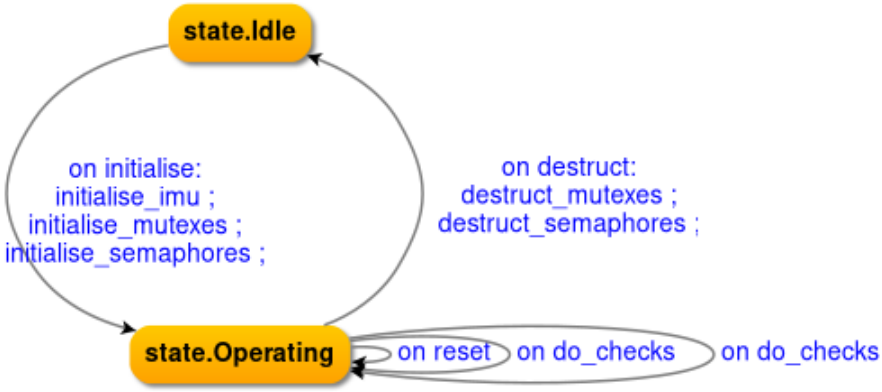
Design and Implementation of a Real-time Safety Module for Care Robots

Kaydo Alders

A thesis presented for the Bachelor degree in Computer Engineering



Computer Engineering
Inholland University of Applied Sciences
Alkmaar, Netherlands
June 6, 2020

Inholland University of Applied Science

Bachelor Thesis Computer Engineering

Design and Implementation of a Real-time Safety Module for Care Robots

Author:
Kaydo Alders

Supervisors:
Cock Heemskerk
Ton Boode
Elmer Hoeksema

June 6, 2020

Research group Robotics,
Bergerweg 200, 1817 MN Alkmaar
+31 (0)72-5195774

Plagiarism statement

Hereby I certify that this research is own written work, except for references to prior research by others, which are properly cited. Any use of figures not made by myself I have been granted to use by their authors. These are properly referenced.

Kaydo Alders
June 6, 2020

Source code availability and documentation

The source code for this research is available under the GPLv3 license. The source code and documentation are can be found at the following URL: https://github.com/Yousousen/safety-module-for-care-robot-rose

Contact information of the author

Name: Kaydo Alders
Student number: 580539
Email address: kaydo1@live.nl
Telephone: +31 6 34712780

Abstract

We developed a real-time safety module to provide an extra layer of safety for service robots. We conducted a feasibility study to determine the feasibility of combining ideas from the fields of mobile robotic safety, verification of system behavior, real-time systems, and to determine the feasibility of applying the combination of these ideas to an implementation of a safety module for service robots. We found a Raspberry Pi 3 with Sense HAT running Raspbian Buster with Xenomai 3 to be a feasible hardware and real-time operating system setup. Sensors on the Sense HAT are used to obtain values to use in the safety criteria. The Sense HATs LED matrix is used to signal misses in meeting the safety criteria. Real-time operation is achieved by running threads in Xenomai's primary mode and by avoiding mode switches with the use of Xenomai's XDDP sockets.

Contents

List of Figures

List of Tables

1. Introduction

Research group Robotics at Inholland Alkmaar wants to improve the safety of care robots, as in the current state of affairs the care robots cannot guarantee safety. When a component of a care robot fails, like the grip arm, motion sensor or navigational equipment it can result in major consequences like damage to the robot's environment or injury of patients. Without guarantees that these consequences will not take place the care robots cannot be used in practice. Research group Robotics therefore wants to develop a safety module that can be attached to a care robot and will operate as fail operational system to prevent major consequences like damage to the care robots environment or patient injury. The safety module will constantly check if the behavior of the care robot is safe given the situation. The safety module is a safety critical real-time system, as its actions are required to be completed in a given time. The safety module provides an extra layer of safety over existing safety measures on the care robot.

The goal of this research is to lay down the basis of this safety module and to design and implement a prototype of the safety module that will detect situations where the care robot performs unsafe actions, which the safety module will then signal to the environment. We will focus on an implementation for the semi-autonomous care robot *Rose*, a care robot that is able to assist elderly people with every day tasks. Figure 1.1 shows a picture of robot Rose helping an elderly woman and Figure 1.2 shows robot Rose pouring milk into a cup. Robot Rose is located at Innovatielab at Inholland Alkmaar.

The structure of the thesis is as follows. In Chapter 2 we give an in depth elaboration of the assignment. In Chapter 3 we analysis the problem and set up our research questions from this analysis. In Chapter 4 we elaborate the research methodology of the research, starting with the general methodology of the research in Section 4.1 followed by the methodologies used per subquestion in Section 4.2. Chapter 5 is a broad chapter that starts by explaining various terms used throughout the research in Section 5.1 and Section 5.2. We then present our conducted literature review on safety measures used in care robots, on hardware platforms of care robots, on operating systems used on care robots, and on the Dezyne modelling language, the modelling language used to model the safety module with, in Section 5.3, Section 5.4, Section 5.5 and Section 5.6 respectively. In Chapter 6, we show the results of the research, elaborated per subquestion in Section 6.1, Section 6.2, Section 6.3 and Section 6.4. In Chapter 7 we present the conclusion of the research. In Chapter 8 we present the recommendations of the research. Lastly, a summary of the research can be found at the very end of the thesis.

Figure 1.1.: Robot Rose assisting an elderly woman, source: HIT, 2020

Figure 1.2.: Robot Rose pouring yogurt into a cup, source: HIT, 2020

2. Assignment Specification

In this chapter, an in depth elaboration of the assignment is given.

The assignment is to design and implement a prototype of a safety module for care robots. The prototype of the safety module will focus on the semi-autonomous care robot Rose. Later models of the safety module will also have to work on different care robots, semi- and fully autonomous alike.
The safety module should detect situations where robot Rose behaves in an unsafe manner. For this assignment unsafe behavior is given the following definition:

Unsafe behavior is behavior that could result in injury to humans or damage to the environment by collision, clamping or pinching of the care robot with an object or human.

The head, body and platform can collide or clamp with an object. Pinching of an object can be caused by the grip arm's gripping force or torque.
It should be noted that Rose already has safety measures built into her system. For example, collision with a person and electric shock due to spilling of water are prevented by using navigation path planning and making the external housing of electrical parts water resistant (Heemskerk, 2020b). The latter is a mechanical safety measure. The former is a software safety measure, but it is not a real-time safety measure due to the fact that Rose uses ROS. In cases such as the former the goal of the safety module is to provide a second-layer real-time safety check that operates independently from Rose. The keyword real-time is important here, as this really is the added benefit over the regular safety measures. Throughout the research, we use the following definition of real-time:

A real-time system is a system that must react to stimuli from a controlled object within time intervals dictated by its environment. (Juvva, 1998).

Thus a real-time system is given limited time to react to given stimulus. In context of the safety module the stimulus would be, for example, the acceleration sensor on Rose detecting a large increase in velocity. In such a case the safety module would execute a real-time task to signal this large increase as unsafe behaviour to its environment. The time before which the safety module must have completed the task of signalling the environment is called the *deadline* of the task. (Juvva, 1998). In real-time systems a distinction is often made between hard, firm and soft real-time tasks. Buttazo, 2004 gives the following definitions for these: A real-time task is said to be hard if producing the results after its deadline may cause catastrophic consequences on the system under

control, it is said to be firm if producing the results after its deadline is useless for the system, but does not cause any damage and it is said to be soft if producing the results after its deadline has still some utility for the system, although causing a performance degradation. These are the definitions of hard, firm and soft real-time tasks that are used throughout the research.
From our earlier given definition of the unsafe behavior that the the safety module should detect, namely that it should detect unsafe behavior that could result in injury to humans, it seems quite unethical to give the real-time safety module any other designation then hard real-time, as injury to patients is severely catastrophic. Indeed, we think the final safety module should be hard real-time. In this assignment, however, we are designing and implementing a prototype, and we will accept soft real-time behavior to be good enough for two reasons. The first reason is that the safety module will communicate with Rose to receive sensor data. Rose runs Robot Operating System (ROS), and since Robot Operating System is not real-time (ahendrix, 2014), we cannot achieve hard real-time behavior. The problem with an inherently non-real-time system like Robot Operating System is that they cannot guarantee any response times. This is a bottleneck, and a large part of the research is designated to accommodating for inherently non-real-time aspects of the robot-and-safety-module system like this. The second reason is that we will develop a prototype and that the development time is too small to investigate and realise the requirements for hard real-time systems. Hard real-time systems require dedicated hardware which would first have to be determined, then software would have to be developed for that dedicated hardware.

2.1. Deliverables

The deliverables of this research are as follows:

- A thesis containing the results of the research.
- A prototype implementation of the safety module.
- Software models of the safety module's behavior and functionality.
- Installation documentation of hardware and software used for the safety module
- Documentation of the source code, either in the respective source code files or as a separate document

2.2. Involved parties

The following table represents the parties involved in the research and their function in the research. The involved parties of this research are given in Table 2.1. For each party, the university or company they work for, their name, their email address and their function in this research are listed.

University / Company	Name	Contact	Function in research
Inholland	Cock Heemskerk	cock.heemskerk@inholland.nl	Supervisor company
Inholland	Ton Boode	ton.boode@inholland.nl	Supervisor company
Inholland	Elmer Hoeksema	elmer.hoeksema@inholland.nl	Supervisor university
Inholland	Pieter van der Hoeven	pieter.vanderhoeven@inholland.nl	Financial insights
TU Twente	Jan Broenink	j.f.broenink@utwente.nl	Implementation support
Verum	Bert de Jonge	bert.de.jonge@verum.com	CEO Verum
Verum	Johri van Eerd	johri.van.eerd@verum.com	Modelling support
Heemskerk Innovative Technology	Dimitris Karageorgos	d.karageorgos@heemskerk-innovative.nl	Care robot support
DigiNova	Jeroen Wildenbeest	j.g.w.wildenbeest@heemskerk-innovative.nl	Acquaintance with mobile care robots

Table 2.1.: Involved parties and their function in this research

3. Problem Analysis and Research Questions

In this chapter, we provide an analysis of the problem and set up our research question based on this analysis.

3.1. Hardware setup

A Hardware setup that can handle the real-time constraints is required for the assignment. With real-time constraints we mean restrictions on the timing of tasks such that they meet their deadline (Hsiung, 2001). Although the prototype safety module will be designed and implemented for care robot Rose, eventually the safety module has to work with a variety of care robots. The hardware should therefore be extensible so that it can work with different care robots as well.
The following are all the requirements for the hardware platform as they are relevant to our research:

1. The hardware supports environments with real-time constraints.
2. The hardware is able to communicate with care robot systems that run Robot Operating System to receive sensor data from such care robots.
3. The hardware has or supports the addition of a WiFi module for communication with the operator that controls the care robot.
4. The hardware has or supports the addition of a 4G network module.
5. The hardware has or supports the addition of an accelerometer, a gyroscope and a camera.
6. The hardware can be attached to a care robot without impact on the operation of the care robot.
7. The costs for the safety module do not exceed €300,-

Item 1 specifies that the hardware should support an environment with real-time constraints. In Section 2 we elaborate why this is required for the safety module.
Item 2 specifies that the hardware is required to communicate with care robot systems that run Robot Operating System to receive sensor data from. Robot Rose runs ROS,

and many other care robots and mobile servant robots do too ((Li et al., 20180201), (Triebel et al., 2015), (Delgado et al., 2019), (Karageorgos, 2017) to name a few), so if the safety module is extended to work with other care robots in the future it is very useful to support interaction with ROS.
Item 3 specifies that the hardware should support a connection over WiFi to communicate with the operator of Rose. This is because Rose is a semi-autonomous care robot robot and receives navigation input from her operator over a WiFi network connection.
In Item 4 the requirement of a 4G network module is listed. This 4G module serves as backup module for when the WiFi connection is very weak or completely gone. Research into an appropriate 4G module is out of the scope of this assignment and will be done in a later version of the safety module. Nonetheless, the hardware platform should support the addition of such a 4G module.
Item 5 lists the requirement that the safety module should support the addition of sensors with which it can utilize independently from the care robot sense the environment. This should be at least an accelerometer to measure acceleration with, a gyroscope to measure angular velocity with, and a camera that allows the safety module to look around. These are required for two reasons. The first reason to have independent sensors on the safety module is so that the safety module does not depend on the correct operation of the sensors of the care robot, and the second reason is that there may be a lot of latency in receiving sensor data as it received from ROS. The camera need not be used in this assignment, but the safety module should support the addition of a camera for future versions. Item 6 refers to the fact that the safety module should be portable, that is, it should be easy to attach to a care robot. It should not have an impact on the performance of the care robot. This means it should be light (relative to the care robot) and should not have large components sticking out (also relative to the care robot).
Lastly, Item 7 specifies the maximum costs of the safety module.
Section 5.4 shows the literature review we did to see what others have used as hardware setup for their robots, this can be taken as an inspiration and to determine the hardware to use for the safety module.

3.2. Operating system of the safety module

As a hardware setup to handle the real-time constraints of the safety module is required, likewise a software setup that can handle the real-time constraints is required. This mainly concerns the operating system the safety module will run on. Research will be conducted into operating systems for the safety module that can handle these constraints.

3.3. Retrieving sensor data from Rose

Rose runs a Linux distribution with Robot Operating System (ROS) as middleware. To retrieve data from Rose her sensors we have to look into the available means of communication with this system. ROS is not real-time (ahendrix, 2014), therefore it is important

to research the impact this has on the safety module and whether communication over the available interfaces is fast enough for a system with real-time constraints.

3.4. Modelling the safety module

In a software model the safety module and its relation to the care robot should be modelled. A well made software model helps in determining the complexity of a problem, and breaking the complexity of the problem up in smaller parts. It is preferable to do as much verification on the state of the safety module and its actions as possible. Such verification encompasses going through all the states of the sytem and verifying no deadlocks and livelocks occur. Deadlock occurs in a system when all its constituent processes are blocked. Another way of saying this is that the system is deadlocked because there are no eligible actions that it can perform (Magee and Kramer, 2006). One example of a deadlock in a system is a system in which two threads each hold a resource that the others requires to continue execution. Livelock occurs in a system where a thread's further execution requires the action of another executing thread, but the other thread is constantly served more work to do without executing the required action, meaning the first thread cannot continue to execute (Oracle, 2019).

3.5. Principal research question and research subquestions

Having made our problem analysis we can setup our principal research question and research subquestions. To fulfill the goal of the assignment described in Chapter 1 and Chapter 2, the following question will be central to the research and will serve as the principal research question:

How can a real-time safety module be designed and implemented that provides an extra layer of safety in order to prevent injury to patients or damage to the environment because of collision, clamping or pinching caused by care robot Rose?

To answer the principal research question, four subquestions are set up. The first subquestion deals with the hardware and operating system that can be used for the implementation of the safety module. Because the safety module is a safety critical real-time system, we need hardware and an operating system that can handle the real-time constraints of the care robot. The goal of the first subquestion is to provide an answer to this matter:

1. *What hardware and real-time operating system can be used to implement the safety module?*

Having determined our hardware setup and operating system, we can look at ways to communicate with care robot Rose and the sensors available on our hardware. Concretely, we will look at how sensor and actuator data can be retrieved and how it can be used to detect unsafe situations. Therefore, the second subquestion is as follows:

2. *How can sensor data be retrieved and how can sensor be used to check for unsafe behavior?*

After subquestion two we can start designing the safety module with a software model. In the software model we want to do as much verification on the state of the safety module and its actions as possible. Moreover, we have to take into account that the safety module will be a safety critical real-time system, as described in Section 2. We therefore have to concern ourselves with the real-time constraints while modelling the safety module. The third subquestion, is, therefore, as follows:

3. *How can the operational behavior of the safety module be modelled in a formal specification language?*

The fourth and final subquestion will bring everything together into an implementation of the safety module. Here we will implement the safety module on the chosen hardware and operating system and according to the model of the previous subquestion. The last subquestion is, therefore, as follows:

4. *How can the safety module safety module be implemented, using the formal specifcation, on a real-time operating system?*

4. Research Methodology

In this chapter, we elaborate the research methodology of the research, starting with the general methodology of the research in Section 4.1 followed by the methodologies used per subquestion in Section 4.2

4.1. General methodology of the research

The general methodology of this research is a feasibility study in the sense that we want to explore existing research in robotic safety, real-time systems that deal with inherently non-real-time elements and verification of system behavior. The goal is to apply these existing ideas to our case with care robot Rose, namely to implement a prototype safety module to show the feasibility of applying the combination of these these ideas to the case of care robot Rose. Because of this broad focus we only touch the surface of the vast sea of knowledge that is contained in each of these subjects. To elaborate, in the context of real-time systems we will not aim for hard real-time performance of the safety module, despite care robot Rose being perceived as a hard real-time system, as this requires much more specialised attention to implement, which would deviate from our aim to explore a multitude of ideas. Instead, we aim for the safety module to be a soft real-time system that is able to show the feasibility of the system as a whole. In the context of robot safety, we will look at the safety criteria and safety measures in existing research, but we will only apply a rudimentary form of these safety measures criteria. Likewise, we explore the ideas of system behavior verification in so far this gives guarantees on the correct operation of the safety module.

The results of this research and the prototype safety module produced by this research can be used to build further upon the ideas explored. Specifically, this research can be used to continue the development of the safety module into a product that provides an extra layer of safety to different care robots and perhaps different mobile service robots as a whole category. Chapter 8 elaborates follow-up research options in more depth.

To provide answers to the subquestions stated in Chapter 3, a mixture of research methods are used. These are elaborated per subquestion in Section 4.2.

4.2. Methodologies per subquestion

The first subquestion stated in Chapter 3 is: 'What hardware and real-time operating system can be used to implement the safety module?'. For this subquestion we will

conduct a literature review to determine what kind of hardware previous researches use in their robotic applications. Concretely, we would like to to find hardware that is both able to handle the real-time constraints of the care robot and that can interact with ROS. A comparative study will be conducted to determine what is the best choice for the heart of the system, the processing unit, RAM and related peripherals. From here on we will determine what measuring devices can be used with this hardware, and we will conduct another literature review and comparative study to find an appreciate real-time operating system for this hardware of the safety module. The literature review on the hardware and software can be found in Section 5.4 and Section 5.5 respectively. The results of the comparative studies can be found in the Results chapter in Section 6.1.1 and Section 6.1.3 respectively.

The second subquestion, 'How can sensor data be retrieved and how can sensor be used to check for unsafe behavior?', builds upon the results of the first subquestion and uses the results of the comparative studies of the first subquestion to find out how, with the chosen hardware and software, we can interact with sensors to retrieve data and how we can use that data to detect unsafe behavior. The answer to this subquestion is provided in Section 6.2 of the Results chapter.

The third subquestion, 'How can the operational behavior of the safety module be modelled in a formal specification language?', aims to model the behavioral aspects of the safety module. The result of this subquestion will be a formal specification that can put some guarantees on the behavior of the safety module. An elaboration of the toolset that will be used to make this formal model is give in Section 5.6. The results of this subquestion can be found in Section 6.3.

The fourth and last subquestion, 'How can the safety module safety module be implemented, using the formal specifcation, on a real-time operating system?' will put everything together in an implementation of the formal specification on hardware and software. A simulation will be used to emulate the interaction with care robot Rose her sensors. The results of this subquestion can be found in Section 6.4.

5. Theoretical Background and Literature Review

In this chapter we provide the theoretical background and literature reviews that corroborate our research. We will first present definitions of terms that are used throughout this research. After that, we present our conducted literature reviews on the topics of service robot safety, hardware platforms and operating systems used in service robots. Then we provide an overview of the Dezyne toolset and modelling language, the toolset and modelling language used to model the behavioral aspects of the safety module.

5.1. General terms and definitions

Throughout the research we make use of a variety of terms and definitions. The definitions we use for *real-time systems* and *hard, firm and soft* real-time tasks were given in Section 2. Likewise, the definition we use for *unsafe behavior* was given in in Section 2 as well.

Previous research frequently uses the terms *mobile service robot*, *domestic robot* or *personal care-robot* for mobile robots used for medical (Najarian et al., 2011) and military purposes (Murphy et al., 2009), or for use in personal care (Graf et al., 2002). Tadele, 2014 defines a personal care-robot as a service robot with the purpose of either aiding or performing actions that contribute toward the improvement of the quality of life of an individual. Tadele, 2014 defines a domestic care robot as a personal-care robot with or without manipulators that operate in home environments and is often mobile. When we refer to *care robots* in our research, we refer to the definition of a domestic care robot as given by Tadele. We sometimes refer to service robot as well, if that is the specific term used by the research we refer to.

Multiple times in this research we refer to the concepts of *processes* and *threads*. A process is a sequence of execution in a computer, it is a program in execution. A process has its own address space and system resources (Karl-Bridge-Microsoft, 2018). A thread is also a sequence of execution, but a thread can be seen as a subset of a process. A process can create multiple threads and each thread shares their address space and system resources with other threads (Karl-Bridge-Microsoft, 2018). We sometimes use the term *task* as a synonym for *thread* in this research.
Two other concepts related to the execution of threads are the concepts of *starvation* and *race-conditions*. Starvation describes a situation where a thread is unable to gain

regular access to shared resources and is unable to make progress (Oracle, 2019). A race condition occurs when two threads access a shared variable at the same time (Microsoft, 2012). Race conditions can be prevented by use of mutexes or semaphores. A mutex is a lockable object that is designed to signal when critical sections of code need exclusive access, preventing other threads with the same protection from executing concurrently and access the same memory locations (CppReference, 2020). Such a critical section can be a shared variable as with a race condition. A semaphore object is a synchronization object that maintains a count between zero and a specified maximum value (Microsoft, 2018). Both mutexes and semaphores can be used to prevent race conditions.

Definitions and meanings for terms and acronyms like, *ROS*, *MCU* and are given in our Glossary and Acronyms lists which can be accessed by clicking on the respective term or acronym.
Other definitions of terms will be given as we meet them.

5.2. Robot Operating System

Care robot Rose uses Robot Operating System (ROS) as its middleware. The definition of Robot Operating System (ROS), as given by ROS O. S. R. Foundation, 2018 is as follows: *ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used func tionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.*
The advantages of ROS over other robotics platforms are its distributed computation model, the ability to reuse software and the rapid testing of software build with ROS. (O'Kane, 2018).

5.3. Safety measures used for care robots

There are three widely used measures for safety in care robots. These are: acceleration-based criteria, force-based criteria and energy/power-based criteria. Other criteria include human pain tolerance, maximum stress and energy density (Tadele, 2014). Of the acceleration-based criteria, the head injury criteria (HIC) is most widely used. It is a measure of the head acceleration for an impact that lasts for a certain duration (Gao and Wampler, 2009). Force-based criteria consider excessive force as the possible cause of collision injury (Ikuta et al., 2003). Lastly energy/power based criteria is a criteria that considers the rate of transfer of kinetic energy on body impact as the cause of possible injury to a human (Birk and Carpin, 2006).
The prototype safety module will use a very rudimentary form of force-based and energy-based safety criteria. It will calculate the kinetic energy of the body of the care robot and force used by the grip arm of the care robot and will use this as the basis to determine

what is unsafe behavior. The reason we only use this rudimentary form is because research into the topics of the various safety criteria is not the focus of this research. Instead, the focus of this research is to provide second-layer real-time safety checks on care robot Rose, meaning a safety checks that are independent of the state of Rose and her operating system. We describe the goal and specification of the assignment in more depth in Section 1 and Section 2 respectively.

5.4. Hardware in service robots

Previous research has used a variety of hardware for their real-time mobile service robot applications. Delgado et al. used a Raspberry Pi 3 as their control architecture for its open source environment. They mention its advantages over (also open-source) embedded hardware. Developing a real-time environment on embedded hardware is complex due to its limited availability of systematic documentation and technical support. Although manufacturers often provide Linux kernel sources, often the system lacks compatibility with other software, which is an open problem in embedded development in general. Delgado et al. further note that the Raspberry Pi 3 offers great compatibility with Robot Operating System (ROS), as ROS offers pre-built binaries for Ubuntu systems, and the Raspberry Pi 3 is able to run Ubuntu. ROS supports Raspbian, the native operating system of the Raspberry Pi, as well, but there are no pre-built binaries for this operating system, which increases development time.
Bouchier, 2013 researched into ways to integrate ROS into a robot. Their research applied to a broader field of robots than just care robots, but their research is relevant nonetheless. They mention three architecture styles, namely embedding ROS on an integrated personal computer, the use of a proprietary embedded system with custom interface and the extension of ROS messaging and APIs. The embedded PC architecture style allows for full integration of ROS on a Linux distribution with the ability to run others ROS nodes besides the robot system as well. Still, default Linux is not real-time, and requires patches and frameworks to achieve soft or firm real-time performance. For the second option, the use of a custom proprietary embedded system, Bouchier, 2013 describes a ROS device node can be setup to allow translation between ROS nodes and the propriety embedded interface. A recent development of the time had setup such an architecture on an Arduino (Bouchier, 2013). Ferdoush and Li, 2014 also use an Arduino board in their wireless sensor network control system for environmental sensing of temperature and humidity. The third architecture option describes communication with ROS nodes by use of Remote Procedure Calls (RPC). Rosserial is an approach that can be used to relay messages between ROS nodes to a embeddded system. Rosserial offers support for any platform that supports the C++ programming language.
The architecture styles that Bouchier describes are focused on the control system of a robot itself. In our case, the safety module is an addition to the already existing care robot system. The system of robot Rose most closely resembled the first architecture Bouchier describes, but without real-time patches or a real-time framework installed.
Another architecture not described by Bouchier but which can be found in, for example,

Triebel et al., 2015 is the use of a base CPU in the robot itself, which communicates with another PC to receive input to determine which actions to take. Triebel et al. used this in a service robot that guides and informs passengers in airports. Robot Rose has a similar architecture in that she receives navigational commands from the operator connected to her via a laptop on the WiFi network.
Instead of a PC one could also use a Field Programmable Gate Array (FPGA) as main processing unit in a service robot, as Asami et al. have done in their patrol service robot, in which the FPGA is used for image processing to calculate a travel route for the patrol robot.
Lastly, microcontrollers are also used in robotic control applications (Honegger et al., 2013) (Kaliński and Mazur, 2016). An advantage of using a microcontroller for the safety module would be that it does not add yet another PC with operating system to the system (Rose already has `rosepc1` and `rosepc2` which are PCs running Linux distributions) and a microcontroller is lower on power.

5.5. Operating systems on service robots

Service robots often run Linux distributions. As Delgado et al. describe, this is due to the fact that Linux operating system are the most popular open-source operating systems, and due to the development of real-time extensions for the Linux kernel to increase its response time and make it more suitable for real-time applications. Another reason Linux operating systems are popular is because ROS runs on Linux. In Section 5.4 we have seen that Triebel et al., 2015 and Delgado et al., 2019 use ROS as middleware on their service robots, and others do too, for example Karageorgos, 2017 in their Human aware autonomously navigating care robot, Hendrich et al., 2015 in their architecture design for care robots used for elderly care, Zhang et al., 2019 in their service robot for customer services in hotels, Fu and Zhang, 2015 in their social robot used in research and education, and many more. Since all these studies use ROS as their middleware, they use Linux distributions as their operating system. The most used Linux distribution for ROS systems is Ubuntu, as it is best supported Linux distribution (R. P. Foundation, 2020), but ROS can be compiled for Raspbian as well (Delgado et al., 2019).

On Linux distributions, there are two prominent options to add real-time support to the operating system. The first is Real-time Linux (Community, 2016), which applies a set of patches to the Linux kernel to make normally blocking sections of the Linux kernel preemptive. Blocking sections in the kernel refer to sections that are waiting for some event to happen before they continue executing. A preemptive kernel is one that can be interrupted while executing its code (Stallings, 2017). This is advantageous for real-time systems in that no running code will block the entire operating system waiting for some resource or event. The Real-time Linux approach is a single kernel approach, opposed to the second prominent option: the co-kernel approach of Xenomai 3 (Kiszka, 2020). In the co-kernel approach of Xenomai, a second kernel, by the name of *Cobalt*, runs alongside the standard Linux kernel. The cobalt kernel schedules real-time threads and

handles all time-critical activities. Real-time operation is achieved because the Cobalt kernel's activities have a higher priority than that of the standard Linux kernel (Kiszka, 2020).
Xenomai imitates various other real-time operating systems by providing interfaces of them using APIs, called Xenomai *skin*s. Among the Xenomai skins there is the POSIX skin, which provides POSIX functionality wrapped in real-time Xenomai functionality, and the native skin, which is Xenomai's own interface to real-time functionality. All Xenomai skins share a common core (Gerum, 2019), making the skins as the name suggests, just an interface to the real-time functionality Xenomai provides.

Brown and Martin, 2018 compare the Real-time Linux and Xenomai approaches and provided a number of important results. They found that for hard real-time performance Xenomai performed better than the real-time patched Linux kernel. They further made the distinction between 95% hard real-time performance and 100% hard real-time performance, where 95% hard real-time means the real-time requirements should be met 95% of the time, and where 100% hard real-time means the real-time requirements should be met 100% of the time. [1] They also made the distinction between code running in userspace and code running in kernelspace. userspace code is code that runs outside of the kernel and that the kernel is not aware of. On the contrary, kernelspace code is code that is run and maintained by the kernel (Stallings, 2017). Their tests show that for 95% hard real-time requirements Xenomai userspace performed better than real-time patched Linux kernel userspace. For 100% real-time requirements Xenomai kernelspace performed the best. So if one wants to have the best real-time performance one should go for Xenomai kernelspace. The downside of Xenomai kernelspace is that it is very maintenance intensive due to the use of custom built kernels, compared to Xenomai userspace (Brown and Martin, 2018).

5.6. Modelling and Dezyne

In this section we present a short overview of the Dezyne toolset that is used in this research to model behavior of the safety module. We think it is important to show a simple example, one that is less complex than the safety module which is shown in Section 6.3 of the Results chapter, to get across the effectiveness of the toolset and its relevance to our research. The Dezyne toolset is developed by Verum (Verum, 2019a), a partner company of this research. Dezyne allows us to verify the behavior of the safety module, by tracing every possible execution sequence and stopping at a problem. Dezyne's verification checks the system on the occurrences of race conditions, deadlocks, livelocks, mismatches between specification and implementation and asserts that the

[1] It is interesting that the paper by Brown and Martin makes the distinction between 95% and 100% hard real-time, while the notions of soft and hard real-time themselves already exist to avoid ambiguity between 'X%' and 'Y%' real-time. We think they make the distinction between 95% and 100% in order to avoid having to deal with the even more hard real-time notion they defined: life-safety hard real-time, which they explicitly stated to not focus on in the paper.

behvior of a program is deterministic. Dezyne has the ability to generate source code from the model specification, which makes sure that the source code is free from these problems (Verum, 2019b). To provide this verification Dezyne internally makes use of the process algebraic language *mCRL2*, a formalism for behavioral specification and verification of concurrent and distributed systems (Groote and Mousavi, 2014). mCRL2 extends the algebra of communicating processes with various features including notions of data, time, and multi-actions .

5.6.1. Process algebras and Dezyne

Before we go over some of the Dezyne language's basics, let us first compare Dezyne to process algebraic languages. Let us use as examples the languages mCRL2, which was mentioned in this section's introduction, and *FSP*, an open-source language and model checking toolset (Magee and Kramer, 2006). Both mCRL2 and FSP are formal specification languages that can be used to model system behavior with. System behavior is the series of actions a system can perform in a particular state. Both mCRL2 and FSP allow you to verify this modelled behavior. Dezyne does not differ from the functionality of these toolsets in the sense that it allows you to model and verify behavior as well. However, where it does differ is in language and syntax. Process algebras like mCRL2 and FSP have very specific language and syntax that requires one to be versed in math. Dezyne differs from these languages in that the Dezyne modelling language's syntax resembles the syntax of the C programming language families. This allows developers who are not experts at modelling system behavior to create deterministic deadlock- and livelock free programs. We will look at the syntax of the Dezyne modelling language in Section 5.6.2.
Another important matter where Dezyne differs from the process algebras mCRL2 and FSP is that Dezyne allows one to generate source code from the verified model. This source code can then be integrated into the system. In contrast to having a well verified model and hoping that one do not introduce any bugs in the process of converting this model into executable code, Dezyne's generated source code checks whether one have implemented all aspects of the model correctly. Furthermore, unlike process algebras like mCRL2 and FSP, Dezyne provides suggestions as to where to how to solve verification errors. This feature helps programmers to find and solve errors in a more convenient manner. We will look at this feature in Section 5.6.4.

5.6.2. The Dezyne modelling language

We will now look at the basics of the Dezyne modelling language and verification mechanisms. For this we will construct a short model that turns a LED on, off or lets it blink. Inspiration for this model has been taken from the Dezyne documentation (Verum, 2019b). The model is shown in Listing 1.
First of all, notice the notions of an *interface* and a *component* from Lines 1 to 21 and 23 to 38 respectively. Dezyne breaks logical control problems into components and interfaces. An interface defines the following: how a component interacts, the events that can

```
interface ILED {
    enum State { Off, On };
    in void turnOn();
    in void turnOff();
    in State getState();

    behaviour {
        State state = State.Off;

        [state.Off] {
            on turnOn: { state = State.On; }
            on turnOff: { state = State.Off; }
            on getState: { reply(state); }
        }
        [state.On] {
            on turnOn: { state = State.On; }
            on turnOff: { state = State.Off; }
            on getState: { reply(state); }
        }
    }
}

component LED {
    provides ILED iLed;

    behaviour {
        ILED.State state = ILED.State.Off;

        [state.Off] {
            on iLed.turnOn(): state = ILED.State.On;
            on iLed.getState(): reply(state);
        }
        [state.On] {
            on iLed.turnOff(): state = ILED.State.Off;
            on iLed.getState(): reply(state);
        }
    }
}
```

Listing 1: Dezyne model that turns a LED on and off

be communicated and the interaction protocol (Verum, 2019c). In other words, it defines the behavior of a component, the events that it can execute and in what states it can execute those events. In Line 2 we can see an enum declaration. This enum defines the states the system can be in. Below that, in Lines 3 to 5 we see the event declarations of the system. From Line 7 to 20 we see the *behaviour* block of the interface. Here we define in what states certain events are allowed, or what events should bring the system into a certain state. In this example we have chosen to declare behavior state-first, meaning we specify the state of the system first and then we specify which actions can occur in that state, but the language allows to declare behavior event-first as well, in which where we would specify the event first and then specify the states. In Line 8 we declare that the system starts off in the `State.Off` state. Then in Line 10 to Line 14, we specify that when the system is in the `State.Off` state, we allow the `turnOn` and `getState` events to fire, but the `turnOff` is `illegal`, which as the keyword suggests makes the event illegal to fire. This makes sense, as when we are already in the `State.Off` state we cannot turn the system off. The `turnOn` event sets the state to `State.On`, and the `getState` event, using the `reply` keyword, returns the current state of the system when the event fires. in Lines 15 to 19 we specify what should happen when the system is in the `State.On` state. When the system is in the `State.On` state, the `turnOn` state firing should be illegal, as we cannot turn the system on when it is already on. The `turnOff` state turns off the system by setting the state to `State.Off`, and the `getState` event returns the current state of the event, just like when the system is in the `State.Off` state.

We will now look at the component declared in Lines 23 to 38. A component is a unit of definition and instantiation (Verum, 2019d). A component defines zero or more *ports*, where a port is a named instance of an interface (Verum, 2019d). `provides` is the keyword used to define a port. In our example the defined port is `iLed`, an instance of the interface `ILED`. The idea of the behavior of a component is that it conforms to the behavior of its specification, or interface. We therefore make it start in the same state as the interface does in Line 27. In Line 29 to Line 36 we specify the behavior of the component. Notice how, in this example, the behavior is the same, except that the events that are illegal are not specified. These need not be specified because the component infers what is illegal behavior from its provided interface. This is because verification of a component always goes through its interface, as we will see in Section 5.6.3. In this example the component's behavior block is almost exactly the same as the interface's behavior block. This is because the example is simple, more complex models will show more differences between the behavior blocks of the component of the interface. Nonetheless, the behavior of the component should always conform to the behavior of the interface.

Listing 5.1 shows the state chart of our example LED model.

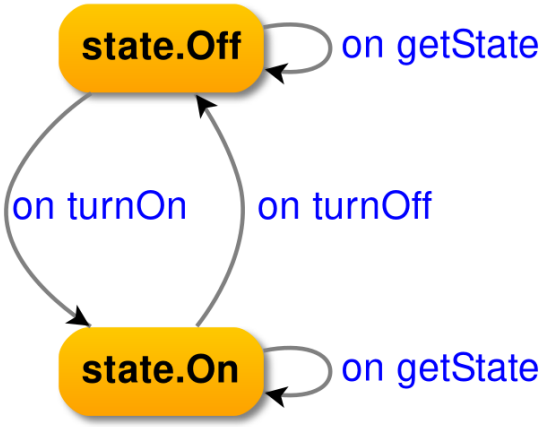


Figure 5.1.: State chart for the LED model

In Listing 5.1 we can see the `State.Off` and `State.On` states and the `turnOn` `turnOff` and `getState` events we declared in code of the model in Figure 1.

5.6.3. Verifying a Dezyne model

We will now look at the verification features of the Dezyne toolset. We will look at these features in context of the model in Figure 1. Let us first verify the interface. When we verify the interface as it is given in Figure 1, we get the message back that there are no verification errors in the interface. But things are not always so straightforward. Suppose that we forget to specify the `getState` event on Line 13. If we then verify the model we will get the error message shown in Figure 5.2.

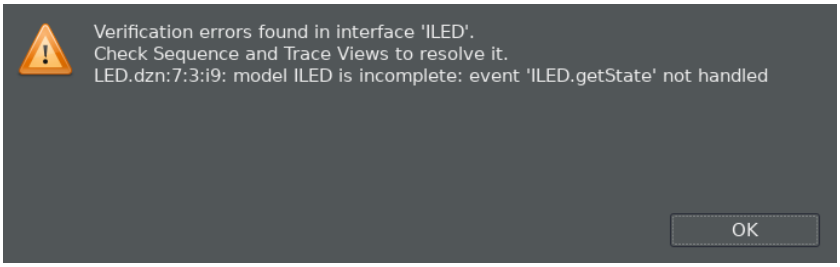


Figure 5.2.: Interface verification error: Not specifying all events

The reason we get this verification error is that a well defined interface should specify the behavior of any of its declared events, and that behavior should be specified for every state of the system. When we do not declare the `getState` event on Line 13, we have not specified what behavior should take place if the `getState` event fires while the system is in the `State.Off` state.

Let us return to the correct code as given in Figure 1 and let us now verify the component. The code of the component as it has been given in Figure 1 shows no first

Check	Action	Time	States	Transitions	Done	Result
ILED						
Deadlock		0:00	7	11	100%	✓
Livelock		0:00	7	11	100%	✓
LED						
Deterministic		0:00	7	11	100%	✓
Illegal		0:00	7	11	100%	✓
Deadlock		0:00	7	11	100%	✓
Livelock		0:00	7	11	100%	✓
Compliance		0:00	7	11	100%	✓

Figure 5.3.: Table showing no verification errors in the LED component

verification errors. In case a component gets through verification without errors we see a table like the one shown in Figure 5.3.
In Section 5.6.2 we noted how verification of a component is always in context of its interface. We can see this fact in Figure 5.3, as interface `ILED` is first checked for deadlocks and livelocks before we check the `LED` component. The `ILED` interface shows no errors so the verifier goes on to the LED component. We can see that the component is checked for determinism, illegal statements, deadlocks, livelocks and compliance. Determinism in computer science refers to deterministic behavior of an algorithm. In Dezyne determinism means that for every state there is a unique implementation for every event. There can not be multiple definitions of a particular state in a event. The illegal statements refer to whether the component tries to execute an event that is deemed illegal by its provided interface. The definitions of deadlock and livelocks we have given in Section 5.1. Compliance refers to whether the component conforms to the behavior as specified in the interface.
Let us now look at an instance of verification of a component where we get errors in the verification. Suppose we add the line: '`on iLed.turnOff(): state = ILED.State.Off;`' above Line 30 in the code of Figure 1 to try and turn the LED off while it is in the off state. Since the interface the component provides does not specify that the `iLed.turnOff()` event can fire in the `State.Off` state, we get an error showing that the `iLed.turnOff()` event is not allowed by port iLed, as shown in Figure 5.4. Additionally, Dezyne shows a sequence diagram of a trace that leads to the error, as shown in Figure 5.7. In this case, the trace is very short as the first event that executes, namely `turnOff`, is the cause of the problem.

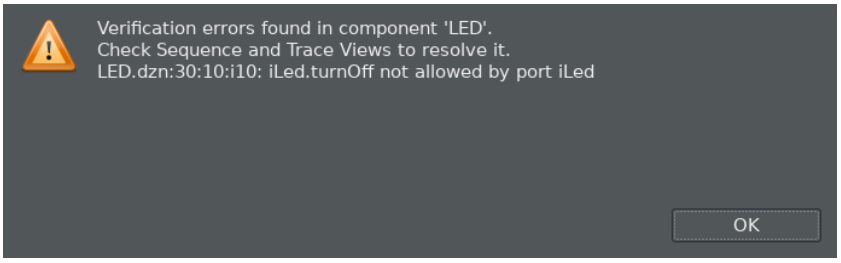


Figure 5.4.: Component verification error: Not confirming to the interface

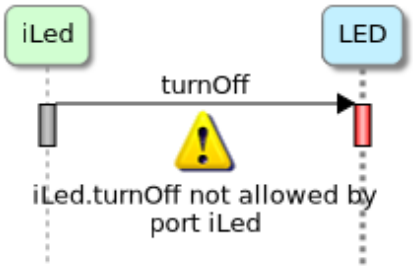


Figure 5.5.: Sequence diagram showing that the component does not conform to the interface

From this we can see that the interface prevents us from executing actions (events) that are not allowed by it. Let us remove the `turnOff` action, so that we end up with the code in Figure 1 again. Suppose we removed Line 35: `on iLed.getState(): reply(state)`, because we do not want to be able to get the state when the LED is in the `State.On` state anymore. If we then verify the model, we get an error saying that getState is an implicitly illegal action, as we can see in Figure 5.6. The accompanying sequence diagram is shown in Figure 5.7.

Figure 5.6.: Component verification error: getState implicitly illegal

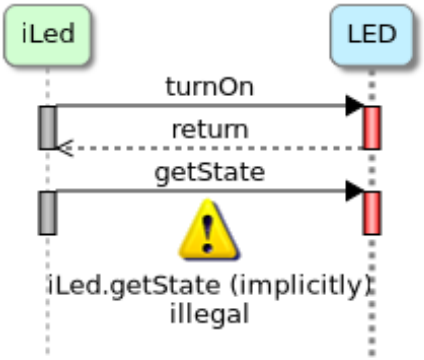


Figure 5.7.: Sequence diagram showing that getState is an illegal action in the component

The reason we get the error shown in Figure 5.6 and Figure 5.7 after removing Line 35 is because not specifying an event means that the event is illegal in Dezyne. But this is in contrast to what we specified in the `ILED` interface. We did not specify that `getState` is illegal in the interface, therefore the error says that `getState` is *implicitly* illegal. The lesson learned from this example is that a component must always conform to what is specified in its provided interface.

Lastly, let us look at a sequence diagram for the correctly verified model of Figure 1. The sequence diagram is shown in Figure 5.8. This is also called a simulation in Dezyne, because we simulate the system's control logic, taking a particular walk through the available actions. In this research, when we refer to a component's actions, we mean the same thing as the events we have declared for it. At the bottom of the Figure 5.8 we can see the particular actions we can execute, that is, events that can fire in this system's state, and on the right we see a watch window that shows values of the variables of the interface and the component. Since this example consists of just a `LED` component and its interface, the sequence diagram does not give us much information that we could not already deduce from the state diagram in Listing 5.1. However, for more complex systems like our safety module given in Section 6.3, the sequence diagrams will prove to be a very useful visualisation of the system.

In Section 5.6.4 we will look at the C++ code we can generate from our LED example model.

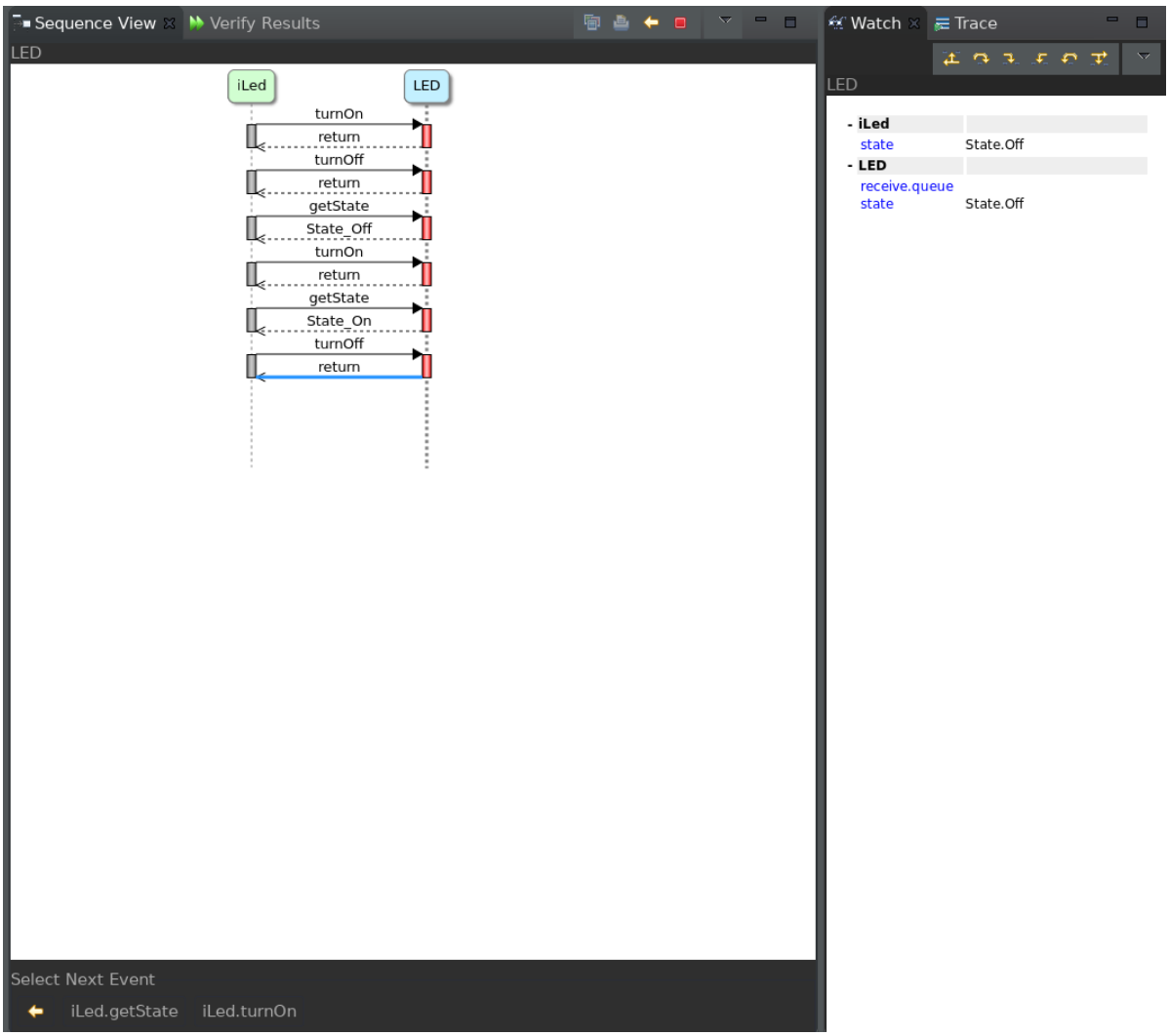


Figure 5.8.: Sequence diagram of the LED component

5.6.4. Generating code from a verified Dezyne model

In this section we will look at the code generation feature of Dezyne. We will make use of the model from Figure 1 for the generation of the code. To make the demonstration simpler, instead of letting the `turnOff` and `turnOn` events be illegal on Lines 12 and 16, we simply switch the state to `State.Off` and `State.On` respectively. we also added those actions to the component. Furthermore, to demonstrate the system component, which will be useful in code generation, we added a `IController` interface, a `Controller` component and a system component above the `ILED` interface. These additions are shown in Listing 2. The system component generates a System view, which is shown in Figure 5.9.

```
component System {
        provides IController iController;
        system {
                Controller controller;
                LED led;
                iController <=> controller.iController;
                controller.iLed <=> led.iLed;
        }
}

interface IController {
        in void trigger();
        in void stop();
        behaviour {
                on trigger: {}
                on stop: {}
        }
}

component Controller {
        provides IController iController;
        requires ILED iLed;
        behaviour {
                on iController.trigger(): iLed.turnOn();
                on iController.stop(): iLed.turnOff();
        }
}
```

Listing 2: Addition of interface IController, component Controller and system component

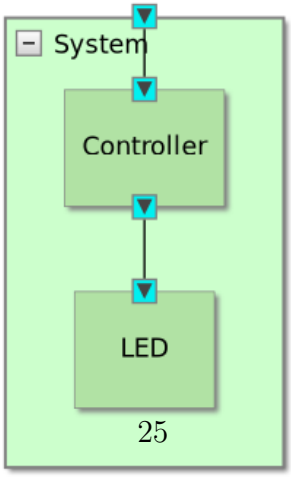


Figure 5.9.: System view of the controller and LED system

The Controller's job is to simply turn the LED on and off with its `trigger` and `stop` functions. We can see that it does that by calling the `iLed.turnOn` and `iLed.turnOff` events. In Line 22 of Listing 2 we specified that the controller component will call events from `ILED` with the `requires` keyword. In the component called `System` we see some more new syntax. The system section, as indicated by the keyword `system`, consists of two parts: instantiations of subcomponents, followed by all the bindings among the sub-components and ports of the component. This results in the system view shown in Figure 5.9. This system view shows all the components of the system. We will see how the system view is useful to us in the generated code shortly.

We will now generate source code from the model. Dezyne provides the option to generate code in a variety of languages. For this example we choose to generate C++ code, as C++ is the language used for the safety module as well. In Listing 3 we show a very minimal main function to use the generated code and Figure 5.10 shows the execution of the compilation of this code. The generated code itself can be found on our github repository.

```
pi@raspberrypi:~/led_example $ ./led_example
On
<external>.trigger -> LedExampleSystem.controller.iController.trigger []
LedExampleSystem.controller.iLed.turnOn -> LedExampleSystem.led.iLed.turnOn [0]
LedExampleSystem.led.iLed.return -> LedExampleSystem.controller.iLed.return [1]
LedExampleSystem.controller.iController.return -> <external>.return []
Off
<external>.stop -> LedExampleSystem.controller.iController.stop []
LedExampleSystem.controller.iLed.turnOff -> LedExampleSystem.led.iLed.turnOff [1]
LedExampleSystem.led.iLed.return -> LedExampleSystem.controller.iLed.return [0]
LedExampleSystem.controller.iController.return -> <external>.return []
```

Figure 5.10.: Execution of minimal LED example code

Lines 7 to 11 have to do with the allocation and use of the Dezyne runtime. Usage of the Dezyne runtime is elaborated extensively in the documentation of the toolset (Verum, 2019c). On Line 10 we declare a system object. The system object is useful because we can use it to check whether every function from dezyne is bound to a function in our C++ code and we can access our dezyne functions, `trigger` and `stop` through it, as is done on Line 18 and Line 20 respectively. In our simple example we have not bound any Dezyne functions to C++ code. In a real system we would bind the LED's `turnOn` and `turnOff` events to C++ functions that are able to turn a real LED on and off. Examples of the binding of Dezyne events to functions can be found in the Dezyne documentation or in our Results on the implementation of the safety module in Section 6.4.2.
The executed code in Figure 5.10 shows how we simulate turning on and off a LED by giving the program the input `On` and `Off`. We can see that the controller calls `iLed.turnOn` and `iLed.turnOff` internally as we call `iController.trigger` and `iController.stop`.

There are many useful features of Dezyne we have not discussed. For example, the Dezyne runtime gives an exception when one tries to call an illegal that is declared illegal. Moreover, components implemented in source code can inherit skeleton classes to

```
#include <iostream>
#include <dzn/runtime.hh>
#include <dzn/locator.hh>
#include "LED_example.hh"

int main() {
    dzn::locator loc;
    dzn::runtime rt;
    loc.set(rt);
    System s(loc);
    s.dzn_meta.name = "LedExampleSystem";
    s.check_bindings();

    std::string input;
    while(1) {
        std::cin >> input;
        if (input == "On")
            s.iController.in.trigger();
        else if (input == "Off")
            s.iController.in.stop();
    }
    return 0;
}
```

Listing 3: Example source code

make sure every function that needs an implementation is actually implemented. Many examples are given in (Verum, 2019b) and many of the features we use in our model of the safety module. The strength of the toolset really comes from the ability to carry over the verification of the model to the implementation code, making sure that what we verified stays correct in the implementation.

6. Results

In this chapter we present the results of our research, elaborated per subquestion in Section 6.1, Section 6.2, Section 6.3 and Section 6.4 respectively.
All code, documentation and models can be found on our GitHub repository. Some imporant snippets of code are given in Appendix B.

6.1. Choice of hardware and operating system

In this section we elaborate on the hardware and software choices that were made for the assignment.

6.1.1. Main component

We will now look at the choices made for the hardware of the safety module. We will first look at the main component. The main component of the safety module specifies the heart of the system: the CPU, RAM, and related peripherals. In Chapter 5 we explored previous research to see what others have used as their main component. In Section 5.4 we saw that Delgado et al. used a Raspberry Pi 3 in their control architecture on their mobile service robot. In Section 5.4 we also noted that previous research describes the main component of the robot itself, while in our research the main component of the robot already exists; namely the robot, Rose, has two Ubuntu systems, `rosepc1` and `rosepc2` which together form the main component of the system, consisting of the CPU, RAM and related peripherals. `rosepc1` and `rosepc2` are connected to the grip arm Rose uses to grab items with, the Kinect camera Rose uses to see her environment with and the WiFi router which connects Rose to her operator. `rosepc1` and `rosepc2` correspond with what Bouchier described as the first architecture in 5.4, a complete Linux distribution, but without real-time patches or a real-time framework. The safety module we implement in this research is an addition to the already existing system of Rose.

We will now look at available hardware solutions to determine what is the best choice for the main component of the hardware based on the requirements listed in Section 2. The Raspberry Pi is a small computer with various software solutions for real-time computing. Raspbian, the default operating system of the Raspberry Pi, is an operating system based on Debian, a Linux based operating system. The Linux kernel can be configured for real-time computing through the `CONFIG_PREEMPT_RT` parameter (Community, 2016) and by use of a scheduler with priority queues (Budak, 2020). Another

option is to configure the Linux kernel for real-time computing is to use the Xenomai project framework, which is an open source software project aiming to build a real-time framework for Linux (Kiszka, 2020).
The Raspberry Pi 3 offers support for the addition of peripherals through its 40 pins GPIO header. It has a built in WiFi module and Ethernet interface. Various resources on the internet provide guides for embedding a Raspberry Pi in a project, and the Raspberry Pi Foundation (R. P. Foundation, 2020) offers clear documentation on the available features of the Raspberry Pi. The Linux kernel and the Xenomai project offer thorough documentation as well. Our literature study in Section 5.4 showed that multiple studies used a Raspberry Pi in their mobile robotic applications. These options will be compared in Section 6.1.3.

The Arduino Uno is a microcontroller board based on the Atmel 8-bit ATmega328P microcontroller ("Arduino Uno Rev 3", 2020). The board provides a USB interface to program the microcontroller without the need of a microcontroller programmer and contains various other utilities to ease interaction with the microcontroller on the board. Through the use of ARTe (Arduino Real-time extension) the Arduino Uno can be extended to support multitasking and real-time preemptive scheduling ("ARTe Arduino Real-Time extension", 2018). At the time of writing, the real-time extension for the Arduino framework lacked up-to-date and clear documentation however. Just like the Raspberry Pi, the newest version of the Arduino Uno has built in WiFi module. In Section 5.4 we saw that Ferdoush and Li, 2014 used an Arduino in their robotic control system.

In Section 5.4 we also found that microcontrollers are used in robotic applications. Microcontrollers like the Microchip PIC series can be used for real-time systems as they are highly configurable which allows them to fit well to the needs of a project. (Microchip, 2020) The programmer can program the task scheduler of the microcontroller himself to fit the needs of the system. Microchip offers an extensive feature list in their datasheets of the various microcontrollers they offer. A disadvantage for usage of a microcontroller in this project is that a microcontroller has no built in WiFi, 4G, and ethernet module which means these have to be added in addition to the measuring devices required for the project. Furthermore, because microcontrollers generally do not run an operating system, considerable time has to be put into developing a real-time scheduler for the microcontroller.

To make a decision for the main component of the hardware we set up a table to compare the components and see how well they fit in fulfilling the requirements listed 3.1. As a first selection we chose the Raspberry Pi 3B+, the Arduino Uno SMD R3 and the PIC16F15386 microcontroller. Earlier this section we described why the Raspberry Pi and Arduino Uno are good choices. The PIC16F15386 we selected because Microchip recommends it for real-time applications (Microchip, 2020). In Table 6.1, all three components have been given points based on how well they fit the requirements and their advantages and disadvantages. They get a score of 1 if they fit the requirement not very

well, 2 if they fit well enough, and 3 if they fit the requirement very well. The component with the largest total points will be chosen as the main component for the hardware. We left out Requirement 2, the requirement that the hardware should be able to communicate with ROS, as all components support this. We also left out Requirement 5, for all components are lightweight and small. Lastly we estimated the total cost after addition of peripherals and gave a score for their price. Table 6.1 shows the results.

Main component comparison			
	Raspberry Pi 3B+	Arduino Uno SMD R3	PIC16F15386 MCU
Real-time support	3	1	3
Ethernet support	3	3	1
WiFi and 4G support	3	2	1
Peripheral support	3	3	3
Estimated total costs	2	2	3
Total score	14	11	11

Table 6.1.: Table to compare potential hardware choices

We gave the Arduino Uno SMD R3 only one point for *real-time support* because at the time of writing we found the documentation for the real-time framework to be unclear and outdated. We gave the PIC16F15386 microcontroller 1 point for *WiFi and 4G support* because it requires a lot of development time relative to the Arduino Uno and Raspberry Pi. The PIC16F15386 has three points for *estimated total costs* because it is cheaper than the other two. None of the choices exceeded the maximum of €300,-, with all three choices averaging below €100,-.
The last row shows the total score of each component. We can see that the score Raspberry Pi 3B+ scored the highest, we therefore think the Raspberry Pi 3B+ is the best choice for the main component of the safety module.

6.1.2. Measuring devices

Having chosen on the main component of the safety module system in Section 6.1.1, we looked into measuring devices for the Raspberry Pi to fulfill Requirement 5 listed in Section 3.1, namely the addition of an accelerometer for measuring acceleration and the addition of a gyroscope for measuring angular velocity.
There are various options to add additional measuring devices to the Raspberry Pi. One option is to buy the devices separately. Another option is to buy an add-on board commonly called Raspberry Pi HATs with multiple sensors on it. We went for the second option, as it greatly decreases development time and is extensible for future releases of the safety module. We chose the Raspberry Pi Sense HAT, as it was the most readily available Raspberry Pi HAT that contained both an accelerometer and a gyroscope. In

addition to this, the Raspberry Pi Sense HAT has a number of advantages over using separate measuring devices:

- The Sense HAT has a magnetometer for measuring the magnetic field at a particular location
- The Sense HAT can fuse accelerometer, gyroscope and magnetometer data to reduce uncertainty in measurements
- The Sense HAT comes with a library that can be used to receive data from the sensors on the Sense HAT.
- The Sense HAT has a LED matrix which can be used to show the results of safety checks

Furthermore, the Sense HAT has a humidity sensor, a pressure sensor and an onboard joystick. These are not needed for the prototype of the safety module, but could be useful for further releases of the safety module. For example, the joystick can be used for adding minimal control support for when the joystick of Rose her operator does not work. The humidity sensor and pressure sensor could be used to get more detailed information on the environment Rose is located in.
Figure 6.1 shows the Sense HAT. The components of interest for us in this research are the the inertial measurement unit (IMU), labeled `ACCEL/GYRO/MAG`, which contain the accelerometer, the gyroscope and the magnetometer, and the LED matrix. The name of the IMU itself is *LSM9DS1*. Figure 6.2 shows the orientation of the IMU. The orientation is such that if we lay down the Raspberry Pi with Sense HAT attached in front of us, in such a way that the Ethernet port points towards us, then that is the direction labeled X. As we can see, the Z direction denotes the up -down direction, and the only other direction left is the Y direction. The fact that we live in a universe with three spatial dimensions complicates the way we want to measure acceleration a bit. The measuring of the quantities of acceleration and angular displacement will be elaborated in Section 6.2.

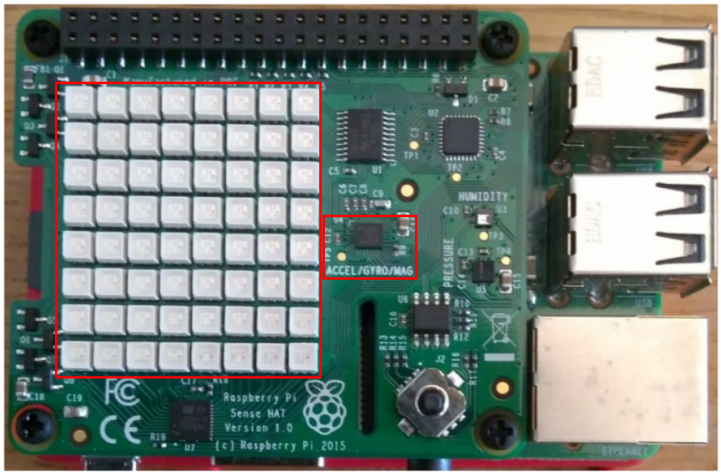


Figure 6.1.: Raspberry Pi with Sense HAT

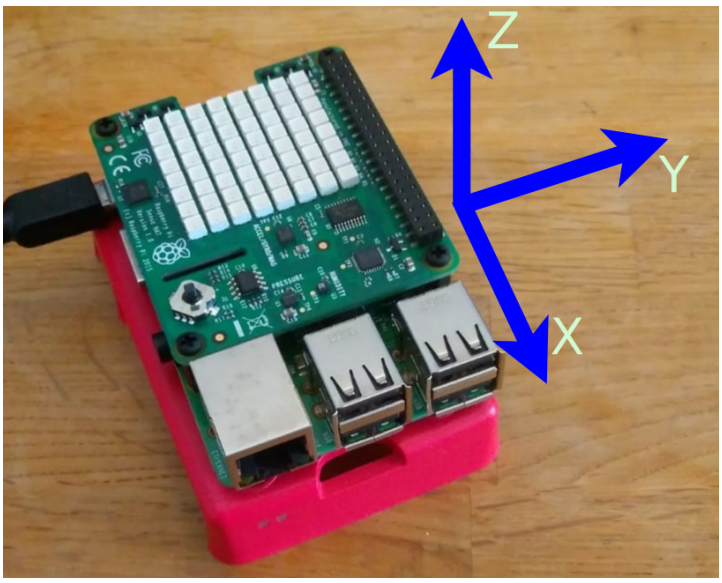


Figure 6.2.: Axis of the IMU on the Sense HAT

6.1.3. Operating System

In our literature review in Section 5.5 we saw that Linux distributions are often used in care robot systems, because ROS is a popular middleware and ROS runs best on Linux distributions. We choose to run a Linux distribution on the Raspberry Pi of the safety module, because care robot Rose runs ROS as well, and the safety module needs to interact with care robot Rose. Specifically, we choose to run Raspbian, the Debian based Linux distribution for the Raspberry Pi, as it is optimized for the Raspberry Pi's hardware (Raspbian, n.d.).
As was shortly mentioned in the description of the Raspberry Pi in Section 6.1.1 and as was elaborated in detail in the literature review of Section 5.5, the two prominent options for adding real-time support to Linux distributions are real-time Linux and the Xenomai framework. In Section 5.5 we also saw a comparison between these two options, with as result that Xenomai performed better in terms of meeting hard real-time requirements. Now, for the prototype safety module that is developed in this research, we only aim for soft real-time performance. Still, further versions of the safety module will need to support hard-real-time performance, and we therefore choose to use the co-kernel setup of the Xenomai framework over Real-time Linux as our operating system.
As was shortly mentioned in Section 5.5, Xenomai provides various skins. These skins are interfaces to the real-time functionality that the Xenomai framework provides. We choose to work with the POSIX skin. The POSIX skin provides the functionality of the Xenomai framework through the interface of a subset of POSIX system calls. The following services are provided (Kiszka, 2018):

- Clocks and timers
- Condition variables

- Message queues
- Mutual exclusion
- Process scheduling
- Semaphores
- Thread management
- Scheduling management

These POSIX services are wrapped around with Xenomai versions of the calls. For example, if one calls the `pthread_create` function to create a new thread in the C programming language or the C++ programming language, what is actually called is a real-time implementation of the system call provided by the Xenomai framework.
The reason we choose to work with the POSIX skin as opposed to other skins is that the POSIX skin avoids the need to rewrite device drivers written for standard, non-Xenomai Linux, as such device drivers often use the POSIX services given in the previous list. In our case, this avoids the need to rewrite the driver for the Sense HAT. Likewise, ROS is targeted for standard Linux and not Xenomai, so using the POSIX skin also avoids having to rewrite ROS services. Section 6.4 shows how we implemented the safety module using the xenomai framework.

We have built ROS for Raspbian and installed it on our system. For this research we will simulate interaction with care robot Rose and ROS and we will not use this installation, however. The reason we installed ROS nonetheless is to show the feasibility of using the system of the Raspberry Pi using Raspbian with Xenomai, as a whole.
Figure 6.3 gives a high level overview of this system as a whole, visualised as a stack. On the bottom we have the Raspberry Pi with the Sense HAT, this is our hardware. On top of that there are the Linux kernel and the Xenomai co-kernel, which run next to each other. Then there is Raspbian Buster, the operating system. On top we see RTIMULib, Dezyne and ROS. RTIMULib is the library used to communicate with the Sense HAT. Dezyne is the modelling language we use, which uses its own runtime library. The use of Dezyne is elaborated extensively in its own section, Section 6.3. The reason RTIMULib and ROS are shown on top, despite the fact that both communicate with the hardware of the bottom layer, is because RTIMULib and ROS are both libraries called from user space. We can say that anytime we want to access a hardware device in our proram we traverse the stack from top to bottom.

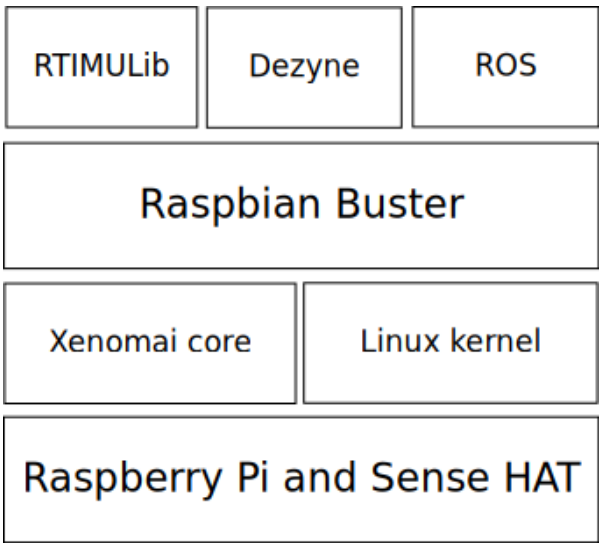


Figure 6.3.: Safety module system stack

Let us look at a communication scheme of the system and how the Xenomai framework provides us with real-time operation of the safety module. In Figure 6.4 an overview of the communication scheme of the system is given. On the left side we can see our program and the Xenomai core. These run in *primary mode*, primary mode is the real-time domain; the part of the system that operates real-time, because it is governed by the Xenomai co-kernel. This is due to that all scheduling of tasks in primary mode is handled by the Xenomai core. The Xenomai framework provides us with the ability to make a subset of system calls, the ones from the items listed in List 6.1.3 without leaving primary mode. We depicted Dezyne to be inside our program, in primary mode. The code that Dezyne generates from the model is essentially the control loop of the safety module system. This control loops executes five checks, this is why there are five lines of communication going through the entire system in the overview of Figure 6.4. These checks are checks on five aspects of the robot: the kinetic energy, the rotational energy, the grip arm strength, both force and torque, and grip arm position. These checks are elaborated extensively in the subsequent Results sections. RTIMULib and ROS communicate with device drivers written for the standard Linux kernel, meaning they do not run in primary mode, but *secondary mode*. Secondary mode is the non-real-time domain; the part of the system that does not operate in real-time and is governed by the standard Linux kernel. This brings us to a problem. Our program needs to communicate with these libraries, and therefore with these device drivers, but if it does so it would make a mode switch by moving from primary mode to secondary mode. One solution would be to reimplement all used device drivers to work with the Xenomai framework, so that they can run in primary mode. This solution, however, takes a lot of development time and research into how to convert standard linux drivers to real-time drivers. Luckily, Xenomai provides us with another solution: the use of XDDP sockets. XDDP sockets allow the part of the system running in primary mode to communicate data with the part of the system running in secondary mode, without the former leaving

primary mode. This is good because now we can set up the system as follows. In primary mode the control loop runs in real-time governed by the Xenomai co-kernel. In secondary mode, threads retrieving data from the sensors are running and provide their data on the XDDP socket, which can then be read by the threads of the program running in primary mode. This set up does away with the problem of our control loop needing to leave primary mode to access sensor data. However, now, there is another problem hiding in the background. What should we do when the sensor data is not provided by the non-real-time thread when a real-time thread requires it? We go into more detail on this problem in Section 6.4, but essentially the real-time checks just keep going regardless of whether there is data provided by the non-real-threads. That is to say, the checks do not block if there is no data available.

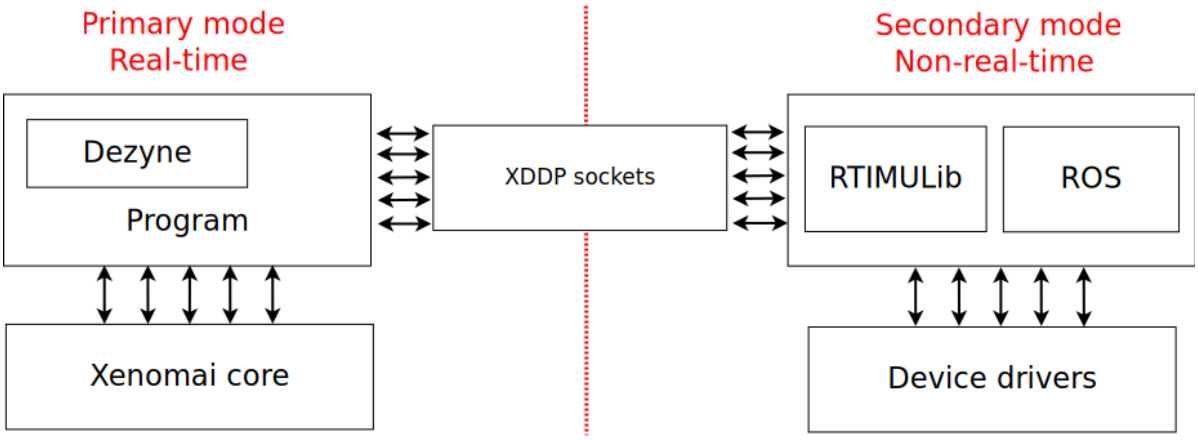


Figure 6.4.: Distinction between primary mode and secondary mode

6.2. Retrieving and Using Sensor Data

In this section we elaborate our results on how to retrieve sensor data and use this data to check whether the safety module has performed an unsafe actions.
In Section 2 we defined unsafe behaviour as behavior that could result in injury to humans or damage to the environment by collision, clamping or pinching of the care robot with an object or human. The goal of the safety module is to provide a second layer of safety checks upon the already existing safety checks by the care robot itself. In Section 5.3 we explored how existing research measures safety in service robots. We implement a rudimentary form of some of these safety checks by looking at five aspects of the care robot. These are: its kinetic energy, its rotational energy, its grip arm force and torque and its grip arm's position. These aspects were given on the Risk Analysis for additional safety measures of robot Rose (Heemskerk, 2020a).

6.2.1. Sensors on the safety module

The sensors of the Sense HAT of the safety module are used to check the kinetic energy and rotational energy of the robot. The safety module is attached to the care robot, and should be attached in a way such that the x-axis (see Figure 6.2) points forward; along with the front of Rose.
We make use of the RTIMULib library, an open-source library that provides us an application programming interface (API) to interact with the accelerometer, gyroscope and magnetometer of the LSM91DS1, the inertial measurement unit of the Sense HAT (RPi-Distro, 2015). With the use of a Kalman-filter the library provides fused data of the accelerometer, gyroscope and magnetometer. Kalman-filtering is a technique to provide an accurate measure of, in our case, the angular orientation of the safety module.

6.2.2. Kinetic energy

First we will look at the aspect of kinetic energy. In Newtonian mechanics, kinetic energy is defined as follows:

$$E_k = \frac{1}{2}mv^2 \tag{6.1}$$

where m is the mass of the object undergoing linear translation and v the velocity of the object undergoing linear translation. Robot Rose's mass is $95kg$ (HIT, 2020), the Raspberry Pi's mass is $45g$ (Piltch, 2019) and the Sense HAT's mass is $20g$ (own measurement). That gives us $m = m_{tot} = 95.065kg$. Velocity we do not measure directly, instead we measure the acceleration using the accelerometer on the Sense HAT. To obtain velocity $\vec{v}$ from acceleration $\vec{a}$, we numerically integrate $\vec{a}$ using the following equation:

$$\vec{v} = \int_{t_0}^{t_f} \vec{a}\ dt \approx \sum_{t=t_0}^{t_f-1} \frac{\vec{a}(t) + \vec{a}(t+1)}{2} \Delta t \tag{6.2}$$

where t_0 is the initial time, t_f is the final time and Δt is a small change in time. Equation 6.2 is called the *Trapezoidal rule*. For smaller Δt the approximation becomes more accurate. We have set Δt such that the time taken to sample the acceleration is a multiple of the recommended poll interval of the IMU as given by RTIMULib.
Various sources report the occurrence of drift error due to sensor noise by using integration to obtain velocity from acceleration (CHRobotics, n.d.) (Sachs, n.d.) (Woodman et al., 2007). However, we can still use this method of determining velocity for our purposes, for the obtained velocity is accurate enough to determine kinetic energy and we discard the value of velocity after determining the kinetic energy gained from it. Every time we calculate the kinetic energy, we do so by obtaining the kinetic energy that is gained from the change in acceleration. Our error does not get increasingly worse, because we only measure changes in acceleration and do not take into account the velocity the robot may already have.
A subtle, but important point that follows from this, is that we do not actually measure the kinetic energy of the robot, but the change in kinetic energy. Furthermore, with discarding the previous value of velocity we mean we set the initial velocity to be zero every time we want to calculate the kinetic energy. The result is that the following equation holds:

$$\Delta E_k = \frac{1}{2}m(\Delta v)^2 \implies \Delta E_k = \frac{1}{2}mv^2, \quad if\ \vec{v_i} = 0 \tag{6.3}$$

where Δv is the change in velocity and $\vec{v_i}$ is the initial velocity. Δv is not depicted as a vector here because squaring Δv eliminates the sign of the velocity. In fact, as we only use the velocity to calculate kinetic energy, we actually only need to obtain the speed. Still, we obtain the velocity for scalability and in case someone wishes to do calculations with velocity in the future.
Excessive change in kinetic energy will make the robot injure a human or damage an object upon collision, which we defined as a form of unsafe behavior. To avoid this, the obtained change in kinetic energy is compared to the maximum allowed change in kinetic energy. For our purposes, this maximum allowed change in kinetic energy is the same for every situation. The maximum allowed change in kinetic energy is given as input to the safety module either in an input file or as command line argument. The value is not determined by us, but by the company or organisation who employs the safety module, who has tested and determined an appropriate value for this variable. We have chosen a value that most clearly demonstrates the use of the check on kinetic energy.

6.2.3. Rotational energy

The rotational energy is determined analogous to the kinetic energy, but using angular variables instead, and we calculate the actual rotational energy, not the change in

rotational energy. The rotational energy is calculated using the following equation:

$$E_{rot} = \frac{1}{2} I\omega^2 \tag{6.4}$$

where I is the moment of inertia and ω is the angular velocity. For I we approximate the robot as if it were a solid cylindrical object. I can then be calculated as follows:

$$I = \frac{1}{2} mR^2 \tag{6.5}$$

where m is the mass of the robot and R the radius of the robot. R we approximate to be 1 meter, taking the scenario into account where Rose fully stretches her arm.
The fused accelerometer, gyroscope and magnetometer data gives us an accurate measure of the angular displacement $\vec{\theta}$ of the safety module. With $\vec{\theta}$ we can calculate the angular velocity $\vec{\omega}$ using numerical differentiation as follows:

$$\vec{\omega} = \frac{d\vec{\theta}}{dt} \approx \frac{\Delta\vec{\theta}}{\Delta t} = \frac{\vec{\theta}(t + \Delta t) - \vec{\theta}(t)}{\Delta t} \tag{6.6}$$

Here Δt is a small change in time, which we set to be a multiple of the poll rate of the IMU, just like with Δt used in the calculation of the linear velocity for the kinetic energy.
Excessive rotational energy will cause the linear momentum of the robot to be too great when the robot makes a turn, which can result in damage to the environment or injury to humans. Therefore, just like with kinetic energy, the rotational energy is compared to a maximum allowed value which is given in an input file or command line argument.

6.2.4. Sensors on robot Rose

Ultimately, Rose's arm sensors will be used to retrieve the strength and position of the arm. These values are received from ROS. ROS provides a publisher-subscriber type of pattern in which a subscriber can listen on a *topic* on which the publisher publishes data. In our case the subscriber would be the safety module and the publisher would be Rose. Topics in ROS are collections of messages concerning a particular subject, where a message is the means of communication of data in ROS (O'Kane, 2018).
Currently we use generated values from within the safety module itself and do not communicate with robot Rose through ROS. We have still installed ROS on the safety module to show the feasiblity of the system as a whole. Documentation on the installation of ROS on the safety module can be found on our github repository.

6.2.5. Force and torque

Strength can be expressed in two different kind of forces. One is the force of the arm against an object, the other is a rotational force, commonly called *torque*, that is used to grab an object. We need not calculate the force and torque like we did with kinetic

energy and rotation energy, because ultimately care robot Rose provides these values through ROS.
Excessive force can cause the arm to clamp with an object, and excessive torque can cause the arm to pinch an object. Both are cases we want to avoid. We, therefore, just like with kinetic energy and rotational energy, set a maximum allowed value for the force and torque, which should not be exceeded.

6.2.6. Position

The last check on unsafe behavior concerns the arm position of the robot. When the robot is moving, the arm should be in a folded position. This means it is not stretched, and in a position that it can do the least amount of harm. This value, too, is ultimately provided by Rose through ROS, and need not be calculated by the safety module.

6.2.7. When the result of a check gives unsafe behavior

What should we do when a check is executed and the maximum allowed value for the kinetic energy, rotational energy, force, torque is exceeded, or the arm is not folded when the robot is moving? These are the situations we call unsafe behavior. In such a case, the LED matrix receives a signal to turn red. On default, the LED matrix is low. When a check is executed and the result of the check is that there is no unsafe behavior, the LED matrix receives a signal to turn blue. Only when the result is that there is unsafe behavior does the LED matrix receive a signal to turn red. Checks are executed in sequence. An important point to note here is that situations exist, where, for example, the result of the first of the five checks gives unsafe behavior, while the subsequent checks give safe behavior. In such a case, the LED matrix should turn red after the first check, and stay red after the subsequent checks. This is because unsafe behavior of one safety check is not supposed to be overridden with safe behavior of another safety check. We therefore specify that once the result of a check gives unsafe behavior, the LED stays red until an acknowledge action is executed. This acknowledge action, among the other actions of the system, is elaborated in Section 6.3.

6.3. Modelling of the safety module

In this section we elaborate how we modelled the system. In Section 6.3 we present a formal model of the behavior of the safety module made with the modelling toolset *Dezyne*. Some auxiliary models can be found in Appendix A, and all models, in general, can be found on our GitHub repository. We avoid to show the syntax of the Dezyne language itself wherever possible, to make these results readable for a greater public. One can take a look at the GitHub repository to see the syntax of the modelling language. The Dezyne files can be recognized by their `.dzn` extension.

The basics of the Dezyne toolset and modelling language were elaborated extensively in Section 5.6.2. There we also explained what Dezyne's behavior verification does. In short, it checks the system on the occurrence of race conditions, deadlocks, livelocks, mismatches between specification and implementation and concepts related to deterministic behavior of the system. A prototype of the language has also the capability to check the system on functional behavior. That is to say, it checks whether a system, after executing a set of actions, is in the expected system state. If it is not, this will be output as a functional verification error. In this section, we will present our formal behavior model of the safety module. We will first present an overview of the system, which shows us all components of the system, then we will work from inside out, going through the individual components of the system.

6.3.1. System overview

Figure 6.5 shows the system overview. An alternative representation of this same view is given in Appendix A.

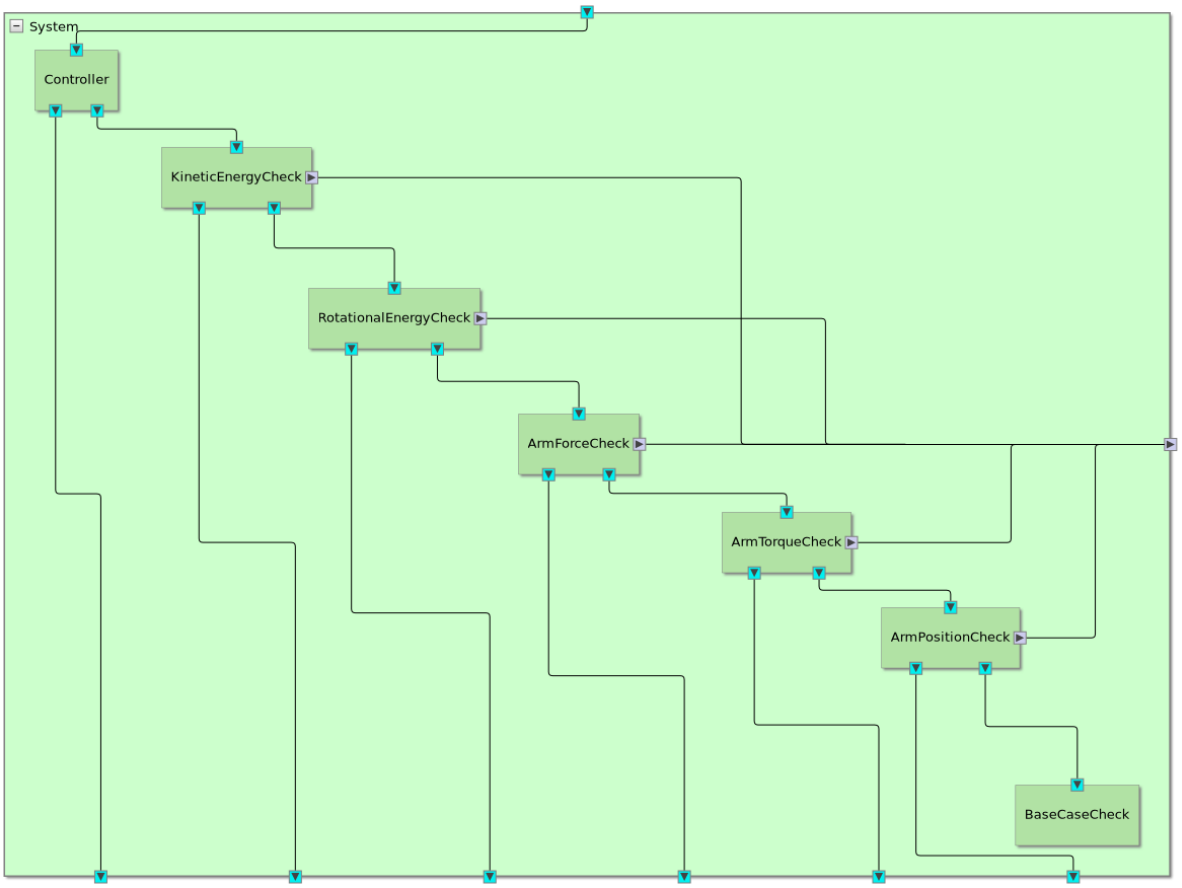


Figure 6.5.: System view of the safety module system

In Figure 6.5 we can see (if we squint our eyes a little) the hierarchy of all components that are implemented in Dezyne. On top there is the `Controller`. The `Controller` governs the entire system, and executes safety checks in sequence. These checks are checks on kinetic energy, rotational energy, force of the grip arm, torque of the grip arm and position of the grip arm, like we established in Section 6.2. Each check is implemented like an element in a linked list. A linked list is a data structure in which each element of the list points to the next element. So, in this case, each check points to and executes the next check. This is to provide the system a level of abstraction and generality. When one wishes to add a new check to the system, one does not need to make changes to the `Controller`, and the only thing that needs to be changed is the pointer that points to the next check at the location in the list one wishes to add the new check. No changes need to be made to any other aspect of the system logic outside of the added check itself.
Every check returns a value, `Behavior.Safe` or `Behavior.Unsafe`, which indicates whether the executed action of the care robot results in safe or unsafe behavior.
The model can be simulated, which shows a trace of a system along a certain path of actions which results in a sequence diagram, as we will see in subsequent subsections. In such a simulation we can choose whether a particular check returns safe or unsafe

behavior ourselves. In the real non-simulated system, however, the system will return safe or unsafe behavior based on whether the retrieved value of the quantity the check examines exceeds the maximum allowed value, or, in the case of the arm position, whether the arm is or is not in folded position.
In Figure 6.5 we can see blue squares going into and out of components. These are called *ports*. In Section 5.6.2 we mentioned how source code can be generated from Dezyne models. Ports on the the edges of the system need to be bound in implementation code to generated source code. With implementation code we mean the C++ code of the safety module program that will run on the Raspberry Pi. The single port at the top of the system view denotes the `Controller` port that we may intract wit to bind to a function in implementation code in order to execute the dezyne system, or, in other words, the safety checks. The six ports at the bottom are ports that also need to be bound in implementation code, but these are executed by Dezyne internally, that is, we assign a function for Dezyne to execute by binding these ports. The binding of ports is elaborated in Section 6.4.2.
In Section 5.6.2 we mentioned how a Dezyne model consists of interfaces and components, where interfaces specify the behavior a component should conform with and which events or actions a component can execute.

The system consists of the following interfaces:

- `ILEDControl`
- `ISafetyCheck`
- `ISensors`
- `IResolver`
- `IController`

We will first elaborate `ILEDControl`, `ISafetyCheck`, `ISensors` and `IResolver`. Then we will elaborate all components as they are found in Figure 6.5. `IController` we will elaborate alongside its component, `Controller`.

6.3.2. ILEDControl

`ILEDControl` is an interface that specifies the behavior of the safety module in context of controlling the LED matrix. It has no component in Dezyne. An implementation of this interface, in other words, a component, is mostly concerned with communicating with the LED matrix framebuffer. Communicating with hardware through device drivers is not a task for Dezyne, as it does not involve serious control logic, and only involves the retrieval of data through a device driver. So the component of `ILEDControl` is actually completely implemented in implementation code. As we mentioned in the previous section, there is still control logic to verify for Dezyne in interfaces. In Figure 6.6 the

event table of ILEDControl is shown. In this table the blue rows are the events that the ILEDControl interface can execute. The *States* column denotes the state we are in; and behavior is different depending on which state the interface is in. The second column shows guard conditions. These are conditionals that further influence what kind of behavior is allowed in a certain state. The *Code* column shows the actual code that is executed, and the *Next* column is the state the interface will be in after executing the code. Let us look at the first event, initialise_framebuffer. This event, as the name suggests, initialises the framebuffer that is used to communicate with the LED matrix. When the system is in the Idle state, this event will result in a transition to the Operating state. If the system is already in the Operating state when the initialise_framebuffer action is executed, then this is an illegal action, as denoted by the illegal keyword in the Code column. This makes sense, as we do not want to initialise an already initialised framebuffer, and once the ILEDControl is operating, the framebuffer should already be initialised. One of the key features of Dezyne is that, once we generate source code from this model, we are not allowed to execute the initialise_framebuffer action in implementation code either, because in the model we specified that this should not be possible. It is these kind of checks from Dezyne that make sure the transition from model to implementation happens smoothly and without error.
In contrast to the initialise_framebuffer event, the destruct_framebuffer event is only allowed to be executed if the interface is in the Operating state. If the system is in the Idle state, there is nothing to destruct, for in the Idle state we have not initialised the framebuffer.

ILEDControl has an important task that is expressed in its guard conditions. It has the task to prevent the system from overriding a red LED with a blue LED. We mentioned this briefly in Section 6.2.7. Once a certain executed safety check gives as result that there is unsafe behavior, the LED will turn red. If a subsequent check then gives as result that subsequent behavior is safe behavior, it cannot turn the LED blue, because we do not know if the unsafe behavior has yet been recognized by the system or someone controlling the system. The fact that a blue LED cannot override a red LED is specified in the rows below the light_led_blue event. The system can only turn the LED blue if the system state is Operating and if the LED state is Low or Blue. If the LED state is Red, setting the LED blue is an illegal action, and can thus not happen.
How can one, then, ever turn the LED blue once it has turned red? This is where the reset_led event comes into play. The reset_led event turns the LED state to Low, so that the LED has the option to turn blue again.
Building this prevention to override a red LED with a blue LED at the abstraction level of ILEDControl makes sure that any component that uses the ILEDControl interface cannot somehow define behavior that would override a red LED with a blue LED. The use of this we will see when we elaborate the Controller component.

States	Guard	Code	Next
initialise_framebuffer			
[systemState.Idle]		systemState = State.Operating;	systemState.Operating
[systemState.Operating]		illegal;	
destruct_framebuffer			
[systemState.Idle]		illegal;	
[systemState.Operating]		systemState = State.Idle;	systemState.Idle
light_led_red			
[systemState.Idle]	[ledState.Low]	illegal;	
[systemState.Idle]	[ledState.Blue]	illegal;	
[systemState.Idle]	[ledState.Red]	illegal;	
[systemState.Operating]	[ledState.Low]	ledState = LedState.Red;	systemState.Operating
[systemState.Operating]	[ledState.Blue]	ledState = LedState.Red;	systemState.Operating
[systemState.Operating]	[ledState.Red]	ledState = LedState.Red;	systemState.Operating
light_led_blue			
[systemState.Idle]	[ledState.Low]	illegal;	
[systemState.Idle]	[ledState.Blue]	illegal;	
[systemState.Idle]	[ledState.Red]	illegal;	
[systemState.Operating]	[ledState.Low]	ledState = LedState.Blue;	systemState.Operating
[systemState.Operating]	[ledState.Blue]	ledState = LedState.Blue;	systemState.Operating
[systemState.Operating]	[ledState.Red]	illegal;	
reset_led			
[systemState.Idle]		illegal;	
[systemState.Operating]		ledState = LedState.Low;	systemState.Operating

Figure 6.6.: The event table of `ILEDControl`

We made an alternative view that shows the events per system state, as opposed to system states per event, as is the case in Figure 6.6. This alternative view is given in Appendix A. We have made an events per state and states per events view for all interfaces and components. We show the ones here that we think convey the purpose of the interface or component the best. The alternative view can, in general, be found in Appendix A unless the views show exactly the same, which is the case when there is just one system state and one event.

We made a sequence diagram of a particular trace through the events of `ILEDControl`. This sequence diagram is shown in Figure 6.7. One should note that the Dezyne behavior verifier goes through every possible sequence of events, as opposed to just one particular sequence, as is the case in the sequence diagram. The sequence diagram is useful nonetheless as a simulation of the model and to visualise certain aspects of the system. Below the sequence diagram at the bottom of Figure 6.7 we can see the next actions that can be executed at the particular state of the system. We show these possible next actions here because in the case of the `ILEDControl` interface it highlights how stating an event as `illegal` prevents one from executing the event in a trace of the system. For subsequent sequence diagram figures we will not show the possible actions explicitly, and we will only show the sequence diagram itself. As we can see in Figure 6.7, the only available next actions are `destruct_framebuffer`, `light_led_red` and `reset_led`. `light_led_blue` cannot be executed because we declared it to be illegal to execute `light_led_blue` if the LED is already red. It is only after a `reset_led` action that we can turn the LED blue again.

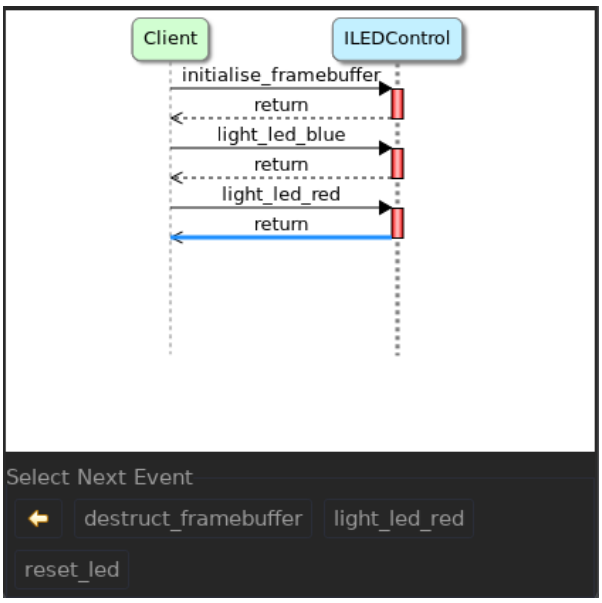


Figure 6.7.: Sequence diagram showing a particular trace through `ILEDControl` and the next possible actions

6.3.3. ISafetyCheck

The next interface we will look at is `ISafetyCheck`. `ISafetyCheck` is a simple interface that specifies the behavior of the five check components, namely `KineticEnergyCheck`, `RotationalEnergyCheck`, `ArmForceCheck`, `ArmTorqueCheck` and `ArmPositionCheck`. These five safety check components implement the `ISafetyCheck` interface. We will look at these components later in this section. The single event of the `ISafetyCheck` interface is shown in Figure 6.8. In the blue row we see the `<state>.<Initial>` state. This denotes, as the name suggests, the initial state of the interface. This state is shown if an interface or component does not define any states of its own. In the case of `ISafetyCheck`, the interface simple and need not keep track of any states of its own. The only event that the `ISafetyCheck` interface can execute is the `do_check` event. This event denotes the execution of a safety check. Since this is an interface, `do_check` is just a generic notion of a safety check. When we look at the components that implement `ISafetyCheck` we will see that the component associates a specific action with `do_check`. In the Code column we can see that `do_check` replies either the `Behavior.Safe` or `Behavior.Unsafe` behavior type. The behavior types `Behavior.Safe` and `Behavior.Unsafe` are just like boolean types, but with more indicative names.

Events	Guard	Code	Next
[<state>.<Initial>]			
do_check		reply(Behavior.Safe);	<state>.<Initial>
do_check		reply(Behavior.Unsafe);	<state>.<Initial>

Figure 6.8.: State table of `ISafetyCheck`

Figure 6.9 shows the sequence diagram we created for the `ISafetyCheck` interface. Here we make a trace through the interface, and, because this is a simulation of the model, we are free to return either `Behavior_Safe` or `Behavior_Unsafe` after the execution of `do_check`.

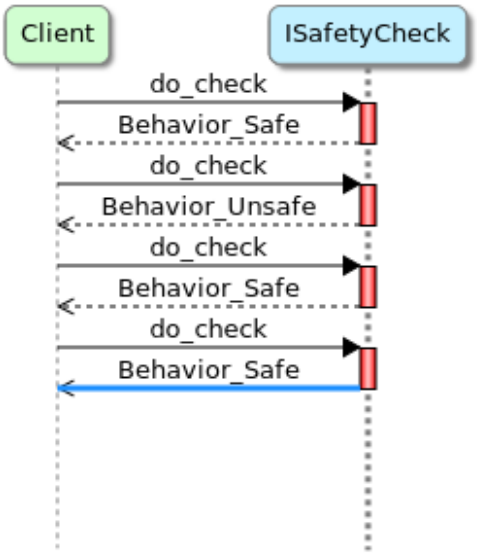


Figure 6.9.: Sequence diagram of ISafetyCheck

6.3.4. ISensors

`ISensors` defines five interfaces that correspond to a sensor or measuring device used in a safety check. The check on kinetic energy makes use of the accelerometer of the Sense HAT, the check on rotational energy makes use of the fused accelerometer, gyroscope and magnetometer data of the Sense HAT which are then used to obtain angular displacement, the arm force, torque and position are to be retrieved from robot Rose. Each `Sensor` defines only one event; this event retrieves the quantity that is to be checked. The sensors do not define any system states. In Figure 6.10 the `retrieve` events of respectively `KineticEnergyCheck`, `RotationalEnergyCheck`, `ArmForceCheck`, `ArmTorqueCheck` and `ArmPositionCheck` are given. In Figure 6.11 their sequence diagrams are given in the same order. One may notice that the code columns define no code for these interfaces. That is because the retrieval of data is a purely implementation dependent task, and there is no control logic to check for Dezyne. In Section 6.4.2 we shall see how the retrieve events are bound to functions and how they are implemented.

Events	Guard	Code	Next
[<state>.<Initial>]			
retrieve_ke_from_acc		{}	<state>.<Initial>

Events	Guard	Code	Next
[<state>.<Initial>]			
retrieve_re_from_ang_vel		{}	<state>.<Initial>

Events	Guard	Code	Next
[<state>.<Initial>]			
retrieve_arm_force		{}	<state>.<Initial>

Events	Guard	Code	Next
[<state>.<Initial>]			
retrieve_arm_torque		{}	<state>.<Initial>

Events	Guard	Code	Next
[<state>.<Initial>]			
retrieve_arm_pos		{}	<state>.<Initial>

Figure 6.10.: State table showing the `retrieve` events of the five safety checks

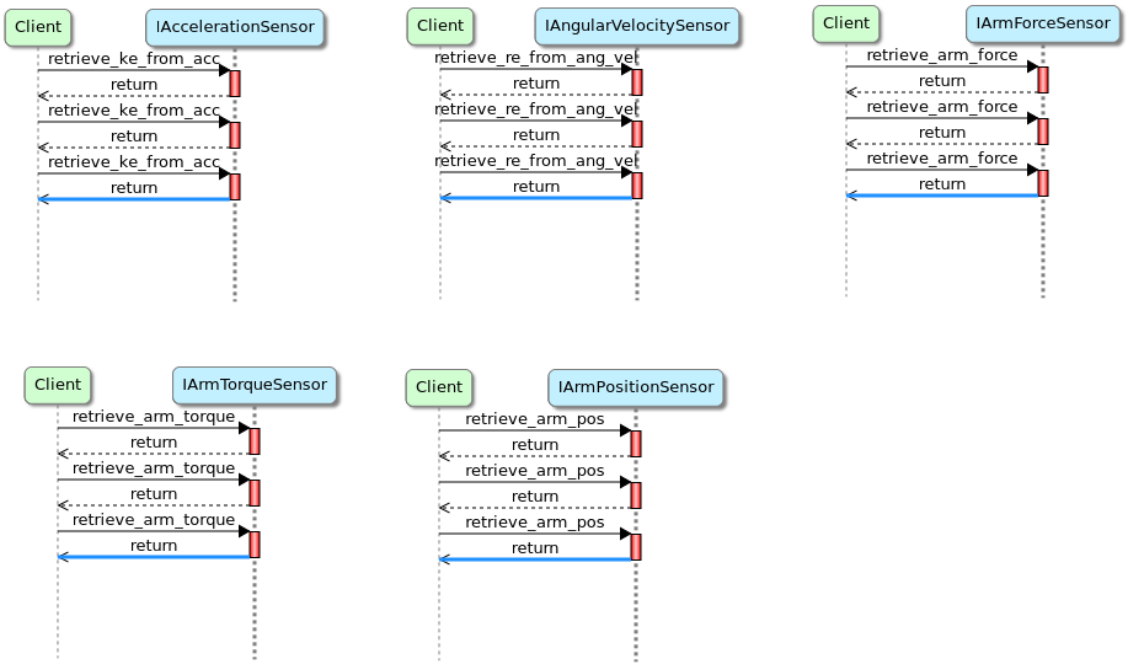


Figure 6.11.: Sequence diagrams of the `ISensor` interfaces

6.3.5. IResolver

`IResolver` is a single interface with five events that resolve values retrieved from sensors. Resolving a value in this context means translating a retrieved sensor value into something the Dezyne language can use in its control logic. Concretely, this means that the re-

solve events convert an obtained sensor value into `Behavior.Safe` and `Behavior.Unsafe` behavior types. This can be seen in the state table of Figure 6.12. Every resolve event either returns `Behavior.Safe` or `Behavior.Unsafe`. Later we will see that the `Controller` uses this to turn the LED blue in case of a `Behavior.Safe` reply from the resolve event and to turn the LED red in case of a `Behavior.Unsafe` reply from the resolve event. The safety check components execute their own respective resolve event. `KineticEnergyCheck` executes `resolve_ke_from_acc`, `RotationalEnergyCheck` executes `resolve_re_from_ang_vel`, `ArmForceCheck` executes `resolve_arm_force`, `ArmTorqueCheck` executes `resolve_arm_torque` and `ArmPositionCheck` executes `resolve_arm_position`.
Just like with `ISafetyCheck` we are free to reply either `Behavior.Safe` or `Behavior.Unsafe` in a simulation of the model, as can be seen in the sequence diagram of Figure 6.13.

Events	Guard	Code	Next
[<state>.<Initial>]			
resolve_ke_from_acc		reply(Behavior.Safe);	<state>.<Initial>
resolve_ke_from_acc		reply(Behavior.Unsafe);	<state>.<Initial>
resolve_re_from_ang_vel		reply(Behavior.Safe);	<state>.<Initial>
resolve_re_from_ang_vel		reply(Behavior.Unsafe);	<state>.<Initial>
resolve_arm_force		reply(Behavior.Safe);	<state>.<Initial>
resolve_arm_force		reply(Behavior.Unsafe);	<state>.<Initial>
resolve_arm_torque		reply(Behavior.Safe);	<state>.<Initial>
resolve_arm_torque		reply(Behavior.Unsafe);	<state>.<Initial>
resolve_arm_pos		reply(Behavior.Safe);	<state>.<Initial>
resolve_arm_pos		reply(Behavior.Unsafe);	<state>.<Initial>

Figure 6.12.: State table of `IResolver`

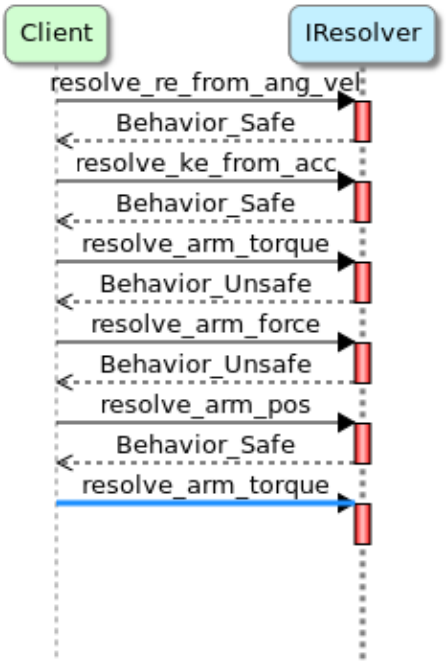


Figure 6.13.: Sequence diagram of `IResolver`

One might wonder why `ISensor` consists of five separate interfaces, while `IResolver` has all resolve events in just one interface. This has to do with a useful modelling feature we used for `IResolver`. Since every safety check has to convert its retrieved sensor value to a Dezyne control logic type, and since every `resolve` action does roughly the same thing, we generalized it to one interface that is used by all checks. If we refer back to the system view of Figure 6.5, we can see this usage of `IResolver`. All checks, except `BaseCaseCheck` (`BaseCaseCheck` does not actually check anything and serves as the base case or default element for the linked list of checks), are connected to the grey square on the right edge, this is the port of `IResolver`. This port is to be bound in implementation code to functions that actually resolve or convert retrieved sensor values into Dezyne control logic. We will see this binding in Section 6.4.2.

6.3.6. BaseCaseCheck

We have now arrived at the first component, `BaseCaseCheck`. As was just mentioned, `BaseCaseCheck` does not check any behavior of the care robot and serves as the base case or default element for the linked list of checks above it. It should always be the last element. If one wishes to add a new safety check to the system, it can be added anywhere in the list of checks, so long as it is above `BaseCaseCheck`. `BaseCaseCheck` implements the `ISafetyCheck` interface. `BaseCaseCheck` does one thing, and it does it well; in conformance with its interface, it executes the `do_check` event. The `do_check` event of `BaseCaseCheck` always returns `Behavior.Safe`, this is the so called base case. This is because every time we execute a check the result of this check and the directly

following check will be evaluated. If one or both of the checks replies `Behavior.Unsafe`, then this`Behavior.Unsafe` value is replied further down to the controller. One might ask why one check needs to evaluate the result of the next check. This is a consequence of the linked list structure of the checks. To be clear, the controller just tasks the first check to execute. That check then executes the next check, and the next check the following check, and so towards the base check. So it is really a particular check that executes the next check and evaluates both itself and that next check, the controller does not do this evaluation the controller. The controller only tasks the first check to execute. The task of the controller will be elaborated later this section.
Figure 6.14 shows the state table we created for `BaseCaseCheck`. Here we see that it has only one event and always returns `Behavior.Safe`. Because the behavior of BaseCaseCheck is trivial we have omitted it here. It can be found in Section A.3 of Appendix A.

Events	Guard	Code	Next
[<state>.<Initial>]			
do_check		reply(Behavior.Safe);	<state>.<Initial>

Figure 6.14.: State table of `BaseCaseCheck`

6.3.7. KineticEnergyCheck

We will now look at the `KineticEnergyCheck` component. `KineticEnergyCheck` implements the `ISafetyCheck` interface and it uses the `IResolver` and `ISensors` interfaces. The state table of `KineticEnergyCheck` is shown in Figure 6.15. Elaborating `KineticEnergyCheck` will be straightforward, as we have already elaborated all pieces it is composed of because we work through the system inside out. In Figure 6.15 we can see that `KineticEnergyCheck` defines a single event, `do_check`, in conformance with the interface it implements. Let us look at the Code column. Here we see that it calls the `retrieve_ke_from_acc` event from the acceleration sensor interface to let the acceleration sensor obtain sensor data from the accelerometer, which can then be used to calculate kinetic energy. The raw value of kinetic energy is not usable in the Dezyne modelling language, so we convert this value to Dezyne usable control logic by calling the resolver's `resolve_ke_from_acc` event. This event returns a Dezyne usable control logic type, namely `Behavior.Safe` or `Behavior.Unsafe`, which we store as the result of the `KineticEnergyCheck`. We then then refer to our pointer, `iNext` which points to the next check in the list, and call the this next check. Looking back at the system view of Figure 6.5, we can see that the next check is `RotationalEnergyCheck`, but it might as well be any other check that we may add it the future. After calling the next check, we execute the the `and_safety_states` function, which performs a boolean AND [1] on the Behavior types of `KineticEnergyCheck` and the next check. The purpose of this function has to

[1]This is a boolean AND if we define `Behavior.Unsafe` to be 0 and `Behavior.Safe` to be 1. A boolean AND will only return 1 if both proponents are 1. In terms of our function, the `_and_safety_states` function will only return `Behavior.Safe` if both proponents have value `Behavior.Safe`.

do with what we mentioned when we elaborated `BaseCaseCheck`. There we mentioned that if one or both of the checks results in behavior type `Behavior.Unsafe`, then the `and_safety_states` returns `Behavior.Unsafe`. Otherwise, it returns `Behavior.Safe`, which would indicate that both individual checks replied `Behavior.Safe`. Finally, since the Controller is the component that calls `KineticEnergyCheck`, the return value of the `and_safety_states` function is returned to the controller. The controller will decide what to do with his value.

Events	Guard	Code	Next
[<state>.<Initial>]			

do_check		Behavior currSafetyState = Behavior.Safe;
		iAccelerationSensor. retrieve_ke_from_acc ();
		currSafetyState = iResolver. resolve_ke_from_acc ();
		Behavior nextSafetyState = iNext. do_check ();
		Behavior res = and_safety_states(currSafetyState,nextSafetyState);
		reply(res);
			<state>.<Initial>

Figure 6.15.: State table of `KineticEnergyCheck`

`KineticEnergyCheck` is a good opportunity for us to show the Dezyne verification results. The verification results are shown in Figure 6.16. Verification goes from top to bottom. This indicates that, to verify `KineticEnergyCheck`, the `IAccelerationSensor`, `ISafetyCheck` and `IResolver` has to be verified first. It is only if these interfaces contain no errors that `KineticEnergyCheck` can possibly be given a positive result. It is thus convenient to have model the system in such a way that is divided into chunks that can be individually verified.

Check	Action	Time	States	Transitions	Done	Result
IAccelerationSensor						
Deadlock		0:00	2	2	100%	✓
Livelock		0:00	2	2	100%	✓
ISafetyCheck						
Deadlock		0:00	3	4	100%	✓
Livelock		0:00	3	4	100%	✓
IResolver						
Deadlock		0:00	3	12	100%	✓
Livelock		0:00	3	12	100%	✓
KineticEnergyCheck						
Deterministic		0:00	13	16	100%	✓
Illegal		0:00	13	16	100%	✓
Deadlock		0:00	13	16	100%	✓
Livelock		0:00	13	16	100%	✓
Compliance		0:00	13	16	100%	✓

Figure 6.16.: Verification results for the `KineticEnergyCheck` component

Let us now look at the sequence diagram of we created for `KineticEnergyCheck`. This sequence diagram is shown in Figure 6.17. This sequence diagram visualises what we tried to explain concerning the `and_safety_states` function. `iKineticEnergyCheck` executes a `do_check` event, which, after calling `resolve_ke_from_acc`, returns either `Behavior.Safe` or `Behavior.Unsafe`. Then the `do_check` event of the next check is executed. This next `do_check` also returns either `Behavior.Safe` or `Behavior.Unsafe` (internally, the next check's `do_check` event also executes a resolve action. The result

of this resolve is what is actually returned by the next `do_check`, but this is not shown). The first part of the sequence diagram of Figure 6.17, starting at `iKineticEnergyCheck` all the way to `iResolver` and back to `iKineticEnergyCheck`, returns `Behavior.Safe`, because the two executed checks both returned Behavior.Safe. The second time we make this round, the first check returns `Behavior.Safe` and the second check returns `Behavior.Unsafe`. This results in `Behavior.Unsafe` being returned to `iKineticEnergyCheck`, because there was a check that returned unsafe behavior. The same holds for the third round. The fourth round returns `Behavior.Safe` again, because the two checks both returned `Behavior.Safe`. Although this is not shown in the sequence diagram, if both checks return `Behavior.Unsafe`, `Behavior.Unsafe` is returned to `iKineticEnergyCheck` as well.

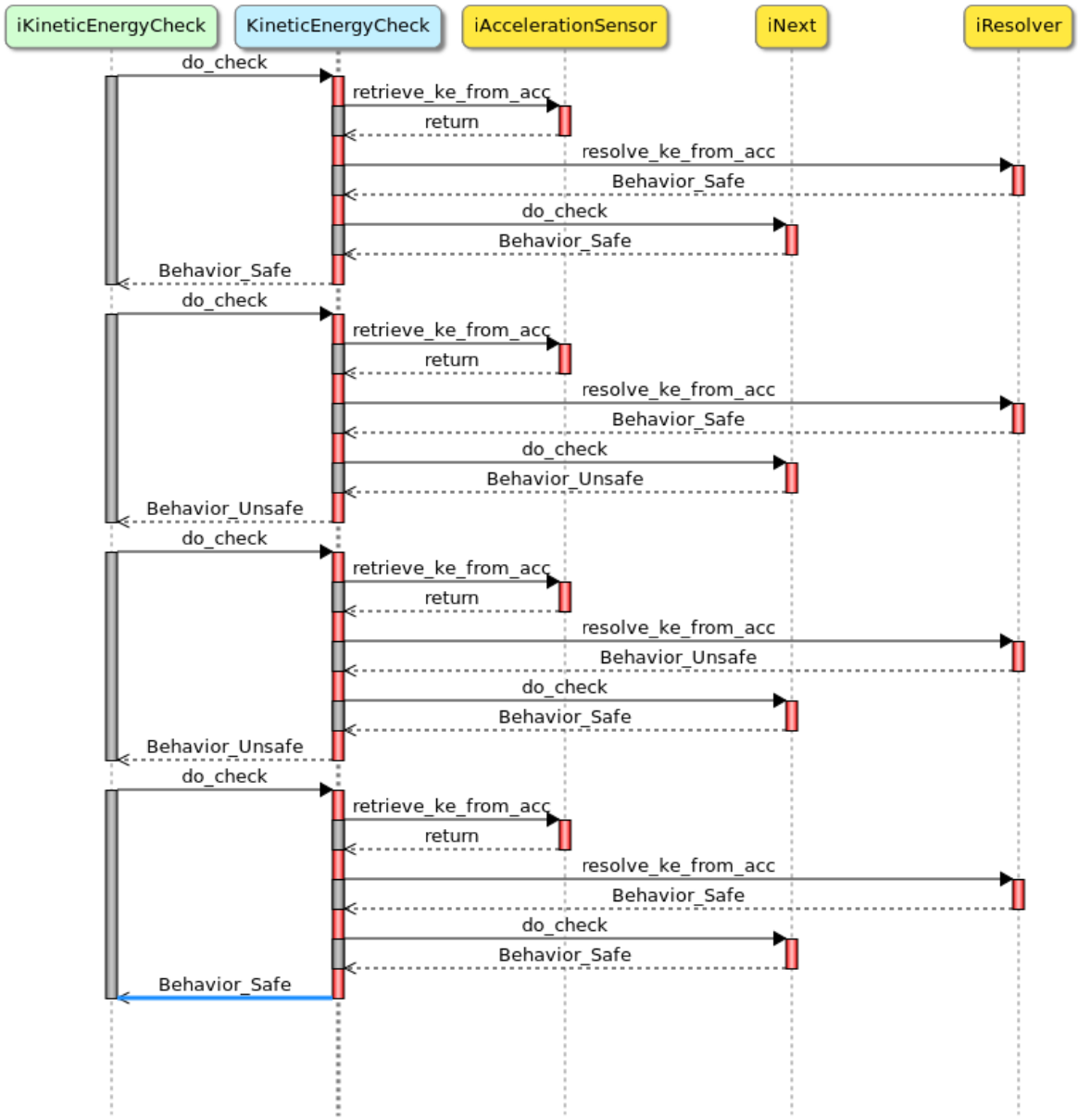


Figure 6.17.: Sequence diagram of `KineticEnergyCheck`

6.3.8. RotationalEnergyCheck, ArmForceCheck, ArmTorqueCheck and ArmPositionCheck

`RotationalEnergyCheck`,`ArmForceCheck`, `ArmTorqueCheck` and `ArmPositionCheck` all behave very similarly to `KineticEnergyCheck`, except that they call the the retrieve event and resolve event for their respective quantities that they want to check. Figure 6.18, Figure 6.19, Figure 6.20 and Figure 6.21 for `RotationalEnergyCheck`, `ArmForceCheck`, `ArmTorqueCheck` and `ArmPositionCheck` respectively. We have omitted the sequence diagrams for these checks here. The sequence diagrams for these checks

can be found in Section A.7, Section A.5, Section A.6 and Section A.8 of Appendix A respectively.

Events	Guard	Code	Next
[<state>.<Initial>]			

do_check		Behavior currSafetyState = Behavior.Safe;
		iAngularVelocitySensor. retrieve_re_from_ang_vel ();
		currSafetyState = iResolver. resolve_re_from_ang_vel ();
		Behavior nextSafetyState = iNext. do_check ();
		Behavior res = and_safety_states(currSafetyState,nextSafetyState);
		reply(res);
			<state>.<Initial>

Figure 6.18.: State table for `RotationalEnergyCheck`

Events	Guard	Code	Next
[<state>.<Initial>]			

do_check		Behavior currSafetyState = Behavior.Safe;
		iArmForceSensor. retrieve_arm_force ();
		currSafetyState = iResolver. resolve_arm_force ();
		Behavior nextSafetyState = iNext. do_check ();
		Behavior res = and_safety_states(currSafetyState,nextSafetyState);
		reply(res);
			<state>.<Initial>

Figure 6.19.: State table for `ArmForceCheck`

Events	Guard	Code	Next
[<state>.<Initial>]			

do_check		Behavior currSafetyState = Behavior.Safe;
		iArmTorqueSensor. retrieve_arm_torque ();
		currSafetyState = iResolver. resolve_arm_torque ();
		Behavior nextSafetyState = iNext. do_check ();
		Behavior res = and_safety_states(currSafetyState,nextSafetyState);
		reply(res);
			<state>.<Initial>

Figure 6.20.: State table for `ArmTorqueCheck`

Events	Guard	Code	Next
[<state>.<Initial>]			

do_check		Behavior currSafetyState = Behavior.Safe;
		iArmPositionSensor. retrieve_arm_pos ();
		currSafetyState = iResolver. resolve_arm_pos ();
		Behavior nextSafetyState = iNext. do_check ();
		Behavior res = and_safety_states(currSafetyState,nextSafetyState);
		reply(res);
			<state>.<Initial>

Figure 6.21.: State table for `ArmPositionCheck`

If one wants to add a new check to the system, this is likely going to be its base control structure also. The differences will be in the functions the retrieve and resolve functions are bound to.

6.3.9. IController

The final interface and component of the safety module system are `IController` and `Controller` respectively. We will first elaborate the `IController` interface. As `IController`

is the interface of `Controller`, it specifies what behavior the controller should conform to. The controller is the component that governs the safety module system, it starts the check sequence and controls the LED matrix.
In Figure 6.22 the state chart of `IController` is given. Here we see that when the `initialise` event fires, the `initialise_imu`, `initiliase_mutexes` and `initialise_semaphores` are called (`initialise_framebuffer` of `ILEDControl` is called in the `Controller` but not in its interface. This is a consequence of the way the system is modelled. It has no implications for the operational behavior of the system, though) and the system jumps from the `Idle` to the `Operating state`. Calling `initialise_imu`, `initiliase_mutexes` and `initialise_semaphores` on the `initialise` event ensures that the IMU, mutexes and semaphores used by the safety module are always properly initialised before the system will execute its control loop in the `Operating` state. The safety module makes use of mutexes and semaphores because the implementation is multithreaded. In a similar manner as `initialise`, when the `destruct` event fires, the `destruct_mutexes` and `destruct_semaphores` are called. The IMU need not be explicitly destructed. This destruct action is not supposed to execute, however, as the safety module is expected to run indefinitely in the `Operating` state, or at least until Rose shuts down.
In the `Operating` state `do_checks` calls the first check of the linked list of checks. `do_checks` is shown twice in Figure 6.22, because the interface specifies that `do_checks` either returns `UnsafeTriggerered.Yes` or `UnsafeTriggered.No`. The notions of `UnsafeTriggerered.Yes` and `UnsafeTriggered.No` are used to monitor whether one or more of the checks returned unsafe behavior.
Now, to explain the controller further, it is much more useful to go right to the `Controller` at this stage, then to go through the event table and sequence diagram of `IController`. Therefore, we will omit these diagrams of `IController` here.

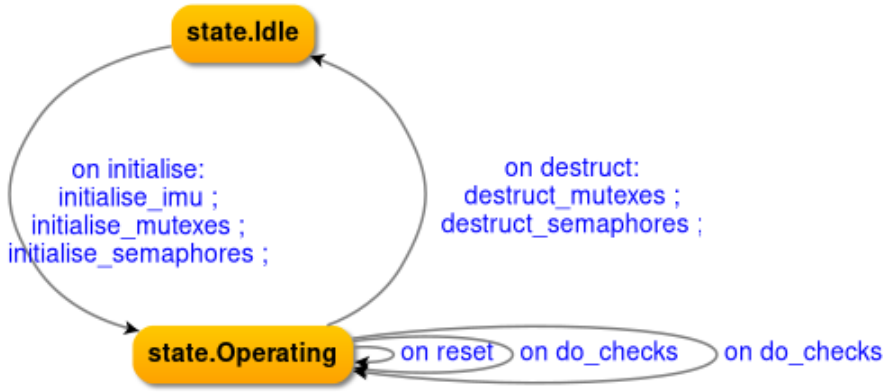


Figure 6.22.: State chart of IController

6.3.10. Controller

Let us now look at the final component: `Controller`. In Figure 6.23 the event table of the controller is shown. We have already elaborated `initialise` and `destruct` in the controller its interface. In addition to the behavior specified by its interface, the controller also calls `initialise_framebuffer` when the `initialise` events fires as well as `destruct_framebuffer` when the `destruct` events fires. Because the event table omits an important conditional in the code column, we will refer to the actual dezyne code we used to create the model for the `do_checks` event, which is listed in Code 4. For the `reset` event, we refer to Figure 6.23.

Let us first look at `do_checks`. Upon entering the `do_checks` event, we call the `do_check` event of `iHead`. `iHead`, `iHead` always points to the first check in the list. On lines 5 to 7 there is an if statement that is entered when the result of `iHead.do_check` is `Behavior.Unsafe`. When will the result of `iHead.do_check` be `Behavior.Unsafe`? Recall that when we execute a check, the check will internally call `do_check` of the next check. So if we start at the Controller, which calls the `do_check` of `KineticEnergyCheck`, `KineticEnergyCheck`'s `do_check` will call `RotationalEnergy`'s `do_check`, `RotationalEnergy`'s `do_check` will call `GripArmForce`'s `do_check`, etc. This will be done all the way to `BaseCaseCheck`'s `do_check`. Recall that `BaseCaseCheck`'s `do_check` always returns `Behavior.Safe`. Further recall that every check executes a boolean AND function on its own and next check's result, returning `Behavior.Unsafe` when either one or both of the checks reply `Behavior.Unsafe`. Since the recursion stops at `BaseCaseCheck`, we can traverse the linked list back up again, finishing all `do_check` calls. Along the way back up, every check will perform the boolean AND function with the subsequently executed check. This means that, ultimately, the result of `iHead.do_check` in the controller is the result of all boolean ANDs from all checks that were called on the way through the linked list. This implies that the result of `iHead.do_check` in the controller is `Behavior.Unsafe` as soon as one or more of the checks result in `Behavior.Unsafe`.

If the result of `iHead.do_check` in the controller is `Behavior.Unsafe`, we enter the if statement on Line 5 of Code 4. Since the safety module has detected unsafe behavior, we will now instruct `iLEDControl` to light the LED matrix red. Signalling unsafe behavior like this is the safety module's most important task, so it has the highest priority in the system.

On Line 7 of Code 4 we set `unsafe_acknowledged`, a boolean variable, to `false`. The boolean variable `unsafe_acknowledged` denotes whether the operator of Rose, or any system that monitors Rose for that matter, has acknowledged or recognized that unsafe behavior has taken place. On Lines 11 to 16 we can see that the controller replies `UnsafeTriggerd.Yes` if `unsafe_acknowledged` has the value of `false` and that the controller replies `UnsafeTriggered.No` if `unsafe_acknowledged` has the value of `true`. The mechanism of `unsafe_acknowledged` is related to something we described when we elaborated the `ILEDControl` interface. When we elaborated the `ILEDControl` interface, we described the built in mechanism of that interface to prevent one to override a red LED with a blue LED. It is for this reason that we can only set the LED to blue when

`unsafe_acknowledged` has the value of `true`, for so long no one or nothing has acknowledged that the robot has performed unsafe behavior the LED matrix should stay red. In Figure 6.23 we can see that the `reset` event sets `unsafe_acknowledged` to the value of `true` and that it instructs the `iLEDControl` to reset the LED matrix. The `reset` action corresponds to the operator of the robot acknowledging the unsafe behavior. `reset` is the only action in the system that can intstruct the `iLEDControl` to reset the LED matrix and to acknowledge unsafe behavior. The notions of `UnsafeTriggered.Yes` and `UnsafeTriggered.No` of Code 4 are related to this. So long as the unsafe behavior is not acknowledged, the controller will return `UnsafeTriggered.Yes`. It is only if no unsafe behavior has yet been detected or if the unsafe behavior has been acknowledged by the operator of the robot that the controller's `do_check` event will return `Unsafe.Triggered.No`.

States	Guard	Code	Next
initialise			
[systemState.Idle]		iLEDControl. initialise_framebuffer (); iController. initialise_imu (); iController. initialise_mutexes (); iController. initialise_semaphores (); systemState = State.Operating;	systemState.Operating
destruct			
[systemState.Operating]		iLEDControl. destruct_framebuffer (); iController. destruct_mutexes (); iController. destruct_semaphores (); systemState = State.Idle;	systemState.Idle
do_checks			

[systemState.Operating]		{
		Behavior safetyState = iNext. do_check ();
		if (safetyState == Behavior.Unsafe) {
		iLEDControl. light_led_red (red);
		unsafe_acknowledged = false;
		}if (!(unsafe_acknowledged)) {
		reply(UnsafeTriggered.Yes);
		}}
			systemState.Operating

reset			
[systemState.Operating]		iLEDControl. reset_led (); unsafe_acknowledged = true;	systemState.Operating

Figure 6.23.: Event table of `Controller`

In the verification results on the `Controller` component shown in Figure 6.24, we see once again that to verify a component, we first need to verify and ensure that none of the interfaces it implements or uses contains verification errors.

```
on iController.do_checks(): {
    Behavior safetyState = iHead.do_check();

    if (safetyState == Behavior.Unsafe) {
        iLEDControl.light_led_red(red);
        unsafe_acknowledged = false;
    } else if(unsafe_acknowledged) {
        iLEDControl.light_led_blue(blue);
    }
    if(!unsafe_acknowledged) {
        reply(UnsafeTriggered.Yes);
    }
    else {
        reply(UnsafeTriggered.No);
    }
}
```

Listing 4: `on iController.do_checks()` definition

Check	Action	Time	States	Transitions	Done	Result
IController						
Deadlock		0:00	12	19	100%	✓
Livelock		0:00	12	19	100%	✓
ILEDControl						
Deadlock		0:00	9	25	100%	✓
Livelock		0:00	9	25	100%	✓
ISafetyCheck						
Deadlock		0:00	3	4	100%	✓
Livelock		0:00	3	4	100%	✓
Controller						
Deterministic		0:00	40	54	100%	✓
Illegal		0:00	40	54	100%	✓
Deadlock		0:00	40	54	100%	✓
Livelock		0:00	40	54	100%	✓
Compliance		0:00	40	54	100%	✓

Figure 6.24.: Verification results for `Controller`

Figure 6.25 shows a sequence diagram of the Controller. Here we first execute the `initialise` action to initialise the system, then we execute `do_checks` multiple times. This sequence diagram does not show us a completely recursive trace through all the safety checks, sadly. This is a result of making the model generic by using a linked list like data structure for the checks. Because we simulate the model to create a sequence diagram of the controller, we get to choose what value `iNext.do_check` returns, and the system need not recursively traverse all checks to determine the correct return value for `iNext.do_check`. Note, however, that the operation of a simulation like this is different from what the verifier does; the verifier *does* need to recursively traverse through the

checks. It furthermore needs to traverse any particular sequence of events as to ensure there are no errors hiding in a particular path through the system.
We added a sequence diagram of a less generic, earlier version of the Controller to show how a sequence diagram looks that shows multiple checks being executed in sequence to Appendix A. This sequence diagram can be found in Section A.10. This sequence diagram uses old naming conventions that are not used anymore.

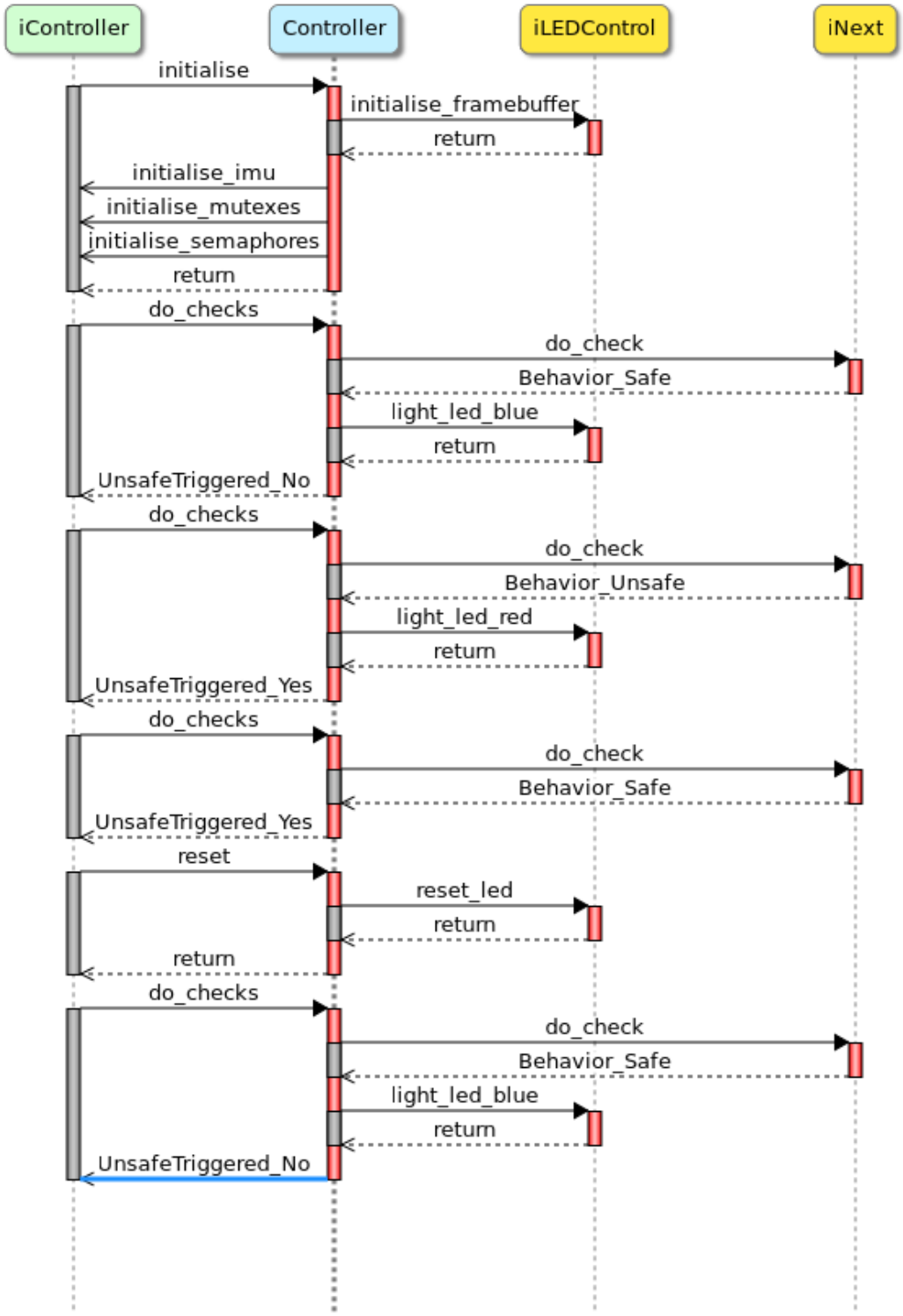


Figure 6.25.: Sequence diagram of the controller

6.3.11. Functional behavior

So far, when we verified the models, we have only looked at static behavior verification. At the time of writing, a prototype version of Dezyne provides us with funcional behavior verification as well. Functional behavior checking concerns itself with checks whether a system, after executing a set of actions, is in the system state that we expect it to be. We made several of these functional behavior checks. Because this is prototype functionality of the toolset, we cannot create a diagram to show the specification of such functional behavior, so we show the specification of the functional behavior in the modelling language itself.

Code 5 shows the functional behavior specification for `KineticEnergyCheck`. Code 6 shows part of a modified version of `KineticEnergyCheck` that conforms with the syntax required to use the functional requirement verifier. Actually, the part that is shown is a modification of `IResolver` in which `IResolver` now part of the safety check itself and no long returns a value, but calls either `iKineticEnergyCheck.BehaviorSafe()`, `iKineticEnergyCheck.BehaviorUnsafe()`, which corresponds to the same behavior as returning `Behavior.Safe` and `Behavior.Unsafe` respectively. In Code 5, the `Requirement` keyword is used to define a functional behavior specification. We gave this specification the name `AnyUnsafe`. This specification specifically checks if the `myState` variable has the value we expect it to be. If, in Code 6, we were to remove line 14, this would get through the static behavior verifier as if nothing happened, but if we run this through the functional behavior verifier, we get an error, because the state does not accord with what we specified in the functional specification.

Similar functional specifications have been made for the other safety check components. For the generated source code that is to be bound to implementation code, we have used the non-functional requirement model, however, as this model has a more convenient syntax to work with.

```
requirement AnyUnsafe {
        on KineticEnergyCheck;
        provides ISafetyCheck iKineticEnergyCheck;
        requires IAccelerationSensor iAccelerationSensor;
        requires ISafetyCheck iNext;
        requires IResolver iResolver;

    behaviour {
        Behavior myState = Behavior.Safe;
        bool expected = false;

        on iKineticEnergyCheck.do_check(): {
            myState = Behavior.Safe;
            expected = true;
        }

        on iNext.BehaviorUnsafe(),
            iResolver.BehaviorUnsafe(): {
            myState = Behavior.Unsafe;
            if(expected) {
                expected = false;
            } else {
                iKineticEnergyCheck.BehaviorUnsafe();
            }
        }

        on iNext.BehaviorSafe(),
            iResolver.BehaviorSafe(): {
            if(expected) {
                myState = Behavior.Safe;
                expected = false;
            } else {
                if(myState == Behavior.Safe) {
                    iKineticEnergyCheck.BehaviorSafe();
                } else {
                    iKineticEnergyCheck.BehaviorUnsafe();
                }
            }
        }
    }
}
```

Listing 5: Definition of `AnyUnsafe` requirement

```
on iResolver.BehaviorUnsafe(),
        iNext.BehaviorUnsafe(): {
        myState = Behavior.Unsafe;

        if(expected) {
            expected = false;
        } else {
            iKineticEnergyCheck.BehaviorUnsafe();
        }
}
```

Listing 6: Edited `iResolver` for functional verification

6.4. Implementation

In this section, we will elaborate how we went from the formal model to an implementation of the safety module. We avoid to show source-code and try to elaborate the results by the use of figures and diagrams to make the text comprehensible to a broader public.

6.4.1. Multithreaded nature of the safety module

In this section, we present various models regarding the multithreaded nature of the safety module. In Dezyne we did not module the system as a set of threads, instead in the Dezyne model there is just one thread that executes checks in a sequential manner, as we mentioned in Section 6.3 we shortly mentioned that the safety checks are executed in sequence by Dezyne. One might wonder what gain a system has from being multithreaded if the control loop, the essence of the system, executes checks in a sequential manner. This is what we would like to elaborate in this section.
To see the idea behind this let us consider the overview of threads in the safety module system in Figure 6.26. On top we see `rt_<task>` and `nrt_<task>`. These do not denote threads, but the general naming convention for the threads of the system. `<task>` denotes the specific task a thread executes. Below the right arrow we see that two of these tasks are to light the LED and to reset the LED. Here we see both two threads prefixed with `rt_` and two threads prefixed with `nrt_`. `rt` stands for real-time and `nrt` stands for non-real-time. These threads thus denote real-time and non-real-time threads. Real-time threads are supposed to run in primary mode and should not make a mode switch to secondary mode, as to ensure that they are only governed by the Xenomai co-kernel and not by the standard Linux kernel. Why would a thread running in primary mode be a real-time thread? This is because the Xenomai kernel handles all time critical activities, and has priority over the standard Linux kernel. All we have to do is to make sure that the tasks that require real-time operation stay in primary mode. In contrast

to the real-time threads, the non-real-time threads are allowed to be in secondary mode. A mode switch to secondary mode is made when the thread makes a kernel call to the standard Linux driver. This happens when a system call is made that is not wrapped with a real-time version by the Xenomai framework. In Section 6.1.3 we showed that a subset of POSIX system calls are wrapped by the Xenomai framework. A thread will make a mode switch to secondary mode when it makes a call to the IMU or to ROS. This is why only non-real-time threads are allowed to communicate with drivers like the IMU and ROS. How then can sensor data be obtained by a real-time thread without leaving primary mode? We answered this question briefly in Section 6.1.3, real-time threads can receive sensor data by use of XDDP sockets. We will look at this more in depth later this section.
In the middle of Figure 6.26, on the left side of the arrow, we have `rt_sample_<quantity>`, `rt_retrieve_quantity` and `nrt_retrieve<quantity>`. These are threads concerning the sampling and retrieval of a quantity. Here `<quantity>` refers to the quantity measured by the five safety checks. So `<quantity>` refers to acceleration, angular displacement, force, torque or position. These threads run periodically, in contrast to the LED threads, which do not run periodically and are initiated by the main thread, the main thread we can see on the far left of Figure 6.26. The main thread is called `sm`. Below that we see the check thread, `rt_do_checks`, which is essentially our Dezyne system running indefinitely, sequentially going through the five checks.

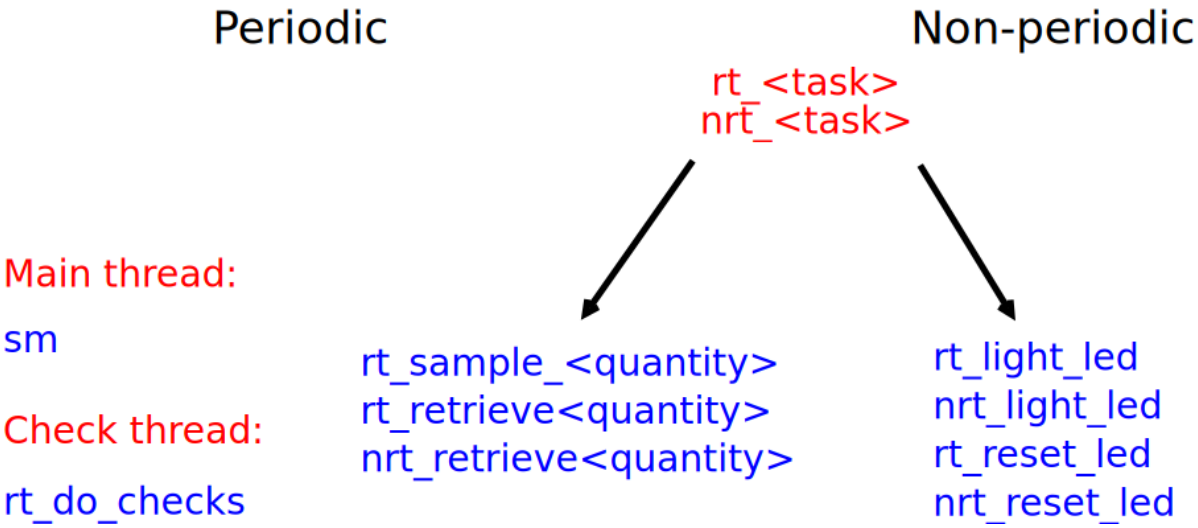


Figure 6.26.: Overview of threads in the safety module

Let us take a closer look at the threads that concern themselves with the LED matrix. In Figure 6.27 we have given a diagram highlighting the communication between the LED threads. At top we see `dzn_light_led` and `dzn_reset_led`. These are in fact not threads; `dzn_light_led` and `dzn_reset_led` denote functions that are initiated from Dezyne by the main thread. `dzn_light_led` corresponds to a `light_led_blue` action or a `light_led_red` action and `dzn_reset_led` corresponds to a `reset_led` action, all three which we have seen in the models of Dezyne. The `dzn` functions initiate a line

of communication between threads up to the LED driver. Both lines of communication on the left side and on the right side have identical operation, differing only in their task, namely lighting the LED or resetting the LED. The `dzn` functions increment a semaphore, both have a semaphore respective for their task, which the `rt` threads decrement. The `rt` threads wait for the semaphore to be incremented before they do any work. The `rt` threads then write over XDDP sockets, also one respective for the task, which actuate the `nrt` threads to write to the LED driver. The XDDP sockets prevent the `rt` threads from making a mode switch to from primary mode to secondary mode, by tasking the `nrt` to do the writing to the LED driver instead. Thus, the `nrt` threads make a mode switch from primary mode to secondary mode, but this is not a problem, as this is what they are designed for. Both `nrt_light_led` and `nrt_reset_led` access the LED driver mutual exclusively.

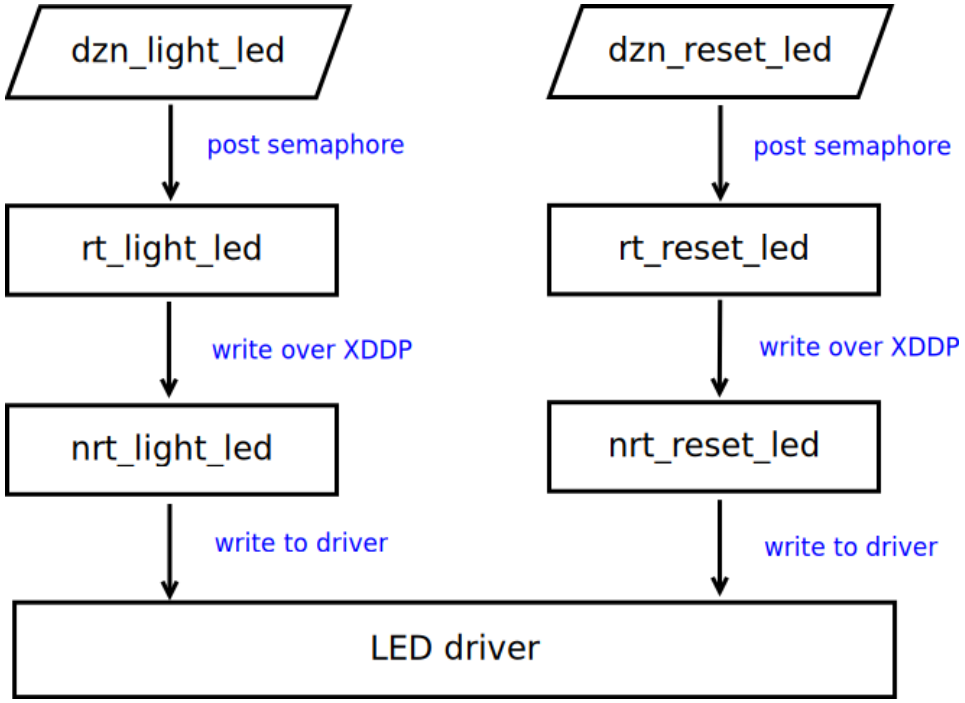


Figure 6.27.: Communication between the LED threads

In Figure 6.27 we have seen one way how the multithreaded nature of the safety module manifests itself. To light the matrix or to reset the matrix the Dezyne loop of the main thread just increments semaphores, which does not take any significant time. It lets the `rt` and `nrt` threads do the actual work. The LED threads are the only threads in the system that are actuated by the main thread. The other threads of the system run periodically at a rate which is a multiple of the IMU poll rate, to provide a common factor of time between the threads.
Let us now look at the communication between the `<quantity>` threads, which shows the other way how the multithreaded nature of the safety module manifests itself. In Figure 6.28 we have given a diagram highlighting the communication between the `<quantity>`

threads. Since `<quantity>` denotes acceleration, angular displacement, force, torque or position, this diagram implies that most threads of the system operate in a similar manner, differing only in the quantity they are involved with. Each thread in the diagram is involved with the same quantity, so there is no communication between threads that are involved with different quantities. In the top left of Figure 6.28 we see `dzn_retrieve_<checked_quantity>`, this is once again a function called from the main thread instead of a thread itself. The `dzn_retrieve_<checked_quantity>` function is called from inside an executing safety check. In the figure a distinction is made between `quantity` and `checked_quantity`. The checked quantity is the quantity that is being checked by one of the five checks. For example, in the case of the kinetic energy check, the quantity is acceleration and the checked quantity is kinetic energy. In the case of the arm checks, the quantity and checked quantity are the same, for that what we receive from the sensors is also the safety metric. The `checked quantity` ellipse at the top right of the figure denotes a variable. This variable is read by the `dzn_retrieve<checked_quantity>` function and updated by the `rt_sample_<quantity>` thread, both in a thread-safe manner. The `rt_sample_<quantity>` thread samples the quantity in real-time. It retrieves the quantity from `rt_retrieve<quantity>` over a socket connection between the two threads. The `rt_retrieve<quantity>` thread retrieves the quantity from `nrt_retrieve_<quantity>`, which in turn retrieves the quantity from the driver. Here we see the same pattern as is used by the LED threads of Figure 6.27. A real-time thread communicates with a non-real-time thread over an XDDP socket so that it can stay in primary mode (if it did not stay in primary mode we would not have called it a real-time thread).

There is a subtle point to make. For the arm force check, the arm torque check and the arm position check there is no `rt_sample_<quantity>` thread, because the quantity is the same as the checked quantity (e.g. the force check retrieves force from the driver and force is also the safety metric). The rest of the figure remains the same for these checks.

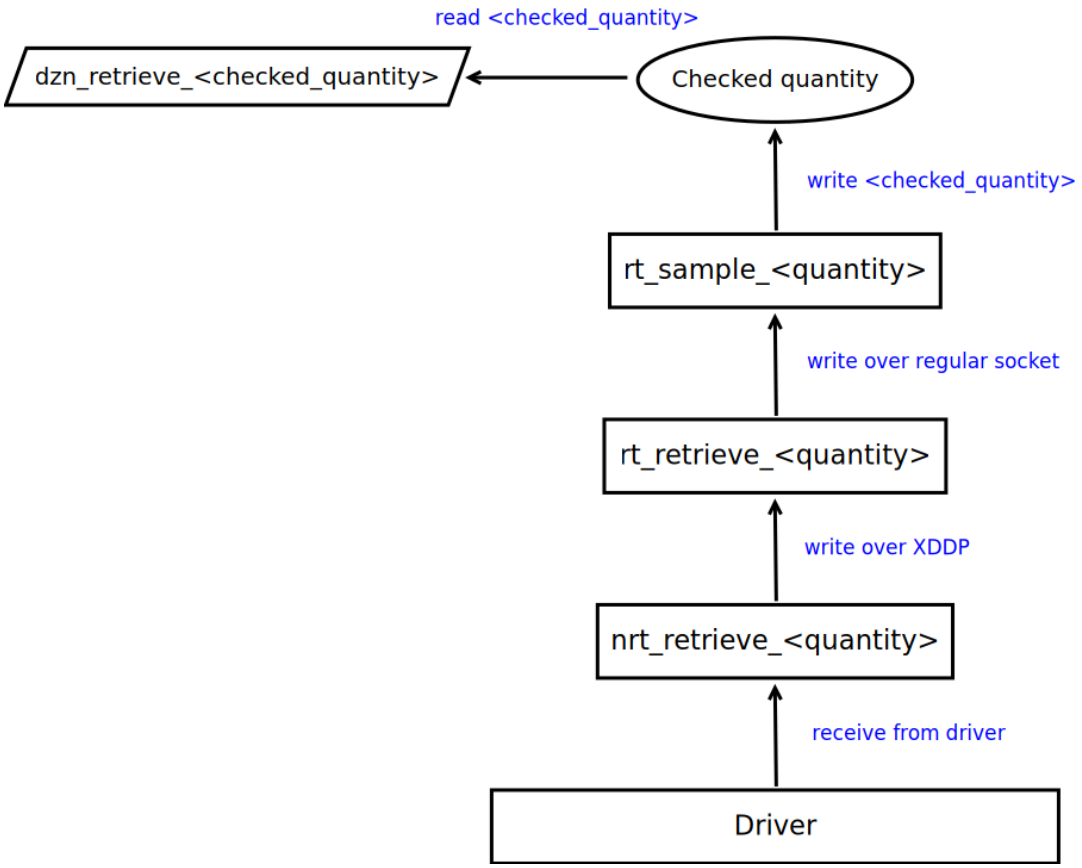


Figure 6.28.: Communication between `<quantity>` threads

We have set up the communication between threads this way so that `dzn_retrieve_<checked_quantity>` need not do any work other than retrieving the checked quantity, which is a cheap operation in terms of time. The respective threads do the work of sampling and retrieving the quantity. Furthermore, the variable containing the checked quantity always has a value, so the `dzn_retrieve_<checked_quantity>` need not block because no data is available. It is in this manner that the multithreaded nature of the safety module manifests itself while working with the sequential nature of the Dezyne control loop.

6.4.2. Binding and calling Dezyne events

In the previous Results sections we have mentioned the interaction of the Dezyne system with the rest of the safety module several times. It is now time to elaborate how exactly the transition is made from the formal model to implementation.
In Section 6.3.1, we saw the System View of the safety module, shown in Figure 6.5. In that section, we talked about ports which are instances of interfaces that are found on the edges of the system. Some of these ports contain events that need to be bound to implementation code. After generating code from the Dezyne models, the binding of these edge ports to functions in the implementation code is essentially the step that needs

to be taken to transition from design to implementation. The Dezyne runtime, which is imported in the implementation code, makes sure we bind every event that needs to be bound. Not all ports on the edges of the System View need to be bound to a function in the implementation, some need only be called from the implementation code. Which events need to be called and which events need to be bound is determined by the location of the ports in the System View. Ports on the top edge of the System View, from which we see only one in Figure 6.3.1, need to be called. The one port on top is the `Controller` port. Calling this port in implementation code will execute the control loop. The ports at the bottom of Figure 6.3.1 all need to be bound to a function in implementation code. These ports on the bottom are the `Sensors` which contain the retrieve events. These retrieve events are bound to their respective `rt_retrieve_quantity` function. These functions are then executed as threads. on the right of the System View we see one gray port, this is the `Resolver`, responsible for translating values to Dezyne control logic. The `Resolver` is actually very similar to the `Sensor` ports at the bottom. It contains events that need to be bound in implementation code, just like the `Sensors`. The difference is that each sensor is implemented as a separate interface (there are five sensors one for each checked quantity), while the resolver is one interface that contains all events required to resolve the values.

6.4.3. Testing the safety module

In this section we elaborate various tests on the real-time operation of the safety module.

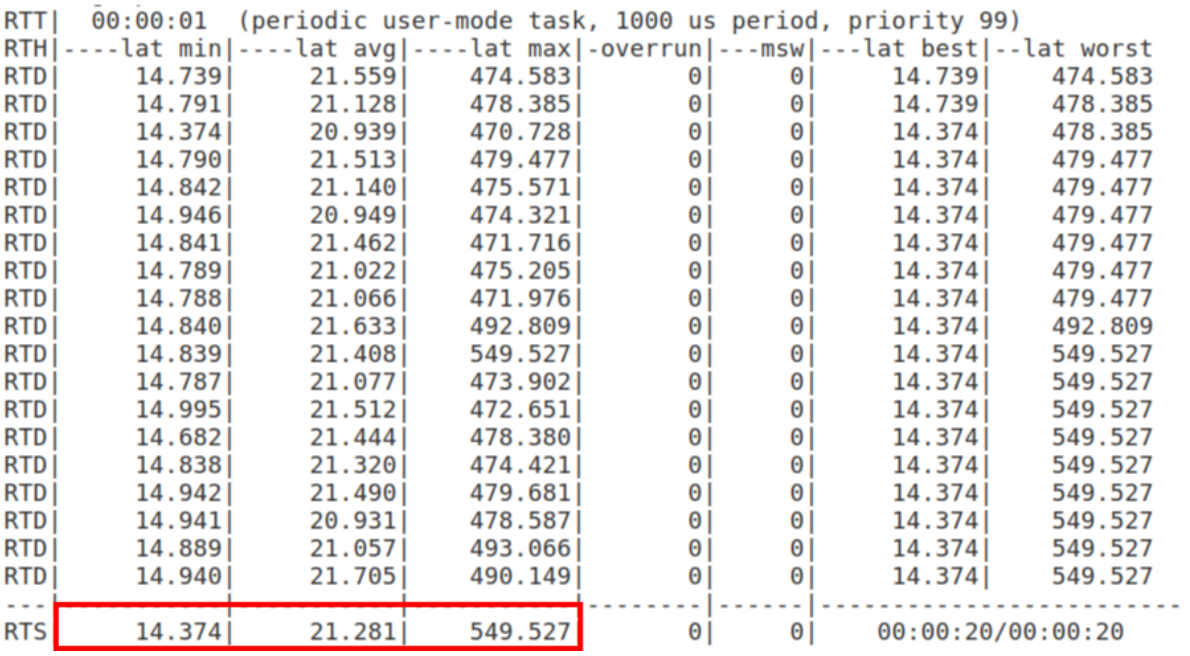
6.4.4. Latency tests

In this section, we elaborate the latency tests we performed on the system. The latency of the system refers to the the time between a stimulus and the system's response. The latency tests were run with Xenomai's test suite program `latency`. In this section, two latency tests are shown. The first test shows the latency of the system without the safety module program running and the second test shows the latency of the system while running the safety module. All shown latency statistic have units of microseconds. Both tests schedule tasks with a period of 1000 microseconds, the highest scheduling priority of 99. The tests are both run for 20 seconds.
Let us first look at the latency of the system without the safety module program running. This test is shown in Figure 6.29. The elements that are of interest to us are the minimum latency, the average latency and the maximum latency, which are highlighted in that order in the red frame. The highlighted minimum latency and highlighted maximum latency are respectively the best and the worst latency of the system in the past 20 seconds. The highlighted average latency is the average of the average latency of the system of the past 20 seconds. We can see that for the system without the safety module program running, the minimum latency is 14.945 microseconds, the average latency is 15.712 microseconds and the maximum latency is 31.504 microseconds.
If we then look at Figure 6.30, the latency test with the safety module program running, we can see highlighted in the red frame that the minimum latency is 14.374 microseconds,

that the average latency is 21.281 microseconds and that the maximum latency is 549.527 microseconds. Thus, there is a slight increase in the average latency compared to the average latency shown in Figure 6.29, and there is an increase by a factor of 17 for the maximum latency. Coincidentally, there is a slight decrease in the minimum latency. In both Figure 6.30 and Figure 6.29, in the fourth column from the right, we can see that the test tasks have no deadline overruns.
Sadly, our safety module's own threads could not be used to test the latency of the system with (the safety module threads are only used to put a load on a system); we instead are limited to using test tasks. However, Section 6.4.5 shows other interesting statistics that are run on the safety module's own threads.

```
RTT|  00:00:01  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|     15.051|     15.694|     22.187|       0|     0|     15.051|     22.187
RTD|     15.103|     15.692|     23.228|       0|     0|     15.051|     23.228
RTD|     15.102|     15.736|     22.603|       0|     0|     15.051|     23.228
RTD|     15.102|     15.671|     23.071|       0|     0|     15.051|     23.228
RTD|     15.049|     15.655|     21.873|       0|     0|     15.049|     23.228
RTD|     14.945|     15.721|     21.924|       0|     0|     14.945|     23.228
RTD|     15.101|     15.710|     22.132|       0|     0|     14.945|     23.228
RTD|     15.048|     15.679|     24.476|       0|     0|     14.945|     24.476
RTD|     15.048|     15.717|     23.746|       0|     0|     14.945|     24.476
RTD|     15.100|     15.681|     21.714|       0|     0|     14.945|     24.476
RTD|     15.047|     15.664|     21.922|       0|     0|     14.945|     24.476
RTD|     15.151|     15.721|     23.380|       0|     0|     14.945|     24.476
RTD|     15.046|     15.711|     22.130|       0|     0|     14.945|     24.476
RTD|     15.098|     15.875|     31.504|       0|     0|     14.945|     31.504
RTD|     15.098|     15.727|     22.754|       0|     0|     14.945|     31.504
RTD|     15.045|     15.767|     22.233|       0|     0|     14.945|     31.504
RTD|     15.097|     15.711|     23.274|       0|     0|     14.945|     31.504
RTD|     15.096|     15.707|     22.336|       0|     0|     14.945|     31.504
RTD|     15.096|     15.700|     23.899|       0|     0|     14.945|     31.504
---|-----------|-----------|-----------|--------|------|-------------------------
RTS|     14.945|     15.712|     31.504|       0|     0|    00:00:20/00:00:20
```

Figure 6.29.: Latency test without running safety module.



```
RTT|  00:00:01  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|     14.739|     21.559|    474.583|       0|     0|     14.739|    474.583
RTD|     14.791|     21.128|    478.385|       0|     0|     14.739|    478.385
RTD|     14.374|     20.939|    470.728|       0|     0|     14.374|    478.385
RTD|     14.790|     21.513|    479.477|       0|     0|     14.374|    479.477
RTD|     14.842|     21.140|    475.571|       0|     0|     14.374|    479.477
RTD|     14.946|     20.949|    474.321|       0|     0|     14.374|    479.477
RTD|     14.841|     21.462|    471.716|       0|     0|     14.374|    479.477
RTD|     14.789|     21.022|    475.205|       0|     0|     14.374|    479.477
RTD|     14.788|     21.066|    471.976|       0|     0|     14.374|    479.477
RTD|     14.840|     21.633|    492.809|       0|     0|     14.374|    492.809
RTD|     14.839|     21.408|    549.527|       0|     0|     14.374|    549.527
RTD|     14.787|     21.077|    473.902|       0|     0|     14.374|    549.527
RTD|     14.995|     21.512|    472.651|       0|     0|     14.374|    549.527
RTD|     14.682|     21.444|    478.380|       0|     0|     14.374|    549.527
RTD|     14.838|     21.320|    474.421|       0|     0|     14.374|    549.527
RTD|     14.942|     21.490|    479.681|       0|     0|     14.374|    549.527
RTD|     14.941|     20.931|    478.587|       0|     0|     14.374|    549.527
RTD|     14.889|     21.057|    493.066|       0|     0|     14.374|    549.527
RTD|     14.940|     21.705|    490.149|       0|     0|     14.374|    549.527
---|-----------|-----------|-----------|--------|------|-----------------------
RTS|     14.374|     21.281|    549.527|       0|     0|    00:00:20/00:00:20
```

Figure 6.30.: Latency test while running safety module.

6.4.5. Mode switches and other statistics

In this section, we elaborate on the statistics of the safety module obtained with Xenomai's `stat` program. These statistics are shown in Figure 6.31, Figure 6.32, Figure 6.33 and Figure 6.34. In all figures, the threads of interest for us, namely the threads of the safety module system, are highlighted with a red frame. The columns that are mostly of interest of us are the `MSW` column and the `%CPU` column. `MSW` stands for *mode switch*. The column denotes the number of mode switches made by a specific thread. `%CPU` denotes the percentage of CPU use by a specific thread. Figure 6.31 and Figure 6.32 show the statistics of the system after watching the `stat` program using `watch` for 5 seconds and 15 seconds respectively. Figure 6.33 and Figure 6.34 show the system after watching for 5 seconds and 15 seconds as well, but after some modifications have been made to the code. We will discuss these modifications shortly, but first, let us first look at the first table of the statistics after 5 seconds shown in Figure 6.31.

Notice that all of the highlighted threads are familiar to us, as we elaborated the purposes of these threads in Section 6.4.1. The only difference is that there are no `(n)rt_retrieve_acceleration` and `(n)rt_retrieve_angular_displacement` threads, instead there are `nrt_retrieve_imu` and `rt_retrieve_imu`. This is because acceleration and angular displacement are obtained from the same IMU and they are therefore both retrieved at once.
One thing that is immediately notable in the table of Figure 6.31 is the high number of mode switches performed by the `rt_checks` thread, which is framed in blue. If we then look at the table of Figure 6.32, which shows the statistics of the system after 15

seconds, we see that the number of mode switches is now roughly three times as high, which means the `rt_checks` thread makes about 100 mode switches every 5 seconds. Recall from Section 6.4.1 that the `rt_checks` thread is the control loop originating from Dezyne that executes the five checks in sequence. Further recall from 6.4.1 that real-time threads are not supposed to be making a mode switch from primary mode to secondary mode, as this transfers control over the thread from the Xenomai cokernel to the standard Linux kernel, which defeats the purpose of a real-time thread.

```
Every 0.1s: cat /proc/xenomai/sched/stat

CPU  PID    MSW        CSW        XSC        PF    STAT       %CPU  NAME
  0  0      0          7318955    0          0     00218000    77.1  [ROOT/0]
  1  0      0          5490147    0          0     00218000   100.0  [ROOT/1]
  2  0      0          6256641    0          0     00218000   100.0  [ROOT/2]
  3  0      0          6708274    0          0     00218000   100.0  [ROOT/3]
  0  8294   1          1          53         0     002600c0     0.0  sm
  0  8299   3          7          21         0     00248042     0.0  rt_light_led
  0  8300   3          19         54         0     00248042     0.0  rt_retrieve_acceleration
  0  8301   3          19         54         0     00248042     0.0  rt_retrieve_angular_displaceme
  0  8302   3          21         60         0     00248042     0.0  rt_retrieve_arm_force
  0  8303   3          20         57         0     00248042     0.0  rt_retrieve_arm_torque
  0  8304   3          20         57         0     00248042     0.0  rt_retrieve_arm_position
  0  8305   1          42825      128558     0     00248040    19.8  rt_sample_acceleration
  0  8306   1          19         125        0     00248044     0.0  rt_sample_angular_velocity
  0  8307   7          8          16         3     002600c0     0.0  nrt_light_led
  0  8308   1          2          34         0     002600c0     0.0  nrt_retrieve_imu
  0  8309   1          1          20         0     002600c0     0.0  nrt_retrieve_arm_force
  0  8310   1          1          19         0     002600c0     0.0  nrt_retrieve_arm_torque
  0  8311   1          1          19         0     002600c0     0.0  nrt_retrieve_arm_position
  0  8312   107        124        408        0     00248044     0.0  rt_checks
  1  0      0          5719329    0          0     00000000     0.0  [IRQ17: [timer]]
  2  0      0          6484900    0          0     00000000     0.0  [IRQ17: [timer]]
  3  0      0          6938630    0          0     00000000     0.0  [IRQ17: [timer]]
```

Figure 6.31.: Thread statistics (5 seconds)

```
Every 0.1s: cat /proc/xenomai/sched/stat

CPU  PID    MSW        CSW        XSC        PF    STAT       %CPU  NAME
  0  0      0          7408303    0          0     00218000    76.8  [ROOT/0]
  1  0      0          5490147    0          0     00218000   100.0  [ROOT/1]
  2  0      0          6256641    0          0     00218000   100.0  [ROOT/2]
  3  0      0          6708274    0          0     00218000   100.0  [ROOT/3]
  0  8294   1          1          53         0     002600c0     0.0  sm
  0  8299   3          22         81         0     00248042     0.0  rt_light_led
  0  8300   3          52         153        0     00248042     0.0  rt_retrieve_acceleration
  0  8301   3          52         153        0     00248042     0.0  rt_retrieve_angular_displaceme
  0  8302   3          56         165        0     00248042     0.0  rt_retrieve_arm_force
  0  8303   3          56         165        0     00248042     0.0  rt_retrieve_arm_torque
  0  8304   3          56         165        0     00248042     0.0  rt_retrieve_arm_position
  0  8305   1          132033     396361     0     00248040    19.8  rt_sample_acceleration
  0  8306   1          54         370        0     00248044     0.1  rt_sample_angular_velocity
  0  8307   37         45         76         18    002600c0     0.0  nrt_light_led
  0  8308   1          2          100        0     002600c0     0.0  nrt_retrieve_imu
  0  8309   1          1          55         0     002600c0     0.0  nrt_retrieve_arm_force
  0  8310   1          1          55         0     002600c0     0.0  nrt_retrieve_arm_torque
  0  8311   1          1          55         0     002600c0     0.0  nrt_retrieve_arm_position
  0  8312   338        391        1296       0     00248044     0.2  rt_checks
  1  0      0          5720210    0          0     00000000     0.0  [IRQ17: [timer]]
  2  0      0          6485857    0          0     00000000     0.0  [IRQ17: [timer]]
  3  0      0          6939312    0          0     00000000     0.0  [IRQ17: [timer]]
```

Figure 6.32.: Thread statistics (15 seconds)

We figured the excessive mode switching has to do with the way Dezyne is run in the safety module, as the only operation of the `rt_checks` thread is to execute the safety checks, of which the control logic is implemented in Dezyne. Note that, as we generate source code from the Dezyne model, in the end the Dezyne model is still just C++ code. We found that the excessive mode switching is caused by the Dezyne runtime's default behavior to print output to the output screen or to print output to a file. Since this output is not needed outside of debugging purposes, it can be disabled without a problem. This can be done by editing the source code of the Dezyne runtime directly, but we instead choose to set the output stream in our safety module code to a null stream object, which, after initialisation of the null stream object, has no operational overhead whatsoever.
The table of Figure 6.33 shows the statistics after running the safety module system for five seconds after a restart of the safety module system with the excessive mode switching patch we just elaborated. In the figure we can see that the mode switching of `rt_checks` has been reduced to just two mode switches, which is a low number as we want from a real-time thread.

```
Every 0.1s: cat /proc/xenomai/sched/stat

CPU  PID    MSW        CSW        XSC        PF     STAT       %CPU  NAME
  0  0      0          7971007    0          0      00218000    77.2  [ROOT/0]
  1  0      0          5490147    0          0      00218000   100.0  [ROOT/1]
  2  0      0          6256641    0          0      00218000   100.0  [ROOT/2]
  3  0      0          6708274    0          0      00218000   100.0  [ROOT/3]
  0  8773   1          1          53         0      002600c0     0.0  sm
  0  8777   3          20         73         0      00248042     0.0  rt_light_led
  0  8778   3          18         51         0      00248042     0.0  rt_retrieve_acceleration
  0  8779   3          18         51         0      00248042     0.0  rt_retrieve_angular_displaceme
  0  8781   3          19         54         0      00248042     0.0  rt_retrieve_arm_force
  0  8782   3          19         54         0      00248042     0.0  rt_retrieve_arm_torque
  0  8783   3          19         54         0      00248042     0.0  rt_retrieve_arm_position
  0  8784   1          40080      120320     0      00248044    19.8  rt_sample_acceleration
  0  8785   1          17         111        0      00248044     0.0  rt_sample_angular_velocity
  0  8786   33         42         68         16     002600c0     0.0  nrt_light_led
  0  8787   1          1          32         0      002600c0     0.0  nrt_retrieve_imu
  0  8788   1          1          18         0      002600c0     0.0  nrt_retrieve_arm_force
  0  8789   1          1          18         0      002600c0     0.0  nrt_retrieve_arm_torque
  0  8790   1          1          18         0      002600c0     0.0  nrt_retrieve_arm_position
  0  8791   2          18         325        0      00248044     0.0  rt_checks
  1  0      0          5725375    0          0      00000000     0.0  [IRQ17: [timer]]
  2  0      0          6491673    0          0      00000000     0.0  [IRQ17: [timer]]
  3  0      0          6943580    0          0      00000000     0.0  [IRQ17: [timer]]
```

Figure 6.33.: Thread statistics (5 seconds, with excessive mode switching fix)

For the remainder of this section, look at the table of Figure 6.34, which shows statistics of the threads of the safety module system after running for 15 seconds and after appliance of the excessive mode switching patch. Using this table we will elaborate on some other statistics that are shown.
Let us first look at the rows of `rt_light_led` and `nrt_light_led`. Notice how `rt_light_led` has made three mode switches and how `nrt_light_led` has made 55 mode switches. This is exactly what we want; the real-time thread stays in primary mode and it is the non-real-time thread that makes the switches to secondary mode to access the LED driver. Looking at the other real-time threads, we see non of them make a lot of mode

switches, which is what we want. Looking at the non-real-time threads, we see that non of these make a lot of mode switches either. This is not surprising for the `arm` threads, as they simulate interaction with ROS by just providing randomly generated sensor data. `nrt_retrieve_imu` on the other hand does interact with real sensors, but it need only make one mode switch.
In fact, we found that all of the mode switching that is performed is performed at initialisation of the system, except in the case of the mode switching performed by `nrt_light_led`, which performs its mode switching throughout the entire execution of the system. This means that after initialisation of the system, `nrt_light_led` is the only thread that performs mode switches.

Let us lastly look at the `%CPU` column shown in Figure 6.34. We can see that, of the safety module threads, `rt_sample_acceleration` the highest percentage of the CPU at this timestamp (in fact, the table refreshes every 0.1 seconds, and every 0.1 seconds `rt_sample_acceleration` takes the highest percentage of CPU time. The rest of the threads take similar CPU percentages as the ones shown in this timestamp). `rt_sample_acceleration` has the highest CPU percentage because of the internal calculation of velocity from acceleration that `rt_sample_acceleration` performs. We can see that `rt_checks` takes a percentage of 0.4 of the CPU and that the rest of the threads appear to be using 0.0% of the CPU. All threads except `rt_sample_acceleration` and `rt_checks` have simple work to do and are rapidly done with their task.

```
Every 0.1s: cat /proc/xenomai/sched/stat

CPU  PID    MSW        CSW        XSC        PF    STAT       %CPU  NAME
  0  0      0          8073312    0          0     00218000    77.0  [ROOT/0]
  1  0      0          5490147    0          0     00218000   100.0  [ROOT/1]
  2  0      0          6256641    0          0     00218000   100.0  [ROOT/2]
  3  0      0          6708274    0          0     00218000   100.0  [ROOT/3]
  0  8773   1          1          53         0     002600c0     0.0  sm
  0  8777   3          31         117        0     00248042     0.0  rt_light_led
  0  8778   3          55         162        0     00248042     0.0  rt_retrieve_acceleration
  0  8779   3          55         162        0     00248042     0.0  rt_retrieve_angular_displaceme
  0  8781   3          60         177        0     00248042     0.0  rt_retrieve_arm_force
  0  8782   3          60         177        0     00248042     0.0  rt_retrieve_arm_torque
  0  8783   3          60         177        0     00248042     0.0  rt_retrieve_arm_position
  0  8784   1          142354     427346     0     00248044    19.6  rt_sample_acceleration
  0  8785   1          58         398        0     00248044     0.0  rt_sample_angular_velocity
  0  8786   55         69         112        27    002600c0     0.0  nrt_light_led
  0  8787   1          1          106        0     002600c0     0.0  nrt_retrieve_imu
  0  8788   1          1          59         0     002600c0     0.0  nrt_retrieve_arm_force
  0  8789   1          1          59         0     002600c0     0.0  nrt_retrieve_arm_torque
  0  8790   1          1          59         0     002600c0     0.0  nrt_retrieve_arm_position
  0  8791   2          59         1055       0     00248044     0.4  rt_checks
  1  0      0          5726252    0          0     00000000     0.0  [IRQ17: [timer]]
  2  0      0          6492999    0          0     00000000     0.0  [IRQ17: [timer]]
  3  0      0          6944516    0          0     00000000     0.0  [IRQ17: [timer]]
```

Figure 6.34.: Thread statistics (15 seconds, with excessive mode switching fix)

6.4.6. Running the safety module

In this section, we elaborate on executing the checks on the safety module. We highly recommend checking out the repository, which demonstrates the kinetic energy and

rotational energy checks in video formats. These demonstrations will provide a clear view of the operation of the safety module. The kinetic energy and rotational energy demonstrations can be found in `demos` directory of the root of the repository. If checking out the repository is not an option, this section presents the kinetic energy check and rotational energy check as a series of frames. These are given in Figure 6.35 and Figure 6.36 respectively.

After initialisation the safety module will run indefinitely, running safety checks one by one. As we have elaborated in Section 6.4.1, the checks themselves are run in sequence, but the retrieval and sampling of sensor data is run continuously parallel to that. There is no requirement for how fast the checks should run, [2]but we have set the period to be exactly the same as the poll rate of the IMU. This is to ensure sensor data is always ready to be read when a check demands it. However, since the drivers the sensor data is received from run in secondary mode, there is no guarantee that data is truly prepared when a check demands it. If the data is not prepared, the check uses the last available data. The system keeps track of the time the last data was retrieved. If the last data was retrieved too long ago the executing check will assume the behavior is unsafe, because it cannot keep track of the behavior of the care robot on time anymore. *Too long ago data* is set to be a time of twice the period that checks are executed, so twice the poll rate of the IMU.
Let us look at two particular checks, the kinetic energy check and rotational energy check. These checks need not be simulated and can be demonstrated in real life. For both kinetic energy and rotational energy we set the limit of maximum energy to be 300 joules. Using Equation 6.3 and Equation 6.4 to solve for velocity $\vec{v}$ and angular velocity respectively, this means robot Rose can have a maximum velocity and angular velocity of 4.8 meters per second. Note that this calculation for velocity is just that what results from a change in velocity. In other words, it does not take into account if Rose is already driving.
In Figure 6.35 we can see the kinetic energy check in action. Here we started shaking the safety module slowly, and increasingly shake it harder, so that the acceleration becomes higher. The last 12 frames of this are shown in the figure. From the fifth frame to the sixth, the velocity obtained from the acceleration causes the kinetic energy to be higher than the allowed maximum, which in turn makes the matrix to turn red to denote that this is unsafe behavior. It then takes us six more frames to stop shaking.
Although this is not the case in the demonstration of Figure 6.35, sometimes small delays are experienced between the high acceleration and the LED turning red.
In Figure 6.36 we can see the rotational energy check in action. This check is similar to kinetic energy, the difference being that it is concerned with angular quantaties as opposed to linear quantities. In the figure we make a quick jerk from left to right, causing the rotational energy obtained from the angular velocity from frame three to frame five to be higher than the allowed maximum, thus resulting in unsafe behavior

[2]This is not part of the assignment and requires further research. More details are in Section 8.0.5 of the Recommendations

and the matrix turning red.

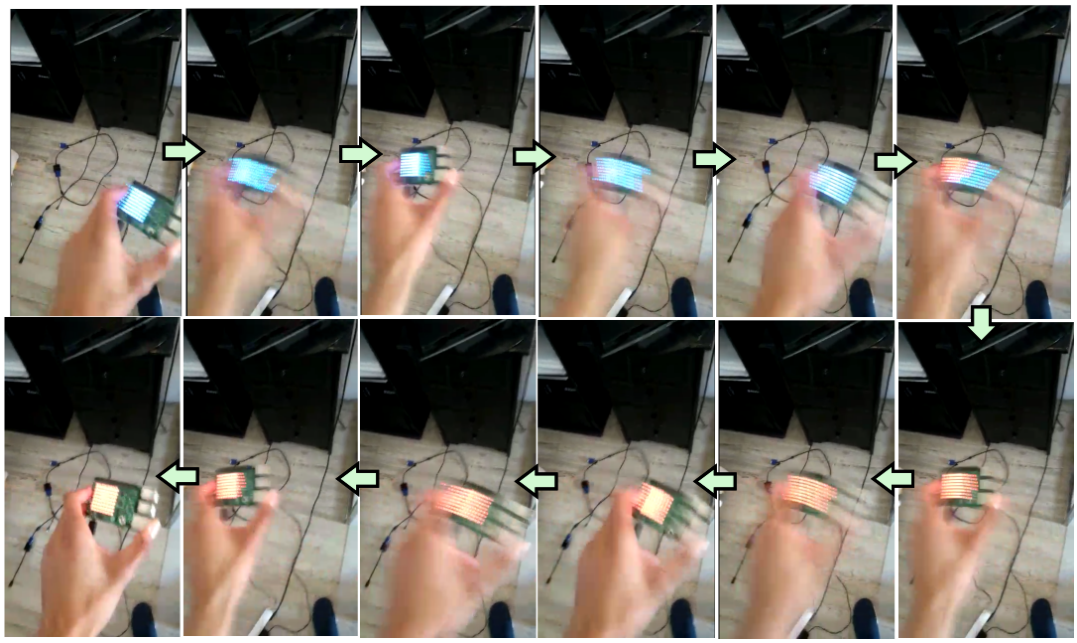


Figure 6.35.: High linear acceleration will cause the matrix to turn red.

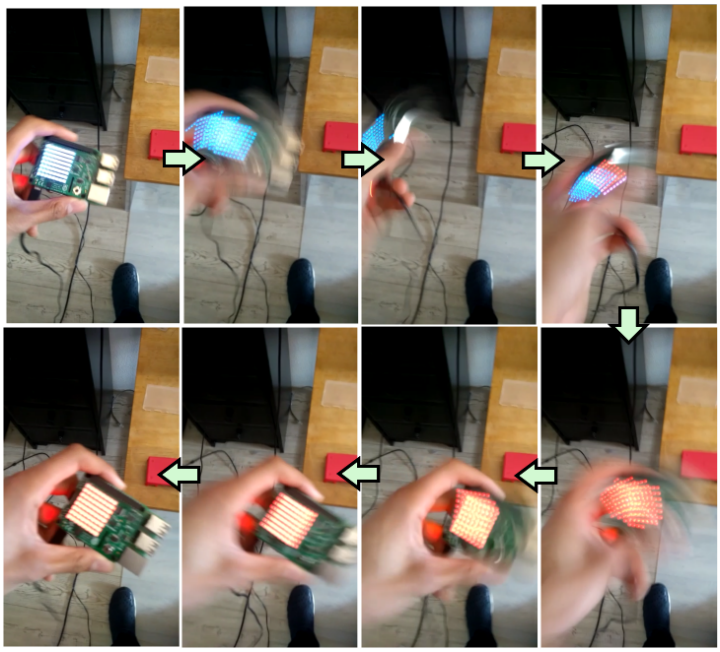


Figure 6.36.: High angular velocity will cause the matrix to turn red.

The matrix cannot turn blue again until we acknowledge the unsafe behavior by clicking the reset button. Successive checks with resets in between are shown in the demonstrations on the repository.

7. Conclusion

In this research, we conducted a feasibility study to determine the feasibility of combining ideas from the fields of mobile robotic safety, verification of system behavior and real-time systems, and applying the combination of these ideas to a safety module for care robot Rose. To determine the feasibility we asked ourselves how a real-time safety module can be designed and implemented that provides an extra layer of safety in order to prevent injury to patients or damage to the environment because of collision, clamping or pinching caused by care robot Rose. This principal research question was worked out into four subsquestions, from which the results were given Section 6.1, Section 6.2, Section 6.4 of the Results. We will now interpret the results of these subquestions.

7.1. On the hardware and the operating system

In Section 6.1 the results of our literature study and comparative study were given. There we found that the Raspberry Pi with Sense HAT is the best hardware platform according to the requirements specified. We think the Raspberry Pi with the Sense HAT is an excellent choice in terms of scalability and further versions of the safety module. We elaborated how the built in WiFi card can be used to communicate with Rose and her operator, and how additional sensors on the Sense HAT can be used to obtain more data from the environment.
In the comparative study, we compared different hardware platforms, but we did not go in depth about different versions of the hardware platform. For example, in the case of the Raspberry Pi, a comparison could be made between the Raspberry Pi Zero W, the Raspberry Pi 3 and the Raspberry Pi 4. The comparison of different versions of the hardware platform is given in Section 8.0.2 of the Recommendations.
In Section 6.1, we elaborated how the Xenomai framework has better real-time performance for our purposes than the Real-time Linux patches to standard Linux. In Section 5.5 we saw that the Xenomai framework is able to provide hard real-time performance in kernelspace. Thus, using the Xenomai framework for the prototype safety module provides a solid ground for further versions of the safety module, in which we may want to port the userspace system to kernelspace. The use of the POSIX skin and XDDP sockets is also scalable; because of the POSIX skin and XDDP sockets new sensors (including sensors from robot Rose through ROS) can be added to the safety module without the need to port their drivers to the Xenomai-framework.

7.2. On the usage of sensor data and the detection of unsafe behavior

In Section 6.2, we elaborated our findings on how to retrieve sensor data and use it to check for unsafe behavior. There we found that the IMU of the Sense HAT can be used to retrieve acceleration and angular velocity from. This is useful, because this provides us with a way to measure the kinetic energy and rotational energy independent robot Rose. Values for the time intervals Δt were determined experimentally and set to be a multiple of the IMU poll rate to provide a common multiple of time throughout the system, but we do not think this is an optimal time interval. In fact, it may be the cause of the small delays that sometimes occur as mentioned in Section 6.4.6. But since our research's objective is to show the feasibility of the safety module instead of immersing ourselves in the timeliness of the system, we did not aim for the most optimal time intervals.
Regardless of the timing, the equations prove to work well to detect unsafe behavior, as has been shown in the demonstrations in Section 6.4.6.

7.3. On the modelling of the safety module's behavior

In Section 6.3, we modelled the behavior of the safety module. The model puts some guarantees on the behavior of the safety module; like the affirmation that there are no deadlocks and livelocks. This affirmation is not limited to the model but propagated to the implementation code as well, because the source code that is generated from the Dezyne models are bound and called from the implementation code. As we aimed for the hardware platform of the safety module to be extensible for the future, we also made sure the model can be easily augmented with new safety checks, by implementing the checks as elements of a linked list. The only downside that comes with this abstraction of checks to elements of a linked list is that the sequence diagrams of the controller no longer show what checks are being executed. This is merely a drawback in the generation of the sequence diagrams; in the implementation code all information about individual checks is still available.

7.4. On the implementation of the safety module

In Section 6.4, we elaborated how we implemented the model of the safety module on the operating system with the Xenomai framework and we elaborated its interaction with the other threads in the system. In Section 6.4 Figure 6.29 and Figure 6.30 showed the latency of the system without running the safety module program and while running the safety module program respectively. We noticed the increase in the maximum latency when running the safety module program. The increase in maximum latency is not a problem, for we have no requirement on what the maximum latency of the system should be. Still, the Figure 6.30 shows that if the system is put under sufficiently large load

the response time to a given stimuli decreases. We also noted that the minimum latency and average latency under load of the safety module program does not differ very much from the minimum latency and average latency when the safety module is not running. The average latency is closer to the the minimum latency in both cases, which means we only reach the maximum latency seldom or it means our executed period test tasks resemble a bimodal distribution skewing to the left. Looking at the `lat min` and `lat avg` columns of Figure 6.30 we can exclude the latter because we see that `lat avg` data is simply just close to the `lat min` data. So we can conclude that maximum latency appears seldom. A few cases of maximum latency is not a problem if we see the safety module as a soft real-time system. [1]
But we have not concluded that the safety module achieves soft real-time operation yet. That the safety module achieves soft real-time operation follows from the last statistics table of Section 6.4, given in Figure 6.34. To explain, in Section 6.1.3, we elaborated how the Xenomai framework achieves real-time operation. In Section 5.5 we saw that Xenomai userspace programs can achieve soft real-time operation at most. The conclusion that can be drawn from these sections is that soft real-time operation of the safety module is achieved by keeping the real-time threads of the safety module running in primary mode. We said that real-time threads should not leave primary mode, for if they would they would not be real-time threads. We defined real-time threads to be threads that do not leave primary mode, and this conforms to what Xenomai thinks of as a real-time thread. We noted at the end of Section 6.4 that the only time any of the `rt` threads make a mode switch to secondary mode, it is at initialisation of the safety module. After initialisation, none of the `rt` threads will make a mode switch to secondary mode. This can be seen by comparing the table in Figure 6.34 to the table in the preceding figure, Figure 6.33. The difference in time between these figures is ten seconds and in those ten seconds non of the real-time threads have made additional mode switches.[2] Because non of the real-time threads make additional mode switches after initialisation, we can conclude that the safety module achieves soft real-time operation.

7.5. Discussion on the final result

Looking at the running safety module shown in Section 6.4.6 of the Results, we think our aim of designing and implementing a safety module that provides an extra layer of safety with the goal to prevent injury to patients or damage to the environment because of unsafe behavior has been achieved. Aside from some minor delays in turning the matrix red that happen every now and then, the safety module works as expected. Moreover, since we achieved soft real-time operation of the safety module, we think we have also affirmed the feasibility of this setup of the system as a whole.

[1] In Section 2 we explained why we aim for soft real-time system and not a hard real-time system.
[2] To truly confirm no additional mode switches are made by the real-time threads we ran it for a couple of minutes and simulated some unsafe behavior. As expected, the mode switch count of any of the real-time threads did not increment.

8. Recommendation

In this section we present our recommendations for further research on the safety module. Most of the recommendations are refinements of the base system that was laid out in this research.

8.0.1. Testing the safety module on care robot Rose

In this research we did not test the prototype safety module on care robot Rose. The prototype safety module makes use of the sensors of its own hardware platform, the Sense HAT, to retrieve data from. Although ROS is installed on the safety module, functions that are to interact with ROS are implemented as stubs. Therefore, the first and foremost recommendation for further research is to create the interface from the safety module to ROS and to test the safety module on care robot Rose.

8.0.2. Raspberry Pi versions and costs

In Section 6.1 we found that the Raspberry Pi was the best hardware platform to work with, but our research did not go in depth about different versions of the Raspberry Pi. We recommend further research to compare different versions of the Raspberry Pi, like the Raspberry Pi Zero W, the Raspberry Pi 3 (used in this research) and the Raspberry Pi 4, in terms of performance and costs. For instance, it could be that there is significant performance to be gained by use of the more expensive Raspberry Pi 4 over the Raspberry Pi 3, but this may not be worth it in terms of finance. Likewise, the cheaper Raspberry Pi Zero W can be considered, but it may not have the required hardware capabilities.

8.0.3. Conversion to Xenomai kernel space for hard real-time performance

In the literature review of Section 5.5 we found that in order to achieve hard real-time performance, the safety module system should be implemented in Xenomai kernelspace. In this research, soft real-time performance is achieved by use of Xenomai userspace tasks. A reasonable step for follow-up research would be to port the existing Xenomai userspace system of this research to Xenomai kernelspace.

8.0.4. Refinement of safety metrics

In this research, we use a rudimentary form of the safety metrics presented in the literature review of Section 5.3. To increase the efficacy of the safety module, we recommend a follow-up research to refine these safety metrics.

8.0.5. Refinement of timeliness of the system

In this research, the periods of threads and time intervals used in calculations (like the Δt's used in the numerical integration and the numerical differentiation) were determined experimentally and set to be a multiple of the IMU poll rate, but this is not necessarily the most optimal timing. A follow-up research could conduct a study to determine what is the most optimal timing for the system in terms of periods of threads, sample time and time intervals in calculations.

8.0.6. Addition of time verification in the behavioral model

In our research, the behavioral module created with Dezyne does not take into account how much time it takes to execute certain actions. In addition to the recommendation to refine the timeliness of the system elaborated in Section 8.0.5, we recommend to verify the timeliness in the behavioral model.

A. Additional Models

This Appendix provides additional models as mentioned in Section 5.6 of the Results.

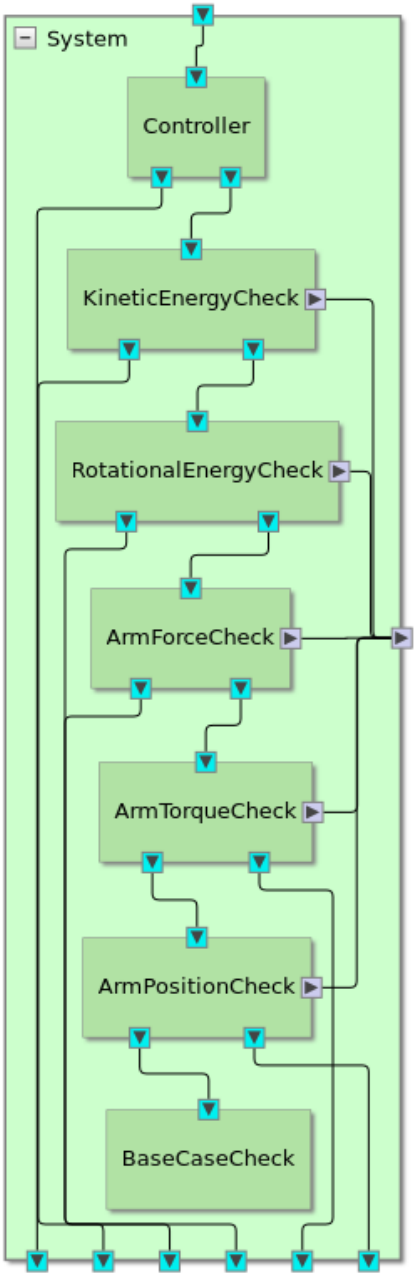


Figure A.1.: Alternative representation of the system view given in Section 6.3.1.

A.1. ILEDControl

Events	Guard	Code	Next
[systemState.Idle]			
initialise_framebuffer		systemState = State.Operating;	systemState.Operating
destruct_framebuffer, reset_led		illegal;	
light_led_red	[&&trueledState.Low]	illegal;	
light_led_red	[&&trueledState.Blue]	illegal;	
light_led_red	[&&trueledState.Red]	illegal;	
light_led_red	[&&falseledState.Low]	ledState = LedState.Red;	systemState.Idle
light_led_red	[&&falseledState.Blue]	ledState = LedState.Red;	systemState.Idle
light_led_red	[&&falseledState.Red]	ledState = LedState.Red;	systemState.Idle
light_led_blue	[&&trueledState.Low]	illegal;	
light_led_blue	[&&trueledState.Blue]	illegal;	
light_led_blue	[&&trueledState.Red]	illegal;	
light_led_blue	[&&falseledState.Low]	ledState = LedState.Blue;	systemState.Idle
light_led_blue	[&&falseledState.Blue]	ledState = LedState.Blue;	systemState.Idle
light_led_blue	[&&falseledState.Red]	illegal;	
[systemState.Operating]			
initialise_framebuffer		illegal;	
destruct_framebuffer		systemState = State.Idle;	systemState.Idle
light_led_red	[&&falseledState.Low]	illegal;	
light_led_red	[&&falseledState.Blue]	illegal;	
light_led_red	[&&falseledState.Red]	illegal;	
light_led_red	[&&trueledState.Low]	ledState = LedState.Red;	systemState.Operating
light_led_red	[&&trueledState.Blue]	ledState = LedState.Red;	systemState.Operating
light_led_red	[&&trueledState.Red]	ledState = LedState.Red;	systemState.Operating
light_led_blue	[&&falseledState.Low]	illegal;	
light_led_blue	[&&falseledState.Blue]	illegal;	
light_led_blue	[&&falseledState.Red]	illegal;	
light_led_blue	[&&trueledState.Low]	ledState = LedState.Blue;	systemState.Operating
light_led_blue	[&&trueledState.Blue]	ledState = LedState.Blue;	systemState.Operating
light_led_blue	[&&trueledState.Red]	illegal;	
reset_led		ledState = LedState.Low;	systemState.Operating

Figure A.2.: State table of `ILEDControl`

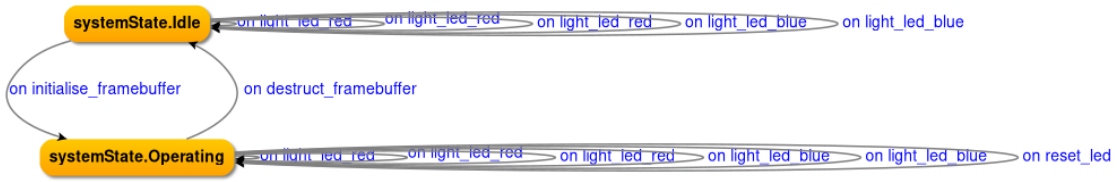


Figure A.3.: State chart of `ILEDControl`

A.2. ISafetyCheck

Figure A.4.: State chart of `ISafetyCheck`

A.3. BaseCaseCheck

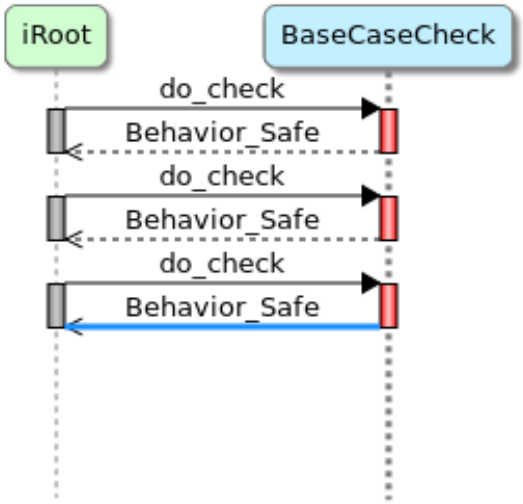


Figure A.5.: Sequence diagram of `BaseCaseCheck`

A.4. RotationalEnergyCheck

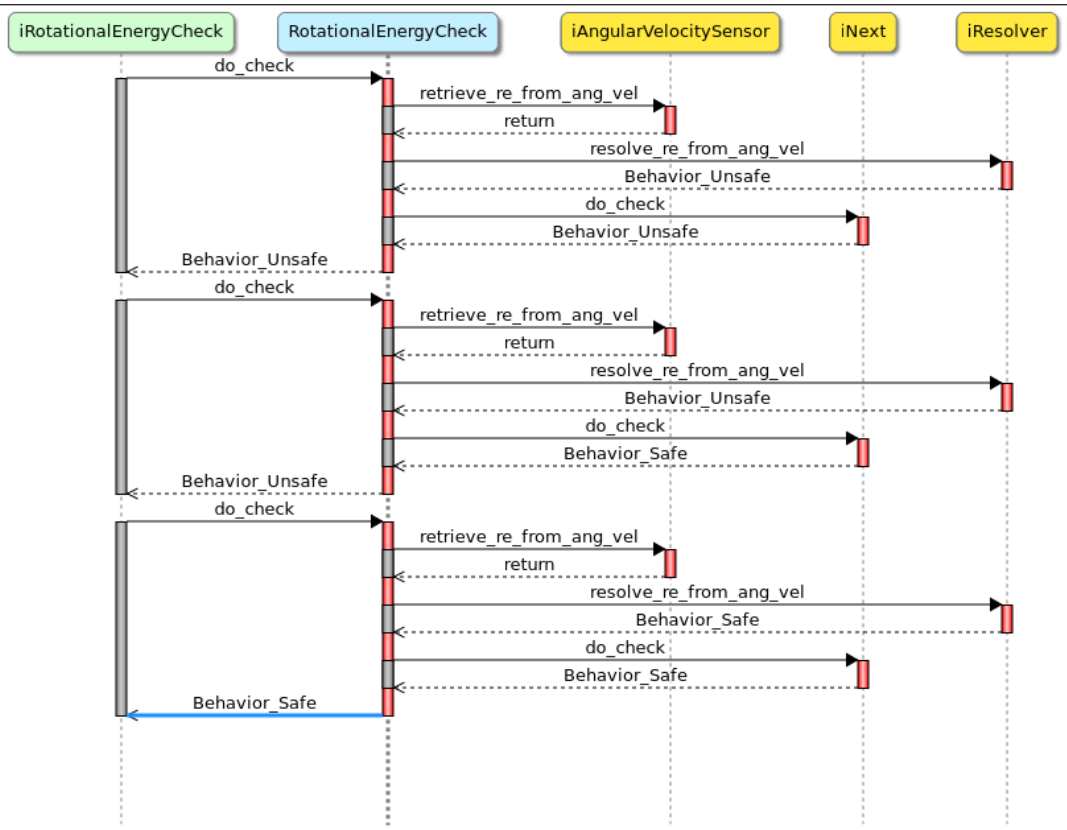


Figure A.6.: Sequence diagram of `RotationalEnergyCheck`

A.5. ArmForceCheck

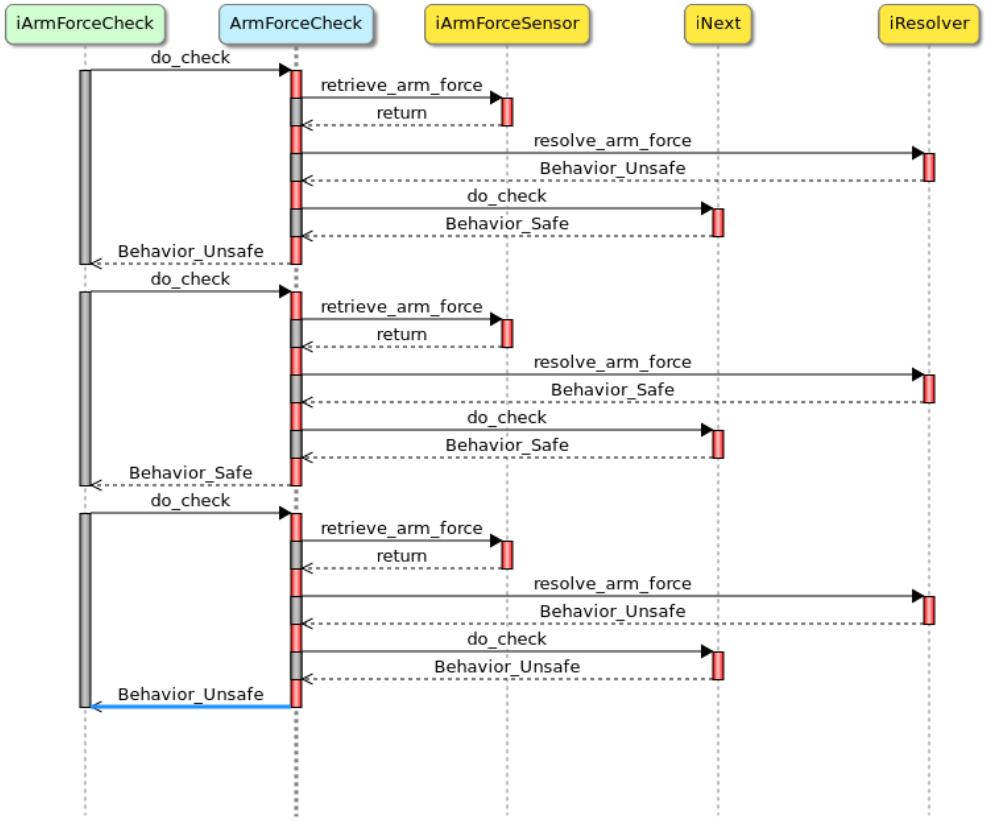


Figure A.7.: Sequence diagram of `ArmForceCheck`

A.6. ArmTorqueCheck

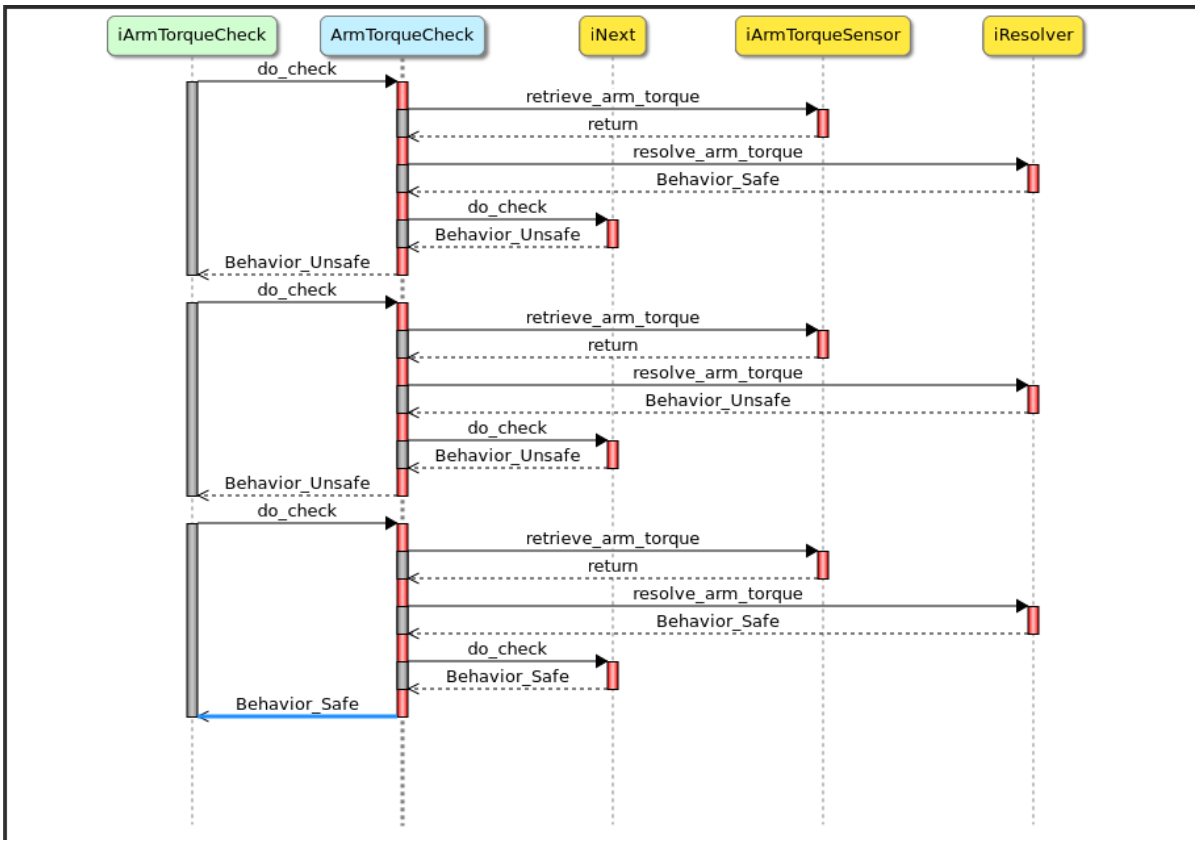


Figure A.8.: Sequence diagram of `ArmTorqueCheck`

A.7. RotationalEnergyCheck

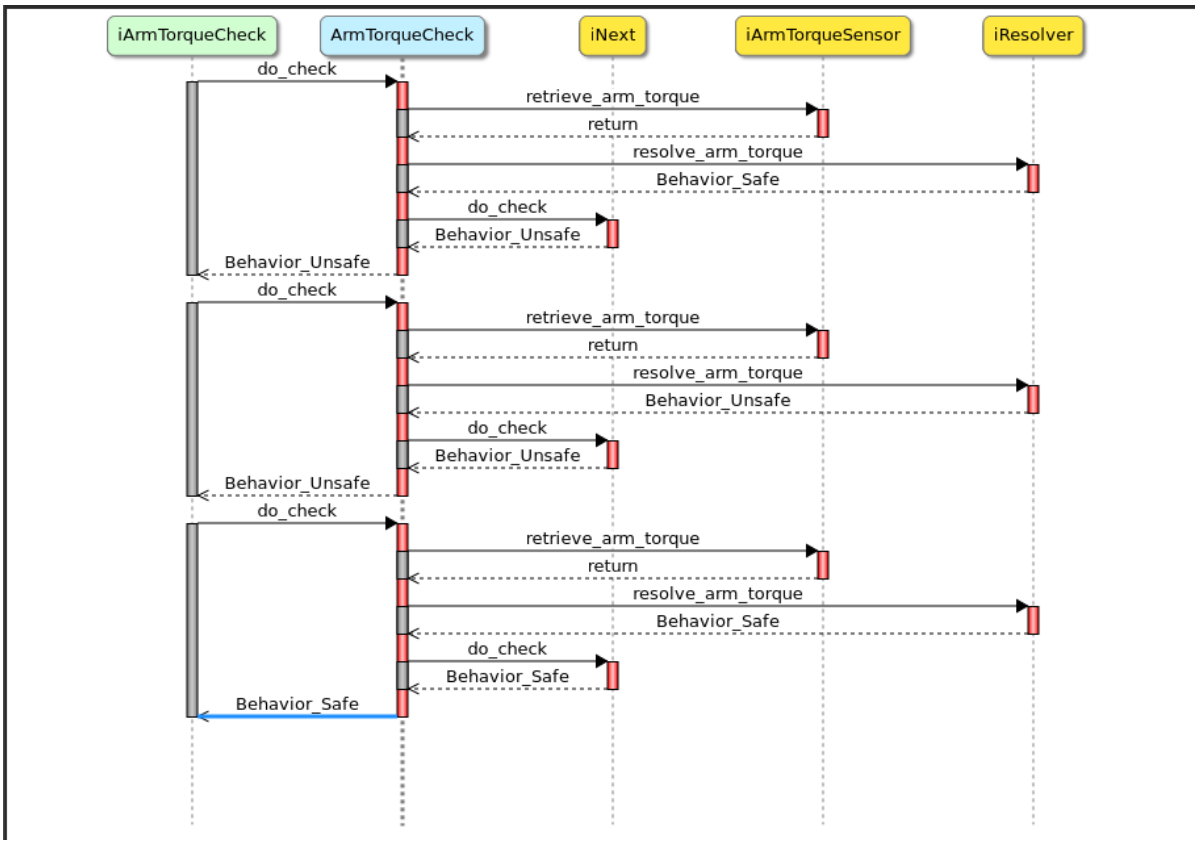


Figure A.9.: Sequence diagram of `ArmTorqueCheck`

A.8. ArmPositionCheck

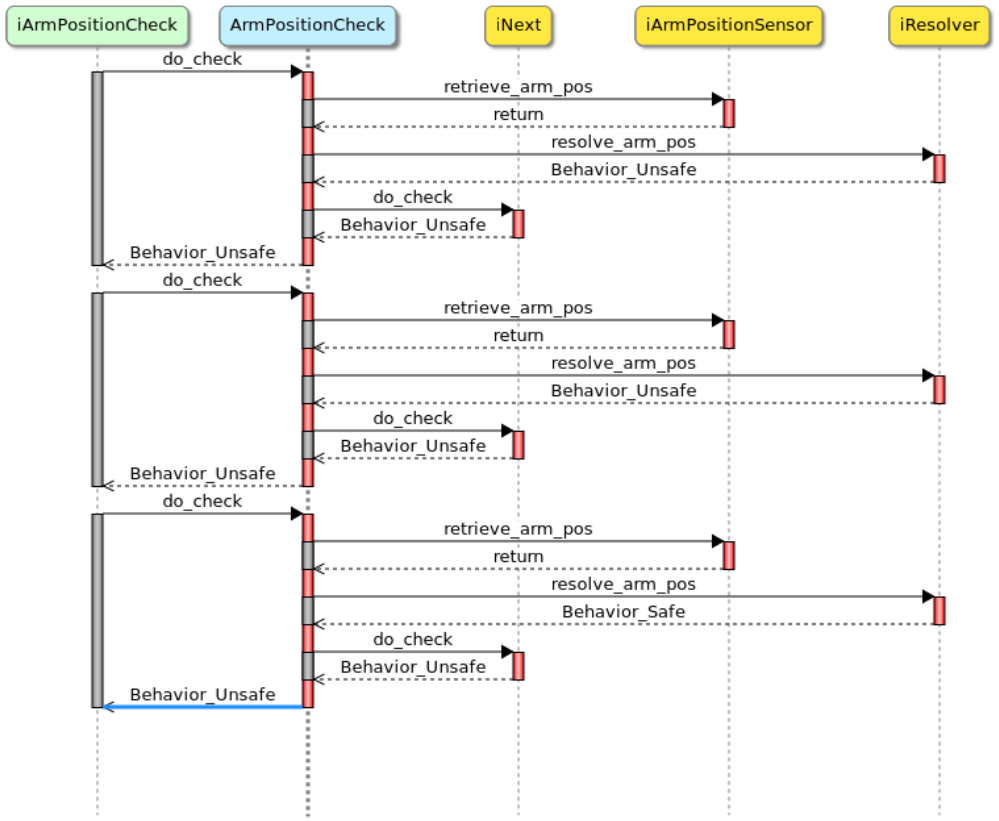


Figure A.10.: Sequence diagram of `ArmPositionCheck`

A.9. IController

States	Guard	Code	Next
initialise			
[state.Idle]		initialise_imu ; initialise_mutexes ; initialise_semaphores ; state = State.Operating;	state.Operating
[state.Operating]		illegal;	
destruct			
[state.Idle]		illegal;	
[state.Operating]		destruct_mutexes ; destruct_semaphores ; state = State.Idle;	state.Idle
reset			
[state.Idle]		illegal;	
[state.Operating]		{}	state.Operating
do_checks			
[state.Idle]		illegal;	
[state.Operating]		reply(UnsafeTriggered.Yes);	state.Operating
[state.Operating]		reply(UnsafeTriggered.No);	state.Operating

Figure A.11.: Event table of `IController`

Events	Guard	Code	Next
[state.Idle]			
initialise		initialise_imu ; initialise_mutexes ; initialise_semaphores ; state = State.Operating;	state.Operating
destruct, reset, do_checks		illegal;	
[state.Operating]			
initialise		illegal;	
destruct		destruct_mutexes ; destruct_semaphores ; state = State.Idle;	state.Idle
reset		{}	state.Operating
do_checks		reply(UnsafeTriggered.Yes);	state.Operating
do_checks		reply(UnsafeTriggered.No);	state.Operating

Figure A.12.: State table of `IController`

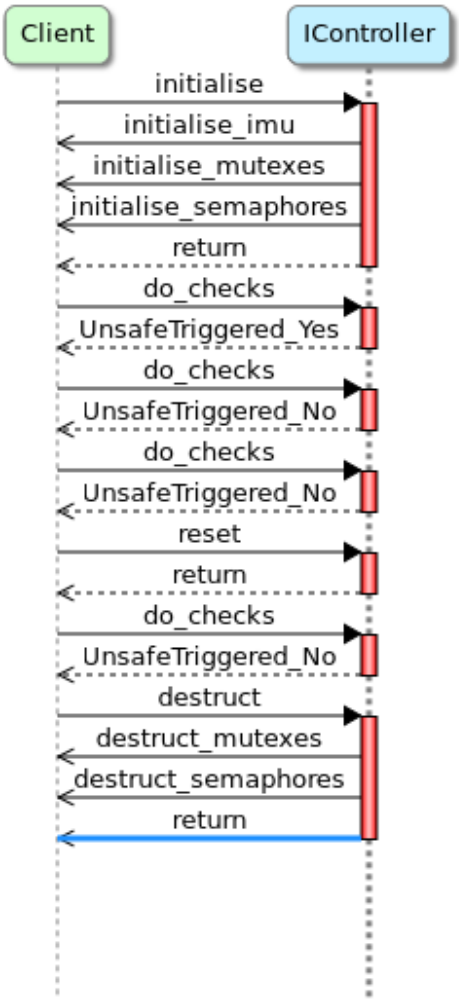


Figure A.13.: Sequence diagram of `IController`

A.10. Controller

Events	Guard	Code	Next
[systemState.Idle]			
initialise		iLEDControl. initialise_framebuffer (); iController. initialise_imu (); iController. initialise_mutexes (); iController. initialise_semaphores (); systemState = State.Operating;	systemState.Operating
[systemState.Operating]			
destruct		iLEDControl. destruct_framebuffer (); iController. destruct_mutexes (); iController. destruct_semaphores (); systemState = State.Idle;	systemState.Idle

do_checks		{
		Behavior safetyState = iNext. do_check ();
		if (safetyState == Behavior.Unsafe) {
		iLEDControl. light_led_red (red);
		unsafe_acknowledged = false;
		}if (!(unsafe_acknowledged)) {
		reply(UnsafeTriggered.Yes);
		}}
			systemState.Operating

reset		iLEDControl. reset_led (); unsafe_acknowledged = true;	systemState.Operating

Figure A.14.: State table of `Controller`

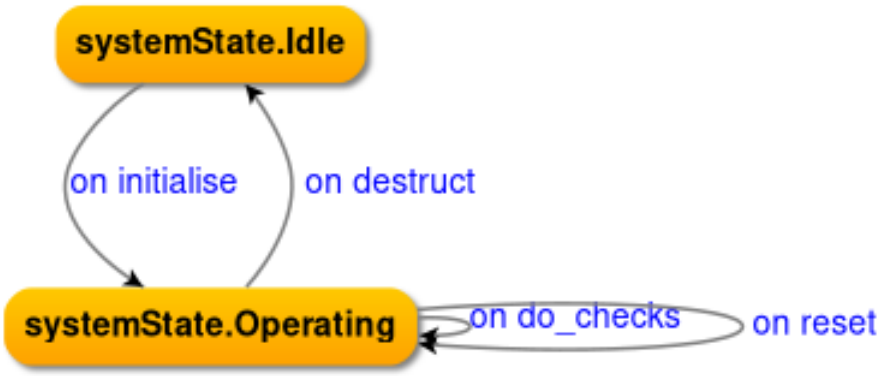


Figure A.15.: State chart of `Controller`

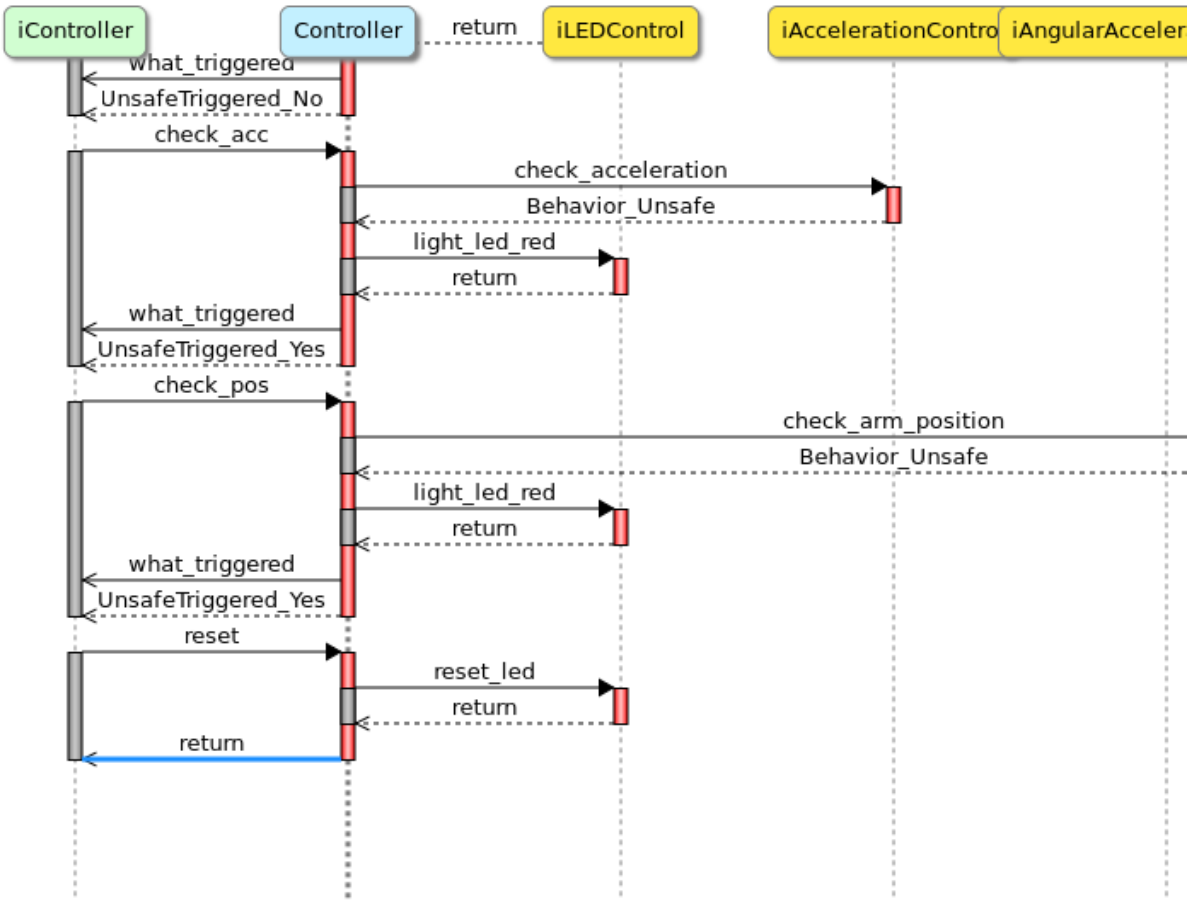


Figure A.16.: Old sequence diagram of `Controller` showing multiple checks being executed in sequence

B. Source code

We have included important code snippets here that are relevant to Section 6.4 of the Results. The code listed here may be outdated. The code on the repository is guaranteed to be up-to-date.

B.1. Main and roll functions

This section contains the `main` and `roll` functions. `main` calls `roll` immediately after initialation of the safety module. The `roll` function does more initialization, like binding the Dezyne resolve, retrieve and light_led events and starting the threads. The main thread runs indefinitely, waiting for input to acknowledge unsafe behavior and reset the LED matrix. Created threads run besides the main thread to continuously retrieve and sample sensor data.

```
auto main() -> int {
    initialise();
    int r = roll();
    if (r != OK) return EXIT_FAILURE;
    destruct();
}

}
ErrorCode_t roll() {
    // Initialise dezyne locator and runtime.
    IResolver iResolver({});

    // Bind resolvers
    iResolver.in.resolve_ke_from_acc = []() -> Behavior::type {
        int r;
        Behavior::type type;
        double ke;

        r = safe_call([&ke]() { ke = kinetic_energy; }, &mutex["ke"]);
        if (r != OK) exit(EXIT_FAILURE);

        if (ke > MAX_KE)
```

```
            type = Behavior::type::Unsafe;
        else
            type = Behavior::type::Safe;
        return type;
    };

    iResolver.in.resolve_re_from_ang_vel = []() -> Behavior::type {
        int r;
        Behavior::type type;
        double re;

        r = safe_call([&re]() { re = rotational_energy; }, &mutex["re"]);
        if (r != OK) exit(EXIT_FAILURE);

        if (re > MAX_RE)
            type =  Behavior::type::Unsafe;
        else
            type =  Behavior::type::Safe;
        return type;
    };

    iResolver.in.resolve_arm_force = []() -> Behavior::type {
        int r;
        Behavior::type type;
        double str;
        bool has_payload;

        r = safe_call([&]() { str = arm_force; has_payload =
                arm_has_payload(); }, &mutex["arm_force"]);
        if (r != OK) exit(EXIT_FAILURE);

        if ( has_payload && str > MAX_FORCE_PAYLOAD)
            type = Behavior::type::Unsafe;
        else if (str > MAX_FORCE)
            type = Behavior::type::Unsafe;
        else
            type = Behavior::type::Safe;
        return type;
    };

    iResolver.in.resolve_arm_torque = []() -> Behavior::type {
        int r;
        Behavior::type type;
        double str;
```

```
67          bool has_payload;
68
69          r = safe_call([&]() { str = arm_torque; has_payload =
70                  arm_has_payload(); }, &mutex["arm_torque"]);
71          if (r != OK) exit(EXIT_FAILURE);
72
73          if ( has_payload && str > MAX_TORQUE_PAYLOAD)
74              type = Behavior::type::Unsafe;
75          else if (str > MAX_torque)
76              type = Behavior::type::Unsafe;
77          else
78              type = Behavior::type::Safe;
79          return type;
80      };
81
82      iResolver.in.resolve_arm_pos = []() -> Behavior::type {
83          int r;
84          Behavior::type type;
85          bool folded;
86          bool is_moving;
87
88          r = safe_call([&]() { folded = arm_is_folded(); is_moving =
89                  robot_is_moving(); }, &mutex["arm_pos"]);
90          if (r != OK) exit(EXIT_FAILURE);
91
92          if (is_moving && !folded)
93              type = Behavior::type::Unsafe;
94          else
95              type = Behavior::type::Safe;
96          return type;
97      };
98
99      dzn::locator locator;
100     dzn::runtime runtime;
101     locator.set(runtime);
102     locator.set(iResolver);
103     auto output = std::ofstream("dzn_output.log");
104     locator.set(static_cast<std::ostream&>(output));
105
106     System s(locator);
107     s.dzn_meta.name = "System";
108
109     /*
110      * Bind dezyne functions with C++ functions.
```

```
111      */
112     /* s.iController.out.what_triggered = what_triggered; */
113     s.iLEDControl.in.initialise_framebuffer = initialise_framebuffer;
114     s.iLEDControl.in.destruct_framebuffer = destruct_framebuffer;
115     s.iLEDControl.in.light_led_red = dzn_light_led;
116     s.iLEDControl.in.light_led_blue = dzn_light_led;
117     s.iLEDControl.in.reset_led = reset_led;
118     s.iController.out.initialise_imu = initialise_imu;
119     s.iController.out.initialise_mutexes = initialise_mutexes;
120     s.iController.out.initialise_semaphores = initialise_semaphores;
121     s.iController.out.destruct_mutexes = destruct_mutexes;
122     s.iController.out.destruct_semaphores = destruct_semaphores;
123     s.iAccelerationSensor.in.retrieve_ke_from_acc = dzn_retrieve_ke_from_acc;
124     s.iAngularVelocitySensor.in.retrieve_re_from_ang_vel =
125         dzn_retrieve_re_from_ang_vel;
126     s.iArmPositionSensor.in.retrieve_arm_pos = dzn_retrieve_arm_pos;
127     s.iArmForceSensor.in.retrieve_arm_force = dzn_retrieve_arm_force;
128     s.iArmTorqueSensor.in.retrieve_arm_torque = dzn_retrieve_arm_torque;
129
130
131     // Check bindings
132     s.check_bindings();
133
134     /*
135      * Initialise system
136      */
137     s.iController.in.initialise();
138
139     /*** Threads related ***/
140     struct sched_param rtparam = { .sched_priority = 42 };
141     pthread_attr_t rtattr, nrtattr;
142     sigset_t set;
143     int sig;
144
145     // real-time thread that starts safety checks periodically.
146     static pthread_t th_rt_checks;
147     // real-time thread for lighting the LED matrix.
148     static pthread_t th_rt_light_led;
149     // real-time threads for retrieving data.
150     static pthread_t th_rt_ret_acc, th_rt_ret_ang_disp;
151     static pthread_t th_rt_ret_arm_force, th_rt_ret_arm_torque, th_rt_ret_arm_pos;
152     // real-time threads for sampling sensor data.
153     static pthread_t th_rt_sample_acc, th_rt_sample_ang_disp;
154
```

```
155     // non-real-time thread for lighting the LED matrix.
156     static pthread_t th_nrt_light_led;
157     // non-real-time thread for retrieving sensor data from the IMU and ROS.
158     static pthread_t th_nrt_ret_imu, th_nrt_ret_arm_force;
159     static pthread_t th_nrt_ret_arm_torque, th_nrt_ret_arm_pos;
160
161     // Information about periodic  threads.
162     static struct th_info th_info;
163
164     // Arguments to threads.
165     struct threadargs threadargs;
166
167     sigemptyset(&set);
168     sigaddset(&set, SIGINT);
169     sigaddset(&set, SIGTERM);
170     sigaddset(&set, SIGHUP);
171
172     pthread_attr_init(&nrtattr);
173     pthread_attr_setdetachstate(&nrtattr, PTHREAD_CREATE_JOINABLE);
174     pthread_attr_setinheritsched(&nrtattr, PTHREAD_EXPLICIT_SCHED);
175     pthread_attr_setschedpolicy(&nrtattr, SCHED_OTHER);
176
177     pthread_attr_init(&rtattr);
178     pthread_attr_setdetachstate(&rtattr, PTHREAD_CREATE_JOINABLE);
179     pthread_attr_setinheritsched(&rtattr, PTHREAD_EXPLICIT_SCHED);
180     pthread_attr_setschedpolicy(&rtattr, SCHED_FIFO);
181     pthread_attr_setschedparam(&rtattr, &rtparam);
182
183     /*** Start threads ***/
184     // Start thread rt_light_led
185     errno = pthread_create(&th_rt_light_led, &rtattr, &rt_light_led, NULL);
186     if (errno)
187         fail("pthread_create");
188
189     // Start thread rt_retrieve_acceleration
190     errno = pthread_create(&th_rt_ret_acc, &rtattr, &rt_retrieve_acceleration,
191             &threadargs);
192     if (errno)
193         fail("pthread_create");
194
195     // Start thread rt_retrieve_angular_displacement
196     errno = pthread_create(&th_rt_ret_ang_disp, &rtattr,
197             &rt_retrieve_angular_displacement, &threadargs);
198     if (errno)
```

```
199         fail("pthread_create");
200 
201     // Start thread rt_retrieve_arm_force
202     errno = pthread_create(&th_rt_ret_arm_force, &rtattr,
203             &rt_retrieve_arm_force, &threadargs);
204     if (errno)
205         fail("pthread_create");
206 
207     // Start thread rt_retrieve_arm_torque
208     errno = pthread_create(&th_rt_ret_arm_torque, &rtattr,
209             &rt_retrieve_arm_torque, &threadargs);
210     if (errno)
211         fail("pthread_create");
212 
213     // Start thread rt_retrieve_arm_position
214     errno = pthread_create(&th_rt_ret_arm_pos, &rtattr,
215             &rt_retrieve_arm_position, &threadargs);
216     if (errno)
217         fail("pthread_create");
218 
219     // Start thread rt_sample_acceleration
220     errno = pthread_create(&th_rt_sample_acc, &rtattr, &rt_sample_acceleration,
221             &threadargs);
222     if (errno)
223         fail("pthread_create");
224 
225     // Start thread rt_sample_angular_velocity
226     errno = pthread_create(&th_rt_sample_ang_disp, &rtattr,
227             &rt_sample_angular_velocity, &threadargs);
228     if (errno)
229         fail("pthread_create");
230 
231     // Start thread nrt_light_led
232     errno = pthread_create(&th_nrt_light_led, &nrtattr, &nrt_light_led, (void*)
233             &threadargs);
234     if (errno)
235         fail("pthread_create");
236 
237     // Start thread nrt_retrieve_imu
238     errno = pthread_create(&th_nrt_ret_imu, &nrtattr,
239             &nrt_retrieve_imu, NULL);
240     if (errno)
241         fail("pthread_create");
242 
```

```
243     // Start thread nrt_retrieve_arm_force
244     errno = pthread_create(&th_nrt_ret_arm_force, &nrtattr,
245             &nrt_retrieve_arm_force, NULL);
246     if (errno)
247         fail("pthread_create");
248 
249     // Start thread nrt_retrieve_arm_torque
250     errno = pthread_create(&th_nrt_ret_arm_torque, &nrtattr,
251             &nrt_retrieve_arm_torque, NULL);
252     if (errno)
253         fail("pthread_create");
254 
255     // Start thread nrt_retrieve_arm_position
256     errno = pthread_create(&th_nrt_ret_arm_pos, &nrtattr,
257             &nrt_retrieve_arm_position, NULL);
258     if (errno)
259         fail("pthread_create");
260 
261 
262 
263     printf("Started running indefinitely.\n");
264     std::string input;
265 #if PERIODIC_CHECKS
266     printf("press: q to quit, r to reset\n");
267 
268     th_info.body = rt_checks;
269     /* th_info.period = 1E6*4;    // 4s */
270     /* th_info.period = 1E6/4;       // 250ms */
271     th_info.period = imu_poll_interval;      // poll interval based period.
272     th_info.s = &s;
273     // Start periodic real-time checks.
274     errno = pthread_create(&th_rt_checks, &rtattr, &rt_periodic_thread_body,
275             &th_info);
276     if (errno)
277         fail("pthread_create");
278 
279     // The safety module runs so long we are in this loop. The LED light can be
280     // reset by inputting 'r'.  We quit the loop correctly by inputting 'q'.
281     while (1) {
282         std::cin >> input;
283         /* input = "a"; */
284         if (input == "q") {
285             break;
286         } else if (input == "r") {
```

```
287             s.iController.in.reset();
288         } else if (input == "i") {
289             // Purposely here to show illegal exception handler.
290             s.iController.in.initialise();
291         } else {
292             printf("Did not understand input.\n");
293         }
294     }
295 #else
296     while (1) {
297         printf("press: q to quit, d to execute all checks, r to reset\n");
298         printf("a to check acc, aa to check ang acc, s to check str, p to check pos\n
299 
300         // Notify nrt_retrieve_acceleration that it can retrieve acceleration.
301         /* sem_post(&semaphore["retrieve_acc"]]); */
302         // Notify rt_sample_acceleration that it can go sample acceleration.
303         /* sem_post(&sem_sample_acc); */
304 
305         std::cin >> input;
306         /* input = "a"; */
307         if (input == "q") {
308             break;
309         } else if (input == "d") {
310             s.iController.in.do_checks();
311         } else if (input == "r") {
312             s.iController.in.reset();
313         } else if (input == "i") {
314             // Purposely here to show illegal exception handler.
315             s.iController.in.initialise();
316         } else {
317             printf("Did not understand input.\n");
318         }
319     }
320 #endif
321     printf("Stopping\n");
322 
323     // Destruct system
324     s.iController.in.destruct();
325 
326     // Kill threads
327     pthread_cancel(th_nrt_light_led);
328     pthread_cancel(th_nrt_ret_arm_force);
329     pthread_cancel(th_nrt_ret_arm_torque);
330     pthread_cancel(th_nrt_ret_arm_pos);
```

```
331     pthread_cancel(th_nrt_ret_imu);
332     pthread_cancel(th_rt_light_led);
333     pthread_cancel(th_rt_ret_acc);
334     pthread_cancel(th_rt_ret_ang_disp);
335     pthread_cancel(th_rt_ret_arm_force);
336     pthread_cancel(th_rt_ret_arm_torque);
337     pthread_cancel(th_rt_ret_arm_pos);
338     pthread_cancel(th_rt_sample_acc);
339     pthread_cancel(th_rt_sample_ang_disp);
340 
341 #if PERIODIC_CHECKS
342     pthread_cancel(th_rt_checks);
343 #endif
344 
345     pthread_join(th_nrt_light_led, NULL);
346     pthread_join(th_nrt_ret_arm_pos, NULL);
347     pthread_join(th_nrt_ret_arm_force, NULL);
348     pthread_join(th_nrt_ret_arm_torque, NULL);
349     pthread_join(th_nrt_ret_imu, NULL);
350     pthread_join(th_rt_light_led, NULL);
351     pthread_join(th_rt_ret_acc, NULL);
352     pthread_join(th_rt_ret_ang_disp, NULL);
353     pthread_join(th_rt_ret_arm_force, NULL);
354     pthread_join(th_rt_ret_arm_torque, NULL);
355     pthread_join(th_rt_ret_arm_pos, NULL);
356     pthread_join(th_rt_sample_acc, NULL);
357     pthread_join(th_rt_sample_ang_disp, NULL);
358 
359 #if PERIODIC_CHECKS
360     pthread_join(th_rt_checks, NULL);
361 #endif
362 
363     return OK;
364 }
```

B.2. Periodic thread functions

This code snippet shows the rt_checks and rt_periodic_thread_body functions. These are used to periodically execute the checks. rt_checks shows us calling the the Dezyne do_checks() function. The thread that starts `rt_checks` itself was shown in Section B.1 of the Appendix.

```c
static void rt_checks(void* arg) {
    struct threadargs *args = (struct threadargs *)arg;
    /*
     * Since do_checks is implemented in Dezyne, if we want to disable one
     * check we have to remove the check from the linked list in System.dzn.
     * This is a consequence of making the system more generic.
     */
    (args->s)->iController.in.do_checks();
}

static void *rt_periodic_thread_body(void *arg) {
    struct periodic_task *ptask;
    struct th_info* the_thread = (struct th_info*) arg;
    struct threadargs threadargs;

    // Copy over dezyne system pointer
    threadargs.s = the_thread->s;

    ptask = start_periodic_timer(0, the_thread->period);
    if (ptask == NULL) {
        printf("Start Periodic Timer");

        return NULL;
    }

    while(1) {
        wait_next_activation(ptask);
        the_thread->body((void*) &threadargs);
    }

    return NULL;
}

struct periodic_task *start_periodic_timer(unsigned long long offset_in_us,
        int period) {
    struct periodic_task *ptask;

    ptask = (struct periodic_task*) malloc(sizeof(struct periodic_task));
    if (ptask == NULL) {
        return NULL;
    }

```

```
43      // Current time is added first, we let the thread wait for its next
44      // activation using absolute time.
45      clock_gettime(CLOCK_REALTIME, &ptask->ts);
46      timespec_add_us(&ptask->ts, offset_in_us);
47      ptask->period = period;
48      return ptask;
49  }
50
51  void wait_next_activation(struct periodic_task *ptask) {
52      // Suspend the thread until the time value specified by &t->ts has elapsed.
53      // Note the use of absolute time.
54      clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &ptask->ts, NULL);
55      // Add another period to the time specification.
56      timespec_add_us(&ptask->ts, ptask->period);
57  }
```

B.3. Lighting LED functions

This snippet shows all function called in a complete line of communication from Dezyne to the driver as was shown in Figure 6.27. `dzn_light_led` is executed from the main thread, `rt_light_led` and `nrt_light_led` are executed from their own threads. The `reset_led` functions are identical in structure and differ only in that they have the task of resetting the LED matrix instead of lighting the LED matrix red or blue.

```
1  void dzn_light_led(char* color) {
2      // Set color
3      int r = safe_call([=]() { strcpy(::color, color); }, &mutex["color"]);
4
5      if (r != OK) exit(EXIT_FAILURE);
6      // Let rt_light_led know that it can send a color to nrt_light_led.
7      sem_post(&semaphore["led"]);
8  }
9
10
11 static void* rt_light_led(void* arg) {
12     int n = 0;
13     int len;
14     int ret;
15     int r;
16     int s;
17     char buf[128];
```

```
18      struct sockaddr_ipc saddr;
19      size_t poolsz;
20
21      struct threadargs *args = (struct threadargs *) arg;
22      // Initialise XDDP
23      /*
24       * Get a datagram socket to bind to the RT endpoint. Each
25       * endpoint is represented by a port number within the XDDP
26       * protocol namespace.
27       */
28      s = socket(AF_RTIPC, SOCK_DGRAM, IPCPROTO_XDDP);
29      if (s < 0) {
30          perror("socket");
31          exit(EXIT_FAILURE);
32      }
33
34      /*
35       * Set a local 16k pool for the RT endpoint. Memory needed to
36       * convey datagrams will be pulled from this pool, instead of
37       * Xenomai's system pool.
38       */
39      poolsz = 16384; /* bytes */
40      ret = setsockopt(s, SOL_XDDP, XDDP_POOLSZ, &poolsz, sizeof(poolsz));
41      if (ret)
42          fail("setsockopt");
43      /*
44       * Bind the socket to the port, to setup a proxy to channel
45       * traffic to/from the Linux domain.
46       *
47       * saddr.sipc_port specifies the port number to use.
48       */
49      memset(&saddr, 0, sizeof(saddr));
50      saddr.sipc_family = AF_RTIPC;
51      saddr.sipc_port = XDDP_PORT_LIGHT_LED;
52      ret = bind(s, (struct sockaddr *)&saddr, sizeof(saddr));
53      if (ret)
54          fail("bind");
55
56      /*
57       * Send a datagram to the NRT endpoint via the proxy.
58       * We may pass a NULL destination address, since a
59       * bound socket is assigned a default destination
60       * address matching the binding address (unless
61       * connect(2) was issued before bind(2), in which case
```

```
62       * the former would prevail).
63       */
64      while (1) {
65          // Wait for an up to send a color to the LED matrix.
66          sem_wait(&semaphore["led"]);
67          char msg[SIZE];
68
69          // Get color
70          r = safe_call([&msg]() { strcpy(msg, color); }, &mutex["color"]);
71          if (r != OK) exit(EXIT_FAILURE);
72
73          len = strlen(msg);
74          ret = sendto(s, msg, len, 0, NULL, 0);
75          if (ret != len)
76              fail("sendto");
77          /* printf("%s: sent %d bytes, \"%.*s\"\n", __FUNCTION__, ret, ret, msg); */
78      }
79      return NULL;
80  }
81
82
83  static void* nrt_light_led(void *arg) {
84      int r;
85      struct threadargs *args = (struct threadargs *)arg;
86
87      char buf[128], *devname;
88      int fd, ret;
89      if (asprintf(&devname, "/dev/rtp%d", XDDP_PORT_LIGHT_LED) < 0)
90          fail("asprintf");
91      fd = open(devname, O_RDWR);
92      free(devname);
93      if (fd < 0)
94          fail("open");
95
96      while (1) {
97          /* Get the next message from rt_light_led. */
98          /* read what to color the led buffer in */
99          ret = read(fd, buf, sizeof(buf));
100         if (ret <= 0)
101             fail("read");
102
103         // Convert hex string to int.
104         int color = (int)strtol(buf, NULL, 16);
105
```

```
106         r = safe_call([=]() { light_led(color); }, &mutex["color"]);
107         if (r != OK) exit(EXIT_FAILURE);
108     }
109 
110     return NULL;
111 }
```

B.4. Retrieving angular displacement functions

This code snippet shows all functions called in a complete line of communication from Dezyne to the driver as was shown in Figure 6.28. `dzn_retrieve_angular_displacement` is called from the main thread, `rt_sample_angular_displacement`, `rt_retrieve_angular_displacment` and `nrt_retrieve_imu` run in their own threads. The other function structures for retrieval of data are identical except that they retrieve (and sample) a different quanity.

```
1  void dzn_retrieve_angular_displacement() {
2      // Currently implemented in sample_angular_displacement itself.
3  }
4  
5  
6  static void* rt_retrieve_angular_displacement(void* arg) {
7      int n = 0;
8      int len;
9      int ret;
10     int r;
11     int s;
12     char buf[BUFSIZE];
13     struct sockaddr_ipc saddr;
14     size_t poolsz;
15 
16     struct threadargs *args = (struct threadargs *)arg;
17 
18     // Initialise XDDP
19     /*
20      * Get a datagram socket to bind to the RT endpoint. Each
21      * endpoint is represented by a port number within the XDDP
22      * protocol namespace.
23      */
24     s = socket(AF_RTIPC, SOCK_DGRAM, IPCPROTO_XDDP);
25     if (s < 0) {
```

```
26          perror("socket");
27          exit(EXIT_FAILURE);
28      }
29
30      /*
31       * Set a local 16k pool for the RT endpoint. Memory needed to
32       * convey datagrams will be pulled from this pool, instead of
33       * Xenomai's system pool.
34       */
35      poolsz = 16384; /* bytes */
36      ret = setsockopt(s, SOL_XDDP, XDDP_POOLSZ, &poolsz, sizeof(poolsz));
37      if (ret)
38          fail("setsockopt");
39      /*
40       * Bind the socket to the port, to setup a proxy to channel
41       * traffic to/from the Linux domain.
42       *
43       * saddr.sipc_port specifies the port number to use.
44       */
45      memset(&saddr, 0, sizeof(saddr));
46      saddr.sipc_family = AF_RTIPC;
47      saddr.sipc_port = XDDP_PORT_RET_ANG_DISP;
48      ret = bind(s, (struct sockaddr *)&saddr, sizeof(saddr));
49      if (ret)
50          fail("bind");
51
52      /*
53       * Retrieve a datagrams from the NRT endpoint via the proxy.
54       */
55      while (1) {
56          // Read packets echoed by the non-real-time thread.
57          /*
58           * This call blocks if there is no data to receive. This however is not
59           * a problem as sample angular velocity is not depended on this thread
60           * blocking or not. That is, sample angular velocity can do its work
61           * independent of this thread.
62           */
63          ret = recvfrom(s, buf, sizeof(buf), 0, NULL, 0);
64          if (ret <= 0)
65              fail("recvfrom");
66          /* printf("   => \"%.*s\" received by peer\n", ret, buf); */
67          n = (n + 1) % (sizeof(buf) / sizeof(buf[0]));
68
69          // Set angular displacement
```

```
70          r = safe_call([&buf]() { set_angular_displacement(atof(buf)); },
71                  &mutex["ang_disp"]);
72          if (r != OK) exit(EXIT_FAILURE);
73      }
74      return NULL;
75  }
76
77
78  static void* nrt_retrieve_imu(void *arg) {
79      struct threadargs *args = (struct threadargs *)arg;
80
81      char *devname;
82      int fd_acc, fd_ang_disp, ret;
83
84      // file descriptor for acceleration.
85      if (asprintf(&devname, "/dev/rtp%d", XDDP_PORT_RET_ACC) < 0)
86          fail("asprintf");
87      fd_acc = open(devname, O_RDWR);
88      free(devname);
89      if (fd_acc < 0)
90          fail("open");
91
92      // file descriptor for angular velocity.
93      if (asprintf(&devname, "/dev/rtp%d", XDDP_PORT_RET_ANG_DISP) < 0)
94          fail("asprintf");
95      fd_ang_disp = open(devname, O_RDWR);
96      free(devname);
97      if (fd_ang_disp < 0)
98          fail("open");
99
100     while (1) {
101         double acc_largest, accx, accy, accz;
102         double ang_disp_largest, ang_dispx, ang_dispy, ang_dispz;
103         // Wait for an up before retrieving acceleration from the sense hat
104         // driver.
105         /* sem_wait(&semaphore["retrieve_acc"]); */
106
107         // Poll at the rate recommended by the IMU.
108         usleep(imu_poll_interval);
109
110         while (imu->IMURead()) {
111             RTIMU_DATA imuData = imu->getIMUData();
112             accx = gforce_to_si(imuData.accel.x());
113             accy = gforce_to_si(imuData.accel.y());
```

```
114             accz = gforce_to_si(imuData.accel.z()-1); // -1 subtracts gravity.
115             // angular velocity is in radians.
116             ang_dispx = imuData.fusionPose.x();
117             ang_dispy = imuData.fusionPose.y();
118             ang_dispz = imuData.fusionPose.z();
119         }
120 
121         /*
122          * For both acceleration ang angular velocity, determine which axis had
123          * the largest increase. The largest increase will be used to
124          * respectively calculate kinetic energy and rotational energy.
125          */
126         accx = fabs(accx);
127         accy = fabs(accy);
128         accz = fabs(accz);
129         ang_dispx = fabs(ang_dispx);
130         ang_dispy = fabs(ang_dispy);
131         ang_dispz = fabs(ang_dispz);
132 
133         if (accx > accy && accx > accz)
134             acc_largest = accx;
135         else if (accy > accx && accy > accz)
136             acc_largest = accy;
137         else
138             acc_largest = accz;
139 
140         if (ang_dispx > ang_dispy && ang_dispx > ang_dispz)
141             ang_disp_largest = ang_dispx;
142         else if (ang_dispy > ang_dispx && ang_dispy > ang_dispz)
143             ang_disp_largest = ang_dispy;
144         else
145             ang_disp_largest = ang_dispz;
146 
147         // Write retrieved acceleration to rt_retrieve_acceleration.
148         write_to_fd(fd_acc, acc_largest);
149         // Write retrieved angular velocity to rt_retrieve_angular_displacement.
150         write_to_fd(fd_ang_disp, ang_disp_largest);
151     }
152 
153     return NULL;
154 }
```

Glossary

4G 4G is the fourth generation of broadband cellular network technology, succeeding 3G.. 29, 30

microcontroller A microcontroller (MCU for microcontroller unit) is a small computer on a single metal-oxide-semiconductor (MOS) integrated circuit chip. 29, 30

Raspberry Pi The raspberry pi is a low cost high performance computer series created by the Rasberry Pi Foundation.. ii, iv, vi, 13, 14, 28–33, 36, 42, 76, 79, 116

WiFi A local area network that uses high frequency radio signals to transmit and receive data over distances of a few hundred feet; uses ethernet protocol. 29, 30, 76

Acronyms

MCU microcontroller. 13, 30

ROS Robot Operating System. 3, 4, 6, 7, 11, 13–15, 30, 33, 38, 39, 64

Bibliography

ahendrix. (2014). *Why is ros not real time?* Retrieved February 4, 2020, from https://answers.ros.org/question/134551/why-is-ros-not-real-time/

Arduino uno rev 3. (2020). Retrieved February 10, 2020, from https://store.arduino.cc/arduino-uno-rev3

Arte arduino real-time extension. (2018). Retrieved February 10, 2020, from http://arte.retis.santannapisa.it/index.html

Asami, K., Hagiwara, H., & Komori, M. (2011). Visual navigation system with real-time image processing for patrol service robot, In *2011ieee 10th international conference on trust, security and privacy in computing and communications*, IEEE.

Birk, A., & Carpin, S. (2006). Rescue robotics - a crucial milestone on the road to autonomous systems. *Advanced Robotics*, *20*(5), 595–605. http://www.tandfonline.com/doi/abs/10.1163/156855306776985577

Bouchier, P. (2013). Embedded ros [ros topics]. *IEEE Robotics Automation Magazine*, *20*(2), 17–19.

Brown, J. H., & Martin, B. (2018). *How fast is fast enough? choosing between xenomai and linux for real-time applications*. Retrieved May 7, 2020, from https://pdfs.semanticscholar.org/9eb5/1dbe38fb23034e80b8664d8281996d2a5ef6.pdf

Budak, A. (2020). *Linux scheduling*. Retrieved February 10, 2020, from https://www.researchgate.net/publication/339079442_Linux_Scheduling

Buttazo, G. C. (2004). *Hard real-time computing systems*.

CHRobotics. (n.d.). *Using accelerometers to estimate position and velocity*. Retrieved May 22, 2020, from http://www.chrobotics.com/library/accel-position-velocity

Community, R. (2016). *Rtwiki*. Retrieved May 6, 2020, from https://rt.wiki.kernel.org/index.php/Main_Page

CppReference. (2020). *Mutex - c++ reference*. Retrieved May 8, 2020, from https://www.cplusplus.com/reference/mutex/mutex/

Delgado, R., You, B.-J., & Choi, B. W. (2019). Real-time control architecture based on xenomai using ros packages for a service robot. *The Journal of Systems Software*, *151*, 8–19.

Ferdoush, S., & Li, X. (2014). Wireless sensor network system design using raspberry pi and arduino for environmental monitoring applications [The 9th International Conference on Future Networks and Communications (FNC'14)/The 11th International Conference on Mobile Systems and Pervasive Computing (MobiSPC'14)/Affiliated Workshops]. *Procedia Computer Science*, *34*, 103–110. https://doi.org/https://doi.org/10.1016/j.procs.2014.07.059

Foundation, O. S. R. (2018). *Ros/introduction - ros wiki*. Retrieved March 9, 2020, from https://wiki.ros.org/ROS/Introduction

Foundation, R. P. (2020). *Raspberry pi foundation*. Retrieved March 20, 2020, from https://www.raspberrypi.org/help/

Fu, G., & Zhang, X. (2015). Rosbot: A low-cost autonomous social robot, In *2015 ieee international conference on advanced intelligent mechatronics (aim)*, IEEE.

Gao, D., & Wampler, C. (2009). Head injury criterion. *IEEE Robotics Automation Magazine*, *16*(4), 71–74.

Gerum, P. (2019). *Xenomai faq*. Retrieved May 6, 2020, from https://gitlab.denx.de/Xenomai/xenomai/-/wikis/FAQ#user-content-what-is-a-xenomai-skin

Graf, B., Hans, A., Kubacki, J., & Schraft, R. (2002). Robotic home assistant care-o-bot ii, In *Proceedings of the second joint 24th annual conference and the annual fall meeting of the biomedical engineering society] [engineering in medicine and biology*, IEEE.

Groote, J. F., & Mousavi, M. R. (2014). *Modeling and analysis of communicating systems*.

Heemskerk, C. (2020a). *Extra safeguards for robot rose*.

Heemskerk, C. (2020b). *Risk analysis robot rose*.

Hendrich, N., Bistry, H., & Zhang, J. (2015). Architecture and software design for a service robot in an elderly-care scenario. *Engineering*, *1*(1), 027–035.

HIT. (2020). *Robot rose official website - heemskerk innovative technology*. http://robot-rose.com/robot-rose/?lang=en

Honegger, D., Meier, L., Tanskanen, P., & Pollefeys, M. (2013). An open source and open hardware embedded metric optical flow cmos camera for indoor and outdoor applications, In *2013 ieee international conference on robotics and automation*.

Hsiung, P.-A. (2001). *Real-time constraints*.

Ikuta, K., Ishii, H., & Nokata, M. (2003). Safety evaluation method of design and control for human-care robots. *The International Journal of Robotics Research*, *22*(5), 281–297.

Juvva, K. (1998). *Real-time systems*. Retrieved February 10, 2020, from https://users.ece.cmu.edu/~koopman/des_s99/real_time/

Kaliński, K. J., & Mazur, M. (2016). Optimal control at energy performance index of the mobile robots following dynamically created trajectories [THEORETICAL AND APPLIED ASPECTS OF MODERN MECHATRONICS]. *Mechatronics*, *37*, 79–88. https://doi.org/https://doi.org/10.1016/j.mechatronics.2016.01.006

Karageorgos, D. (2017). *Human-aware autonomous navigation of a care robot in domestic environments | tu delft repositories*. Retrieved May 3, 2020, from https://repository.tudelft.nl/islandora/object/uuid:849660e5-330a-496b-984b-f9ed94b03142?collection=education

Karl-Bridge-Microsoft. (2018). *About processes and threads - microsoft docs*. Retrieved May 8, 2020, from https://docs.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads?redirectedfrom=MSDN

Kiszka, J. (2018). *Xenomai posix interface*. Retrieved May 7, 2020, from https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group__cobalt__api.html

Kiszka, J. (2020). *Xenomai project*. Retrieved February 10, 2020, from https://gitlab.denx.de/Xenomai/xenomai/-/wikis/home

Li, K., Wu, J., Zhao, X., & Tan, M. (20180201). Real-time human-robot interaction for a service robot based on 3d human activity recognition and human-mimicking decision mechanism.

Magee, J., & Kramer, J. (2006). *Concurrency: State models java programs, 2nd edition*.

Microchip. (2020). *Real-time control | microchip technology*. Retrieved May 4, 2020, from https://www.microchip.com/design-centers/8-bit/applications/real-time-control

Microsoft. (2012). *Description of race conditions and deadlocks*. Retrieved May 8, 2020, from https://support.microsoft.com/en-us/help/317723/description-of-race-conditions-and-deadlocks

Microsoft. (2018). *Semaphore objects - microsoft docs*. Retrieved May 8, 2020, from https://docs.microsoft.com/en-us/windows/win32/sync/semaphore-objects

Murphy, R., Kravitz, J., Stover, S., & Shoureshi, R. (2009). Mobile robots in mine rescue and recovery. *IEEE Robotics Automation Magazine*, *16*(2), 91–103.

Najarian, S., Fallahnezhad, M., & Afshari, E. (2011). Advances in medical robotic systems with specific applications in surgery-a review. *Journal of Medical Engineering Technology*, *35*(1), 19–33. http://www.tandfonline.com/doi/abs/10.3109/03091902.2010.535593

O'Kane, J. M. (2018). *A gentle introduction to ros*. Retrieved March 9, 2020, from https://cse.sc.edu/~jokane/agitr/

Oracle. (2019). *Starvation and livelock the java tutorials*. Retrieved May 8, 2020, from https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html

Piltch, A. (2019). *Raspberry pi 4 review: The gold standard for single-board computing*. Retrieved May 22, 2020, from https://www.tomshardware.com/reviews/raspberry-pi-4-b,6193.html

Poelmans, P., & Severijnen, O. (2013). *De apa-richtlijnen*.

Raspbian. (n.d.). *About - raspbian*. Retrieved May 7, 2020, from https://www.raspbian.org/RaspbianAbout

RPi-Distro. (2015). *Rtimulib - a versatile c++ and python 9-dof, 10-dof and 11-dof imu library*. Retrieved May 22, 2020, from https://github.com/RPi-Distro/RTIMULib

Sachs, D. (n.d.). *Sensor fusion on android devices: A revolution in motion processing*. https://www.youtube.com/watch?v=C7JQ7Rpwn2k

Stallings, W. (2017). *Operating systems : Internals and design principles* (4th ed). Upper Saddle River, NJ, Prentice Hall.

Tadele, T. S. (2014). *The safety of domestic robots a survey of various safety-related publications*. Retrieved February 20, 2020, from https://www.researchgate.net/publication/265555758_The_Safety_of_Domestic_Robots_A_Survey_of_Various_Safety-Related_Publications

Triebel, R., Arras, K., Alami, R., Beyer, L., Breuers, S., Chatila, R., Chetouani, M., Cremers, D., Evers, V., Fiore, M., Hung, H., Islas Ram¿rez, O., Joosse, M., Khambhaita, H., Kucner, T., Leibe, B., Lilienthal, A., Linder, T., & Lohse, M. (2015).

Spencer: A socially aware service robot for passenger guidance and help in busy airports. *Proceedings of the 10th Conference on Field and Service Robotics, FSR 2015*, (Canada).

Verum. (2019a). *About dezyne*. Retrieved February 5, 2020, from https://www.verum.com/dezyne/

Verum. (2019b). *Verum software tools bv*. Retrieved May 9, 2020, from https://www.verum.com/dzndoc/tutorials-index/

Verum. (2019c). *Verum software tools bv*. Retrieved February 20, 2020, from https://www.verum.com/dzndoc/online-help-index/

Verum. (2019d). *Verum software tools bv*. Retrieved May 9, 2020, from https://www.verum.com/dzndoc/online-help-glossary/

Woodman, O. J., Woodman, C. O. J., & Woodman, O. J. (2007). An introduction to inertial navigation.

Zhang, Y., Wang, X., Wu, X., Zhang, W., Jiang, M., & Al-Khassaweneh, M. (2019). Intelligent hotel ros-based service robot, In *2019 ieee international conference on electro information technology (eit)*, IEEE.

Summary

Research group Robotics develops a care robot, called Rose, which is able to assist elderly people with every day tasks. Although Rose has some measures of safety built into her system, robot Rose lacks the kind of guarantees on safety that are required to use Rose in practice. To improve the safety of care robot Rose, research group Robotics at Inholland Alkmaar wants to create a safety module which checks the behavior of care robot Rose and which gives a signal when robot Rose shows an unsafe action, where an unsafe action is characterized by an action that will cause a collision with a human or object, an action that will cause Rose to clamp with a human or object, or an action that will cause Rose her grip arm to pinch a human or a object.

Our research focuses on laying down a basis for this safety module, from which further research can build upon. This research is a feasibility study in the sense that we want to explore existing ideas in mobile robotic safety, real-time robotic systems and verification of system behavior. The goal of this research is to apply these existing ideas to our case with care robot Rose, and to provide a prototype of the safety module to affirm the feasibility of the system as a whole. In order to achieve this goal, we setup the following principal research question: *How can a real-time safety module be designed and implemented that provides an extra layer of safety in order to prevent injury to patients or damage to the environment because of collision, clamping or pinching caused by care robot Rose?*. To provide an answer to this question the research was divided into four parts: obtaining a hardware platform and a real-time operating system for the implementation, devising a way to use sensor data to detect unsafe actions and devising a way to signal this to the environment, creating a formal specification of the behavior of safety module, and creating an implementation using the obtained hardware platform, operating system and created formal specification.

Firstly, we conducted a literature review and comparative study to decide on a hardware platform and real-time operating system to use. The literature reviews and the comparative study showed the Raspberry Pi to be the best hardware platform and Raspbian Buster with the Xenomai framework to be the best real-time operating system to fulfill the goal of the research.
Secondly, we reasoned that unsafe actions can be detected by checking the kinetic energy, rotational energy, grip arm force, grip arm torque and the position of the grip arm of robot Rose. The Raspberry Pi has a Sense HAT with sensors that are used to obtain kinetic energy and rotational energy. The grip arm values are implemented as stubs that simulate force, torque and position values respectively. Unsafe actions are signalled by turning the LED matrix of the Sense HAT red.

Thirdly, a model of the safety module was made in a formal specification language to verify and assert correct and anticipated behavior of the safety module.
Lastly, the safety module was implemented, using the formal model, on the hardware platform and real-time operating system.

Running the safety module showed anticipated behavior; excessive kinetic energy or excessive rotational energy are detected and signalled.

From tests we concluded that the safety module achieves soft real-time operation. Furthermore, we concluded that because of the successful runs of demonstrations of the safety module, we affirmed the feasibility of using this hardware and operating system setup, and we concluded that the goal of designing and implementing a safety module that provides an extra layer of safety in order to prevent injury to patients or damage to the environment because of collision, clamping or pinching is achieved.

The safety module sometimes has issues with the timeliness in signalling unsafe actions. We therefore recommend follow-up research to improve the timeliness of the safety module. Moreover, we recommend follow-up research to ameliorate the safety module by setting sight on achieving hard real-time operation.