whole life and especially during my studies. I also want to thank those who can not see this day, my father and Waltfried.

I wish you all the best and that you live your dreams...

because we can!

Europe, 11th May 2007

Chapter 1

Introduction

In programming control systems one faces the challenge that platforms ("platform" stands for "a hardware configuration together with a real-time operating system" [33]) traditionally only provide weak hardware abstractions and usually only support low-level programming languages. Such languages typically lack strong typing, high-level memory management, and provide limited language-based concurrency concepts in the implementation. Working on such platforms entails problems for the application writer who has to deal with the underlying hardware and restrictions imposed by low-level abstractions.

Our work is aimed at providing hardware abstractions and high-level programming concepts in this process. For this purpose, we propose an infrastructure called *Jarol* that provides facilities to abstract away hardware details and allow the use of different language-supported concurrency models. Furthermore, this infrastructure is written in the high-level programming language Java that enables strong typing, language based concurrency support, and dynamic memory management. We have chosen Java, because it offers all these features. Moreover, considering recent developments like Exotasks [28] [27], we claim that Java will eventually become real-time.

To ensure the interfacing with different platforms and concurrency models, we derived the specific interface requirements of such platforms and models, hence allowing the explicit elaboration of the data and control flow at the interfaces. Therefore, we designed *ports* (data) and *signals* (control), where ports are directed data exchange points, whereas signals are synchronization points. Ports allow the communication between two threads using a non-blocking lock-free message passing scheme. Signals allow threads to synchronize by waiting and signaling each other, using strict semantics. This means that any operation on a signal either succeeds or generates an exception, thus preventing silent failures.

Jarol concepts allow to interface different concurrency models and to provide the abstraction of the hardware. This permits the developer to

(1) focus on the implementation of the control code,

(2) use the benefits of Java,

(3) utilize the features of different concurrency models, and

(4) take the initial step to migrate the whole software architecture to Java.

The structure of the thesis is as follows. First we examine the platforms to derive the necessary requirements for interfacing with different platforms. This examination describes the JAviator and Seascout project (Chapter 2). We proceed with an analysis of Java threads and Exotasks to gather information needed to create an interface that incorporates the requirements of platforms and concurrency models (Chapter 3). In Chapter 4 we describe the abstract design of the core concepts we derived to fulfill these requirements. This includes the layering of the Jarol infrastructure and the description of signals, ports, links, and other concepts. We give an overview and discussion of the resulting Java implementation in Chapter 5. In Chapter 6 we show the application of the infrastructure with different platforms and concurrency models. This includes the discussion of the *Jarol JAviator* and *Jarol LAUV* that both use Java threads. Finally, we present the *ExoLAUV* which uses Exotasks to execute the control code. In the conclusions we summarize this thesis (Chapter 7).

1.1 Outline of the Thesis

Chapter 1, Introduction: The introduction gives an outline of this thesis and its terms.

Chapter 2, Platforms: In this chapter we give an overview about the used platforms. We introduce the hardware and software platforms of the projects. We summarize the key disparities, from which we derive the challenge definition and the proposed solution.

Chapter 3, Concurrency Models: The concurrency models that are interfaced with Jarol are presented in Chapter 3. We describe the benefits and drawbacks of Java threads and Exotasks. This allows us to adapt the proposed solution to the needs of the different concurrency models.

Chapter 4, Design: In Chapter 4 we present the concepts that we derived to meet the challenge of designing a general platform and concurrency interface.

Chapter 5, Implementation: The implementation of the concepts in Java is the content of the Chapter 5. It gives an overview about the Jarol infrastructure and discusses the major algorithms.

Chapter 6, Application: We give a presentation of the Jarol application with the JAviator and Seascout projects. Showing that it is possible to use the Jarol infrastructure with both projects and different concurrency models.

Chapter 7, Conclusions: The last chapter summarizes the contents of this thesis.

1.2 Main Contributions

My personal contributions on Jarol are the evaluation of the JAviator and Seascout platforms, including the description of their hardware and software architecture. Based on this information, I derived the requirements to interface these platforms. Furthermore, I analyzed Java threads and Exotask concurrency models to derive the interface requirements towards a general concurrency interface. Based on the platform and concurrency requirements, I designed the concepts that constitute Jarol. I implemented and documented together with Eduardo Marques these concepts in Java, which resulted in a release of Jarol 0.1 on Friday the 13th of April 2007.

1.3 Definitions of Terms

General Terms

To increase readability we present the general terms that are mentioned in this thesis.

Actuator An actuator is a device that accepts data and translates it into information that realizes an intended effect on a mechanism, like a motor.

AUV An autonomous underwater vehicle (AUV) is a vehicle with no onboard pilot that travels under water. Usually AUVs are oceanographic tools that navigate autonomously and carry various sensors for navigation and oceanographic research. [14]

ASV An autonomous surface vehicle (ASV) is a vehicle with no onboard pilot that travels across water. ASVs are sometimes used to coordinate with AUVs [24] [31].

DUNE The DUNE Uniform Navigational Environment is a general framework for on-board software in autonomous vehicles developed by the USTL (see below).

Exotasks "Exotasks are a novel Java programming construct that achieve deterministic timing, even in the presence of other Java threads, and across changes of hardware and software platform." [27]

Fin "A fin is a surface used to produce lift and thrust or to steer while traveling in water, air, or other fluid media." [15]

Gumstix A Gumstix is a small computer that is widely used in embedded devices.

Gyroscope "A gyroscope is a device for measuring or maintaining orientation, based on the principle of conservation of angular momentum." [16]

I2C I2C "is a multi-master serial computer bus invented by Philips that is used to attach low-speed peripherals to a motherboard, embedded system, or cellphone. The name stands for Inter-Integrated Circuit" [17].

Inertial Unit See IMU.

IMU "An Inertial Measurement Unit (IMU) is a closed system that is used to detect altitude, location, and motion. Typically installed on [an] aircraft or [an] UAVs, it normally uses a combination of accelerometers and angular rate sensors (gyroscopes) to track how the craft is moving and where it is." [18]

JAviator The JAviator is an UAV developed by the Computational Systems Group at the University of Salzburg [7].

MVS "The MVS is a multiple vehicle simulation system" [25] developed by the USTL.

Neptus Neptus is a distributed command and control framework for operations with vehicles, sensors and human operators developed by the USTL.

PWM Pulse width modulation (PWM) is a technique to control analog circuits with digital outputs. [10] We call the result of this conversion PWM signals.

Quadrotor "A quadrotor, also called a quadrotor helicopter, is an aircraft that is lifted and propelled by four rotors. Quadrotors are classified as rotorcraft, as opposed to fixed-wing aircraft, because their lift is derived from four rotors. They can also be classified as helicopters, though unlike standard helicopter, quadrotors are able to use fixed-pitch blades, whose angle of attack does not vary as the blades rotate. Control of vehicle motion can be achieved by varying the relative speed of each rotor to change the thrust and torque produced by each." [19]

Robostix A Robostix is a computer board based on an Atmel AVR processor - the ATMega128 [5].

ROV A remote operated underwater vehicle (ROV) is an underwater vehicle without an onboard pilot, but with a connection to a human operator that monitors and steers the vehicle.

RS-232 RS-232 is a standard for transmitting serial data.

Seascout The Seascout is an AUV of the USTL. [12]

Seaware "Seaware is a middleware for network communication in dynamic and heterogeneous network environments, oriented to data-centric network computation." [23]

Sensor A Sensor is a device that reads information from the environment and provides this information in form of data.

UAV "An unmanned aerial vehicle (UAV) is an aircraft with no onboard pilot. UAVs can be remote controlled or fly autonomously based on pre-programmed flight plans or more complex dynamic automation systems." [20]

USTL USTL is the Underwater Systems and Technology Laboratory at the Faculty of Engineering at Porto University (Portugal). [8]

Chapter 2

Platforms

In this chapter we give an overview of the two different platforms for which we want to develop a *control* infrastructure. Each platform consists of a vehicle with a specific equipment and software architecture. The first one is the *JAviator*, also referred to as the helicopter, and the other one is the *Seascout* also referred to as LAUV.

The term *control* in the context of the thesis means to command, direct, or regulate a certain component. In our case we want to control physical components of the vehicles with software. The part of the software that does the actual control is called the *control code*.

The JAviator is a project of the *Computational Systems Group* in the Department of Computer Sciences at the University Salzburg (Austria). It is also the name of the quadrotor - a helicopter with four rotors - that is the heart of the project. Seascout is a project of the *Underwater Systems and Technology Laboratory* (USTL) at the Faculty of Engineering at Porto University (Portugal). The vehicle of the Seascout project is an autonomous submarine called LAUV or Seascout.

We discuss of the challenges that the different platforms presented to us at the end of this chapter.

2.1 Helicopter - The JAviator Project

The JAviator [39] project deals with the research and implementation of the control software for an unmanned aerial vehicle (UAV). The aim of the project is to implement the software of an UAV entirely in Java. The JAviator serves as a test and research platform for the Computational Systems Group (Salzburg) and the IBM T.J. Watson Research Center (Hawthorne, New York) for their Exotasks system that will be explained in Chapter 3.

The JAviator was designed, built, and developed from scratch at the University of Salzburg. The project started in Fall 2005. After initial propulsion tests in April 2006, the first flight was also performed in April 2006. In October 2006 the first all-Java

flight was performed during a visit of Joshua Auerbach and David Bacon from IBM. The JAviator was flying in a rack for security reasons. A video of this event can be found here [9].

2.1.1 Hardware Platform

The JAviator is an electric quadrotor, the rotors surround the inner section in a cross-shaped manner, as can be seen in Figure 2.1. All components except the rotors and motors are contained in the inner section. Rainer Trummer designed and constructed the JAviator from carbon fiber, aircraft aluminum and titanium.

Figure 2.1: JAviator V1

Measurements The JAviator has a total diameter of 1100 mm (max diagonal over spinning rotors). It weighs about 1.5 kilogram including the Robostix, Gumstix and the power-supply board, which will be explained in the following sections. The frame is built with carbon-fiber pipes, body rings, aluminum connectors, aluminum screws and blind rivets. The rotors are constructed with aluminum, titanium and carbon fiber. Figures 2.6 and 2.7 provide a good view on the internals and details of the JAviator.

Propulsion We use custom-made 3-phase synchronous motors with a maximum power of approximately 200 Watt that are controlled by Jeti Spin 22 3-phase controllers. The motor-to-rotor gearing is 6:1. The four motors and other components are powered by a Thunder Power 4-cell lithium-polymer battery with 14.8V and 6000mAh. This allows a maximum thrust of 1000 g per rotor. The maximum payload is approximately 1500 g. The JAviator should be able to lift about 0.5 kilogram and fly for about 15 minutes.

Communication Interfaces Due to the many different hardware components we use different kinds of communication protocols, like RS232, I2C and PWM. In Figure 2.2 the different components and their associated protocol can be seen.

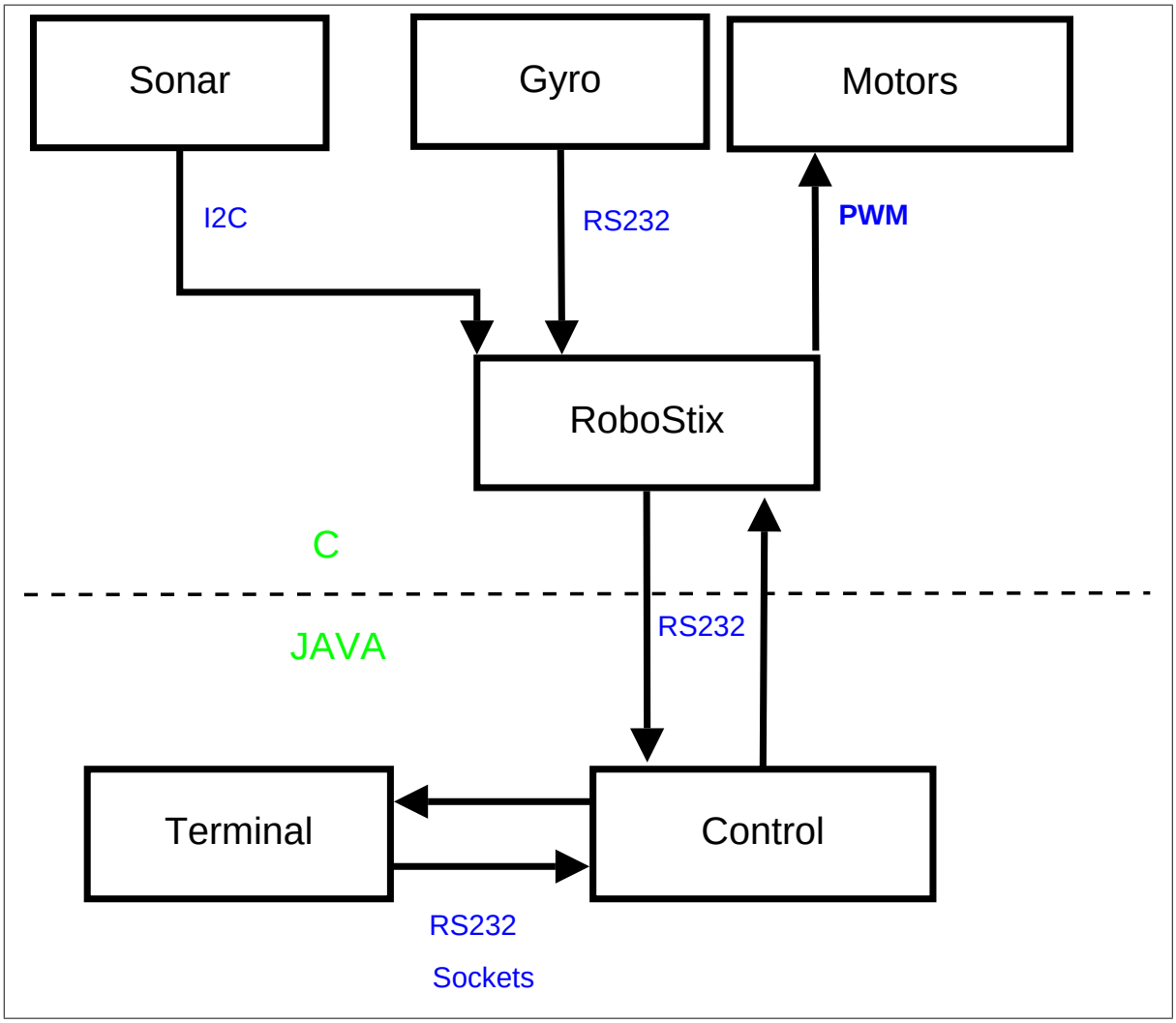

Figure 2.2: JAviator Communications

On-board Computers

Robostix - Robostix TH We use the Robostix TH to do the low-level sensing[1] and actuating[2] for the JAviator. A Robostix is like a small computer without a keyboard or monitor attached. Figure 2.3 shows the back of a Robostix that is mounted on a Gumstix. A Robostix is a board based on an Atmel AVR processor - the ATMega128. The board support communication via PWM, I2C, [5] etc.

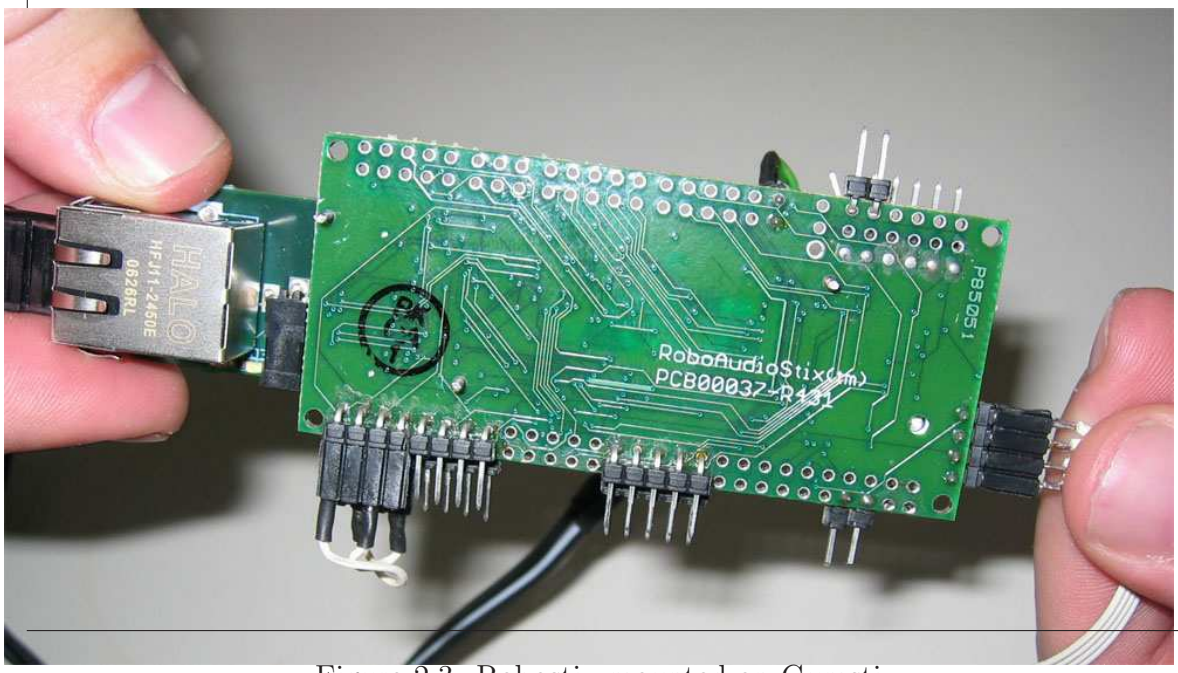

Figure 2.3: Robostix mounted on Gumstix.

Gumstix We use the Gumstix connex400 as on-board computer for the JAviator. A Gumstix is a small computer that uses Linux as operating system, but it lacks a keyboard and monitor, hence it needs to be connected to a common computer to interact with it. Gumstixs are widely used in embedded devices. The connex400 is an Intel PXA255-driven motherboard. It has one 60 pin Hirose I/O connector and one 92-pin bus header. It runs an Intel XScale PXA255 with 400 MHz and has 16MB of flash memory [6].

In Figure 2.2 the Gumstix executes the control code. The current configuration can be seen in Figure 2.4 and 2.5.

Sensors

Sensors are devices that provide information on the environment, this information can be used for various purposes. In the context of control the interest lies in sensor data

[1] Sensing is the activity of acquiring data from sensors.
[2] Actuating is the activity of sending data to actuators.

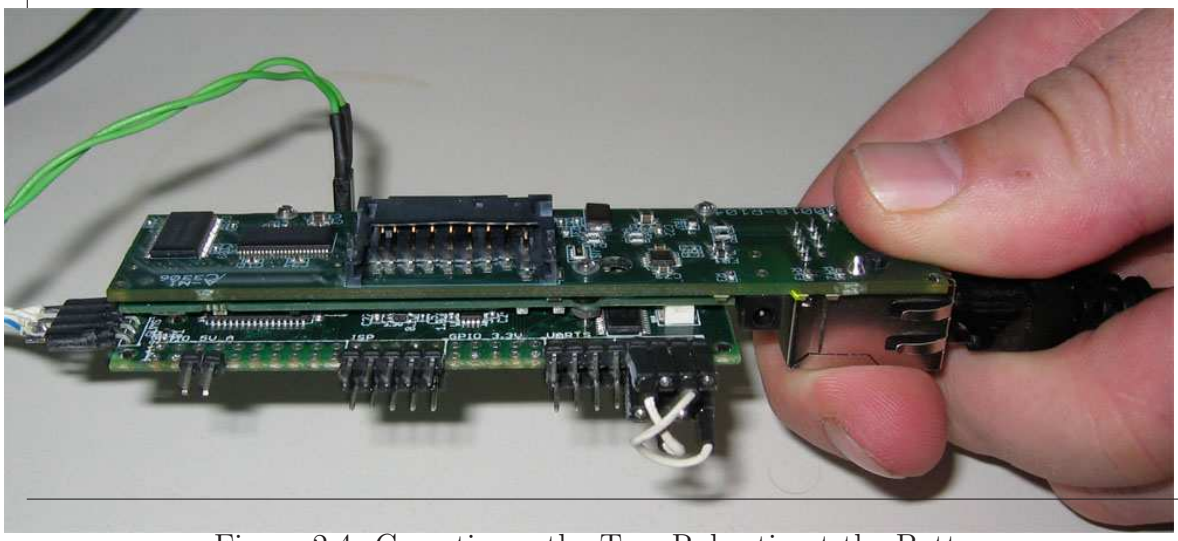

Figure 2.4: Gumstix on the Top, Robostix at the Bottom.

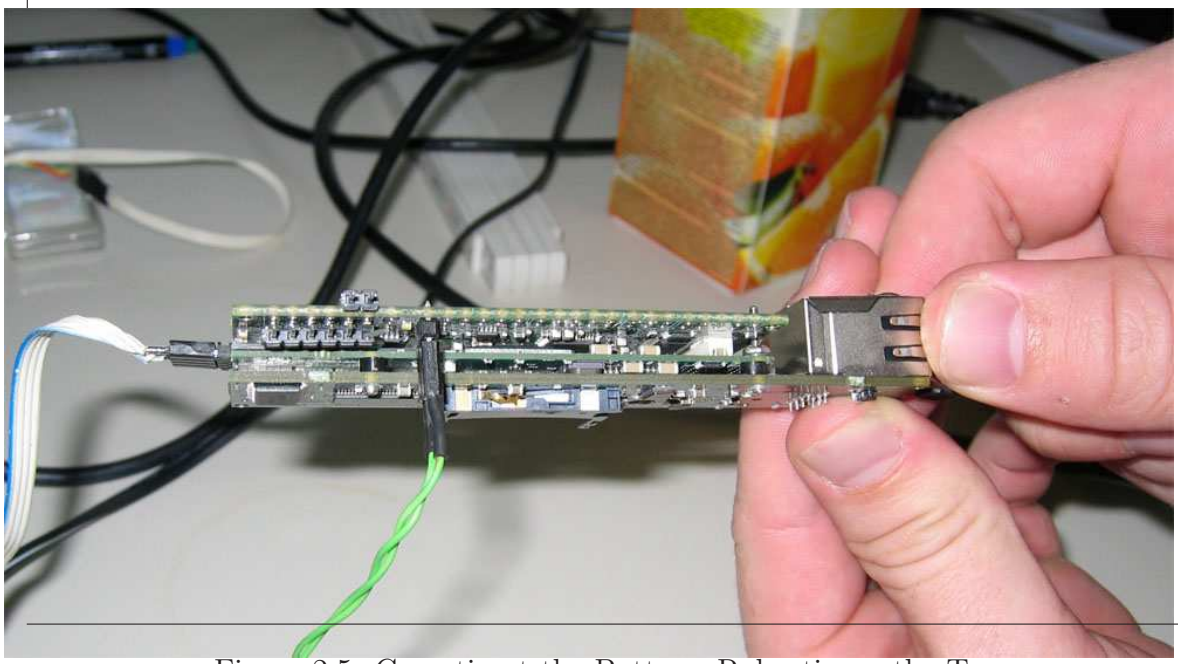

Figure 2.5: Gumstix at the Bottom, Robostix on the Top.

that provides information on the physical state of the vehicle, e.g., altitude, speed, pitch, yaw, roll, etc.

Microstrain 3DM-GX1 We use a Microstrain 3DM-GX1 3-axis gyro-stabilized inertial measurement unit as gyroscope for the JAviator. It is mounted at the top of the inner section of the JAviator, as can be seen in Figure 2.6. The "3DM-GX1 combines three angular rate gyros with three orthogonal DC accelerometers, three orthogonal magnetometers, multiplexer, 16 bit A/D converter, and embedded microcontroller, to output its orientation in dynamic and static environments." [13]

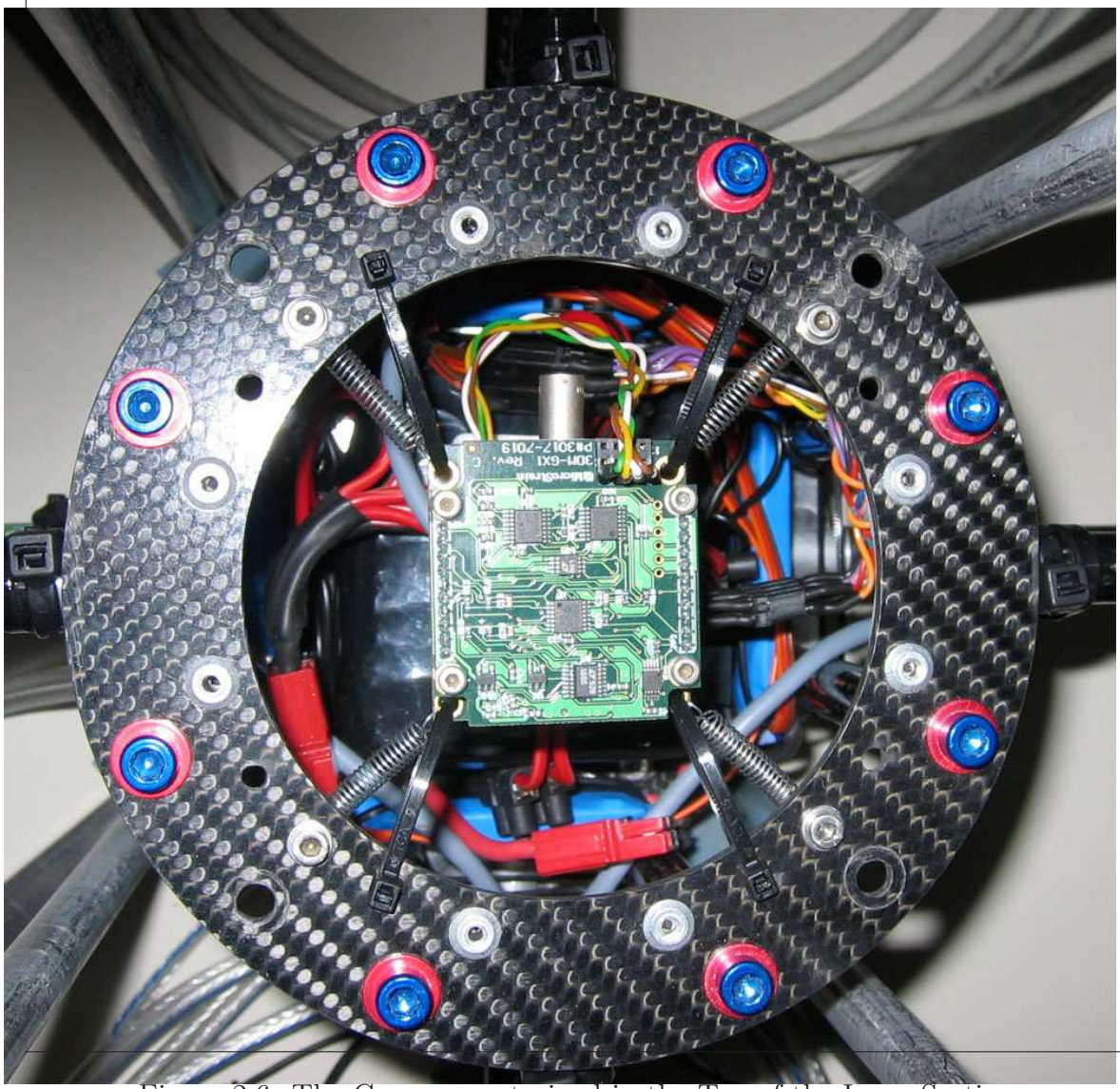

Figure 2.6: The Gyroscope strained in the Top of the Inner Section.

Devantech SRF10 For determining the altitude of the JAviator we use the Devantech SRF10 ultrasonic distance sensor. The sensor is able to determine ranges between

6 cm and 600 cm. It communicates via a standard I2C bus [4]. On each arm of the JAviator resides a sonar that is pointing toward the ground, as can be seen in Figure 2.7. Note that the two small cylinders on the green board are the sonar sensors.

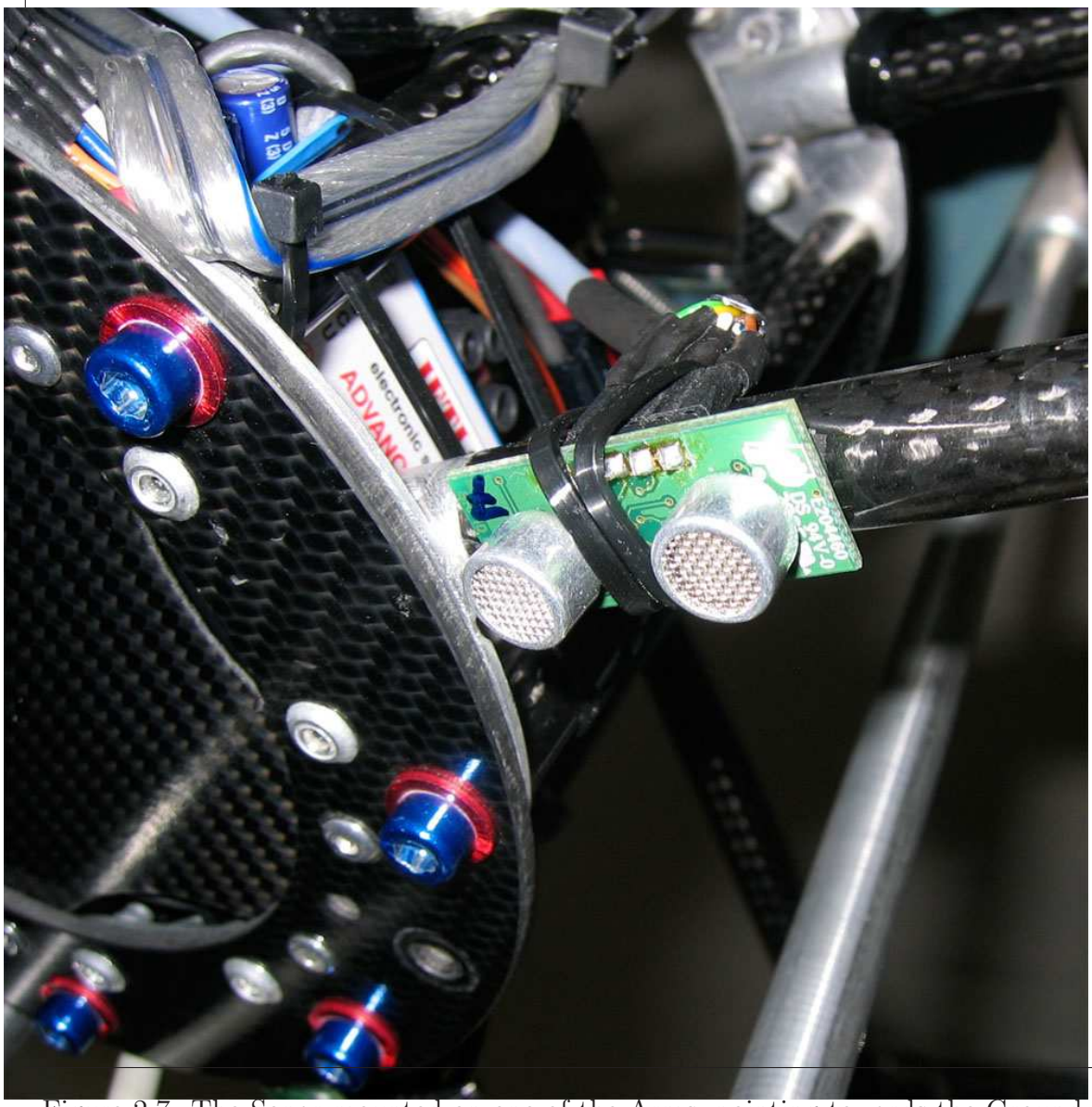

Figure 2.7: The Sonar mounted on one of the Arms, pointing towards the Ground.

Actuators

Actuators are used for actuation, this is the translation from digital output of computers - in this case the Robostix - to an analog or other kind of signals for motors, servos, or other devices.

Jeti Spin 22 3-phase controllers We use four Jeti Spin 22 3-phase controllers, that convert the PWM signals from the Robostix into motor signals. It runs at 14

kHz and converts the DC to AC. Each controller is located in the inner section of the JAviator (see Figure 2.8) next to the arm of the assigned motor.

2.1.2 Software Platform

The JAviator control software is implemented in Java, whereas the low-level code like drivers are implemented in C. As can be seen in Figure 2.2 the control code and the terminal are implemented in Java. The software on the Robostix is implemented in C. It provides an interface for the low-level components that performs a timed sensing and actuating. This allows the JAviator team to implement as much as possible in Java.

Java Control Code

There are two different approaches for the implementation of the control code in the JAviator project. One solution is with ordinary Java threads, the other one uses the Exotasks [27] system from IBM. In Chapter 3, we provide an introduction to Java threads and Exotasks. In order to speed up the development process, large parts of the control code were used for both systems. Since Java threads and Exotasks follow different guidelines the source code is unstructured and hard to read, which decreases code maintainability and the overall understanding of the system.

JAviator Control Terminal

The JAviator Control Terminal is a Java application that allows to monitor, set thresholds and steer the JAviator from a computer that is connected to the JAviator via a RS232 interface or socket connection.

The user is able to monitor the motor signals, the signal offsets, the alignment, roll, pitch, yaw and the altitude of the helicopter. To enhance security the user can set the limits for roll, pitch, yaw and altitude, as can be seen in Figure 2.9. If these limits are exceeded the terminal sends a shutdown message to the JAviator to initiate a safe hard-coded landing procedure that resides on the Robostix. The user can steer the JAviator with a keyboard or a joystick.

MockJAviator

The MockJAviator simulates the physical behavior of the JAviator. It enables verifying control code and testing without using the helicopter. It was written by Rajan Vadakkedathu of IBM.

The MockJAviator can be connected to the JAviator Control Terminal via sockets. It provides sensor data and reacts to the actuator data sent to it, thus allowing the user to verify the reactions on the Control Terminal. Sockets are used for the communication between the MockJAviator and the Control Terminal.

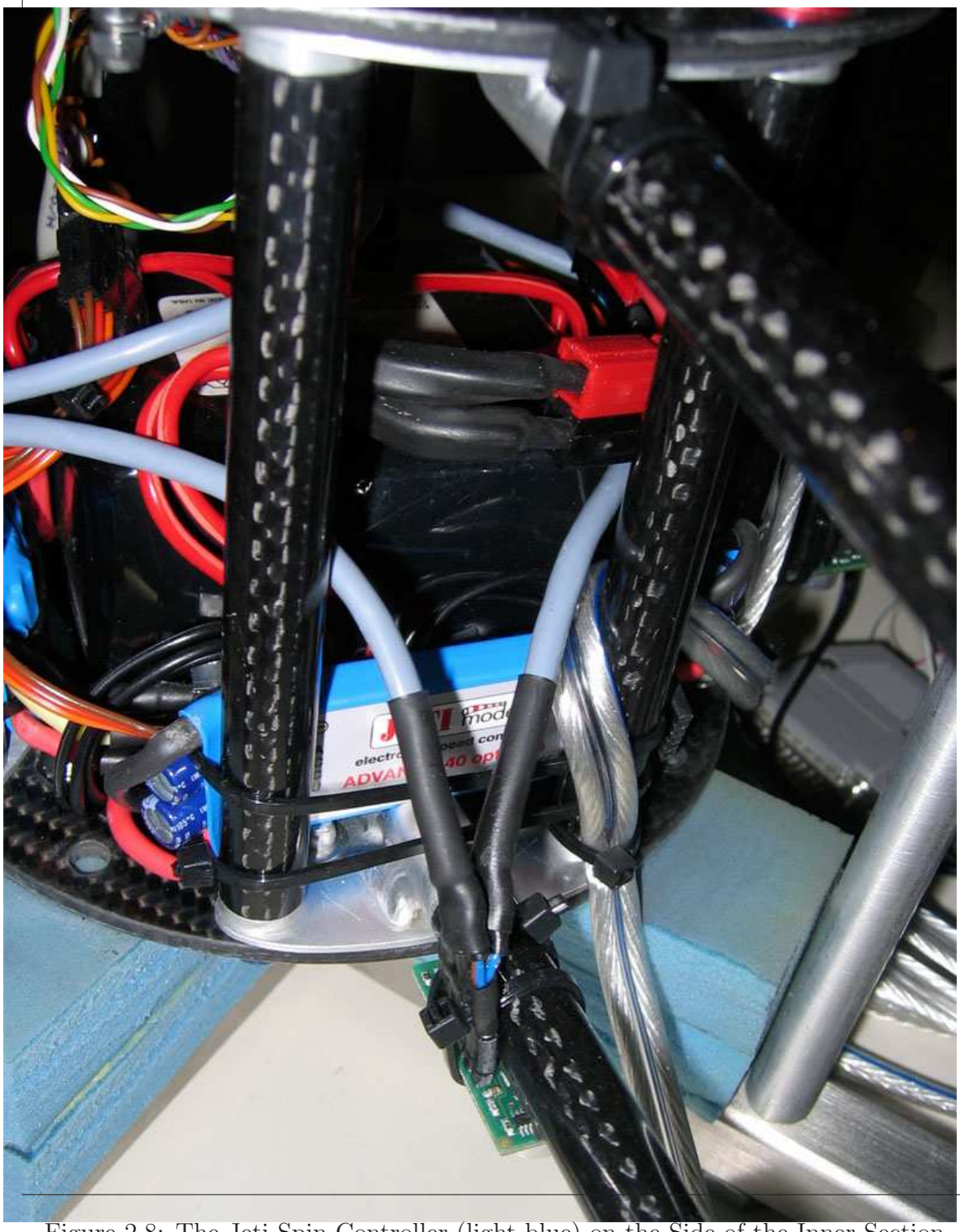

Figure 2.8: The Jeti Spin Controller (light-blue) on the Side of the Inner Section.

Figure 2.9: JAviator Control Terminal

2.2 Submarine - The Seascout Project

The Seascout is an underwater vehicle system that consists of an autonomous underwater vehicle[3] (AUV) named "light AUV" (LAUV) or Seascout (see Figure 2.10 and 2.11), an acoustic positioning system with 2 external acoustic transponders and an operator command and control framework called Neptus [12]. The LAUV "is a small torpedo shaped vehicle optimized for a low cost mechanical structure." [25]

The Seascout was built by the Underwater Systems and Technology Laboratory (USTL) in the Faculty of Engineering at Porto University (Portugal). It was designed for cooperation, swarm-type mission and as a test platform for mixed-initiative concepts. The USTL used previous submarines that were bought from external suppliers and later modified to fulfill the needs of the USTL. The USTL looks back at about 10 years in working with various AUVs, UAVs, ROVs and ASVs [21].

Figure 2.10: LAUV in the Laboratory.

[3] An AUV is a submarine without an operator, it is controlled by an autonomous navigation unit.

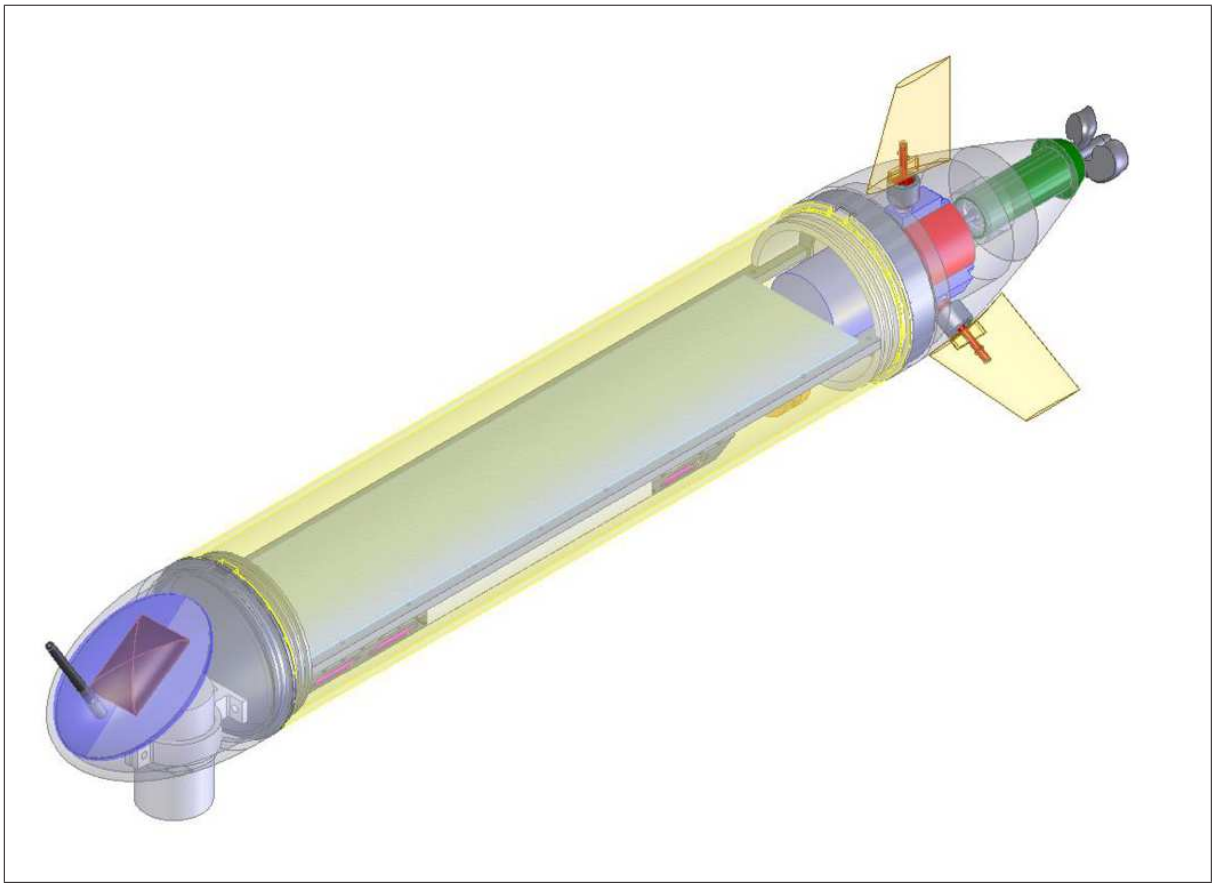

Figure 2.11: LAUV 3-D View

2.2.1 Hardware Platform

The LAUV is a low-cost submarine for environmental and oceanographic surveys. It has an operational depth of about 50 meters and is able to operate for 8 hours. The hull is constructed from aluminum, whereas the tail is built with polyethylene. It can be equipped with various sensors. The onboard computer is based on Gumstixs and is connected to all actuators and sensors [12].

Measurements

The LAUV is torpedo-shaped, measures 108 cm in length and a diameter of 15 cm. It weighs about 18 kilogram. The Figure 2.10 shows the LAUV without fins. In Figure 2.11 a 3-dimensional representation of the complete LAUV can be seen.

Propulsion

The propulsion is provided by one electrically driven propeller. The expected maximum velocity is 2 m/s [25]. A Mclennan BLDC58-50LMK2 brushless motor with integrated drive electronics is used it provides "up to 50 watts continuous output power and variable speed proportional" [1]. It is operating at speeds from 100 to 3000 RPM. Depending on the configuration 3 or 4 fins are used to maneuver the submarine.

On-board Computer

T3SX Stack The T3SX stack is the core of LAUV and consists of various components. Note that we explain the important ones in more detail below.

Overview of T3SX stack components:

- Gumstix - Connex 400xm (GS400J-XM)
- Gumstix - Robostix-TH (BRD00019-TH)
- Gumstix - netCF (BRD00035)
- Kingston - Compact Flash Card PRO 1Gb HighSpeed
- 4 leak sensors
- 4 temperature sensors
- 2 Voltage monitor
- 2 Amperage monitor
- 5 signal relays
- 1 RPM monitor
- 2 buffered and pre-processed RS232 serial ports

- 2 RS232 serial ports
- 1 Thruster controller (i2c interface)
- 4 PWM outputs (for tail servos control)
- 1 Stepper interface

Robostix - Robostix-TH The LAUV uses the same Robostix as the JAviator.

Microstrain 3DM-GX1 The LAUV uses the same gyroscope as the JAviator.

Gumstix The Gumstix is the same model as in the JAviator and it is mounted on the T3SX [25].

Sensors

The LAUV can be equipped with various kinds of sensors, we explain only those that are important from the control perspective.

CTD Sensor CTD is the abbreviation for conductivity, temperature and depth. A CTD sensor provides important data on the depth of the submarine. The LAUV uses a Glomo SW100 OEM. For further information see the specifications here [3].

Actuators

Actuators are used for actuation, this is the translation from digital output of computers - in this case the Robostix - to an analog or other kind of signals for motors, servos, or other devices.

Servos The LAUV uses Futaba S3003 servos to control the fins.

Thruster Controller The BLDC 58 brushless motor has integrated drive electronics. It provides up to 50 watts of "continuous output power and a variable speed proportional to a 0-5 V control signal" [2].

2.2.2 Software Platform

The Seascout resides on a vast software architecture that was developed and maintained over the years at the USTL. This architecture provides support for various aerial and nautical vehicles. The low-level part of the architecture is provided by the *DUNE Uniform Navigational Environment* (DUNE). It interacts with Seaware that is used for the communication between numerous vehicles and multiple operators. Operators work on Neptus terminals, which are used for mission planing, monitoring and steering the vehicles.

DUNE - DUNE Uniform Navigational Environment

DUNE is a general framework for on-board software in autonomous vehicles. It provides a platform abstraction layer in C++ to allow portability for different computer architectures and operating systems, thus it is used to abstract away the hardware details of the LAUV. Figure 2.12 gives a good impression of the layers in DUNE. Additionally it provides facilities for low-level communication with sensors, actuators, navigation, guidance, maneuver and mission control. DUNE supports multiple Operating Systems (Linux, QNX, ...) and target architectures (Gumstix, PC104,...). It allows the control of different vehicles (ROVs, AUVs, ASVs, ...). Additionally, it includes a large amount of drivers for heterogeneous hardware, like CAN, GPS, IMU, I2C sensors, etc [12] [25] [24].

DUNE is multithreaded (one thread per task) and uses a lock-free message bus. Related logical operations are divided into isolated sets called tasks. Tasks can be attached and detached by name at any time [12] [25] [24]. "Tasks are executed in a concurrent or serialized fashion and may also be grouped into single concurrent or serialized execution entities." [25]

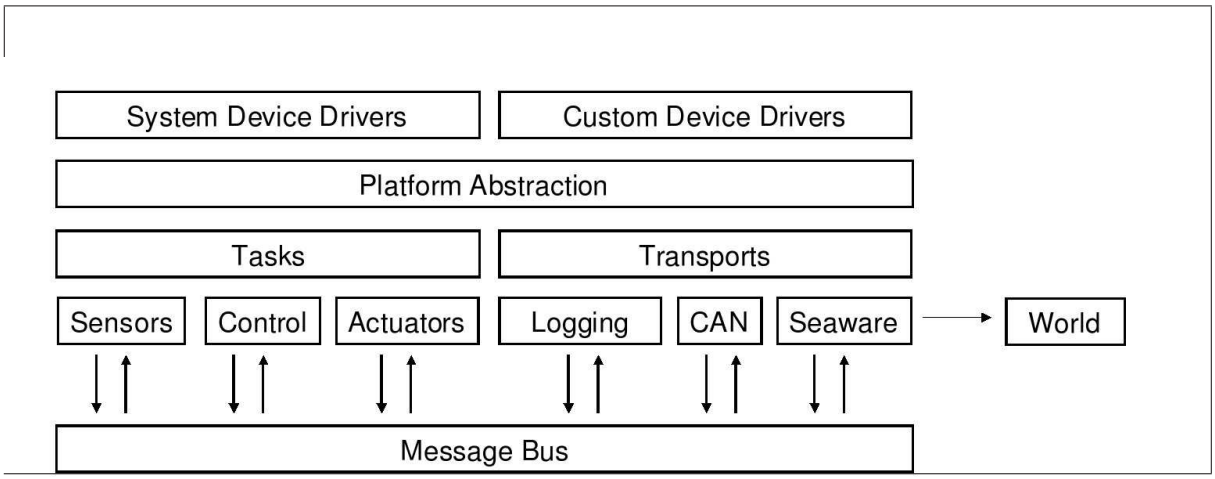

Figure 2.12: DUNE

The LAUV Simulator

The LAUV simulator is called *multiple vehicle simulation* system (MVS) [25] developed by the USTL. The simulator runs as a periodic DUNE task, where it replaces the sensors and actuators.

The simulation omits any collision detection and is limited to one vehicle. When the simulation is initiated the dynamic world is created with global properties like gravity, also obstacles and boundaries are added. The vehicle with its initial state is defined. After the initialization a loop is started that runs until termination of the simulation, it performs the following three steps:

(1) Apply forces to the vehicle.

(2) Take simulation step.

(3) Read the vehicle's position, orientation and velocity.

Neptus - Seascout Edition

Neptus [22] is a distributed command and control framework for operations between vehicles, sensors and human operators. The USTL provided us with a Neptus application for the Seascout: "Neptus - Seascout Edition".
The Neptus framework provides applications for world representation and modeling, planning, simulation, execution control, and post-mission analysis. It uses XML for data representation and XLST generators to translate XML into the appropriate vehicle language [22]. Neptus uses the Seaware middleware to communicate with the attached vehicles [23].

The Neptus Seascout Edition allows the user to control the submarine via keyboard or joystick. It provides a 3-dimensional view of the mission area, various sensor data, and control information as can be seen in Figure 2.13.

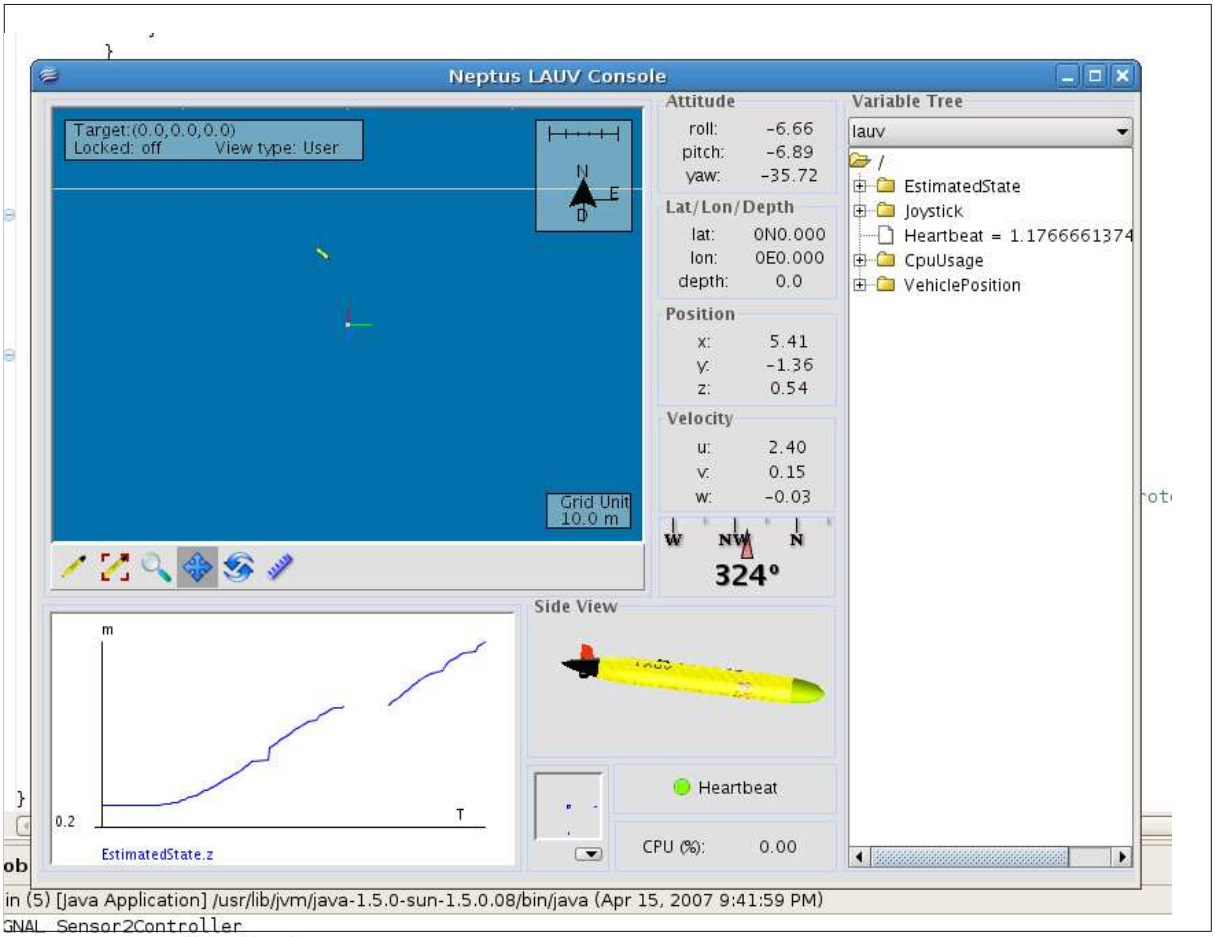

Figure 2.13: Neptus Seascout Edition

2.3 Summary

We discussed the JAviator and Seascout platforms on which we want to apply our control infrastructure. On the hardware side we have a strong disparity between the two projects: not only that we consider an aerial versus an underwater vehicle, there are also many differences in the experience of both groups in dealing with vehicle platforms. This is similar on the software side. The software of the JAviator project was developed from scratch, in contrast to the Seascout project that relies on a vast amount of existing infrastructures and tools.

The differences helped us to see the benefits and drawbacks of the different approaches. We also profited from the exchange of experience, especially in those areas where we use a common architecture like the Gumstix, Robostix, and the gyroscope.

The aim of the Jarol project lies in providing an infrastructure for developing control code in a more convenient way. We are interested in creating an interface that offers an abstraction that hides all the aspects unnecessary for control.

2.4 Challenge Definition

In order to define the challenge we need to take a look at the initial situation.

Deriving from the evaluation of the hardware and software platforms, we have:

- Two heterogeneous vehicles that use different hardware and software platforms.
- Two different concurrency systems in the JAviator.

The challenge is to deal with heterogeneous platforms and different concurrency models. Therefore, we need to derive platform and concurrency interfaces that allow the implementation of control code in Java. We are interested in:

- Developing control code for both vehicles in Java.
- Using different concurrency systems for the execution of the control code.
- Migrating more source code from low-level languages to Java.

2.5 Proposed Solution

Examining the challenge definition, we propose the development of an infrastructure that provides:

- Abstraction of the hardware details of the platform.
- Abstraction of the software details of the platform.
- Explicit data flow at the interfaces - to allow the use of different concurrency models and generic programming.
- Explicit control flow at the interfaces - to allow the use of different concurrency models and provide support for deterministic implementations.

The abstraction of the underlying hardware and software is realized by defining a message set that is derived from the existing projects and converted into Jarol compatbile objects. This enables extracting the data and control flow information from the messages in order to perform and synchronize the computation of the control code.

Chapter 3

Concurrency Models

Providing an interface to different concurrency models is a major aspect of Jarol. It is needed to enable the developer the use of Java threads or Exotasks for the execution of the control code. Hence the developer can choose the model most appropriate for his project.

3.1 Data Flow and Control Flow - Communication and Synchronization

In a concurrent program we have a data and a control flow. The data flow contains all the necessary data that is used by the program to fulfill its goal. It is needed for *what* the program does. The control flow is all the information that is needed to coordinate the program internally. It reflects *how* parts of the program are interacting with each other.

We will introduce the concept of process communication and synchronization and their relation to the data and control flow. The data flow relies mainly on communication, whereas the control flow relies mainly on synchronization.

Synchronization is the coordination of different threads within the system. "In its widest sense, synchronization is the satisfaction of constraints on the interleaving of the actions of different processes" [30]. Communication refers to exchanging data between different threads. "Communication is the passing of information from one process to another." [30]

Data flow and control flow are two separated concepts, but on the implementation level they are typically not separated from each other. The literature notes that the "two concepts [synchronization/communication] are linked, since some forms of communication require synchronization, and synchronization can be considered as contentless communication" [30].

Synchronization and communication are either realized with *shared variables* or *message passing*. Shared variables is a concept that implies that more than one process

has access to a certain object. Message passing is the explicit exchange of data in form of a message that is passed between two processes [30].

3.2 Java Threads and Exotasks

We are interested in deriving the requirements for using different concurrency models for the execution of the control code. Therefore we investigate the concurrency models that are used in the JAviator project. These are standard Java threads and Exotasks [27]. Using the requirements for both systems, we are able to create a concurrency interface that allows us to interact with both systems.

Java threads are a general concept in contrast to Exotasks, which have a specific purpose. Java threads are used to provide concurrent execution of different threads, whereas Exotasks are a programming model for the development of real-time systems. Therefore the restrictions, requirements and features are quite different. In communication and synchronization, Java threads use the shared variables approach, whereas Exotasks follow a message passing scheme.

3.3 Java Threads

Java threads allow the developer to execute different threads of execution in one program. Threads in Java are a base concept of the language [40]. This section gives an overview about Java threads, their features and limitations.

3.3.1 Definitions

We assume that the reader is familiar with the concept of threads, therefore we only provide a short definition:
"A thread is a call sequence that executes independently of others, while at the same time possibly sharing underlying system resources such as files, as well as accessing other objects constructed within the same program." [37]

3.3.2 General

"A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently." [11] In Java every program consists of at least one thread. This thread runs the main method of the class that is used as a startup argument for the Java Virtual Machine [37].

The notion thread is a shortcut of "thread of execution". Hence a thread executes a subset of the source code of a given program. In Java a thread is an object. Java threads have the disadvantage that the Java policy "write once run anywhere", usually

fails for them. This is discussed further below.

3.3.3 Synchronization and Communication with Monitors

Java supports two kinds of synchronization: *mutual exclusion* and *cooperation*, both build upon monitor semantics. Mutual exclusion is realized with object locks, whereas cooperation is supported by the `wait()` and `notify()` primitives of monitors.
We assume that the reader knows the concept of a monitor and its associated wait set.

Mutual Exclusion "Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor." [32]
A monitor is associated with an object and with a subset of the source code. This subset is called a monitor region. In Java a monitor region is defined by a synchronized block or a synchronized method. A monitor region is not executed atomically, but is "executed as one indivisible operation with respect to a particular monitor" [40]. Thus the execution of a monitor region of one monitor by a thread disallows the entering of any monitor region of the same monitor by any other thread. "A monitor enforces this one-thread-at-a-time execution of its monitor regions." [40] In Java locks are owned per thread, thus calling a monitor region from within a monitor region of the same monitor does not block. [26]

Cooperation Java also provides cooperation with monitors. Cooperation is needed to adjust different threads towards fulfilling a common goal. A simple example is one thread that reads data from a buffer, whereas another thread writes data to the same buffer. If data is written to the buffer, the writing thread should notify the reading thread when the writing process is finished, preventing the reading thread from unnecessary waiting and/or polling on the buffer.

In Java cooperation is bound to monitors, because the primitives `wait()` and `notify()` can only be called from a thread that is inside a monitor. When a `wait()` is called, the calling thread is suspended. This means that the thread stops its execution and enters the wait set. The thread is suspended (waiting) until another thread inside the same monitor calls a `notify()`. After the call of `notify()` the calling thread still keeps the monitor until it either calls a `wait()` or leaves the monitor by completing the execution of the monitor region. Then the waiting thread is awakened and continues after the `wait()` call [40].
Note that "a thread can only execute a wait command if it currently owns the monitor, and it can't leave the wait set without automatically becoming again the owner of the monitor." [40]

3.3.4 Nondeterministic Thread Behavior

In this section we will take a look at the thread behavior that is guaranteed by the Java Language Specification (JLS). We explain which aspects of the JLS cause problems for the development of multi-threaded applications in Java. Additionally, we address the main issues, related to platform-independent and deterministic behavior.

The "JSR-133: JAVA^{TM} Memory Model and Thread Specification" states:

> "These semantics do not describe how a multithreaded program should be executed. Rather, they describe the behaviors that multithreaded programs are allowed to exhibit. Any execution strategy that generates only allowed behaviors is an acceptable execution strategy." [38]

The Java compiler is allowed to reorder instructions as long as "this does not a affect the execution of that thread in isolation" [32]. To determine the legality of a thread in execution, the implementation of a thread is evaluated as if it would be executed as a single thread. "[I]ntra-thread semantics are what determine the execution of a thread in isolation; when values are read from the heap, they are determined by the memory model." [38] However, when two threads are interacting this can result in a wrong execution order, if the code is not properly synchronized. Reordering can have multiple causes, the just-in-time compilation or the the processor may be the reason for code rearrangements, additionally the memory hierarchy of the architecture on which the virtual machine reside may make the code rearranged [32].
The "JSR-133: JAVA^{TM} Memory Model and Thread Specification" adds on this [38]: "[T]he behavior of a correctly synchronized program is much less dependent on possible reorderings. Without correct synchronization, *very* strange, confusing and counterintuitive behaviors are possible." Sounds fun.

Thus deterministic behavior and platform independence for multithreaded Java programs cannot be guaranteed.

3.3.5 Platform Independence

As already mentioned above, Java threads are not platform independent, thus complicating the platform-independent development of multithreaded applications. It is necessary to know "something" about the run-time environments to run the applications properly. This does not mean it is impossible to write platform-independent multithreaded applications in Java, but it is challenging [34]. We need to take this into account, when we generate facilities for synchronization and communication.

3.3.6 Thread States

A Java thread enters various states during its lifecycle. We provide a short overview of theses states. According to the JAVA API 2 Platform Standard Edition 5.0 [11], a thread can enter the following states:

- NEW: “A thread that has not yet started is in this state.”
- RUNNABLE: “A thread executing in the Java virtual machine is in this state.”
- BLOCKED: “A thread that is blocked waiting for a monitor lock is in this state.”
- WAITING: “A thread that is waiting indefinitely for another thread to perform a particular action is in this state.”
- TIMED_WAITING: “A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.”
- TERMINATED: “A thread that has exited is in this state.”

3.3.7 Summary

Java threads are a general concept for concurrent programming, but they provide no guarantee for deterministic or platform-independent behavior. The monitor semantics do not provide the necessary strictness we need for synchronization.

To meet these challenges we propose:

1. A mechanism to transfer consistent data without locking between different threads.
2. A mechanism to allow a strict synchronization protocol for threads.

We want that all interacting threads execute in isolation with respect to each other, in order to have *sequences of threads* instead of loosely-organized concurrent threads. Enabling deterministic behavior is also one of our goals, therefore it is important that our synchronization scheme allows us to provide a deterministic sequence of the executed threads, e.g., a guarantee that Thread 1 is executed before Thread 2.

3.4 Exotasks

Exotasks are a Java programming construct that was developed by IBM in cooperation with the Computational Systems Group at the University of Salzburg. The aim is to combine the convenience of the Java programming language with real-time guarantees and platform independence, thus extending the "write once run anywhere" philosophy to real-time applications in Java. This is achieved by providing time portability and deterministic execution behavior.

Exotasks are a novel concept, for a better explanation we show the characteristics that distinguishes an Exotask program from a program with common Java threads.
In a common Java program the developer has to implement the data and control flow on his own, e.g., threads can access each others variables if they are public. This accessing is an operation that is part of the data flow of a program. Since there is no automated protection against race conditions, the threads need to be synchronized. This synchronization must be implemented by the application writer and is part of the control flow of the program.

In an Exotask program the data flow is defined with explicit and deterministic semantics and timing behaviors. The developer defines the data flow between the Exotasks explicitly via connections.[1] Additionally, the control flow is on an even higher level, it is realized by the Exotask system that follows the constraints the developer has defined, e.g., at which time an Exotasks has its deadline.

The main difference between Java threads and Exotasks is that a Java thread is a *common object* with the capability to perform an execution concurrently, thus having shared memory regions like any other object, whereas an Exotask is an *isolated object* that is embedded in a system structure and can only communicate via explicitly defined connections.

The Exotask model is entirely defined in the Java language. The main feature of Exotasks is deterministic timing with time-portability. Any Exotask program that is schedulable can be executed on different platforms in the same manner and execution behavior, assuming that enough resources are available. Hence there is no re-certification needed if there are underlying hardware or software changes. This makes Exotasks platform-independent [27].
Exotasks provide essential real-time properties in combination with memory semantics that are close to the original Java memory semantics. Hence, they provide a high-level programming concept with the guarantees needed for real-time applications. To properly run Exotasks a modified Java Virtual Machine (JVM) is needed that is capable of enforcing Exotask memory isolation.

[1] A graphical tool is provided to assist the developer in defining the data flow, e.g., the developer draws the graph and connects two Exotasks with a connection.

3.4.1 Isolation

Exotasks are isolated in two ways. They are physically isolated in space (memory) and logically isolated in time.

Memory isolation is done to preserve deterministic behavior and enforce memory protection. The inspection or modification of non-final static fields is not allowed. Exotasks are *stateless functions*, which means that the output of the function is only determined by the input of the function. The access to a non-final static field would break this statelessness, because the output could be influenced from elements (the non-final static fields) residing outside the function. The memory isolation is enforced by a verifier that checks if tasks are inspecting or modifying any non-final global variables, which would break the isolation, thus failing the garbage collection. This leads to certain restrictions when programming with Exotasks, which are discussed in detail below.

The creation of threads is also forbidden, because it would prevent an accurate scheduling. The schedule for an Exotask graph is calculated before the execution, but standard threads do not provide facilities to determine their execution time, breaking the calculated schedule.

The timing isolation is more complex and out of the scope of this thesis.

3.4.2 An Exotask Program

Each Exotask program consists of three major components:

- The specification graph.
- The user code.
- The timing annotations.

The specification graph[2] describes the structure of the Exotask program. It holds the information of the structure and interaction of the system. A graph consists of several nodes representing Exotasks that are connected via connections. Each Connection has an associated data type. Figure 3.1 shows an Exotask graph with five Exotasks, four communicators and one compute task, which we describe further below.

The user code is the implementation of the functional behavior of the Exotask program. In Figure 3.1 only the ExoLAUVController (a compute task) has an associated class, which contains user code. Note that there are other classes in an Exotask program that provide functionality and are not represented in the graph, like distributers, which are described below.

[2]The graph can be graphically manipulated with an Eclipse plugin. Since it is stored in an XML file it is also possible to edit the graph with a common text editor.

The timing annotations specify the timing behavior of the graph. The timing is determined by the user and is preserved when the Exotask program moves to a different platform. Annotations are attached to Exotasks, connections, or the whole graph.

Memory Allocation

Every Exotask has a private heap and a garbage collector of its own. These garbage collectors can be executed on-demand or scheduled. The private heap follows the standard Java memory model, thus allowing the convenient way of Java memory allocation. This means the developer can allocate memory freely and unreachable memory regions get garbage collected.

Inter Exotasks Communication - Connections and Ports

The nodes of the graph specify the Exotasks. Nodes are connected via directed edges called connections. These explicitly defined connections are the only way how Exotasks can communicate with each other, thus enforcing the isolation of each Exotask. Any object that is transfered via a connection is deep copied, hence there is no sharing between Exotasks.

Connections connect the output port of one Exotask to the input port of another Exotask. In Figure 3.2 the various levels of the communication abstraction can be seen. Connections are defined by the developer in the graph definition, whereas the developer uses (reads from and writes to) the associated ports in the actual implementation.

Ports are attachment points for connections. A port is directed, it is either an input or an output port. Two ports are connected via connections in the graph and must be of the same data type. Data types that are transfered via ports must implement a method that provides deep copying. This is not necessary for data types that are supported by the Exotasks system.

Compute Tasks

Compute tasks are Exotasks that are used for computation of the user code, which are for example - in the context of the JAviator - control algorithms. A compute task has both input and output ports to communicate with other Exotasks.

Communicators

Communicators are special Exotasks that have one input and one output port of identical type. Communicators are mainly used as attachment points for so called *distributers* that allow to extend an Exotask graph over machine boundaries, as explained in the following section.

3.4.3 Extending the Exotask Graph

To extend an Exotask graph over the isolation boundaries a so called *distributer* is used. A distributer links objects residing outside of the Exotask system to Exotasks. It uses anchor points (`ExotaskRoots`) that are connected to communicators. These communicators are used to exchange data with the external system. Note that to connect Java threads with an Exotask system it is also necessary to use a distributer.

Connections are used for inter-Exotask communication, whereas channels are used for communication between Exotasks and components that are outside of the Exotask graph. Channels connect anchor points (`ExotaskRoots`) of the distributer to the communicators, thus allowing the data to pass from outside into the Exotask graph, as can be seen in Figure 3.3.

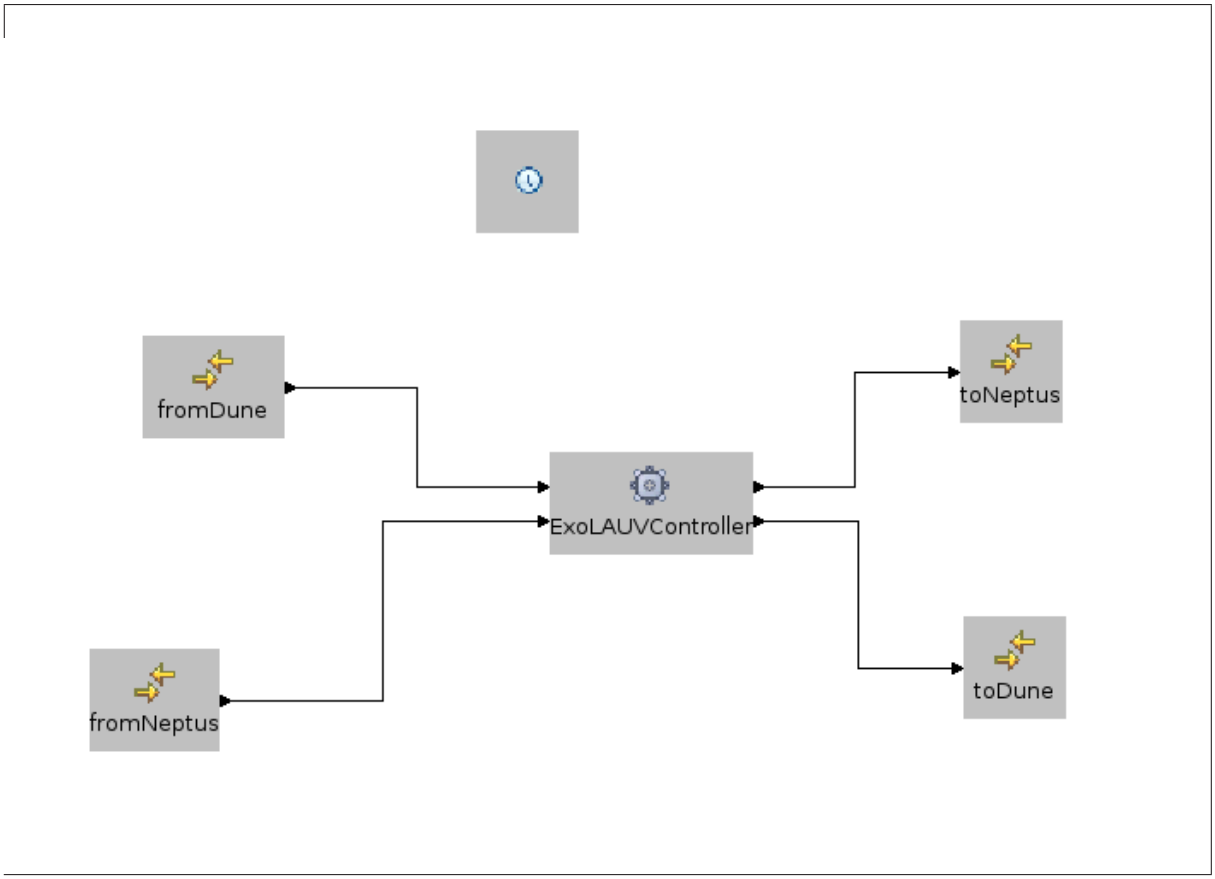

Figure 3.1: The Exotask Graph of the ExoLAUV.

3.4.4 Programming with Exotasks

Programming with Exotasks is similar to programming common Java programs with certain distinctions. A difference is that the program design is not solely done in the source code. In the Exotask system the developer uses a graphical editor or directly manipulates the XML file to define the graph. The graph constitutes of different Exotasks and their connections to each other, it also contains the timing grammar and

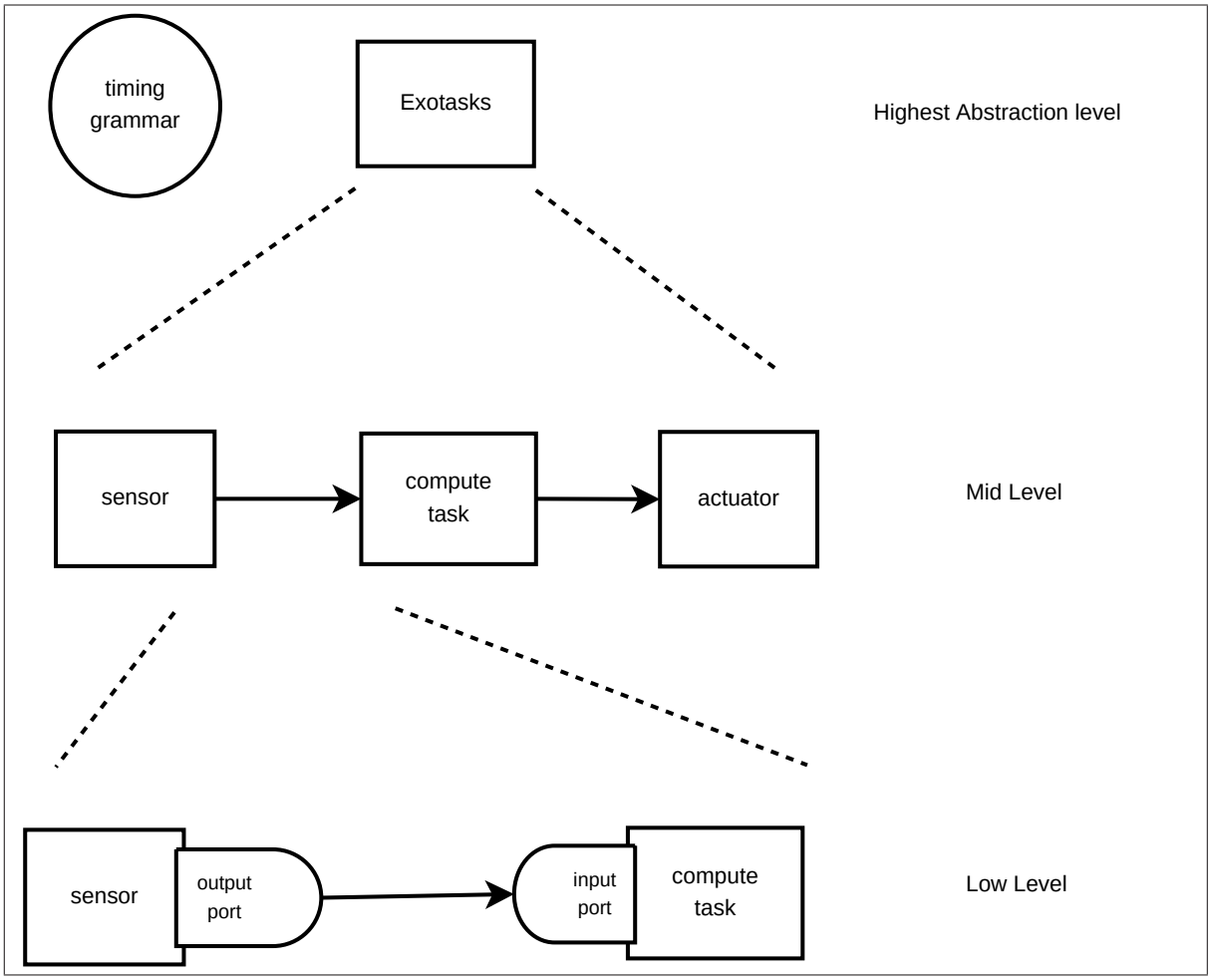

Figure 3.2: Exotask Abstraction Levels

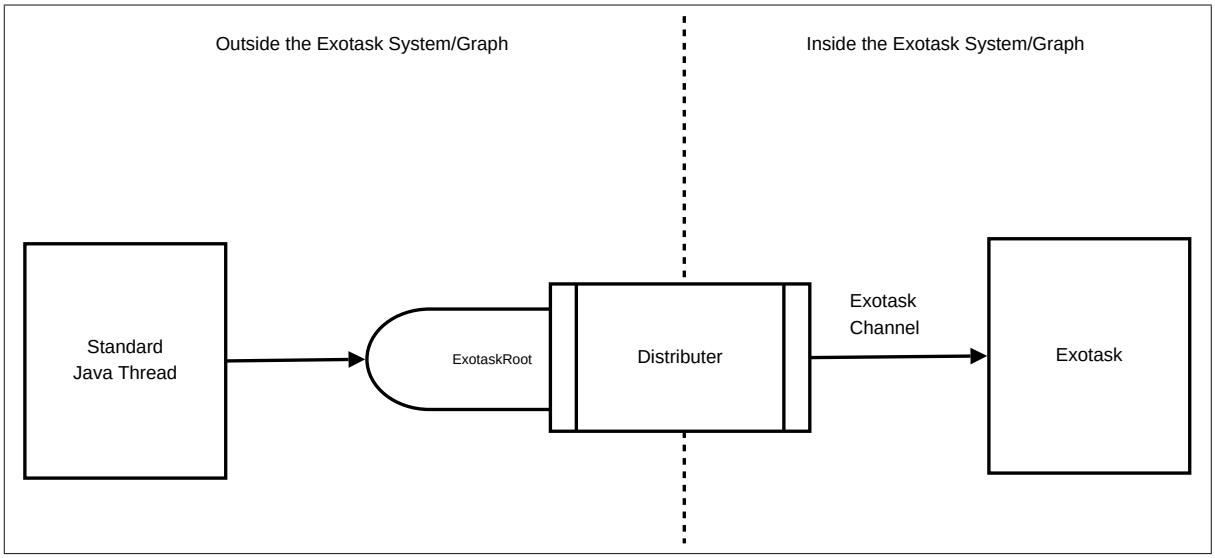

Figure 3.3: Exotask Channel

timing annotations. The actual implementation of the user code is nearly identical, besides the call of some Exotask-specific methods and the restrictions enforced by the Exotask system.

Designing the Graph

Programming an Exotask program begins with the design of the key parts of the Exotask program and its interfaces towards the platform. The graph makes the data flow between the various Exotasks explicit. The developer also defines timing annotations that enforce the execution deadlines.

User Code

For every Exotasks that needs to be implemented, e.g., a compute task, an associated class is created. This class has objects for the input and output ports. Note that these ports need to be defined in the graph with the connections between two Exotasks. The developer can only use these ports to communicate between Exotasks. Ports offer methods to get and set values, thus presenting a simple interface for the developer. The implementation is similar to the implementation of a Java thread. The class needs to implement the `Runnable` interface and the Exotask code is located in the `run()` method.

Restrictions

There are two major restrictions:

(1) The handling of non-final static fields is disallowed.

(2) The creation of threads is also prohibited.

3.4.5 Summary

Exotasks are a specific concept in Java for providing deterministic execution and time-portability of written programs. The isolation model implies certain restrictions in programming and interfacing Exotasks.

We need an interface that allows us to transfer data in and out of the Exotask graph. Thus we need to design and implement that allows to connect Exotasks across machine boundaries.

3.5 Towards a General Concurrency Interface

The goal is to support the mentioned concurrency models (and possibly more) in Jarol, in order to provide a generic data-exchange and synchronization interface for the different concurrency models it is necessary to make the data and control flow as explicit as possible.

We want to reduce the amount of thread locking to a minimum, hence protecting shared memory via locks is not an option. The common synchronization techniques are too limited and do not provide support for deterministic behavior, since wrong synchronization in Java leads to "very strange, confusing and counter-intuitive behavior". We need a model that is very strict upon synchronization failures to prevent any silent-failure in context of synchronization. A mechanism to share or transfer this data between different Java threads is necessary.

The Exotask system relies on isolation. To preserve isolation, Exotasks ports are used for transferring data between different Exotasks. Hence we need mechanisms to provide deterministic synchronization and for exchange of data, between threads and into Exotasks.

Chapter 4

Design

We outlined and explained the hardware and software platforms in Chapter 2. Then we introduced and described the concurrency models that we use for the execution of the control code in Chapter 3. In this chapter we describe the concepts that we derived from the analysis of the previous chapters.

4.1 System Structure

Figure 4.1 shows the system structure of Jarol, it constitutes of the external system and the three Jarol layers:

- The external system is the underlying platform, e.g., a vehicle with its software architecture. Color code, red.
- The Jarol Interface Ring is the layer that abstracts away the control-irrelevant message information. Color code, green.
- The Jarol Adaptation Layer translates the synchronization logic of the platform to the Jarol synchronization information. Color code, yellow.
- The Jarol Core performs the execution of the control code. Color code, blue.

To improve readability every layer has an assigned color, this color is used throughout the thesis for the different components that reside in these structural parts.

4.1.1 External System

The external system is constituted by the the hardware and software of a project - the platform. Each platform uses different kinds of messages to communicate with the Jarol Interface Ring, these messages contain control-relevant and irrelevant information, e.g., the message encoding.

Figure 4.1: Jarol System Structure

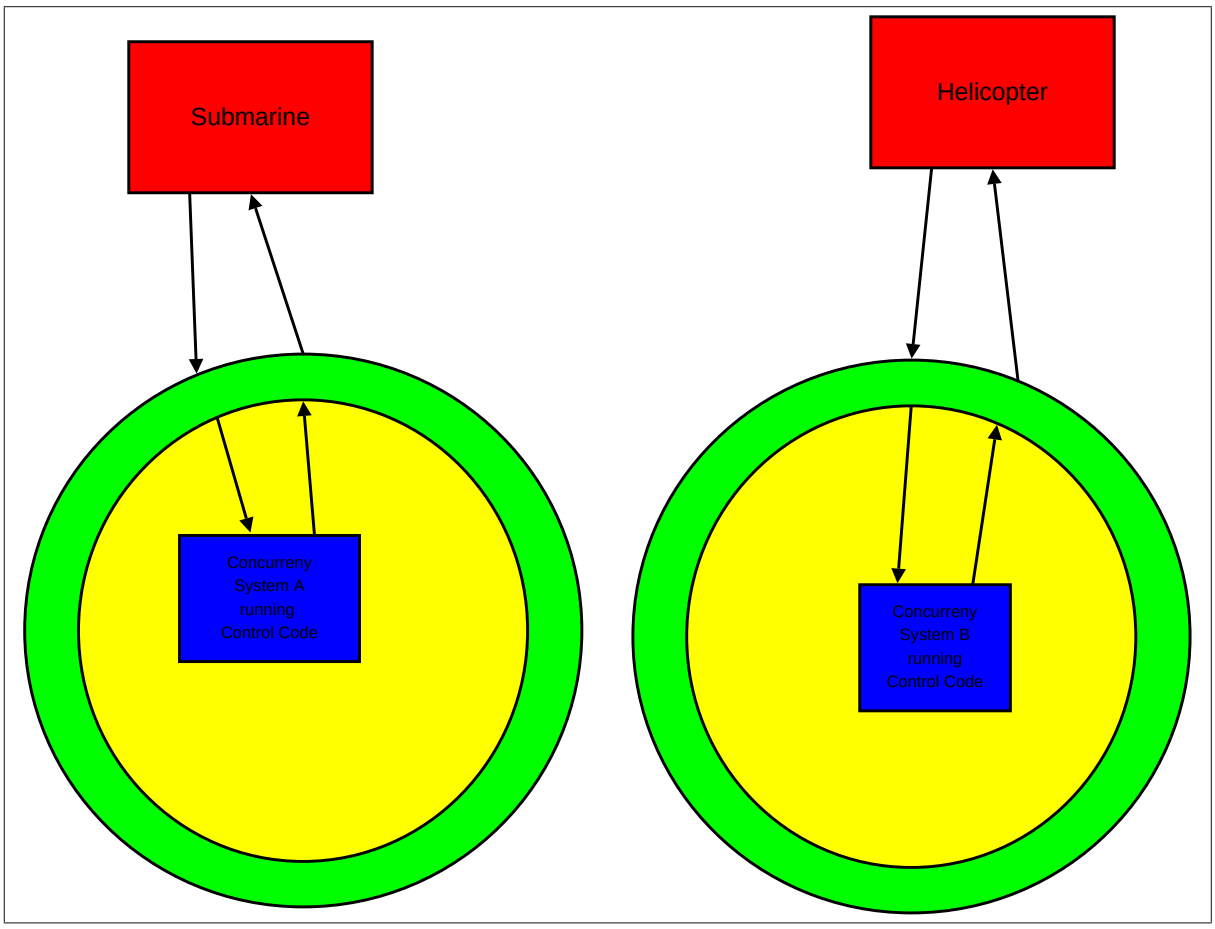

Figure 4.2: Two Jarol Systems exemplified.

4.1.2 Jarol Interface Ring

The main task of the Jarol Interface Ring is to abstract away the control-irrelevant message information. Additionally, it converts the incoming data to Jarol Messages. Thus all the information that passes through the Jarol Interface Ring is data that follows Jarol standards.

The extracted data flow still contains some implicit project-specific information that cannot be handled without further interpretation. Therefore the data is passed to the Jarol Adapation Layer.

4.1.3 The Jarol Adaptation Layer

The Jarol Adaptation Layer uses the data flow from the Jarol Interface and extracts the synchronization information. Since the synchronization information is encoded in a platform dependent way, a project-wise implementation is necessary to identify the control specific information and use it to generate Jarol-specific control information. A scheme of the message conversion can be seen in Figure 4.3.

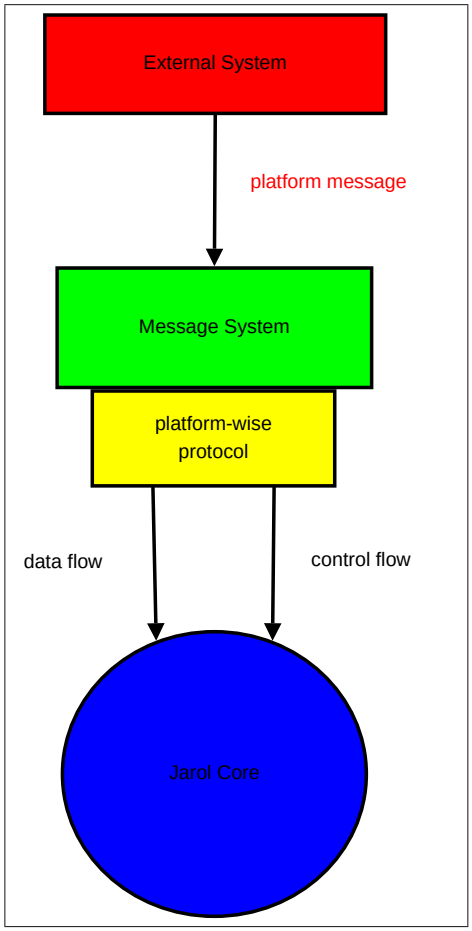

Figure 4.3: Message Conversion

4.1.4 Jarol Core

The Jarol Core is where the control code is executed. The Jarol Core can be implemented with different concurrency models, like standard Java Threads or the Exotask System.

4.2 Concepts

We explained the Jarol layers, their purpose and responsibilities. A discussion follows that describes the concepts used to connect these layers.

4.2.1 Thread

Figure 4.4 shows the symbol that is used throughout the thesis for threads.

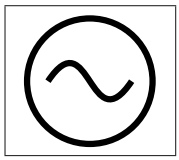

Figure 4.4: Symbol for Thread.

4.2.2 Signals

A signal in Jarol is a synchronization point that allows different threads to synchronize with each other. Figure 4.5 shows the symbol that is used throughout the thesis for signals.

The notion "signal" is widely used in control engineering and computer science. Signal in control engineering is a message. A signal in computer science is used for interprocess communication, but with weaker semantics and on operating system level. When we refer to signal we mean the definition given at the top of this section, if not otherwise stated.

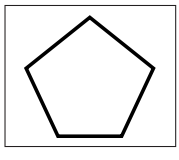

Figure 4.5: Symbol for Signals.

A signal S is a pair $(State, n)$, where:

- $State$ is the current state of the signal, $State \in \{normal, waiting\}$,

- $n \in \mathbf{N}$: the number of waiting threads on this signal.

A signal provides two operations to threads: await and signal. See Figure 4.6 for a schematic of two threads on using a signal for synchronization.

When a thread calls **await** on a signal:

- The calling thread is suspended.
- *State* is set to *waiting*.
- The number n is increased by one.

When a thread calls **signal** on a signal with one or more waiting threads:

- The calling thread notifies all awaiting threads and then continues its execution.
- The number n is decreased by one of the awakened thread.
- If $n < 1$ the awakened thread sets *State* to *normal*.

When a thread calls **signal** on a signal without a waiting thread ($n <= 0$):

- A synchronization error is reported.

Signals in Jarol enforce a cooperation protocol that is strict. Signals are synchronization points that are used for coordination with stronger semantics than common signals, like described in [29]. This means that every operation on a signal has either a positive or negative reaction for the caller. This prevents silent failures and any break of synchronization is noticed immediately. Signals are used to make the control flow explicit and ensure an exact execution sequence or any break in it.

A signal is used for synchronization of different threads. A signal has two primitives: await and signal. The await primitive causes the caller to pause until a signal is emitted from a different caller. The signal primitive wakes up all callers of the await primitive. Signaling a signal in Jarol without some thread waiting for it, results in an error, because this means that the execution is out of order.

4.2.3 Ports

A Jarol port is an universal directional data exchange point, that provides non-blocking and lock-free operations for a single writer and single reader. It provides a simple way to buffer messages that can be used for data transfer between different components. Thus breaking down the data flow to a simple form. Figure 4.7 shows the symbol that is used for ports throughout the thesis.

A port P is an n-tuple $(s, m_0, m_1, m_2, ..., m_{s-1})$, where:

- $s \in \mathbf{N}$ is the size of the buffer,
- $\forall m$: m is a Jarol Message.

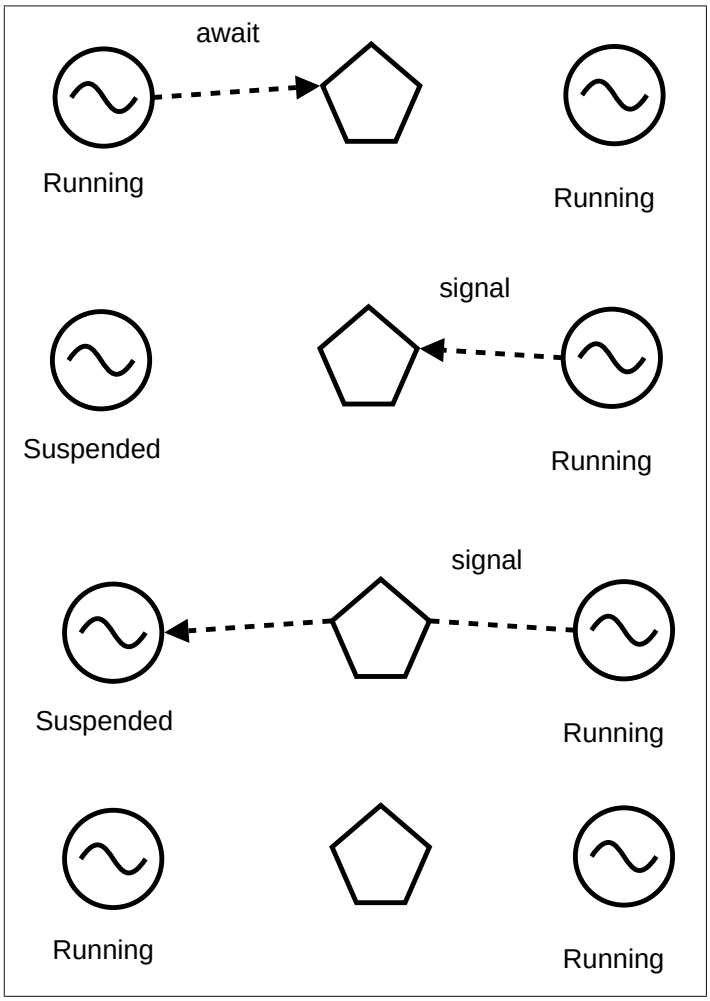

Figure 4.6: Two Threads synchronizing via a Signal.

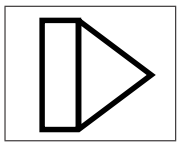

Figure 4.7: Symbol for Port.

A port provides two operations: read and write. See Figure 4.8 for a schematic of two threads using a port for data transfer.

When read is called on a non-empty port:

- The oldest element of the buffer is read and returned.

When read is called on an empty port:

- The value *null* is returned.

When write is called on a port, which is not full:

- The object is cloned and written into the next free slot of the buffer.

When write is called on a full port:

- The value *null* is returned.

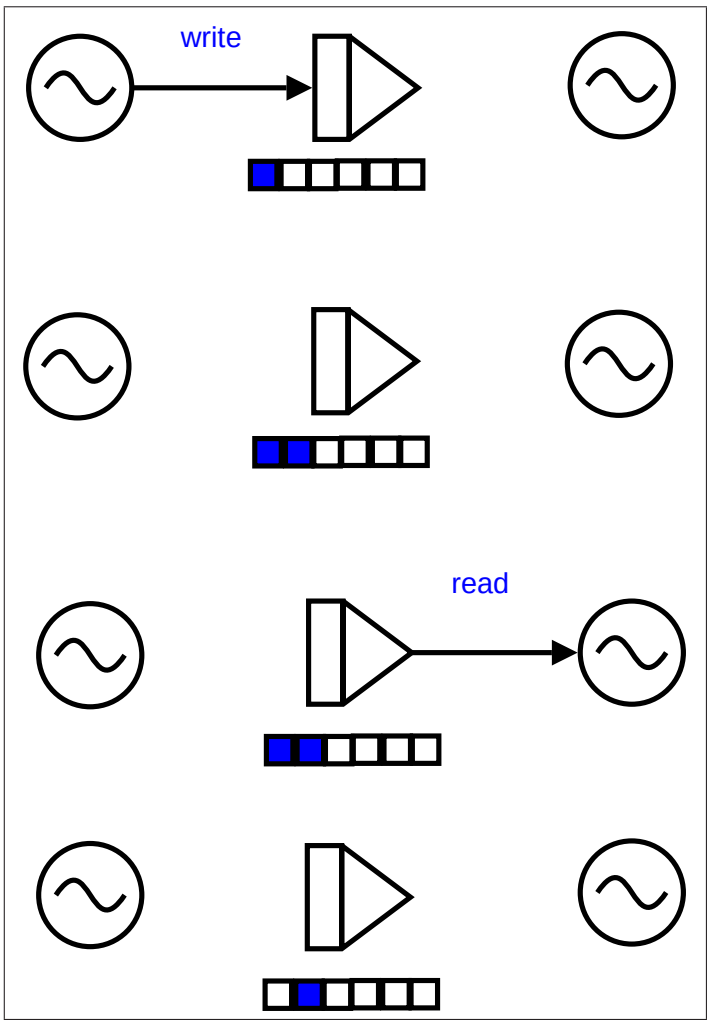

Figure 4.8: Two Threads using a Port.

Ports are solely used for data that actually contains information that is relevant for doing control. Implicit information like transfer information and synchronization information is stripped away by the Jarol Interface Ring and/or the Jarol Adaptation Layer.

4.2.4 Time Triggers

A time trigger generates a periodic signal for its awaiting threads. Figure 4.9 shows the symbol that is used throughout the thesis for time triggers.
A time trigger allows threads to wait on it and be signaled at a determined period. A time trigger is interally composed of a signal and a thread. The thread signals the signal at a given period to awake all waiting threads of the time trigger.

A time trigger TT is a 3-tuple (s, p, $Thread$), where:

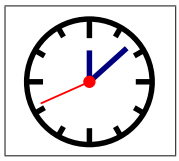

Figure 4.9: Symbol for Time Trigger.

- s is a signal S,
- $p \in \mathbf{N}$: the period that indicates how long the *Thread* sleeps until it emits another signal,
- *Thread* is a thread.

A time trigger provides one operation for a thread: await. See Figure 4.10 for a schematic of a thread on using a time trigger for time-triggered wakeup signal.

When a thread calls **await** on a time trigger:

- The calling thread is suspended.
- *State* is set to *waiting*.
- The number n is increased by one.

4.2.5 Links

A link provides a convenient way to abstract away different underlying connection schemes. Figure 4.11 shows the symbol that is used throughout the thesis for links. A link acts as a translator and forwarder from the external system to Jarol and back again. It relies on the use of ports and signals. Interally it uses threads for communication with the external system, thus reducing the complexity of the Jarol Core and Jarol Adaptation Layer implementation.

A link L is a 6-tuple $(RecvThread, SendThread, InputPort, OutputPort, s, Translator)$, where:

- $InputPort$ is a port P used for buffering messages received from the external system,
- $OutputPort$ is a port P used for buffering messages to send to the external system,
- $RecvThread$ is a thread that receives data from the external system and places it into the $InputPort$,
- $SendThread$ is a thread that sends data to the external system,
- s is a signal S that emits signal to the $SendThread$, causing the messages in the $OutputPort$ to be sent,

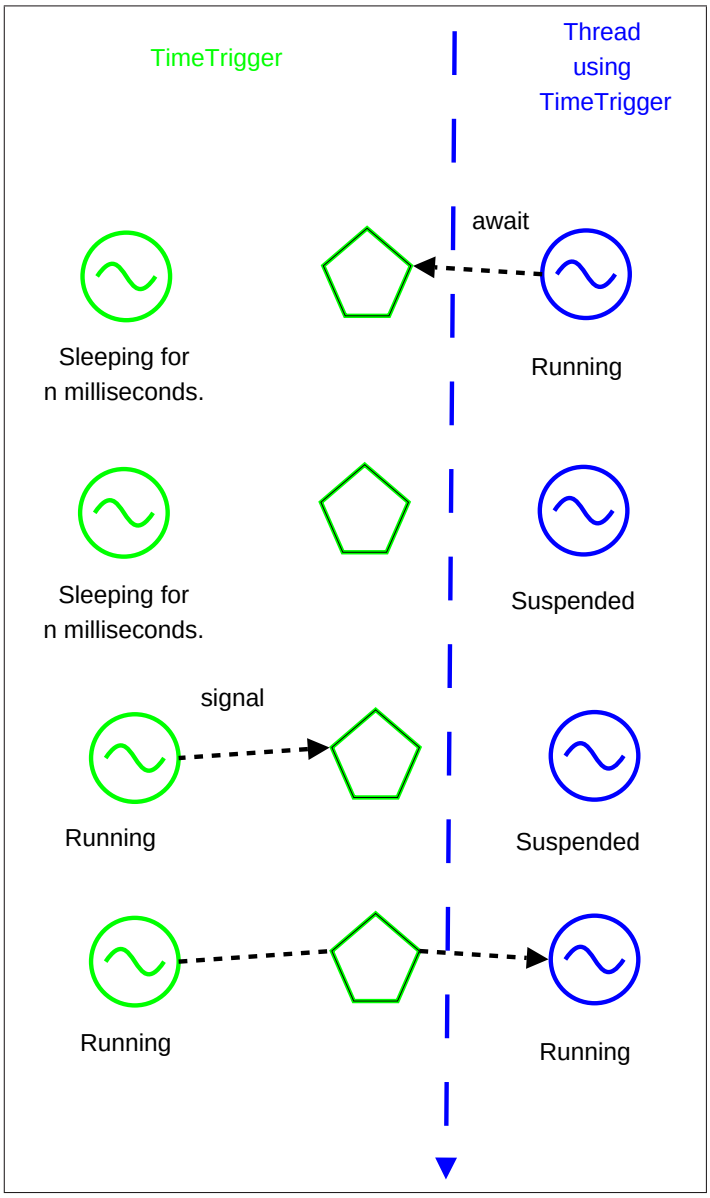

Figure 4.10: The Composition and Interaction of a Time Trigger with a Thread.

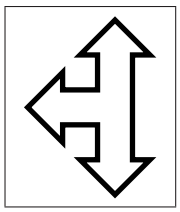

Figure 4.11: Symbol for Link.

- *Translator* is a translator that translates messages from the external system into Jarol Messages and the other way round.

A link provides two operations: connect and disconnect.

When **connect** is called on an unconnected link:

- A connection is set up.
- *RecvThread* and *SendThread* are created and started.

When **connect** is called on a connected link:

- Nothing.

When **disconnect** is called on a connected link:

- The link waits for any pending sending and receiving operations to complete.
- Stops the *RecvThread* and *SendThread*.
- Disconnects.

When **disconnect** is called on an unconnected link:

- Nothing.

Figure 4.12 shows the composition of the link and its usage of Jarol components.

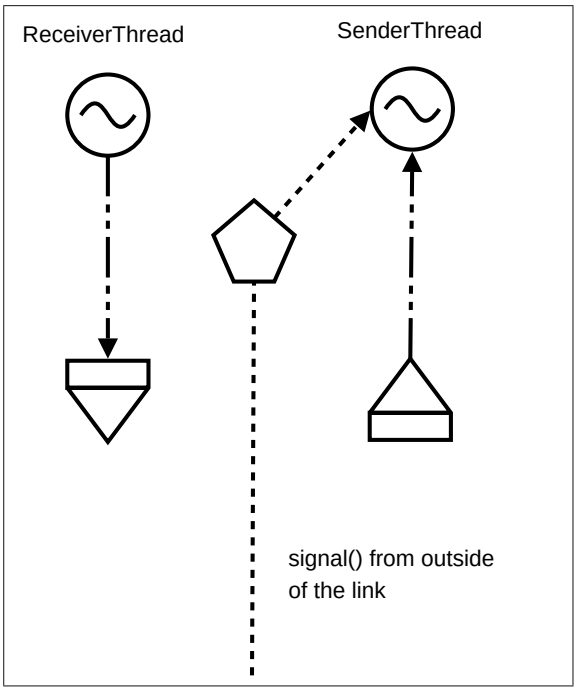

Figure 4.12: Link Composition

4.2.6 Jarol Messages

A Jarol Message is an arbitrary class that implements a set of operations, thus allowing the use of generic code in the port implementation. Jarol Messages are a convenient way for message transfer and manipulation in the Jarol system.

4.2.7 Message System

The main functions of the message system are the translation of project-specific messages into Jarol Messages, the creation of Jarol Messages, and communication between the external system and Jarol. Therefore, it is used in all Jarol layers, e.g., Jarol Interface ring where it converts the messages from the external system to Jarol Messages, inside the Jarol Core when new messages are created with the use of factories.

The message system manages all incoming messages from the external system, in order to do this, it is necessary to define any message of the project precisely. Thus allowing the conversion of all known incoming messages into Jarol Messages. Jarol Messages implement a small set of operations to support the management of the Jarol-specific data flow.

When a Jarol Message is sent to the external system, it is translated back into the appropriate format, so that it can be used by the external system. To ease up the message creation there are also message factories available that allow to generate Jarol Messages in an easy and convenient way.

Additionally, the message system provides facilities to create and manage connections with the external system that are represented as so called links. Links provide a set of operations to perform message passing into and out of Jarol with implicitly converting the messages into the appropriate format.

4.3 Formal Definitions of the Layers

To extend the theoretical view onto the layers, we formalized the layers according to the scheme used for the concepts. Since the layer definitions build upon the concept definitions they are not provided earlier.

4.3.1 External System

An external system EXS is a pair ($Sender$, $Receiver$), where

- $Sender$ is a component that sends messages to the $RecvThread$ of a link L,
- $Receiver$ is a component that receives messages from the $SendThread$ of a link L.

4.3.2 Jarol Interface Ring

A Jarol Interface Ring JIR is an n-tuple (L_1, L_2, L_3, ..., L_n), where:

- $\forall L$: L is a link.

4.3.3 Jarol Adaptation Layer

A Jarol Adaptation Layer JAL is an n-tuple ($Adapter_1$, $Adapter_2$, ..., $Adapter_n$), where:

- $n \in \mathbf{N}$ is the number of adapters.
- $\forall Adapter$: $Adapter$ is a thread.

4.3.4 Jarol Core

A Jarol Core JC is a 3-tuple ($Reader$,$Writer$,$Synchronizer$), where:

- $Reader$ is a runnable object that can read from ports P,
- $Writer$ is a runnable object that can write to ports P,
- $Synchronizer$ is an interface that can wait and signal on signals S.

Chapter 5

Implementation

In this chapter we provide a description of the Jarol implementation in Java. The implementation consists of several Java packages that provide classes, abstract classes, interfaces and exceptions. The corresponding classes are implementations of the concepts mentioned in Chapter 4, that are mainly used in the Jarol Interface Ring. The interfaces and abstract classes provide structure to implement the Jarol Core with common Java threads.

The packages are:

- `jarol` - Provides classes and interfaces for the Jarol infrastructure without messaging functionality.
- `jarol.exceptions` - Provides classes for the exception handling in the Jarol infrastructure.
- `jarol.messages` - Provides core messaging functionality.

5.1 Package: jarol

The `jarol` package provides the facilities to realize the Jarol Interface Ring and Jarol Adaptation Layer. Additionally it provides interfaces and abstract classes to ease up the structure of the Jarol Core implementation.

Overview of interfaces:

- `ActuatorInterface` - The `ActuatorInterface` should be implemented by any class whose instances should be an actuator interface.
- `JarolCoreInterface` - The `JarolCoreInterface` should be implemented by the class that performs the control operations.
- `MessageInterface` - `MessageInterface` should be implemented by any class whose instances should provide Jarol Message functionalities.
- `NavigationInterface` - The `NavigationInterface` should be implemented by any class whose instances should be a navigation interface.

- `SensorInterface` - The `SensorInterface` should be implemented by any class whose instances should be a sensor interface.
- `TerminalInterface` - The `TerminalInterface` should be implemented by any class whose instances should be a terminal interface.
- `TimeTriggerInterface` - `TimeTriggerInterface` should be implemented by any class whose instances should provide a periodic "tick" on watchers.

Overview of classes:

- `JarolCore` - Abstract class for controlling a vehicle, includes fields and implemented methods for a system with a vehicle and a terminal.
- `JarolCoreWithLink` - Abstract class for controlling a vehicle with the support of UDPLinks, includes fields and implemented methods for a system with a vehicle and a terminal.
- `Port` - `Port` is a data transfer point that provides concurrent access for a single reader and a single writer.
- `PortEnhanced` - `PortEnhanced` is a data transfer point that provides concurrent access for a single reader and a single writer with enhanced semantics that never refuse a write operation.
- `Signal` - `Signal` is a synchronization point, the `await()` call causes the calling thread to be suspended until a `signal()` call is performed by an arbitrary thread.
- `TimeTrigger` - `TimeTrigger` is a periodic signal.

5.1.1 ActuatorInterface

The `ActuatorInterface` is an interface that provides methods for structuring threads that implement interfaces that represent platform actuators.

Overview of methods:

- `void actuatePlant(MessageInterface[] actuatorData)` - Forwards the actuator data to the actuator unit.
- `void awaitJarolCore()` - Awaits a signal from the Jarol Core to continue.
- `MessageInterface[] readFromJarolCore()` - Reads the data that was provided by the Jarol Core.

5.1.2 JarolCoreInterface

The `JarolCoreInterface` is an interface that provides methods for structuring threads that implement the Jarol Core that executes the control code. Depending on the architecture some methods need not to be implemented, e.g., `signalNavigation()` is only needed if the design requires that the navigation unit is activated at a deterministic point.

Overview of methods:

- `void awaitNavigation()` - Waits for a signal from the navigation interface to continue.
- `void awaitSensor()` - Waits for a signal from the sensor interface to continue.
- `MessageInterface[] computeActuationData(MessageInterface[] sensorData, MessageInterface[] navigationData)` - Calculates the information (e.g., motor signals, thruster, fin positions, ...) and returns an array of messages containing this information.
- `void controlLoop()` - Should be called in the `run()` method and contains a loop that calls all the necessary methods of `JarolCoreInterface`.
- `void forwardToActuator(MessageInterface[] actuatorData)` - Forwards the actuation data to the actuator interface.
- `void forwardToNavigation(MessageInterface[] sensorData)` - Forwards sensor data to the navigation interface.
- `void forwardToTerminal(MessageInterface[] dataBundle)` - Forwards data to the terminal, e.g., sensor data, actuator data, etc. that should be used by the terminal.
- `MessageInterface[] readFromNavigation()` - Reads navigation data from the navigation interface.
- `MessageInterface[] readFromSensor()` - Reads data from the sensor interface.
- `void signalActuator()` - Signals the actuator interface to continue.
- `void signalNavigation()` - Signals the navigation interface to continue.
- `void signalTerminal()` - Signals the terminal interface to continue.

5.1.3 MessageInterface

The `MessageInterface` is an interface that provides three methods that allow generic programming in the port implementation and support for the Exotask system. Every instance of a class that implements the `MessageInterface` can be transfered along

with ports and used with Exotasks and Java threads.

Overview of methods:

- `Object clone()` - Clones this object, needed for Jarol port compatibility.
- `Object deepClone()` - Deep clones this object, needed for Exotask compatibility.
- `String toString()` - Returns a string representation of this object.

5.1.4 NavigationInterface

The `NavigationInterface` is an interface that provides methods for structuring threads that implement interfaces to navigation units.

Overview of methods:

- `void awaitExternalNavigationData()` - Awaits navigation data from the external navigation unit.
- `void awaitJarolCore()` - Awaits the Jarol Core to continue.
- `void forwardNavigationData(MessageInterface[] navigationData)` - Forwards the navigation data to the Jarol Core.
- `void forwardSensorDataToNavigationUnit(MessageInterface[] sensorData)` - Forwards the sensor data to the external navigation Unit.
- `MessageInterface[] readNavigationData()` - Reads the navigation data from the navigation unit.
- `MessageInterface[] readSensorData()` - Reads the sensor data from Jarol Core.
- `void signalJarolCore()` - Signals the Jarol Core.
- `void signalNavigationUnit()` - Signals the external navigation unit.

5.1.5 SensorInterface

The `SensorInterface` is an interface that provides methods for structuring threads that implement interfaces to sensors.

Overview of methods:

- `void awaitExternalSensor()` - Waits for input from the sensor unit.
- `void forwardToJarolCore(MessageInterface[] sensorData)` - Forwards the sensor data to the Jarol Core.
- `MessageInterface[] readSensorData()` - Reads the sensor data.
- `void signalJarolCore()` - Signals the Jarol Core to continue.

5.1.6 TerminalInterface

The `TerminalInterface` is an interface that provides methods for structuring threads that implement interfaces to terminals.

Overview of methods:

- `void awaitJarolCore()` - Awaits Jarol Core to continue.
- `void forwardDisplayData(MessageInterface[] displayData)` - Forwards the sensor data to the terminal application.
- `MessageInterface[] readDisplayData()` - Reads the display data.
- `void signalTerminal()` - Signals the terminal.

5.1.7 TimeTriggerInterface

The `TimeTriggerInterface` is an interface that provides methods for threads that should provide a periodic "tick" on watchers.

Overview of methods:

- `void await()` - Causes the calling thread to be suspended until the period is finished.
- `int getPeriod()` - Returns the current period.
- `void setPeriod(int period)()` - Sets the period in milliseconds.
- `void stopExecution()` - Stops the execution of the time trigger.

5.1.8 JarolCore

The abstract class `JarolCore` provides fields and implements some of the methods of the `JarolCoreInterface`. It provides the developer with base implementations for various methods. It supports the structuring of threads that implement the Jarol Core. Depending on the architecture some methods need not to be implemented.

Overview of methods - see `JarolCoreInterface`.

5.1.9 JarolCoreWithLink

The abstract class `JarolCoreWithLink` inherits the abstract class `JarolCore` and provides two links for the communication with the platform. One is supposed to be used with the vehicle and one with the terminal.

Overview of methods that are not included in `JarolCore`:

- `UDPLink setupUDPLink(MessageFactory messageFactory, java.lang.String host, int localPort, int remotePort, int receiveBufferSize, int sendBufferSize)` - Creates an `UDPLink` on the `localPort` to a given host on its `remotePort`.

5.1.10 Signal

A signal is a synchronization point that allows different threads to synchronize and coordinate with each other. Presenting the developer with a scheme that allows him to implement a strict synchronization protocol. This is supported by strong semantics. Every operation on a signal either produces the indented reaction or creates an exception that includes information why the intended operation could not be performed.

Overview of methods:

- `await()` - Causes the caller to be suspended until `signal()` is called.
- `signal()` - All calling thread of `await()` call are resumed.

If the intended behavior cannot be fulfilled a subclass of `SignalException` or an `InterruptedException` is thrown. There are four different cases:

1. `signal()` is invoked and there is no thread waiting. A `SignalNotAwaitedException` is emitted.
2. `signal()` is invoked while signaling is in progress. This is the result of a not completed execution of a previous signal call, when a signal is called. This means that the wake up phase of the awaiting thread(s) was interrupted. Hence a `SignalDuringSignalingException` is thrown.
3. `await()` is invoked while signaling is in progress. This is the result of a not completed execution of a previous signal call, when an await is called. This means that the wake up phase of the awaiting thread(s) was interrupted. Resulting in the throw of a `AwaitDuringSignalingException`.
4. The awaiting thread is interrupted, this happens if an `interrupt()` is called.

We implemented signals with the use of `await()`, `notify()`, a counter (`watching`) and a `condition` variable.

signal()

The algorithm in Figure 5.1 presents the source of code of the `signal()` method. The `watching` variable indicates how many threads are awaiting this signal. The `condition` variable is set to false at initialization. Its purpose is to indicate if the thread is allowed to continue its execution and to detect a signaling conflicts.

At the beginning the condition variable is checked. If it is `true` a signal was performed before and no `await()` was called meanwhile, thus we emit a `SignalDuring-`

`SignalingException`. Else we check if there are any threads watching on this signal, if not we emit a `SignalNotAwaitedException`, because a signal must be awaited. If no exception occurred, we set the condition `true` and notify all threads waiting on this monitor.

```
public synchronized void signal()
   throws SignalDuringSignalingException, SignalNotAwaitedException
{
   if (condition)
      throw new SignalDuringSignalingException();
   if (watching < 1)
      throw new SignalNotAwaitedException();
   condition = true;
   notifyAll();
}
```

Figure 5.1: Source code for signal().

await()

The algorithm in Figure 5.2 presents the source of code of the `await()` method. The variable `watching` indicates how many threads are awaiting this signal. The condition variable is set to false at initialization. It indicates if the thread is allowed to continue its execution and to detect signaling conflicts.

First we check if the `condition` variable is set correctly, if it is `true` this would indicate that a signal was called and an `await()` was called afterwards before a thread could awake from the evoked signal. Thus we emit an `AwaitDuringSignalingException`. Then we increase the number of watching threads by one. In the try block we use a while loop that checks `condition` in order to prevent wake-ups from wrong notifications.
The finally block is executed in any situation, thus when a threads awakes after calling `wait()` the watching counter is decremented, when it is smaller than zero the condition is set to `false` again, indicating that the last thread has been awaked (and the last signal was used up).

5.1.11 Port

A port acts as a data transfer point that provides concurrent access for a single reader and a single writer. Objects that implement the `MessageInterface` can write into and read from it. It contains a circular FIFO buffer, if its capacity is reached the writer is rejected, until an object is removed from a port. We also implemented a port with different buffer semantics (see `PortEnhanced`). Since Java uses call-by-reference, the object written into the buffer needs to be cloned. The method for cloning is provided by the `MessageInterface`.

```
public synchronized void await()
   throws AwaitDuringSignalingException
{
   if (condition)
      throw new AwaitDuringSignalingException();
   watching++;

   try
   {
      while (!condition) //in order to avoid "wrong" notifications/wake-ups
         wait();
   }
   catch (InterruptedException e)
   {
      throw new RuntimeException("awaiting interrupted.");
   }
   finally //executed in anyway
   {
      watching--;
      if (watching < 1)
         condition = false;
   }
}
```

Figure 5.2: Source code for await().

The port main operations are `read()` and `write()`, which are non-blocking. Additionally there is a method to retrieve the buffer size.

Overview of methods:

- `int getBufferSize()` - Returns the buffer size.
- `MessageInterface read()` - Reads the oldest element from the buffer.
- `boolean write(MessageInterface item)` - Writes an item into the buffer.

5.1.12 PortEnhanced

PortEnhanced is a data transfer point that provides concurrent access for a single reader and a single writer with enhanced semantics that will never reject a write operation. The internal buffer we call *Circular Pushing Buffer*. This kind of buffer does not reject a write operation even if the buffer is full. In case of a full buffer a write results in the overwriting of the oldest object with the new object.

Overview of methods:

- `int getBufferSize()` - Returns the buffer size.
- `MessageInterface read()` - Reads the oldest element from the buffer.
- `boolean write(MessageInterface item)` - Writes an item into the buffer.

The Circular Pushing Buffer is a circular buffer for a single writer and a single reader. Contrary to existing implementations like [35] our writer is able to overwrite old values, thus pushing the reader forward, when it is too slow or inactive. This provides a storage for current data within a user-determined boundary.

We assume that simple integer operations are performed atomically, this is supported by a Java implementation compliant with the Java Language Specification [37]. We do not use any synchronization primitives, like test-and-set, monitors, semaphores, etc.

The algorithm follows these principles:

- The writer is always allowed to write, this means the write operation is always successful.
- The reader attempts to read. The read operations returns `null` if the buffer is empty, else it retries until a valid value is read.
  - The reader checks in the read attempt, if its index is still in a valid range:
    - Valid range: read value and move read pointer one slot forward.
    - Invalid range: *push forward*: sets the read index to the oldest element, a normal read (see valid range) is performed afterwards.

We use four integer indices to provide protection of the critical section, manage the access to the buffer elements and determine if the read index is in a valid range. These indices are composed of two indices for the actual element in the buffer (called `readIndex` and `writeIndex`) and two indices to save the number of wraparounds of the former. We call the number of wraparounds *levels* (`readLevel` and `writeLevel`), since they indicate the age of the related index.

Algorithm 1 void write(MessageInterface item)

```
1: //ASSUMPTION: writeIndex is even
2: writeIndex := writeIndex + 1;
3: writeBuffer(writeIndex >> 1, item);
4: if (writeIndex + 1 = bufferSize * 2) then
5:    writeLevel := writeLevel + 1;
6:    writeIndex := 0;
7: else
8:    writeIndex := writeIndex + 1;
9: end if
```

The write() Method can be seen in Algorithm 1. It uses a `writeIndex` that is similar to a concurrency control field as seen in [36]. The writeIndex has two purposes: (1) as an index for the writer and (2) as a concurrency control field that indicates the entry in and exit of a critical section, similar to [35].

When `write()` is called we assume, that `writeIndex` is even. At the beginning `writeIndex` is increased to an odd value, this indicates that writing is in progress.

Then the write to the buffer is executed. The `writeLevel` is increased if a wraparound will occur (this is the case if `writeIndex` would exceed the buffersize). The update of the level needs to be done in the critical section. Finally, the `writeIndex` is set to an even value, to indicate the exit of the critical section.

Algorithm 2 MessageInterface read()

```
1: while true do
2:   savedWriteIndex := writeIndex;
3:   savedWriteLevel := writeLevel;
4:   if (savedWriteIndex is even) then
5:     //have we read a consistent pair of index and level?
6:     if (savedWriteIndex = writeIndex) AND (savedWriteLevel = writeLevel)
       then
7:       if (savedWriteLevel = readLevel) then
8:         if (readIndex = savedWriteIndex) then
9:           //buffer empty
10:          return NULL;
11:        end if
12:      else
13:        //ASSUMPTION: Integer.MIN_VALUE = Integer.MAX_VALUE + 1
14:        if (savedWriteLevel != readLevel + 1) OR (savedWriteIndex > readIn-
           dex) then
15:          //writer passed reader, set reader to oldest element
16:          readIndex := savedWriteIndex;
17:          //ASSUMPTION: Integer.MAX_VALUE = Integer.MIN_VALUE - 1
18:          readLevel := savedWriteLevel - 1;
19:        end if
20:      end if
21:      item := readBuffer(readIndex >> 1);
22:      //have we read a consistent copy of the buffer element?
23:      if (savedWriteIndex = writeIndex) AND (savedWriteLevel = writeLevel)
         then
24:        readIndex = readIndex + 2;
25:        if (readIndex = bufferSize*2) then
26:          readIndex := 0;
27:          //ASSUMPTION: Integer.MIN_VALUE = Integer.MAX_VALUE + 1
28:          readLevel := readLevel + 1;
29:        end if
30:        return item;
31:      end if
32:    end if
33:  end if
34: end while
```

The read() Method can be seen in Algorithm 2. It returns `null` if an empty buffer is detected or loops until a successful read is performed. It begins with the saving of the current `writeIndex` and `writeLevel` in the local `savedWriteIndex` and `savedWriteLevel`. In line 4 it determines if there is no writing in progress.[1] Then we check the consistency of `savedWriteIndex` and `savedWriteLevel`. Next we check for equality of the `savedWriteLevel` and `readLevel`. If they are equal, we check if `savedWriteIndex` and `readIndex` are also equal. If this is the case, they point at the same slot, thus we have an empty buffer. If `savedWriteIndex` and `readIndex` are not equal, we still know that the levels are equal, hence the `readIndex` is in a valid range, therefore the value is read out of the buffer in line 21.

If the `savedWriteLevel` and `readLevel` are not equal, we need to check if the distance between them is in range, if not a *push forward* of the `readIndex` and `readLevel` is necessary. Since we know, that `savedWriteLevel` and `readLevel` are not equal, we can do an optimized checking. If the `writeLevel` is not bigger by one than the `readLevel`, we are out of range, hence we need to *push forward*. If it is not the case (`writeLevel` = `readLevel` + 1), it is still possible that the writer has overtaken the reader, this would be the case if the `savedWriteIndex` is bigger than the `readIndex`, thus we need to do a *push forward*. This is done in line 16 to 18. Else we would just continue to line 21.

In line 21 the value from the buffer is read. Next is a test on `savedWriteIndex` and `savedWriteLevel` to determine if no write occurred meanwhile (before and during the read). If this holds the read value is valid. In this case, the following is performed: The `readIndex` is increased by two, then we check if `readIndex` is two times the size of the buffer, this indicates a wrap around, hence `readIndex` is set to 0 and `readLevel` is increased by one. Finally we return the read item.

The theoretical limitation in this algorithm is the following: Assume the writer writes so often, that even the `writeLevel` is wrapped around. Hence, the reader appears to be in the valid range, thus there is no accurate update of the `readIndex` and `readLevel`. Practically the probability of this is situation is unlikely: it is $\frac{2}{bufferSize * MaxSizeOfInt}$.

5.1.13 TimeTrigger

A `TimeTrigger` is a signal that performs the signaling on its own every n milliseconds. It is similar to a signal with the difference that it only provides an `await()` call and no primitive for signaling. The `TimeTrigger` has a timed thread that performs a signal every n milliseconds to all awaiting threads. The `TimeTrigger` is used to introduce time-triggered behavior without the explicit use of `sleep()` by the user.

Overview of methods:

- `void await()` - Causes the calling thread to be suspended until the period is finished.

[1] In all following cases if the check is not passed the reader loops back to the beginning, except it is noted otherwise.

- `int getPeriod()` - Returns the current period.
- `void run()` - Performs the sleeping and waking up.
- `void setPeriod(int period)()` - Sets the period in milliseconds.
- `void stopExecution()` - Stops the execution of the internal timing thread.

5.2 Package: jarol.messages

The `jarol.message` Java package provides core messaging functionality. The package defines classes and interfaces needed to operate with messages. This package was implemented by Eduardo Marques.

Overview of interfaces:

- `MessageFooter` - The `MessageFooter` interface should be implemented by any class whose instances should be encoding the footer of the message set.
- `MessageHeader` - The `MessageHeader` interface should be implemented by any class whose instances should be encoding the header of the message set.
- `MessagePart` - The `MessagePart` interface should be implemented by any class whose instances should be encoding of the parts of the message set.

Overview of classes:

- `Buffer` - Buffer class for serialization.
- `Link` - Jarol-wise, non-blocking, lock-free bidirectional message link abstract class.
- `Message` - Base class for messages.
- `MessageFactory` - Message factory interface.
- `SerializationHandle` - Application parameterization for message factories and serialization.
- `TCPClientLink` - TCP client socket message link.
- `TCPServerLink` - TCP server socket message link.
- `UDPLink` - UDP socket message link.

5.2.1 Link

The abstract class `Link` implements a large part of the communication with the external system, it also converts the incoming messages into Jarol Messages. This conversion is done with message factories. Link uses Jarol ports and signals. Additionally, it uses threads to read and write to the external system in a non-blocking way. It is an abstract class that allows to implement actual network links. In Jarol we have concrete links that implement links for UDP and TCP.

Overview of methods:

- `void connect()` - Connect the message link.
- `boolean connected()` - Check if link is connected.
- `void disconnect()` - Disconnect the message link.
- `void getRecvPort()` - Get receive port.
- `void getSendPort()` - Get send port.
- `void getSendSignal()` - Get send signal.
- `abstract int recv(byte[] buf, int off, int len, int timeout)` - Abstract method used to receive data.
- `abstract void send(byte[] data, int off, int len)` - Abstract method used to send data.
- `abstract void start()` - Abstract method for starting the link.
- `abstract void stop()` - Abstract method for stopping the link.

Overview of classes that inherit `Link`:

- `TCPClientLink` - This message link allows simple bi-directional communication using a client TCP socket connected to a specified remote TCP server socket.
- `TCPServerLink` - This message link allows simple bi-directional communication using a server TCP socket.
- `UDPLink` - This message link allows simple bi-directional communication between UDP ports. The local host will listen on a specified local port for messages and send messages to a specified UDP port on a remote host.

5.2.2 MessageFactory

The abstract class `MessageFactory` defines factory methods to instantiate messages, message headers and footer. It provides methods for serialization and de-serialization enabling a convenient facilities to communicate with the external system.

Since `MessageFactory` is used inside of links we omit the listing of the methods.

5.3 Package: jarol.exceptions

The `jarol.exceptions` Java package contains the Jarol specific exceptions.

Overview of classes:

- `AwaitDuringSignalingException` - This exception is thrown by the signal to indicate that an `await()` was called while signaling was in process.
- `SignalDuringSignalingException` - This exception is thrown by the signal to indicate that a `signal()` was called while signaling was in process.
- `SignalException` - Thrown by signal to indicate a signaling violation.
- `SignalNotAwaitedException` - This exception is thrown by the signal to indicate that a `signal()` was called without an awaiting thread on this signal.

Chapter 6

Application

In this chapter we demonstrate the use of the Jarol infrastructure with the JAviator and Seascout systems. Additionally we use different concurrency models to show the flexibility of the infrastructure. Thus we implement:

(1) the JAviator with Java threads,

(2) the LAUV with Java threads,

(3) the LAUV with Exotasks.

6.1 The Jarol JAviator

In the Jarol JAviator application we use the Jarol infrastructure to interface the control code of the JAviator. The goal is to leave the code base of the JAviator unchanged and interface the existing control code within the Jarol infrastructure.

We use the JAviator control code, the MockJAviator, and the Control Terminal. For details on these components please refer to Chapter 2.

6.1.1 Structure

As can be seen in Figure 6.1 the Jarol JAviator has two elements that represent the external system (red). These are the MockJAviator that simulates the JAviator and the Control Terminal that reads user input and displays the sensor and actuation data. In this case the external system is directly connected to the Jarol Adaptation Layer (yellow) that consists of three threads: SensorThread, ActuatorThread and TerminalThread. These threads use signals and ports to interact with the Jarol Core thread (blue), where the control code is executed.

The data flow inside of Jarol is solely organized via ports, thus the different threads only read from and write to ports. The control flow is made explicit with signals. Signals are used for thread synchronization.

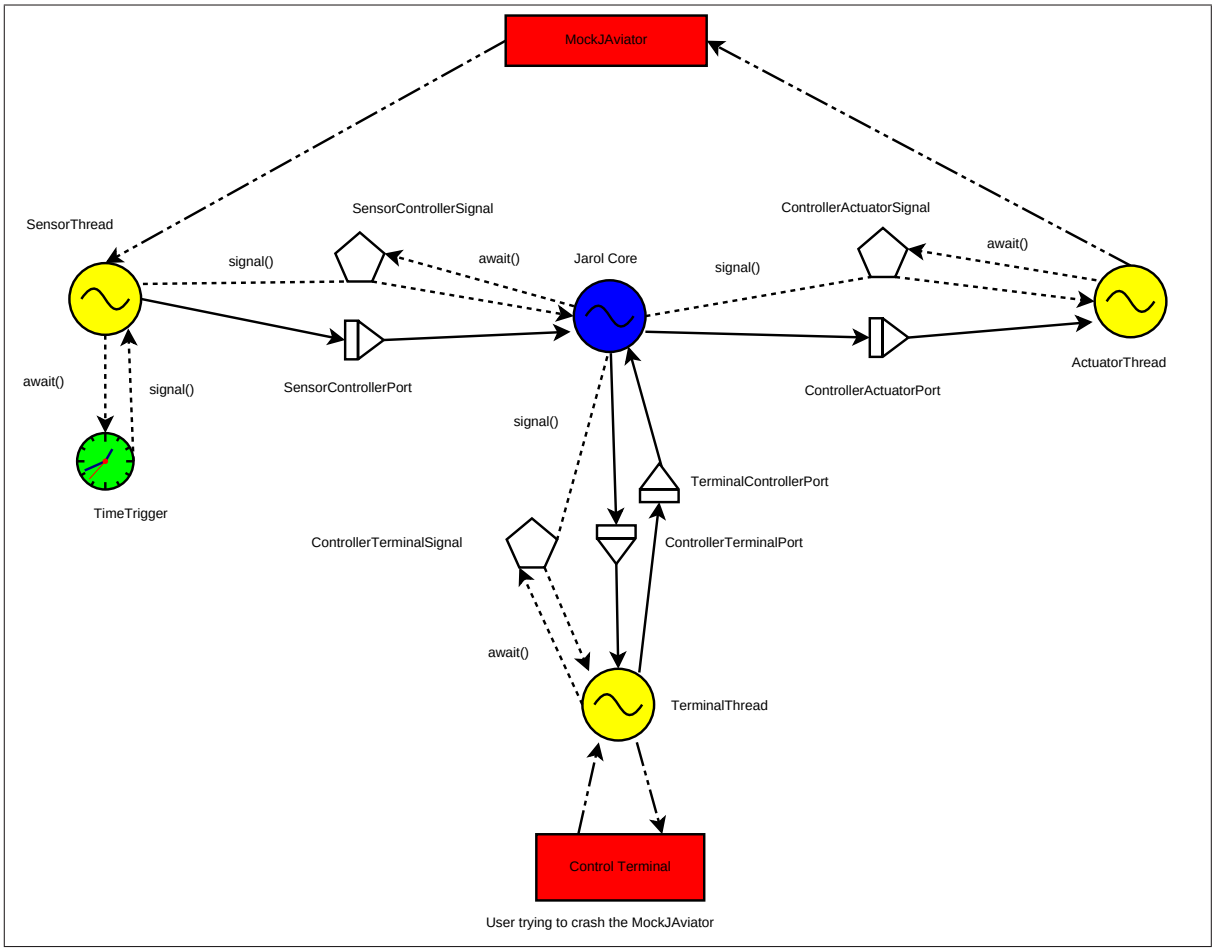

Figure 6.1: Jarol JAviator

6.1.2 Jarol Adaptation Layer

The Jarol Adaptation Layer (yellow) is the layer between the Jarol Interface Ring and the Jarol Core. In this case the layer contains three threads: SensorThread, ActuatorThread and the TerminalThread. Within this layer the translation of the JAviator protocol to an explicit data and control flow is performed. This means the JAviator messages are converted into Jarol Messages (data flow) and signals are emitted on JAviator messages that contain synchronization information (control flow).

SensorThread

The SensorThread is used to read messages from the MockJAviator, it synchronizes with the Jarol Core for which it converts and forwards these messages. In Figure 6.1 the SensorThread is located at the left side. It interacts with the Jarol Core via the SensorControllerPort and SensorControllerSignal.

Output ports: SensorControllerPort
Input ports: -
Signaling signals: SensorControllerSignal
Awaiting signals: -

The SensorThread is time-triggered by a TimeTrigger that sends a tick every 20 ms. The received messages are converted into Jarol Messages and written to a port (SensorControllerPort) that is read by the Jarol Core. When the sensor data is read a signal is emitted on the SensorControllerSignal, because the sensor message from the JAviator is also used for synchronization. This signal triggers the Jarol Core that waits on SensorControllerSignal and starts its execution. After signaling the Jarol Core the SensorThreads awaits the next timed signal from the TimeTrigger.

ActuatorThread

The ActuatorThread is used to read messages from the Jarol Core, it converts and forwards these message to the MockJAviator. In Figure 6.1 the ActuatorThread is located at the right side. It interacts with the Jarol Core via the ControllerActuatorPort and ControllerActuatorSignal.

Output ports: -
Input ports: ControllerActuatorPort
Signaling signals: -
Awaiting signals: ControllerActuatorSignal

The ActuatorThread is triggered by the Jarol Core. When the application starts, the ActuatorThread awaits the Jarol Core, going in a waiting state immediately. When it is triggered it converts the Jarol Message read from the ControllerActuatorPort to a JAviator message and forwards it to the external system (MockJAviator). Then it

awaits the next signal from the Jarol Core.

TerminalThread

The TerminalThread is used to interact between the Control Terminal and the Jarol Core. The TerminalThread reads and forwards data from the Jarol Core to the Control Terminal. It also forwards the user input from the Control Terminal to the Jarol Core. In Figure 6.1 the TerminalThread is located near the bottom of the figure above the Control Terminal. It interacts with the Jarol Core via the TerminalControllerPort, ControllerTerminalPort and ControllerTerminalSignal.

Output ports: TerminalControllerPort
Input ports: ControllerTerminalPort
Signaling signals: -
Awaiting signals: ControllerTerminalSignal

The TerminalThread is triggered by the Jarol Core. The Jarol Messages from the Jarol Core are converted into messages that can be interpreted by the Control Terminal. When the application starts, the TerminalThread awaits the Jarol Core thus entering a waiting state immediately. When triggered for the first time it establishes a connection to the Control Terminal. If a connection is established, it reads the data provided by the Jarol Core sending it to the Control Terminal. Then it reads the user input (navigation data) from the Control Terminal converts it to a Jarol Message and forwards it to the Jarol Core. Afterwards it awaits the next signal from the Jarol Core.

6.1.3 Jarol Core

The Jarol Core does the execution of the control code, to do this it requires sensor and navigation data. The Jarol Core is a thread that reads from the SensorThread and TerminalThread, writes to the ActuatorThread and TerminalThread. In Figure 6.1 the Jarol Core is located in the center. It interacts with all three threads from the Jarol Adaptation Layer: SensorThread, ActuatorThread and TerminalThread via various ports and signals.

Output ports: ControllerActuatorPort, ControllerTerminalPort
Input ports: SensorControllerPort, TerminalControllerPort
Signaling signals: ControllerActuatorSignal, ControllerTerminalSignal
Awaiting signals: SensorControllerSignal

The Jarol Core is triggered by the SensorThread. When the application starts, the Jarol Core awaits the SensorThread thus entering a waiting state immediately. When it is triggered it reads the sensor data from the SensorControllerPort and forwards this information via the ControllerTerminalPort to the TerminalThread and signals it on the ControllerTerminalSignal. Then it computes the motor information and forwards it via the ControllerActuatorPort to the ActuatorThread and signals it with the Con-

trollerActuatorSignal. Additionally the current helicopter state is evaluated and sent to the TerminalThread including the actuation data via the ControllerTerminalPort. Then it reads the navigation data from the TerminalControllerPort and awaits the signal from the SensorThread.

6.2 The Jarol LAUV

In the Jarol LAUV we use the Jarol infrastructure to interface the control code for the submarine of the Seascout project. The goal is to leave the Seascout software architecture unchanged, except for the control code that we migrated from DUNE (C/C++) into the Jarol Core (Java). Eduardo Marques did the conversion of the three controllers (depth, heading, speed) written in C/C++ to Java.

We use the Neptus Seascout Terminal and DUNE that runs an environment simulator, but without the control code for submarine. We use the message set provided by the USTL, from which we generated Java classes that implement the `MessageInterface`.

6.2.1 Structure

As can be seen in Figure 6.2 the Jarol LAUV has two elements that represent the external system (red). These are DUNE and the Neptus Seascout. DUNE is running the simulator for the environment, whereas Neptus is a user terminal that displays data and reads user commands for maneuvering the submarine.
The external system is connected to the Jarol Adaptation Layer and the Jarol Interface Ring. We only need one thread in the Jarol Adaptation Layer, because the Jarol Interface Ring is able to fulfill most of the work, due to the use of Jarol links that provide conversion of the messages.

The data flow inside of Jarol is solely organized via ports, thus the different threads and links exchange data via the ports. The control flow is made explicit with signals. Signals are used for thread synchronization, we also use signals to synchronize with the links that use internal threads.

In this approach we used Jarol links, this allowed us to reduce the number of threads needed in the Jarol Adaptation Layer to one in contrast to three in the Jarol JAviator. The links provide the message conversion and threads for non-blocking communication. Thus we have a thick Jarol Interface Ring (green) and a thin Jarol Adaptation Layer (yellow) in contrast to the Jarol JAviator, as can be seen quite clear due to the difference in the color representation in Figure 6.2 and Figure 6.1.

6.2.2 Messages

The Seascout project has a clearly defined message set. This message set is written in XML, Eduardo Marques used an XLST generator to generate Jarol Messages with an

according `MessageFactory` out of the definition file. The `MessageFactory` implements the de-/serialization of these messages. This results in an implicit conversion of the messages when the pass through the links into or out of Jarol.

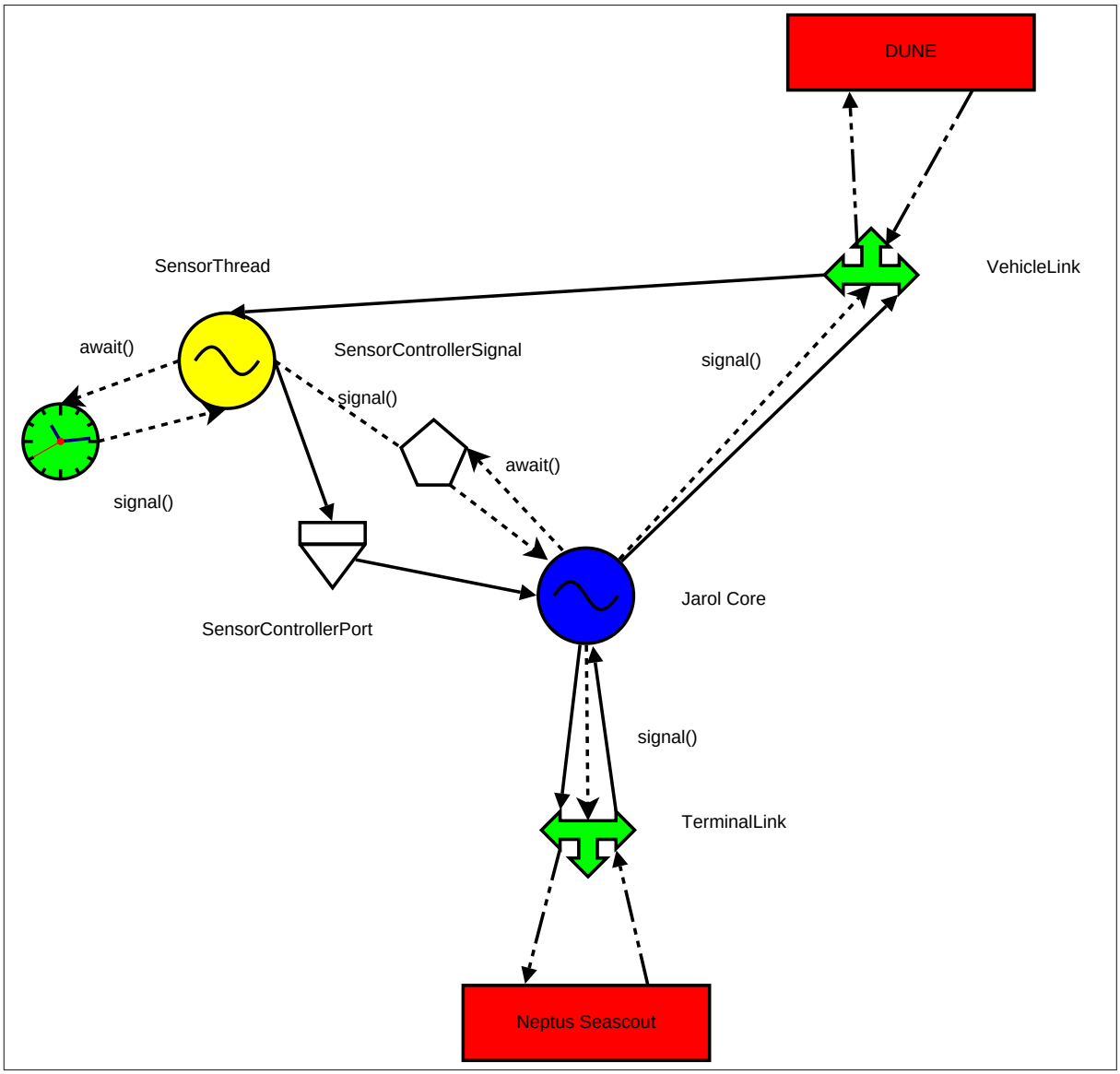

Figure 6.2: Jarol LAUV

6.2.3 Jarol Adaptation Layer

In contrast to the Jarol JAviator implementation the Jarol Adaptation Layer (yellow) in this project is thin. It consists of only one thread. This is due to the use of links that reside inside the Jarol Interface Ring and provide facilities to interface between the external system, the Jarol Adaptation Layer and the Jarol Core. This can seen in Figure 6.2. Notice the difference in comparison with the Figure 6.1 from the Jarol JAviator .

In this layer the translation of the Seascout protocol to an explicit data and control flow is performed. This means the Seascout messages are converted into Jarol Messages (data flow) and signals are emitted on Seascout messages that contain synchronization information (control flow).

SensorThread

The SensorThread reads messages from DUNE via the VehicleLink that provides a port. In Figure 6.2 the SensorThread is located on the left side of the figure. It interacts with the Jarol Core via the SensorControllerPort and SensorControllerSignal.

Output ports: SensorControllerPort
Input ports: One in the VehicleLink
Signaling signals: SensorControllerSignal
Awaiting signals: -

The SensorThread is time-triggered by a TimeTrigger that sends a tick every 20 ms. The messages from the VehicleLink are read and converted into Jarol Messages and written to the SensorControllerPort that is read by the Jarol Core. When the input port of the VehicleLink is read completely - means it is empty - by the SensorThread the Jarol Core is signaled via the SensorControllerSignal. After the signaling the Core the Sensor Threads awaits the next timed signal from the TimeTrigger.

6.2.4 Jarol Core

The main task of the Jarol Core is the computation of the control data to maneuver the submarine. The Jarol Core (blue) interacts with the SensorThread, the VehicleLink and the TerminalLink. In Figure 6.2 the Jarol Core is located in the center of the figure.

Output ports: One in TerminalLink, one in VehicleLink
Input ports: SensorControllerPort, one in TerminalLink
Signaling signals: One in TerminalLink, one in VehicleLink
Awaiting signals: SensorControllerSignal

The Jarol Core is triggered by the SensorThread. When the application starts the Jarol Core awaits the SensorThread. The messages from the SensorThread and the TerminalLink are used to compute the actuation data. Due to the use of links there is only the SensorThread necessary to communicate with the external system.

6.3 The ExoLAUV

The ExoLAUV is the attempt to use Jarol for interfacing the LAUV platform and the Exotask concurrency model. The corresponding Exotask graph can be seen in Figure 3.1. This application shows the flexibility of the infrastructure in regards to the concurrency models.

The setup is as follows, DUNE contains the simulator for the LAUV and executes the control code. The Neptus terminal is displaying the sensor data and reading user input for controlling the submarine. Between DUNE and Neptus is the Exotask graph that transfers the data between the two entities. The challenge with this setup is to bring data from outside the Exotask graph into it and transfer the data from the inside of the Exotask graph to the outside again. The purpose of Jarol is to interface the platform and the concurrency models.

Note that Eduardo Marques has implemented a similar program that implements the control code in an Exotask system. Since his implementation was finished a few days before the submission of this thesis and the fact that we showed that interfacing Jarol and Exotasks can be done, we do not include a discussion of this implementation.

6.3.1 Structure

The ExoLAUV uses two Jarol links to communicate with the external system, since every link has two Jarol ports, these links provide four Jarol ports in total. We use the Jarol ports to transfer the data from the external system to the Exotasks and from the Exotasks to the external system. As connection points we use four Exotasks communicators: fromDune, fromNeptus, toNeptus, toDune as can be seen in Figure 3.1.

These communicators are connected with the ExoLAUVController, which is a compute task. In this case, it just forwards the data received from the input communicators to the output communicators. This is sufficient since the challenge is to pass data through the Exotasks.

In Figure 6.3 we can see the composition of Jarol with an Exotask graph, the Exotask graph is the same as in Figure 3.1. The white boxes are the communicators and the blue box (Jarol Core) is the compute task. The DuneLink and NeptusLink are connected to the external system - not shown in the figure. The links are connected with the distributer that transfers the data from the outside of the Exotask graph to the inside. This distributer is necessary to ensure the isolation requirements of the Exotasks.

6.3.2 Jarol Distributer

Due to the isolation model of the Exotasks it is necessary to have a facility that interfaces between the inside and outside of an Exotask graph. This facility is called

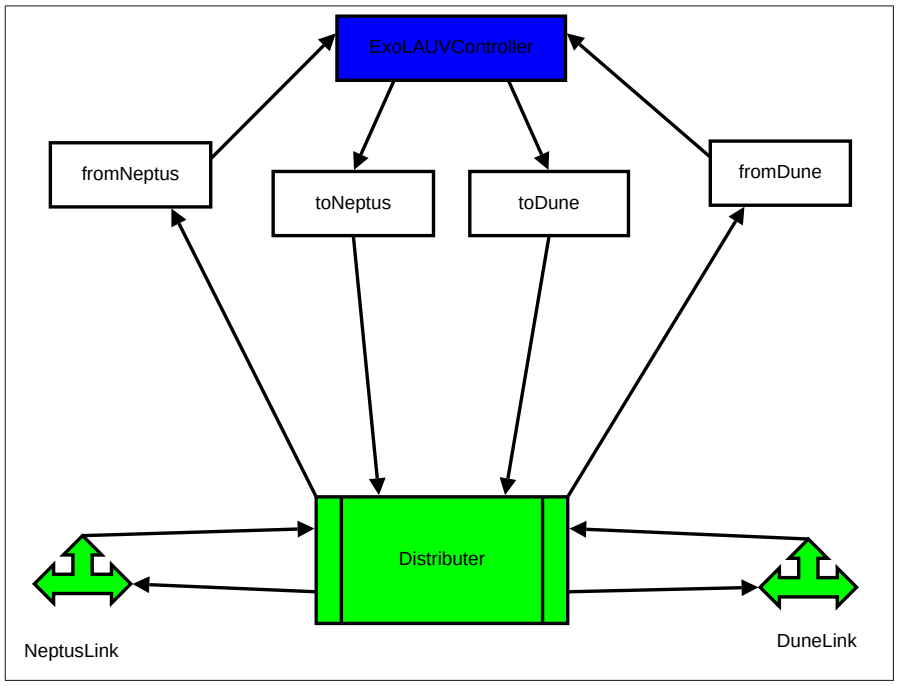

Figure 6.3: ExoLAUV

distributer and is used to extend the Exotask graph across machine boundaries, in this case a virtual machine boundary.

Exotask distributers are still in a development stage, one of the first has been used in the JAviator project. In the future there should be a generic distributer, but for this application we only developed a functional prototype for a Jarol distributer.

The distributer is an Exotask that transfers all the inputs from the links to the communicators and transfers all the outputs of the communicators to the links again. Most of the parts need to be implemented by the developer, but the structure and the use of generic objects like Jarol ports suggest that we should be able to develop a generic Exotask distributers in the near future.

6.3.3 Jarol Core - ExoLAUVController

The ExoLAUVController is a compute task where the control code of the ExoLAUV would reside. Since the Exotask system is still in development and the distributer system is also likely to change in the near future, we just implemented a simple version that shows that we can get data from outside of the Exotask graph into the graph and transfer the data back to the outside system. Note that the major obstacle is the transfer of data from/into the Exotasks system not the execution of the control code.

The structure of the ExoLAUVController user code is as follows: first we read sensor data from the fromDune communicator, then we forward this data to the toNeptus communicator. Next we read navigation data from the fromNeptus communicator.

We forward this data to the toDune communicator. This operation sequence will be extended with the computation of actuation data. Thus we would forward actuation data calculated from the ExoLAUVController with the navigation data from Neptus, instead of doing this calculations inside of Dune.

6.4 Summary

The different implementations of the heterogeneous system demonstrate the flexibility of the Jarol infrastructure. The implementation of the Jarol JAviator makes apparent that the structuring of the source code with abstract classes improves the readability and the understanding of the source code. Due to the use of ports and signals the information flows are now made explicit at the interfaces making it easier to maintain the source code and the whole design.

The Jarol LAUV is a different case where we do not use the source code[1] of the existing system which was implemented in C/C++. The large software architecture and the tool chain are different from the Jarol JAviator project, nevertheless we could apply Jarol with this system. Sometimes it was even easier to apply Jarol since the boundaries are much clearer.

In the ExoLAUV project we demonstrate that the Jarol Core of the Jarol LAUV can be implemented with a different concurrency system without major changes to the Jarol Interface Ring.

[1] Except for the extraction of the control code to convert it to Java.

Chapter 7

Conclusions

We have presented the *Jarol* control infrastructure that allows the developer to focus on the implementation of control code. Jarol provides a platform and concurrency interface that enables abstraction of hardware details and allows the use of different language-supported concurrency models. Implemented entirely in Java, the application writer can rely on strong typing, language-based concurrency support, and dynamic memory management using Jarol. We have implemented Jarol applications with interfaces to Java threads and Exotasks. This shows that Jarol can be used with different concurrency models for execution of control code.

We begun the research with the evaluation of the JAviator and Seascout platforms. During this process we encountered the problem of how to interact with two such heterogeneous systems. To cope with this problem we proposed a platform interface that allows the communication with both platforms. The hybrid control code of the JAviator made obvious that the parallel use of control code by Java threads and the Exotasks creates a hard-to-maintain source code. We were interested to support both concurrency models with Jarol for execution of control code. Hence we analyzed the characteristics of both concurrency models and derived the requirements for a general concurrency interface.

Based on this analysis, we concluded that we need to elaborate the data and control flow at the interfaces. We proceeded to their design and implementation in Java. We designed *ports* that provide non-blocking and lock-free message passing for communication. For the control flow we designed *signals* that allow the implementation of synchronization protocols for threads.

The successful interfacing of the JAviator and Seascout platform with the Jarol intrastructure shows its feasibility for control systems:

- The Jarol-based interface of the JAviator inherits the same functionality as the original, but with increased code structuring, and thus, maintainability.
- The organization of the Seascout with Jarol shows that the migration of software platforms that reside outside the Java realm is feasible.
- The interfacing of the Seascout with Exotasks and Jarol shows the feasibility of

the Jarol concurrency interface. Additionally, this approach had impact on the ongoing development of the Exotask system, in particular on the development of generic distributers.

Bibliography

[1] Brushless Motors - General Purpose DC Motors. http://www.mclennan.co.uk/product/generalpurposedcmotors.html.

[2] Description of BLDC 58. http://whale.fe.up.pt/ rmartins/BLDC58-50LMK2.pdf.

[3] Description of the Glomo SW100. http://www.glomo.dk/prodn4.htm.

[4] Devantech Homepage - Ultrasonic Sensors Page. http://www.robot-electronics.co.uk/shop/Ultrasonic_Rangers1999.htm.

[5] GumstixDocsWiki - Robostix. http://docwiki.gumstix.org/Robostix.

[6] GumstixDocsWiki- Basix and connex. http://docwiki.gumstix.org/Basix_and_connex.

[7] Homepage - Computational Systems Group. http://cs.uni-salzburg.at/ ck/group/.

[8] Homepage - Underwater Systems and Technology Laboratory. http://whale.fe.up.pt/ lsts/wiki/index.php/Main_Page.

[9] IBM Research Metronome Project. http://www.research.ibm.com/metronome/.

[10] Introduction to Pulse Width Modulation. http://www.netrino.com/Publications/Glossary/PWM.php.

[11] Java API 2 Platform Standard Ed. 5.0. http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html.

[12] LAUV Homepage. http://whale.fe.up.pt/seascout/.

[13] Microstain - 3DM-GX1 Product Page. http://www.microstrain.com/3dm-gx1.aspx.

[14] Wikipedia - Autonomous Underwater Vehicle. http://en.wikipedia.org/wiki/Autonomous_Underwater_Vehicle.

[15] Wikipedia - Fin. http://en.wikipedia.org/wiki/Fin.

[16] Wikipedia - Gyroscope. http://en.wikipedia.org/wiki/Gyroscope.

[17] Wikipedia - I2C. http://en.wikipedia.org/wiki/I2C.

[18] Wikipedia - Inertial Measurement Unit. http://en.wikipedia.org/wiki/Inertial_Measurement_Unit.

[19] Wikipedia - Quadrotor. http://en.wikipedia.org/wiki/Quadrotor.

[20] Wikipedia - Unmanned Aerial Vehicle. http://en.wikipedia.org/wiki/Unmanned_aerial_vehicle.

[21] *Operations with multiple autonomous underwater vehicles: the PISCIS project*, 2003.

[22] *Neptus - A Framework to support the Mission Life Cycle*, 2006.

[23] *Seaware: a publish/subscribe based middleware for networked vehicle systems*, 7th IFAC Conference on Manoeuvring and Control of Marine Craft, September 2006.

[24] *SWORDFISH: An Autonomous Surface Vehicle for Network Centric Operations*, 2007. to appear in Oceans'07 Europe, Abeerdeen, Scotland, June 2007.

[25] *Modeling and Simulation of the LAUV Autonomous Underwater Vehicle*, in submission 2007.

[26] Ken Arnold, James Gosling, and David Holmes. *THE Java Programming Language, Fourth Edition.* Addison Wesley Professional, 2005.

[27] J. Auerbach, D.F. Bacon, D.T. Iercan, C.M. Kirsch, V.T. Rajan, H. Röck, and R. Trummer. Java takes flight: Time-portable real-time programming with exotasks. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM Press, 2007.

[28] D.F. Bacon, P. Cheng, D. Grove, M. Hind, V.T. Rajan, E. Yahav, M. Hauswirth, C.M. Kirsch, D. Spoonhauer, and M.T. Vechev. High-level real-time programming in Java. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*. ACM Press, 2005.

[29] Lubomir F. Bic and Alan C. Shaw. *Operating Systems Principles*. Prentice Hall, 2003.

[30] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 3rd edition, 2001.

[31] Reza Ghabcheloo. *Coordinated Path Following Control of Autonomous Vehicles (Thesis Summary)*. PhD thesis, Lisbon, IST, November 2006.

[32] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification - Third Edition*. Java Series. ADDISON-WESLEY, 2005.

[33] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2211 of *LNCS*, pages 166–184. Springer, 2001.

[34] Allen I. Holub. *Taming Java Threads*. Apress, 2000.

[35] K.H. (Kane) Kim. A non-blocking buffer mechanism for real-time event message communication. In *Real-Time Systems - The International Journal of Time-Critical Computing Systems*, volume 32, pages 197–211, March 2006.

[36] Hermann Kopetz and Johannes Reisinger. NBW: A Non-Blocking Write Protocol for Task Communication in Real-Time Systems. Jan. 1993.

[37] Doug Lea. *Concurrent Programming in Java: Design Principles and Pattern Second Edition*. The Java Series. Prentice Hall, 2nd edition, 1999.

[38] William Pugh, Doug Lea, and Sarita Adve. JSR-133: JAVA Memory Model and Thread Specification.

[39] Rainer Trummer. JAviator Homepage. http://javiator.cs.uni-salzburg.at/.

[40] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill Companies, 2000.