University of York Department of Computer Science

Architectural Data Modelling for Robotic Applications

First Year Report

William Barnett

7th September 2019

Supervisors: Prof. Ana Cavalcanti and Dr. Alvaro Miyazawa

Abstract

Robotic systems are being employed in a diverse range of applications, with both the scale and complexity of their software increasing through having to operate in unstructured environments and to provide higher levels of autonomy. In addition, the need for robotic systems to be verified grows as robots are used in applications where they can have significant safety implications. Verification of even small robotic systems software is a challenging problem. Therefore, additional techniques are required to enable the practitioners to produce verified robotic systems.

The use of model-driven engineering and domain-specific languages (DSLs) have proven useful in the development of complex systems. RoboChart is a DSL for modelling the behavior of robot software controllers using state machines. It is distinctive in that it has a formally defined semantics and provides support for automated and semi-automated verification.

So that RoboChart can be used by developers, its support for modelling and verification must scale to real robotic systems. To date, RoboChart has been used to model only a limited number of small systems. Therefore, the pragmatics of using RoboChart are not well-established. Additionally, the verification of large robotic systems using RoboChart's model checking features leads to state explosion. Therefore, new techniques must be developed to enable the verification of real robotic systems using RoboChart.

Our work addresses the issue of scalability by the addition of support for architectural modelling to RoboChart and the development of compositional reasoning techniques that take advantage of architectural structure. We establish the pragmatics of RoboChart by evaluating how robotic systems that employ robotics architectures described in the literature can be modelled, and establishing guidelines for developers on how to use RoboChart to model and verify these systems.

This report provides the context for the research, evaluates the progress made in the first year of work, and sets out a plan for the following two years. We examine the role of robotics software architectures in the development of robotic systems by reviewing five robotics architectures, and five DSLs. A RoboChart model of a software controller for an autonomous vehicle system is presented. Finally, some initial RoboChart guidelines for developers are given, based on the experience gained using RoboChart to model a real system.

Acknowledgements

The B-ACS work has been done as part of the CAVlab project in 2017-2018. The team involved includes Dave Barnett, Servando German Serrano, Ujjar Bhandari, Nastaran Shatti, and Alan Peters. Zeyn Saigol is proposing and developing the B-ACS work as a suitable autonomy verification case study.

Contents

List of Tables

List of Figures

1 Introduction

Robotic systems are being used in an increasingly diverse range of applications, and deployed into more dynamic and unstructured environments. With autonomy and the ability to operate in close proximity to humans, there is an increased risk of these systems causing harm. Furthermore, robotic systems and their software are becoming more complex. We contribute to the verification of robotic systems using a domain-specific language with a formal semantics, namely, RoboChart. We propose an approach to model in RoboChart control software that employ architectures described in the robotics literature. We also show how to take advantage of these architectures to develop compositional reasoning techniques that aid the scalability of verification. Finally, as part of this work, we establish the pragmatics of RoboChart via guidelines on how RoboChart can be used to model and verify robotic systems.

This report reviews the progress made during the first year of a three year research project on architectural and data modelling for robotic applications, which aims to contribute to the verification of robotic systems.

This chapter discusses the motivation and objectives of our work and provides a mapping of the sections of this report onto the first year progression requirements. The structure of this chapter is as follows: Section 1.1 maps the report sections to the progression requirements, Section 1.2 explains the motivation behind our work, Section 1.3 specifies the main objectives of our work, and finally Section 1.4 lays out the structure of this document.

1.1 Overview of the Structure of This Report and Mapping to the Progression Requirements

Table 1.1 provides the mapping between the first year progression requirements [1] and the locations in this report that evidence them.

Progression requirement 1 is evidenced by Section 1.2 (motivation) and Section 1.3 (objectives) . They set the direction of the research and introduce the primary research question.

Table 1.1: Progression requirement mapping

Progression Requirement	Place Evidenced
1	Section 1.2 and Section 1.3
2	Section 5.2 and Section 5.5
3	Chapter 2
4	Section 5.1 supported by Chapter 3 and Chapter 4
5	Appendix A.1
6	Section 5.4

Progression requirement 2 is evidenced by Section 5.2, the plan for year two, and Section 5.5, risk analysis. They detail the overall plan and risk mitigation measures to meet the objectives of the research.

Progression requirement 3 is evidenced in Chapter 2 by the explanation of the role software architecture plays in the development of robotic systems given by Chapter. Chapter 2 provides the context for our work covering: recent robotics architectures, domain-specific languages, and an introduction to RoboChart models.

Progression requirement 4 is evidenced by Section 5.1, an evaluation of progress. The evaluation considers the contribution from each of the previous chapters towards the objectives of our work and the progress made to date. The guidelines defined by Chapter 3 and the autonomous vehicle prototype case study in Chapter 4 support the evaluation providing evidence of the techniques and approaches used.

Progression requirement 5 states all necessary training must be completed, and is evidenced by the SkillForge training log; a reproduction of this log can be found in Appendix A.1.

Progression requirement 6 is evidenced by Section 5.4 and considers the ethical concerns of robotic system.

1.2 Motivation

Advances in technology are enabling the development of robotic systems for an increasingly diverse range of applications. Robotic systems are being deployed in more unstructured and dynamic environments with closer collaboration between people and robots. For instance, manufacturing robots that work alongside humans are being used in the

workplace [2], and robots that assist with care for the elderly in the home are being developed [3].

Additionally, there is a rising demand for more autonomous systems with the ambition to increase productivity, reduce cost, and improve safety. For example, in the transportation sector, driverless vehicles [4], and automated goods delivery robots [5] are being developed.

As these new types of robotic systems become more widespread, interactions between humans and robots will become routine. Safety is a principal concern for any such system, and the more capabilities robotic systems are given to physically interact with their environment, the greater the risk of hazardous situations and hazardous events resulting in harm there will be. Therefore, it is important that robotic systems react to these hazardous situations and events as designed, in order to prevent the widespread disruption of the services that they provide.

Software plays a crucial role in robotic systems for the flexibility it affords, providing many of the desired dynamic features that define the system's behaviour. Therefore, as robotic systems capabilities grow, the software used to control them is becoming increasingly complex.

This increased software complexity poses a significant challenge in verifying robotic systems at design time for all possible scenarios, as the system may encounter situations that have not been considered during design [6]. Typically, complete design time verification of these types of systems is not feasible using existing methods alone [6]. Therefore, in order to enable the successful realisation of forthcoming robotic systems, techniques along with associated tools are required to assist developers in managing the complexity, and ensure that design requirements can be met.

As the scale of software systems has grown, one technique that has been beneficial is the use of a well-defined software architecture. It provides a structural representation of a system that enables the evaluation of different system attributes by its stakeholders [7, p. 5]. The use of views to represent a subset of related architectural structures facilitates communication between stakeholders of the system, and enables evaluation of alternative system designs and modifications [7, p. 10].

Some examples of well-known software architectural patterns include the client-server pattern used by the internet, and the layered pattern [8] used by embedded systems. There has also been the development of standardised domain-specific architectures to ease the integration of system components between stakeholders and promote reuse of produced outputs. One notable example of a standardised architecture is Autosar

from the automotive industry [9].

For the robotics domain, many projects still choose to establish their own architecture. This means that there have been many proposals, but there is no single widely adopted architecture. Some common patterns have emerged: notably, the use of layers for robot control [10, pp. 286–289]. In order for notations and tools that target the robotics domain to be widely accepted and provide the maximum benefit to developers, they must be flexible and support the range of approaches and architectural patterns that are used by developers.

The envisaged tools necessary for the development of prospective robotic systems need to be based upon a suitable development methodology. In many other complex multidisciplinary domains, such as, aerospace and automotive, Model Driven Engineering (MDE) is being used successfully to mitigate complexity [11], [12]. The core principle of MDE is to use abstract models of a system as the primary artefacts of its development process. This promotes identification of the underlying concepts free from specific implementation dependencies. The use of abstract models also facilitates the automation of the software development process. So developers can devote their time to understanding and solving the domain-specific problems.

The flexibility of the MDE approach means that it can be applied to any domain. With such potential diversity between domains, a single language that is general enough to describe all of the required domain concepts leaves the domain-specific concept definitions to each development team, resulting in duplication of work, and hindering the reusability of designs.

Domain-Specific Languages (DSL) address this issue by describing the core concepts required by the target domain, and provide a concise shared representation that is understood by the practitioners of the domain. Over the last twenty-five years, there have been considerable developments in MDE for robotics, with the creation of many DSLs for its different subdomains [13].

Some examples of DSLs for robotics include: RoboML [14], SmartSoft [15], and BCM [16]. These DSLs, like the majority that are available, do not have formally defined semantics. Therefore, the support for formal verification of robotic systems is limited.

A recent literature survey [17], over the last ten years, found sixty-three examples of the application of formal methods within the robotics domain. Formal methods enable the proof of properties of a system's specification through the use of unambiguous mathematical notation [18,

p. 41]. Therefore, formal methods are likely to play an important role in verification of robotic systems.

RoboChart is a DSL for modelling robotics software controllers using state machines [19] that makes innovative use of formal methods for automated verification. The associated tool, RoboTool, provides features of MDE, which include a graphical interface for creating models, and automatic generation of source code. Additionally, RoboChart supports automatic verification of properties such as determinism, deadlock, and livelock using model checking, along with semi-automatic verification techniques using theorem proving.

RoboChart's formally defined semantics coupled with its graphical notation mean that formal models can be automatically generated from a RoboChart model. This means that developers only require a minimal understanding of formal methods to take advantage of the verification capabilities provided by RoboChart. Consequently, RoboChart can provide a multitude of benefits for the development of robotics software, most significantly, a contribution towards rigorously proving that a system conforms to its specification.

To date, RoboChart has been used to model thirteen proof-of-concept case studies, which include robots and multi-robot swarm algorithms [20]. The most complex robot case study is a four-wheeled autonomous chemical detector, and some examples of swarm case studies have included the α-algorithm [21, p. 85] and transport algorithm [22]. All of these case studies have facilitated the development of RoboChart, however, they only represent a small subset of the many diverse robotic systems being developed. Therefore, the pragmatics of using RoboChart to model robotic systems are not well-established.

So that the benefits RoboChart provides can contribute to the wider verification of robotic systems, it must be widely accepted by developers. Therefore, RoboChart must be able to model the software controllers of real robotic systems. When using RoboChart to model larger robotic systems, support for modelling commonly used architectural patterns can enable the structure of systems to be modelled and new compositional reasoning techniques to be developed. This motivates the primary research question for our work:

> How can architectures commonly used in robotics be captured in RoboChart, and how can we take advantage of the architectural composition in verification?

For RoboChart to be accepted by developers it must be accessible to

them, therefore, establishing the pragmatics of RoboChart forms another important component of our work. In the following section the main objectives for the proposed work are specified.

1.3 Objectives

The goal of our work is: to contribute to the advancement of software verification for robotics, to support the creation of robotic systems that are safe and robust, and facilitate certification with emerging safety standards. In order to achieve this goal, we have the following objectives:

- to identify architectural patterns that are used by robotic systems software;
- to evaluate RoboChart's support for the modelling of robotic systems when considering their architecture;
- to contribute to the practice of modelling using RoboChart by providing guidelines to model common architectures, and, if necessary, extending RoboChart to support modelling of these architectures;
- to contribute to the verification aspect of RoboChart by developing compositional verification techniques that make use of architectural structure;
- to establish the pragmatics of RoboChart by the definition of guidelines for developers on how common architectures can be modelled and how the verification features of RoboChart can be best utilised.

Via a comprehensive study of the literature, commonly used architectural patterns will be identified to determine which architectures should be supported by the modelling concepts of RoboChart.

Case studies based on robotic systems from the literature or from industry will be used to evaluate the extent of RoboChart's support for modelling the identified architectures. Any significant limitations identified from the evaluation of RoboChart will be addressed by extending the language to support the modelling of systems that make use of the identified architectural patterns. In any case, precise guidance will be provided to practitioners regarding the use of RoboChart to capture the architecture of their designs.

From the architectural structure provided by RoboChart models, techniques for the compositional verification of robotic system properties will be created to facilitate scalability. We will explore model checking and theorem proving.

From the experience gained using RoboChart, guidelines that cover how to model and verify robotic systems will be created to establish the pragmatics of RoboChart and assist developers in its use.

1.4 Document Structure

The document is structured as follows:

Chapter 2 looks at the role of software architecture in robotic systems development by examining the following areas: robotics architecture, DSLs for robotics, and how to model robotic systems using RoboChart.

Chapter 3 outlines the guidelines that been have been defined as a result of our experience using RoboChart.

Chapter 4 documents the development of an autonomous vehicle case study that is based on real system.

Chapter 5 evaluates the work carried out during the first year and sets out a plan for the next two years, additionally, ethics and potential project risks are considered.

2 The Role of Software Architecture in Robotic Systems Development

DSLs for robotics that can represent the architectural structure of systems provide a mechanism for potentially compositional reasoning techniques to be applied. Compositional reasoning can assist with scalability when verifying robotic systems and the automatic verification of properties can reduce the overall time developers need to verify the system. These factors mean that DSLs which support the representation of architectural structure can contribute the development of increasingly complex robotic systems.

In order to evaluate robotics DSLs support for architectural modelling it is necessary to understand the architectural structures used in the robotics domain architectures.

There have been many definitions for architecture put forward over the last thirty years [23]. We adopt the definition given by ISO 42010:2011 Systems and Software Engineering - Architecture Description:

> "**architecture** <system> fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution." [24]

When developing complex software systems it is important to consider its architecture. Architecture provides a structural representation of a system that enables the evaluation of different system attributes by its stakeholders [7, p. 5]. The documentation of a system's architecture facilitates its communication and understanding among its users.

What constitutes a good architecture depends on the system being developed and its application. Therefore, developing architectural principles that can be widely applied to various domains, such as robotics, is a particular challenge in its own right and one which we are not going to address.

In the robotics domain many different robotics architectures have been proposed with no single architecture fitting all applications [10, p. 283].

Section 2.1 presents robotics architectures that have been used over the last twenty years.

It is also crucial to survey other similar DSLs and to justify the suitability of RoboChart for the work. Section 2.2 reviews a selection of DSLs for robotics.

Since RoboChart is the DSL being used in this work, it is important to understand how RoboChart can be used to model a robotic system. Therefore, Section 2.3 presents an example of how a robot can be modelled using RoboChart. We conclude in Section 2.4, with some final considerations.

2.1 Robotics Software Architectures

Robotic systems are often complex and typically use software as the basis for their control and coordination. Over the last 30 years, many different software architectures for robotic systems have been developed.

Historical architectures include Sense Plan Act (SPA) [10, p. 285] and subsumption [25]. SPA is an example of an architecture that is deliberative: time is taken to plan what to do next, and then the plan is acted out with no sensing or feedback during acting. A robot using SPA in a dynamically changing world can be slow and error prone in response to environmental change and can, therefore, be potentially dangerous.

Conversely, subsumption is an example of an architecture that is reactive, where the environment is constantly sensed and used to directly shape the robot's actions. A robot using the subsumption architecture responds rapidly to a changing world; however, complex actions are difficult to achieve.

More recent hybrid architectures combine the principles from SPA and subsumption architectures in order to benefit from both the deliberative and reactive properties.

In order to determine the important architectural structures that DSLs for robotics should support, the characteristic features of robotics architectures need to be identified. For that, we consider the following architectural aspects for a selection of robotics architectures: structure of the software elements and the relationships between them [7, p. 4], and control approach.

In total, twenty-three robotics specific architectures have been identified from the literature; these are listed in Table 2.1. Five have been selected for discussion based upon evidence of application, reuse, and activity of

development. The collective publications that focus on an architecture have been used to find evidence of application, with the scale of any documented application used to give preference to architectures that have been used in large deployments in the real world. The number of publications where an architecture was used in a new application has been used to asses the architecture's reuse. Preference has been given to architectures with recent activity, determined by the date and frequency of publications where the architecture has been used.

Table 2.1: The architectures identified from the literature.

Architecture	**Focus**	**Year**
CoSiMA [26]	Safe real-time robots	2018
◆ IRSA [27]	Autonomous robots	2018
◆ SERA [28]	Decentralised teams	2018
◆ CARACaS [29]	Autonomous robots	2011
Aerostack [30]	Autonomous unmanned aerial systems	2017
EFTCoR [31]	Service robot control	2006
Syndicate [32]	Autonomous teams	2006
DDX [33]	Distributed robot controllers	2004
◆ CLARAty [34]	Autonomous robots	2001
HARPIC [35]	Autonomous robots	2001
◆ LAAS [36]	Autonomous robots	1998
Remote Agent [37]	Autonomous robots	1998
ORCCAD [38]	Robot control	1998
3T [39]	Autonomous robots	1997
Planner Reactor [40]	Autonomous robots	1995
MIAA [41]	Autonomous robots	1994
CIRCA [42]	Real-time intelligent robots	1993
ATLANTIS [43]	Autonomous robots	1992
Layered Competencies [44]	Autonomous robots	1991
Motor Schema [45]	Robot control	1989
NASREM [46]	Autonomous robots	1989
AuRA [47]	Autonomous robots	1987
Subsumption [25]	Autonomous robots	1986

Legend: ◆ Selected architectures for further discussion.

Sections 2.1.1 to 2.1.5 presents the selected architectures, and provides a review of the discussed important aspects. Finally Section 2.1.6, evaluates

the state of architectures for robotics and their structure.

2.1.1 LAAS

The LAAS architecture was developed at LAAS[1] in 1998 for autonomous robots. A fundamental goal of the LAAS architecture is to provide both deliberative and reactive capabilities required for autonomy [36].

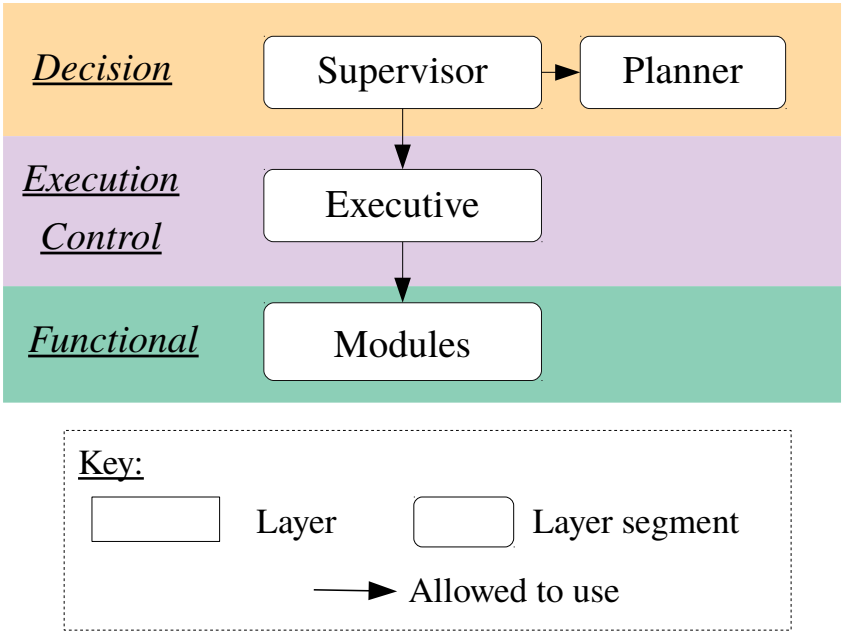
The LAAS architecture is made up of the following three layers:

Functional Layer Provides basic robot actions that are organised into modules consisting of processing functions, task loops, and monitoring functions for reactive behaviour.

Execution Control Layer Selects functions from the functional layer to carry out sequences of actions received from the decision layer.

Decision Layer Plans the sequence of actions necessary to achieve mission goals and supervises the execution of the plans.

Figure 2.1: The LAAS architecture layer diagram



[1]Laboratory for Analysis and Architecture of Systems CNRS

The functional layer, shown in Figure 2.1 as the bottom layer, consists of a network of modules that can be either synchronous or asynchronous. Each module of the functional layer provides a service that relates to a particular sensor, actuator or data resource of the robot [36]. An example of a data resource would be a map or image. All modules have a fixed generic structure made up of a controller and execution engine. Because the structure of modules is fixed, a tool generator of modules (GenoM) can be used to generate module source code. To generate a module's source code, GenoM combines a formalised description of the module along with pieces of code (codels) describing the module's algorithm.

The services of modules are accessed by the executive layer above and other modules from the functional layer, through the use of a non-blocking client-server communication model. The client-server model is well-supported by middleware that uses network protocols; therefore, the implementation of the functional layer can directly correspond to the modelled design.

The execution control layer, shown in Figure 2.1 as the middle layer, bridges the slow, high-level, processing of the decision layer, and the fast, low-level, control of the functional layer. The execution control layer's executive takes sequences of actions from the decision layer, and selects and requests the functions that the functional layer must carry out. The executive receives replies from the functional layer and reports activity progress back up to the decision layer. To enable prioritisation and interruption of functional layer modules, a local execution state database is maintained so that conflicts between modules can be managed.

The Decision Layer, shown in Figure 2.1 as the top layer, is separated into a Supervisor and a planner. The planner creates a sequence of actions to achieve a goal. The supervisor takes the generated sequence of actions and manages their execution by communicating them to the execution control layer, and responding to reports received from the execution control layer.

Along with the sequences of actions, the supervisor also passes down situations to monitor and associated responses that are within the constraints of the plan. For example, for a robot whose mission is to travel from point A to point B in a hospital environment, the mission constraint could be to keep out of areas where the robot is not allowed to go. One situation to monitor would be obstacles blocking the route with the associated response being to avoid the obstacle. On encountering an obstacle, the executive can allow the robot to deviate from the planned path to navigate the obstacle. However, if the obstacle is positioned such that

the only way to avoid it involves violating the mission constraints, the executive would have to notify the supervisor and obtain an updated plan.

These given responses enable the lower layers to react without the need for involvement of the decision layer, therefore, improving response time and reducing unnecessary replanning. The decision layer itself can contain multiple supervisor-planner pairs, for example, Mission, Task, and Coordination with the coordination layer taking into account other robots [36].

LAAS has been used in the implementation of the ADAM rough terrain planetary exploration rover [48], and of three Hilare autonomous environment exploration robots as part of the MARTHA European project [49].

More recently, Behaviour Interaction Priority (BIP) models have been used to verify the functional layer of the LAAS architecture [50]. Functional layers described using $G^{en}oM$ can be automatically translated into a BIP model. The BIP model can then be checked for deadlock freedom and other specified safety properties using BIP's associated tools.

2.1.2 CLARAty

Coupled Layer Architecture for Robotic Autonomy (CLARAty) was developed at JPL in 2001 for planetary surface-exploration rovers. CLARAty is designed to be reusable and to support multiple robot platforms; it consists of two-layers formed by combining the planning and executive layers from a three-layer architecture [34]. A key concept defined by the CLARAty architecture is granularity, which reflects the varying levels of deliberativeness available to the robotic system.

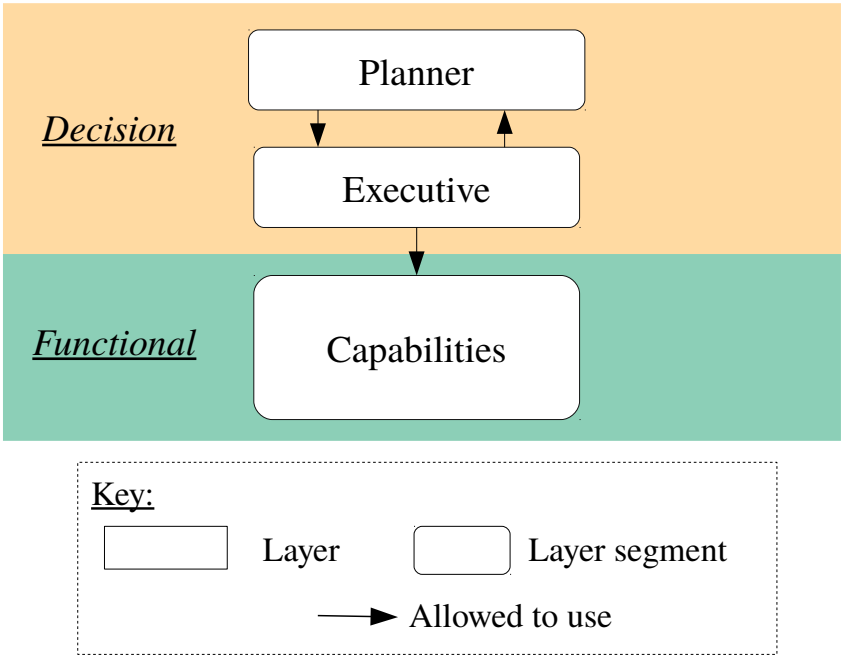
The layers of the CLARAty architecture are as follows:

Functional Layer Is the interface to the systems hardware capabilities.

Decision Layer Decomposes mission goals into task sequences and then into commands for the functional layer.

The functional layer, shown in Figure 2.2 as the bottom layer, provides a software interface to the hardware capabilities of the robot, and it is structured using an object-oriented hierarchy. At the top of the hierarchy is the Robot superclass from which everything inherits. At subsequent levels down the hierarchy, classes are less abstract and each provide functionality for a piece of the robot's hardware. At the bottom of the

Figure 2.2: The CLARAty architecture layer diagram



hierarchy, each class provides access to a specific piece of hardware functionality and its current state.

Classes can provide functionality that requires minimal input from the decision layer, therefore, this type of class can be considered more reactive. For example, the class for a rover may offer a method for obstacle avoidance. Alternatively classes can provide functionality that requires regular input from the decision layer, therefore, the class can be considered more deliberative. For example, the class for a robotic arm may offer a method for setting the position for one of its five motors.

The object-oriented hierarchy of the functional layer allows for a logical mapping onto the robot's physical structure. This complementary relationship assists developers because a correspondence between the robot's hardware and the software is created, reinforcing their understanding. However, the inclination towards the functional view for the lower layer means that the system's behaviours are not emphasised by the architecture.

The decision layer, shown in Figure 2.2 as the top layer, decomposes mission goals into tasks, then into commands that access the capabilities

of the functional layer using a client-server model [51]. The structure of the functional layer means that the decision layer has a choice between selecting more reactive functions or more deliberative functions from the available capabilities. The more reactive functions mean the planning effort required by the decision layer is reduced. By comparison, more deliberative functions provide access to low-level hardware functionality, therefore, more planning effort is required by the decision layer to accomplish a task using them.

The single decision layer enables state information between planner and executive to be shared, which means that the planner becomes tightly integrated with the executive. Consequently, discrepancy between the planner and the functional layer's state is minimised. The CLARAty software architecture has been used for a variety of robot platforms: Rocky 8, FIDO, ROCKY 7, K9 Rovers, and ATRV Jr. COTS platform [52]. The different platforms have a variety of deployment architectures, from a single processor requiring hard real-time scheduling to processor and distributed microprocessors using soft real-time scheduling.

2.1.3 CARACaS

Control Architecture for Robotic Agent Command and Sensing (CARACaS) is an architecture developed at JPL[2] in 2011 for control of autonomous underwater vehicles (AUV), and autonomous surface vehicles (ASV) [29]. CARACaS allows operation in uncontrolled environments ensuring the vehicles obey maritime regulations for preventing collisions. It supports cooperation between different vehicles and it makes use of dynamic planning to adapt to the current environmental conditions and mission goals.

There are five main elements of the CARACaS architecture as follows:

Actuators Interfaces the actuators of the vehicle.

Behaviour Engine Coordinates and enables the composition of behaviours acting on the vehicles actuators.

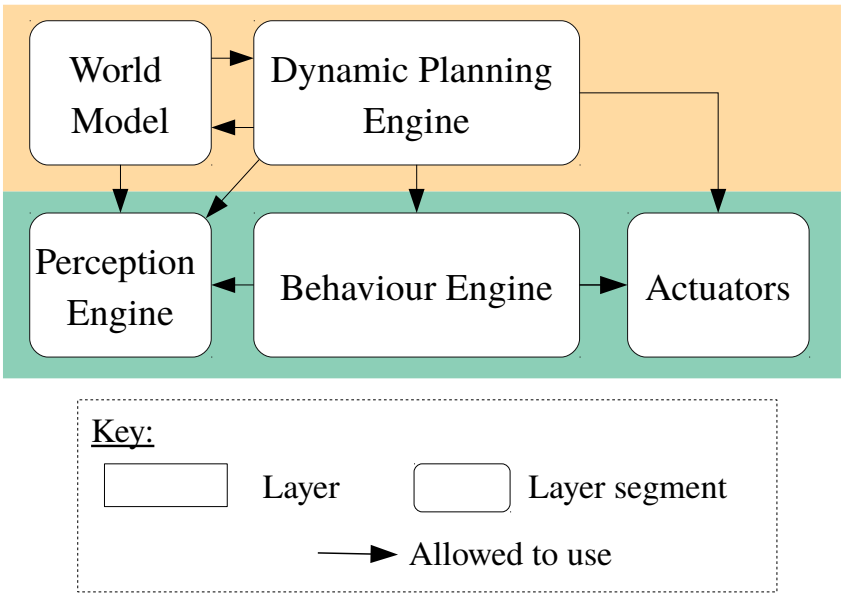
Perception Engine Creates maps for safe navigation and hazard perception from the vehicles sensors.

Dynamic Planning Engine Chooses activities to accomplish mission goals while observing resource constraints.

[2]NASA Jet Propulsion Laboratory

World Model Contains vehicle state information including mission plans, maps, and other agents.

Figure 2.3: The CARACaS architecture layer diagram



The Behaviour Engine makes use of behaviour composition and coordination methods developed as part of a previous multi-agent control architecture CAMPOUT [53]. Control of the vehicle is achieved using algorithms activating and deactivating behaviours. The arbitration mechanisms controlling the enabling and disabling of behaviours supported are subsumption, voting, and interval programming (IvP).

In order to achieve the mission goals, the Dynamic Planning Engine uses Continuous Activity Scheduling Planning Execution and Replanning (CASPER) [54]. CASPER decides on the activities that must be carried out in order to accomplish any mission goals, taking into consideration current resource constraints and rules. The activities are then executed by issuing commands to the Behaviour Engine to enable the behaviours associated with the activity. In the case of plan conflicts, CASPER supports dynamic replanning allowing the system to react to changing events.

CARACaS uses the R4SA real-time embedded system, which runs on real-time operating system QNX [29]. R4SA provides abstractions of the low-level hardware into devices and manages the synchronisation and

scheduling of all elements of CARACaS.

Layers are not strictly defined by [29]; however, CARACaS can be partitioned into two layers as shown in Figure 2.3. At the lowest level, we have a behavioural layer consisting of the Behaviour Engine and Perception Engine elements. The example UAV from [29] uses a stereo vision system and sonar as the main inputs to the Perception Engine for map creation. The second higher level layer consists of the Dynamic Planning and the World Model elements.

Although CARACaS is targeted at autonomous water-based vehicles, it contains all of the required elements to be applied more generally as an architecture for the control of robots.

A notable example of the application of CARACaS is its use as part of an automated patrol demonstration system to the U.S. Navy [55]. The automated patrol system consists of four unmanned boats working together to patrol an area of sea four square miles in size.

2.1.4 IRSA

The Intelligent Robotics System Architecture (IRSA) was developed at JPL in 2018 to streamline the transition of robotic algorithms from development onto flight systems, by improving compatibility with existing flight software architectures [27]. The IRSA architecture uses concepts from the other robotics architectures: CARACaS and CLARAty.

The main elements of the IRSA architecture are as follows:

Primitive Low-level behaviours that can have control loops.

Behaviour Provides the autonomy of the robot, transitioning between multiple states during execution.

Executive Receives and executes a sequence of instruction commands from the planner or other input.

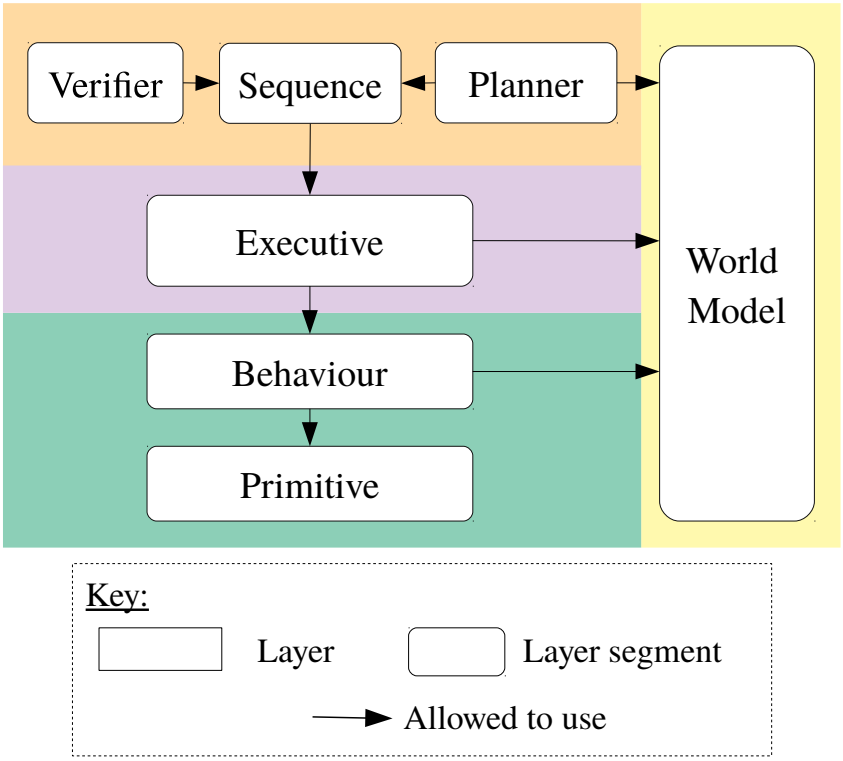
Planner Uses the system state from the world model to produce the sequence of command instructions to be executed.

Sequence Contains the instructions that the robot must perform.

Verifier Verifies the sequence is valid, which can include checking that the robot stays in a safe state.

Robot World Model Maintains a model of the robot that is made up local and global state information.

Figure 2.4: The IRSA architecture layer diagram



Although the IRSA architecture does not strictly define layers, it can be mapped onto a three-layer model with a common world model accessible to all layers, as shown in Figure 2.4.

The IRSA architecture is behaviour focused, with the low-level architectural primitive and behaviour elements responsible for providing the robot's behaviours. The primitive element provides fundamental behaviours that control the robot's hardware. The behaviour element provides hierarchical behaviours composed of those provided by the the primitive. Both the behavior and the primitive elements provide control over the robot, therefore, these two elements can be placed in the bottom layer of the architecture diagram, as shown in Figure 2.4.

The executive receives sequences of commands and manages command execution using the behaviours from the lower layer. Therefore, the executive is placed in the middle layer of Figure 2.4.

Sequences of commands can come from a variety of sources. The

primary source for an autonomous robot would be automated planning and scheduling, depicted as the planner in Figure 2.4. The planner uses the state of the system from the world model to create a sequence of commands that achieves the system's goals. The verifier performs verification checks on the sequence, for instance, validity checking and ensuring the robot maintains a safe state. The resulting command sequence held by the sequence element is communicated to the executive for execution. Therefore, the planner, sequence, and verifier elements can be placed in the layer above the executive; the top layer in Figure 2.4.

The IRSA architecture has been deployed on a variety of test beds (comet surface sample return, Europa lander sampling autonomy, Mars 2020 Controls and Autonomy for Sample Acquisition) and the RoboSimian DARPA challenge. Implementations have used custom middleware (RSAP) which enables inter-process communication via the TCP and UDP network protocols. Time driven and non-time driven tasks are supported, with task execution driven by messages received via inter-process communication. Hierarchical state machines for behaviours and control have been used. The use of ROS2 as an alternative to RSAP is being explored [27].

2.1.5 SERA

Self-adaptive dEcentralised Robotic Architecture (SERA) was developed at the Chalmers University of Technology in 2018 [28]. SERA's primary goal is to support decentralised self-adaptive collaboration between robots or humans, and it is based on the 3-layer self-management architectural model [56]. SERA was evaluated in collaboration with industrial partners who were participating in the Co4Robots H2020 EU project [28]. The layers of the SERA architecture are as follows:

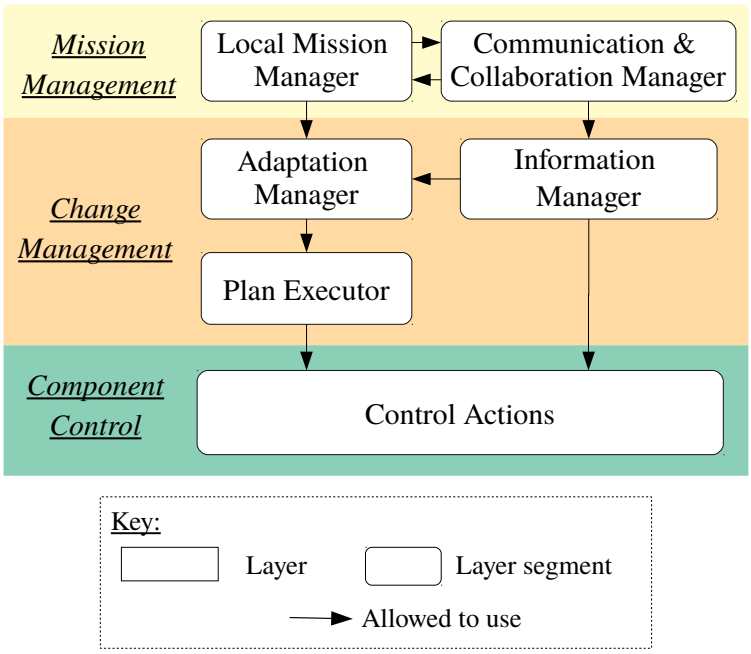
Component Control Layer Provides software interfaces to the robot's sensors and actuators, grouped into control action components responsible for particular areas of functionality.

Change Management Layer Receives the local mission and creates a plan in order satisfy its goals. It executes the plan by calling appropriate control actions from the component control layer.

Mission Management Layer Manages the local mission for each robot and communicates with other robots in order synchronise and achieve

the global mission.

Figure 2.5: The SERA architecture layer diagram



The component control layer, shown in Figure 2.5 as the bottom layer, interfaces to the robot's sensors and actuators through control action components. These components enable autonomous control of the robot through motion planning, object perception, and localisation and mapping.

The change management layer, shown in Figure 2.5 as the middle layer, receives the local mission. Its adaptation manager creates a plan to satisfy the mission goals. If a plan satisfying the mission goals can be created, a plan executor calls the relevant control actions to execute the plan. If it fails to find a plan that satisfies the mission goals, the higher-level mission manager is notified.

The mission management layer, shown in Figure 2.5 as the top layer, receives a local mission specification from a central station in the form of timed temporal logic formulae. The local mission manager checks the feasibility of the received mission and, if it is feasible, passes the mission to the adaptation manager in the layer below. If the mission is infeasible, a communication and collaboration manager communicates and

synchronises with the other robots involved in the mission. During the synchronisation, an updated achievable mission that meets the original mission specification is computed.

This architecture places more functionality in the lowest component control layer, such as, low-level motion planning, mapping, and object detection. The higher level layers are responsible for ensuring the mission is followed. A key feature of the SERA architecture is communication among robots, which takes place at the mission management layer. The communication among robots provides greater flexibility in achieving the mission goals, because, if a mission cannot be completed by an individual robot, a combination of other robots that are able to satisfy the mission can be utilised.

2.1.6 Evaluation

Table 2.2 summarises the primary features of the surveyed robotics architectures. Generally no particular architecture or group of architectures are being widely used across different robotic systems. There is a tendency for each project to establish its own architecture. Between research groups, there is some reuse of architectures, for example, the IRSA architecture is being used for a variety of space testbeds and the RoboSimian robot.

Layers are a common theme among many of the recent architectures. Even if layers have not been explicitly specified, the elements of an architecture can be mapped onto a layered model. All architectures have a functional layer that interacts with the robots sensors and actuators. The upper layers following the functional layer vary in number and purpose.

The functional layer is required by all architectures because every robot requires a means to sense and interact with its environment. From the architectures surveyed, this layer can be categorised as either service or behavioural. CLARAty, LAAS and SERA are all examples of architectures that have a service-based functional layer, whereas, CARACaS and IRSA have behavioural-based functional layers.

Architectures that use a service approach for the functional layer isolate the functional layer from the system state. This has the benefit of simplifying the functional layer and means the state of the system is managed by the upper levels of the architecture. However, this means that the decision layer must manage a large number of states. In CLARAty the decision layer holds a representation of all states, and goals are used as constraints to create the plan to be executed.

Table 2.2: Robotics architectures summary

Architecture	No. Layers	Control	Layers
CARACaS	2	Behavioural	Decision Functional
CLARAty	2*	Service	Decision Functional
LAAS	3	Service	Decision Execution Functional
IRSA	3*	Behavioural	Decision Execution Functional
SERA	3	Service	Mission Decision Functional

Legend: *Mapped onto a layered architecture model.

Architectures that use a behavioural approach for the functional layer rely on responding to environmental changes primarily using the functional layer. This has the benefit of reducing the number of states that the upper layers must manage. However, the functional layer must then arbitrate among the behaviours to share the robotic platform's resources, thereby, increasing the code complexity of the functional layer. The CARACaS architecture supports three techniques to arbitrate behaviours (subsumption, voting, and interval programming), whereas IRSA does not specify any arbitration mechanisms, leaving it open for developers decide for each project.

It is common for the decision layer to be placed directly above the functional layer; for instance, CARACaS, CLARAty, and SERA are structured in this way. They combine the decision and execution layers, therefore, the decision layer generates commands for the functional layer. In contrast LAAS and IRSA have a dedicated executive layer in-between the decision and functional layers that records the state of the system.

Architectures that do not use an executive layer take different approaches to managing the system's state. For instance, SERA and CLARAty use information in the decision layer to hold system state.

Whereas, CARACaS uses a world model layer that is accessible by all other layers to hold system state.

Having a separate execution layer provides no significant differences with regard to functionality, because in either case the functional layer is sent the commands for control and the status from the functional layer is passed to the layer above. Therefore, the primary difference is where the emphasis of concepts used by each architecture is placed.

Some architectures such as SERA have an additional social layer for collaboration between teams of robots. Similarly LAAS supports this through adding supervisor planner pairs, but considers this to be an extension of the decision layer rather than a new layer. Generally the layered architecture lends itself to the addition of new layers for extending the level of system capability.

The review of architectures discussed in this section, provides details on the structure and control techniques used by a selection of robotics architectures. This insight will be used as a foundation to guide the analysis of the use of RoboChart to model robotic systems. The next section provides a review of DSLs for robotics and justifies the use of RoboChart for our work.

2.2 Domain Specific Languages

Modelling languages can either be general purpose or domain specific [57, p. 59]. General Purpose Languages (GPLs) can be used to model a wide range of domains. An example of a GPL is Unified Modelling Language (UML) [58] for modelling software systems.

Domain-specific languages, on the other hand, target an individual domain. This means that DSLs contain only the constructs necessary to represent concepts from the domain of interest. Therefore, the number of semantic elements that users of the language have to remember is reduced [57, p. 70], and the widely used domain-specific concepts can be represented in a common way [57, p. 70]. An example of a DSL is VHDL [59] for modelling electronic hardware.

There is an increasing number of DSLs for robotics [13], therefore, the suitability of RoboChart for use in our work must be justified. For that, we consider the features of interest provided by a selection of DSLs and present an evaluation. The features are as follows: the notation types supported, whether the semantics is formally specified, the aspects of the system that can be modelled, and the artefacts that can be derived from the model.

The DSLs have been selected from one hundred and thirty-seven different robotics DSLs outlined by a survey [13] conducted by Nordmann et al. and a further six from a search of the literature for recent DSLs. The survey covers DSLs published between the years of 1980 and 2015, with our search covering from 2015 and onwards. Five DSLs have been selected for discussion based upon having a documented metamodel, providing support for code generation, and evidence of ongoing support.

A documented metamodel provides a means to analyse and compare the syntactic structure of each of the selected DSLs. Because code generation plays an important role in MDE for improving software quality and reducing development time, the artefacts that a DSL produces can provide an indication of its primary purpose. For DSLs to be widely adopted and used by practitioners they must be accessible and maintained to include the latest concepts from the target domain, therefore, there should be evidence of ongoing support. Evidence for each of the selection criteria has been found using collective publications for each DSL and associated documentation from the DSL's website (where available). Sections 2.2.1 to 2.2.5 present the selected DSLs and provide a review of the discussed important aspects. Finally section Section 2.2.6 evaluates the DSLs.

2.2.1 RoboChart

RoboChart is a notation for modelling robot software controllers using state machines [19]. A notable feature is RoboChart's formally defined semantics, which enables automated and semi-automated verification. RoboChart's semantics is defined using CSP. This is a notation for describing a system in terms of communicating processes while ignoring the computations internal to each process [60]. The CSP notation along with theories of concurrency enable concurrent systems to be analysed, and as a result, properties such as determinism, deadlock, and livelock can be verified [60].

The main structural elements of RoboChart are as follows:

Module The top level component representing the robot, made up of a robotic platform and controller(s).

Robotic Platform Represents observable interactions between the robot and its environment, providing the variables, events, and operations to represent facilities required by the control software.

Controllers Are composed of at least one state machine and represent parallel behaviour.

State Machines represent predominantly sequential behaviour.

RoboChart encourages reusability through modularity using its structural elements [61]. For instance, robotic platforms are independent of controllers. Therefore, a robotic platform in a model can be interchanged, providing that the replacement robotic platform has corresponding variables, events, and operations the controllers from the model require. Similarly, controllers and state machines are self-contained components that can be independently analysed and developed. This is beneficial for the development large systems helping multiple developers work on different areas simultaneously.

CSP has a Unified Theory of Programming [62] making it possible for RoboChart to support reasoning about additional aspects of robotic systems; for example, probabilistic and continuous behaviour using semi-automated theorem proving [19]. The artefacts that can be generated from a RoboChart model include: a CSP model for model checking and verifying system properties, a probabilistic model for analysis in the probabilistic model checker PRISM [63], and a controller implementation in C++ for simulation and deployment onto the target robotic system.

RoboChart models can be created and modified graphically using the Integrated Development Environment (IDE) RoboTool. The ability of RoboChart to automate aspects of system verification utilising formal methods, combined with its graphical modelling capabilities act to widen the application of formal methods. This enables users with minimal expertise in formal methods to make use of the benefits they provide, such as, automated testing and defect detection [64].

2.2.2 RobotML

RobotML is a notation with the primary goal of addressing the interoperability of robotics software to improve its reusability [14]. RobotML achieves this by abstracting the low-level platform-specific hardware and software implementation details, and automatically generating the system's executable code. This allows the developers of a robotic system to focus on the design of the high-level system functionality.

A notable feature of RobotML is the domain model at its core, which is based on an ontology [65] developed as part of the PROTEUS[3] project. The ontology covers all aspects of robotic systems, and is used to extend the UML metamodel. Therefore, RobotML can be used to model a complete system from its mission through to deployment platform. However, there is only an informal correspondence between RobotML and the ontology it uses, therefore, the semantics of RobotML are not precisely defined.

RobotML models are component-based with ports and connectors representing communication between components. They can be created and modified graphically using an Eclipse Papyrus[4] based IDE.

The main structural elements of RobotML are as follows:

Robotic Architecture The top level package describing the robotic system using Robotic Behaviour and Robotic Communications.

Robotic Behaviour The behaviour of components are modelled using finite state machines or algorithms.

Robotic Communications The communications between robotic systems are modelled as either DataFlowPorts (publish/subscribe) or ServicePorts (request/reply).

[3]Plateforme pour la Robotique Organisant les Transferts Entre Utilisateurs et Scientifiques - http://www.anr-proteus.fr/

[4]https://www.eclipse.org/papyrus/

Robotic Deployment The constructs used in the assignment of the robotic system to the target platform, used for code generation.

RobotML's Platform Independent Model (PIM) is made up of three parts that describe a robotic system: sensors, actuators, and the robot control system. Each of the three parts contains one or more components. For the robot control system, the components represent behaviours of the system and each have an associated finite state machine or algorithm defining their behavior, for example, obstacle detection. The communication between all of the components in the system is represented graphically as edges between component ports.

By definition, the PIM does not specify implementation-specific middleware or simulators. Therefore, in order to generate executable code, RobotML uses a Deployment Platform Model (DPM) to map components from the PIM to implementation-specific middleware and simulators.

2.2.3 SmartMARS

SmartMARS (Modelling and Analysis of Robotic Systems) is the notation for the SmartSoft component-based approach to robotics software development. The SmartSoft approach addresses the reusability of robotics software between developers by separating the development process into two activities: component building and systems integration [15]. To enable software components to be integrated in a compositional manner, strictly defined communication patterns are enforced by the component model [66]. These communication patterns informally define the semantics of SmartMARS.

SmartSoft system-level models are component-based with ports and connectors representing provided and required services between components. They can be created and modified graphically using the Eclipse and Papyrus-based SmartMDSD toolchain.

The main elements of SmartSoft system-level models are as follows:

Component A software element which offers and requires services.

Service The instantiation of one of the enforced interaction patterns by a component.

The behaviour of the components is not described by SmartMDSD models. Instead, a component's behaviour is determined by its source code. The SmartMDSD toolchain enables a component developer to graphically

create the structure of the component for the system-level model, and then generate a code outline to implement the behavior of the component.

Other DSLs are available that can be used in conjunction with SmartMARS to provide additional functionality: for example, task decomposition using SmartTCL (Task Control Language) [67]. Dynamic reconfiguration at runtime is supported by dynamic state charts [68], an extension to state charts [69].

2.2.4 BCM

The BRICS[5] Component Model (BCM) is a collection of notations and an associated design methodology to promote model-driven software development in robotics [16]. In particular the BCM defines a component-based structure that maintains the separation of five concerns; that is, the four concerns defined by Radestock and Eisenbach [70] (communication, computation, configuration, and coordination) with the addition of composition.

BCM models are component-based with ports and connectors representing data-flow, services, events, or properties. The semantics of BCM are informally defined following concepts from other component-based approaches that have been used to develop robotic systems. The BRIDE Eclipse based IDE provides a graphical interface for creating BCM models.

The key elements of BCM are as follows:

Component The top-level element that represents a module of a robotic system. A component has ports and represents a function or behaviour.

Services Are provided or required by a component to perform its function; they are represented as a port of a component.

Events Are provided or required by a component for coordination; they are represented as a port of a component.

Data Flow Is the data provided or required by a component to perform its function; they are represented as a port of a component.

Properties Are configuration parameters required by a component to perform its function; they are represented as a port of a component.

The Component Port Connector (CPC) platform-independent notation

[5] Best Practice in Robotics - http://www.best-of-robotics.org/

supports composability by describing the structure of the software without relying on specific middleware or frameworks.

The BCM provides specialisations of the CPC notation for middleware or framework-specific models. The platform-specific models are created using model-to-model transformations. The resulting platform-specific model is used to generate source code for implementation of the robotic system. Because BCM does not provide a means to model the behaviour of its components, only partial source code can be generated.

2.2.5 V^{3}CMM

The three-view component metamodel (V^{3}CMM) addresses the need for improved processes and tools for the development os robotic systems with increasing levels of functionality, while reducing overall development time and cost [71]. V^{3}CMM uses a platform-independent model and a component-based approach to increase the reusability of robotics software. V^{3}CMM 's notation is a subset of UML selected from Alonso, Vicente-Chicote, Ortiz et al. experiences developing robotic systems. It adopts the semantics of UML; as a result, some aspects of the language are left open. V^{3}CMM models are created using a textual notation with an Eclipse based IDE for model-to-model transformations and source code generation.

A notable feature of V^{3}CMM is its concept of three distinct views, each responsible for a particular part of the system model. The three views are as follows:

Structural View Describes the system's structure using components.

Coordination View Describes the event-driven behaviour of components using state machines.

Algorithmic View Describes the algorithms executed during a state from a state machine, using activity diagrams.

There are two types of component in the V^{3}CMM structural view: complex and simple. Simple components can be associated with a behaviour from the coordination view, as opposed to complex components, which can only act as containers for simple components. This means that the behaviour of complex components is defined by the simple components within it.

The behaviour of simple components is defined through the coordination view using the concepts of UML state machines. The state machines for each component provide concurrent event-driven behaviour. The states of the state machines are defined through the algorithmic view, which uses the concepts of UML activity diagrams restricted to sequential execution.

V^{3}CMM focuses on modelling platform-independent behaviour of a robotic system and does not provide views that require additional platform-specific details such as a tasks [71]. For the same reason, model-to-model transformations for middleware are not provided. Features that require platform-specific details can instead be supported by appropriate model transformations [71].

2.2.6 Evaluation

Table 2.3 summarises the features of the DSLs discussed in the previous sections. All of the DSLs reviewed use a component-based approach to model the software structure of robotic systems and facilitate the reusability of robotics software. Graphical modelling capabilities for creating and modifying models are either available or planned.

Each DSL emphasises a different part of the domain, with each taking a unique approach to address challenges in the development of robotics software: RoboChart models capture the behavior of controllers with respect to services provided by a robotic platform, RobotML models capture a complete system based on an ontology, SmartMARS models capture software components structure and the communications patterns used between components, BCM models capture the software components structure, and finally V^{3}CMM models capture the software components structure including their behavior. As a result of these approaches, distinct features are offered which we now consider.

Both SmartMARS and BCM do not model the behaviour of the software components, instead, source code provided by developers defines the components behaviours. The other three DSLs do model the behavior of components, all providing state machines for this purpose. DSLs that do model component behaviour can generate more complete source code.

RoboChart is prominent in that it is the only DSL that has a formal semantics. This means that properties of the modelled system can be mathematically verified using different techniques, for example, model checking and theorem proving. Furthermore, some parts of the verification is automated through the use of an MDE approach.

Table 2.3: Feature comparison of robotics DSLs

DSL	RoboChart	RobotML	SmartMARS	BCM	V^{3}CMM
Domain Emphasis	Controllers	Systems	Communi-cations	Structure	Components
Component-Based	✓	✓	✓	✓	✓
Graphical Modelling	✓	✓	✓	✓	Planned
Model Includes Behaviour	✓	✓	✗	✗	✓
Code Generation	✓	✓	Partial	Partial	✓
Formally Specified	✓	✗	✗	✗	✗
Automatic Verification	✓	✗	✗	✗	✗
Simulation	✓	✓	✓	✗	✗

RobotML is the most comprehensive DSL and aims to cover all aspects of robotics software development. Consequently this means that it is the largest DSL containing the most elements. BCM is the least comprehensive DSL; therefore it is the most concise and has the fewest elements.

RoboChart's formal semantics and support for automatic verification are distinguishing features that make it particularly suitable for our work. Combined with all of the other features in support of an MDE approach, RoboChart provides a foundation for contributing to the verification of robotic systems.

The next section introduces RoboChart using a simple autonomous lawnmower robot example.

2.3 Modelling Robotic Systems Using RoboChart

This section introduces RoboChart's graphical notation through a simple example of a robotic lawnmower. Section 2.3.1 explains the structure and elements of a RoboChart model for the robot lawnmower, and Section 2.3.2 demonstrates the verification of some properties of the robot lawnmower using the RoboChart model.

2.3.1 Modelling

The diversity and often specialised design of robotic systems mean that many different controller configurations and communication methods are used across the domain. RoboChart provides a selection of features and structural elements that make it well-suited to modelling robotic systems.

RoboChart models specify the controller software of a robotic system as a module element that contains three other main types of element: controllers, state machines, and the robotic platform.

- **Controller** elements of a RoboChart model represent controllers from a robotic system.
- **State machine** elements define the behaviour of the RoboChart controllers in a notation typically used by the developers of robotic systems.
- **Robotic platform** is the element which represents the services provided by a particular robotic system that can be used by the controller software.

A RoboChart module can have several controllers that define the concurrent behaviour of the robotic system. Similarly, the behaviour of a controller can be given by one or more state machines that execute concurrently within a controller. State machines define predominantly sequential behaviour, however, in some cases concurrent behaviour is possible, for example, during actions with composite states. To create a RoboChart model of a robotic system it is important to understand the foundational concepts of RoboChart that relate its elements.

RoboChart models are event-based where events can be related to the result of a stimulus from the environment detected by a sensor or to a request from the software to use an actuator. Events only indicate their

occurrence so to communicate values they can have an associated type that provides additional information, for example, the temperature value for a sensor that detects temperature. Events are depicted as boxes □ on the boundaries of elements.

RoboChart operations take parameters and return no value; instead they can affect the state of the robotic system. Operations can be used to represent API calls to the robotic platform, or they can be defined using state machines and provided by controllers. Operations are depicted with its signature prefixed with the symbol **O**.

RoboChart controllers, state machines, and robotic platforms can have variables. Variables can optionally be defined as constant preventing them being modified. Many commonly used primitive types are supported by RoboChart variables including: natural numbers, strings, integers, booleans, and real numbers. Custom primitive types can be created, however, they remain unspecified. Other types supported are enumerations, product types, and datatypes that are made up of fields of types. Finally sets and sequences are also supported. Variables are depicted within elements using an identifying name prefixed with the symbol **x** for variables and **π** for constants.

Related events, operations, and variables can be grouped into an interface. The interface can then be used to specify the events, operations, and variables that a RoboChart element uses. RoboChart has three types of interface: provided, required, and defined. Required interfaces indicate functionality that a controller requires from the robotic platform in order to perform its function and can only contain variables and operations. Provided interfaces indicate the functionality that the robotic platform provides to controllers. Therefore, conceptually provided interfaces are the reverse of required interfaces and similarly can only contain variables and operations. Defined interfaces indicate the events and variables that an element uses to perform its function. Where interfaces are used they are depicted within elements using an identifying name prefixed with a symbol indicating the interface type. The symbols used to indicate interface type are as follows: provided interfaces Ⓟ, required interfaces ®, and defined interfaces ⓘ.

The support RoboChart provides for interfaces makes the services a controller requires from the platform in order to operate clearly visible. Additionally, the support provided by defined interfaces to group together related events and variables further contributes to the structure of RoboChart models and their comprehensibility.

The flow of events is depicted by the connections between the robotic

platform, controllers, and state machines shows the relationship between the elements that they connect. This is useful when modelling more complex systems, particularly where there are multiple controllers and state machines, in being able to understand the structure of the robotic system and the distribution of data among the components. RoboChart supports both synchronous and asynchronous connections that follow the commonly used communication approaches of the robotics domain. Asynchronous connections are labelled as 'async' whereas synchronous connections are unlabelled.

A notable feature of RoboChart state machines is the support for the specification and verification of time properties. The timed features include the concept of clocks, budgets, and deadlines. Clocks record the number of time units that have passed since the clock was last reset; they are represented with an identifying name prefixed with the ◷ symbol. The budgets and deadlines provide a way to specify how long an action can take or the amount of time required for a transition trigger to occur.

State machines define behaviour using states, junctions, and the possible transitions among them. The states and transitions make use of the other elements that make up state machines which are: events, variables, required interfaces, defined interfaces, and clocks; these other elements can be used by the states and transitions to accomplish the state machines function.

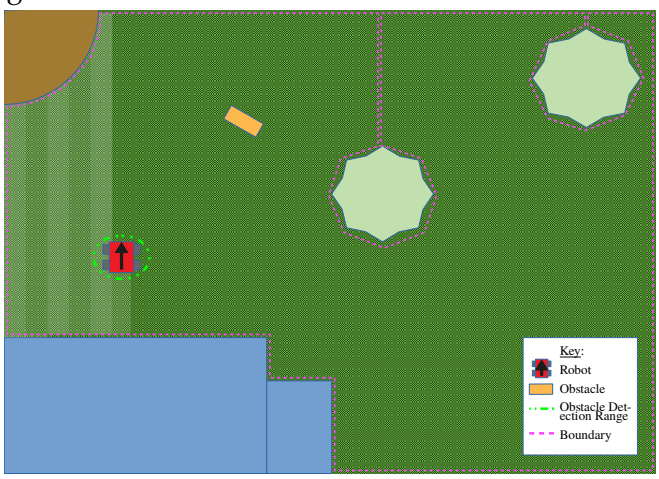
The actions of states can be specified as being executed on entry, during, or on exit of the state. Actions are defined using a simple action language which contains among other things: operation calls, conditionals, event input and output, and assignments [72, p. 25]. States can also be composite and so contain a state machine that is executed when that state is entered.

Transitions connect states and junctions and they can have any combination of triggers, guards, or action statements that specify the conditions when a transition will occur. Triggers cause a transition to be taken on the occurrence of a particular event. Optional start and end deadlines can be given to triggers supporting the specification of time properties of the system. The guards are a boolean expression that only allow a transition to be taken when it evaluates to true, providing greater control over the transitions between states. The action statement enables any required actions to be executed on the occurrence of a transition.

Now the features of RoboChart have been introduced we will apply them to create a RoboChart model of a simple autonomous lawn-mowing system.

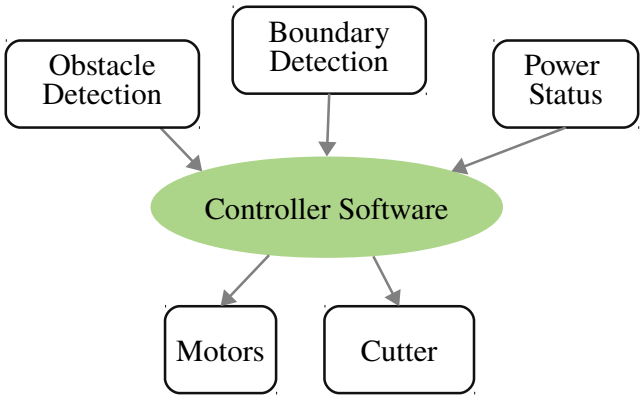
The robotic platform for the lawn-mowing system is a robot that is four wheeled, differentially driven and battery powered. The battery is charged using a solar panel on top of the robot. Additionally, the robot has a cutter, which can be switched on or off, for performing its main function of cutting the grass. The lawn-mowing system is expected to operate autonomously in a typical residential garden as shown in Figure 2.6. To keep the robot within a specific area of grass, the user must install a boundary wire. Inside the robot's area of operation there could be obstacles that the robot must avoid.

Figure 2.6: The environment of the lawnmower robot.



To enable autonomous operation in its environment the robot features two types of sensor: a Very Low Frequncy (VLF) sensor for detecting the boundary wire, and ultrasonic sensors for obstacle detection. The software controllable inputs and outputs of the lawnmower robot are shown in Figure 2.7. An overview of the lawnmower's behaviour is as follows. When the charge in the battery is sufficient the robot moves forward with the cutter enabled. When a boundary is reached the robot turns around to cut the next strip of grass parallel to the last. If the robot encounters an obstacle it turns to avoid the obstacle and then continues moving forward. For safety the lawnmower has bumper switches on each side that if activated disconnect the power; this requires a user to reset and is not under control of software.

Figure 2.7: The inputs and outputs of lawnmower robot's controller software.



As introduced earlier important elements of a RoboChart model are the robotic platform, controllers, and state machines. Therefore to model the lawn-mowing system a logical starting point is the robotic platform and the structure of controllers of the system.

To model the robotic platform of the lawn-mowing system the services that the lawnmower robot provides must be described using events, operations, and variables. Inputs to the robot's controller software originate from two places either from the environment via the robot's sensors, or internal state information provided by the robot's hardware. All of the inputs to the controller software can be modelled using events.

The events from the environment include: a boundary event that occurs when the lawnmower reaches the boundary wire, and an obstacle event that occurs when an obstacle has been detected along the robot's path. The internal events from the robot's hardware include a lowPower event that occurs when the battery level is too low to cut grass, and a fullPower event that occurs when the battery has been fully charged by the solar panel. Because both the lowPower and fullPower events relate to power they will be defined together in an interface. Now the inputs to the controller software that the robotic platform provides can be modelled, the outputs of the controller software need to considered.

The lawnmower robot software API provides methods for controlling the robot's cutter and motors. The methods from the software API can be modelled using operations and organised into two interfaces MotorControl and CutterControl which reflect the robot's actuators. The

interface definitions can be found in Figure 2.8.

Now that all of the services the lawnmower robotic platform provides can be modelled, the controller structure of the system must be considered. To minimise the cost of the lawnmower robot it has a single embedded microcontroller, therefore, this can be modelled as a single controller in the RoboChart model which we will call Mower. Because there is only a single controller in this system it must handle all events and require all of the interfaces from the robotic platform in order to fulfil its lawn-mowing function.

Figure 2.8: The interfaces and the data types of the lawnmower robot.

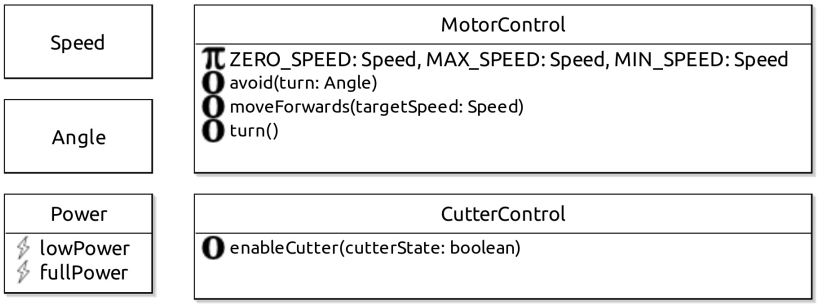


Figure 2.9 shows the RoboChart module for the lawnmower system; it is made up of the Lawnmower robotic platform and the single controller Mower. The directionality of the events is indicated by the connection arrows between events of the controller and the state machine. It is worth noting that the event names for connected events between the elements can be different, however, the types associated with connected events must match.

Now that the controller structure has been specified the structure of state machines that make up the Mower controller must be decided. Because the behaviour of the lawn-mowing system is simple and consists of managing the cutter and movement that requires no concurrent control, a single state machine can be used to define the behaviour of the Mower controller. Figure 2.10 shows the Mower controller and the its state machine which has been called MowManager.

Figure 2.9: The module of the lawnmower robot.

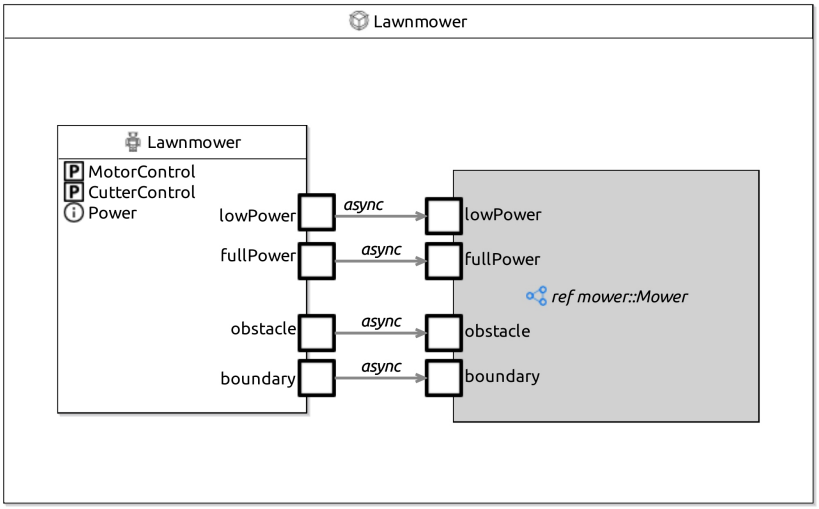
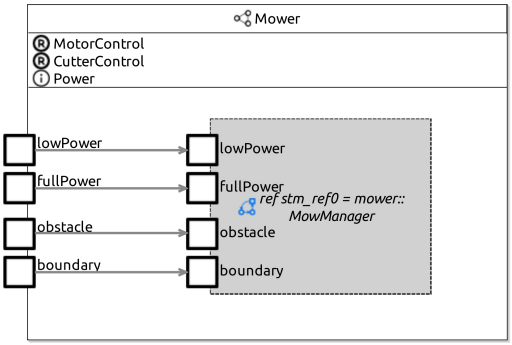


Figure 2.10: The controller of the lawnmower system.



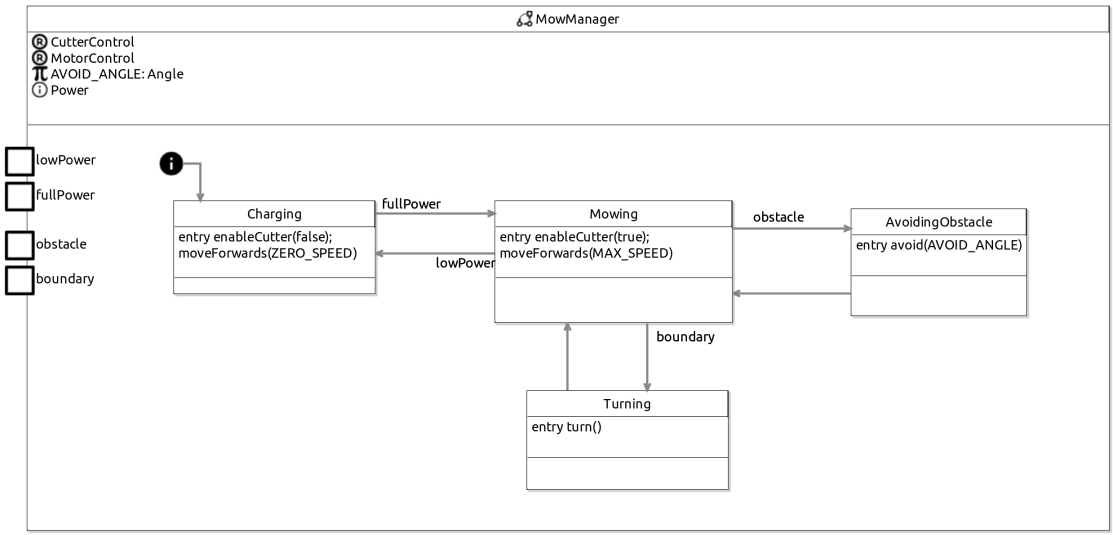
The MowManager machine requires all of the interfaces and must handle all of the events of the system, because, a single controller and a single state machine are being used to model the system. The behaviour of the lawnmower robot can be modelled using four states: charging, mowing, avoiding obstacle, and turning. Figure 2.11 shows the MowManager state machine that implements the systems behaviour.

In addition to the required and defined interfaces a constant AVOID_ANGLE is defined, indicated by the π symbol, which defines a constant of type Angle. The AVOID_ANGLE constant is used as the angle for obstacle

avoidance; its exact value is not specified in this model.

The Charging state from Figure 2.11 is the initial state indicated by the special junction. On entering the charging state, the cutter is disabled using the enableCutter operation; subsequently the robot is stopped using the moveForward operation.

Figure 2.11: The state machine of the lawnmower robot.



The Charging state from the MowManager machine in Figure 2.11 will transition to the Mowing state when a fullPower event occurs. Similarly, when in the Mowing state and a lowPower event is received, the machine will transition back into the Charging state. The complete behaviour of the MowManager machine can be summarised as follows:

- The robot will be stationary with the cutter switched off in the Charging state, until the fullPower event is triggered.
- On triggering of the fullPower event the Mowing state will be entered, the cutter will be enabled and the robot will begin moving forward.
- If the robot is in the Mowing state and a boundary event is triggered, the robot will enter the Turning state. When the turn has completed the Mowing state will entered.
- If the robot is in the Mowing state and an obstacle event is triggered,

the robot will enter the AvoidingObstacle state. Once the obstacle has been avoided the Mowing state will be entered.

- If the charge runs low when when in the Mowing state then the Charging state will be entered and the cutters will be switched off and the robot will stop.

The next section demonstrates some of the properties that can be verified of using the RoboChart model of the lawnmower robot just discussed.

2.3.2 Verification

The RoboTool IDE automatically generates a CSP model from the RoboChart model enabling verification of a system's properties. Basic properties can be automatically verified and other properties of interest can be included in the verification following manual specification in CSP. For the lawn-mowing system the generated CSP model consists of 54 files and 1,813 lines of code. The CSP model in combination with a set of assertions can be used to automatically verify the following basic properties of the lawn-mowing system:

Determinism that the resulting behaviour of a module, controller or state machine only depends on the inputs provided to it.

Divergence Freedom that a module, controller or state machine does not enter an infinite loop of performing only internal actions.

Deadlock Freedom that a module, controller or state machine does not get to a point where it is unable to progress refusing all interactions.

Termination that an operation, state machine, or controller can successfully complete reaching a final state.

Reachablilty that it is possible to enter the state of a machine with some combination input events.

The set of assertions to be verified are specified using RoboTools's assertion language, which is a controlled natural language based on English. For example, to check the MowManager machine is deterministic the corresponding statement in the RoboTool assertion language is:

assertion MM_1: mower::MowManager **is deterministic**

Keywords are highlighted in boldface and MM_1 is a user-defined label given to the assertion for easy identification. The full specification for the assertion language format is given in [73].

From the set of RoboChart assertions RoboTool automatically generates corresponding CSP assertions. The generated CSP assertions along with the CSP model can then be used to verify the specified software properties using a model checker such as FDR [74].

In addition to the basic properties, other properties interest can be verified by specifying the custom property as a CSP expression, and then refinement checking can be used to verify the CSP expression against the CSP model. For example, the lawnmower system should avoid an obstacle when an obstacle is detected, this can be expressed as follows.

$$
\begin{aligned}
&propertyAvoidObstacle = \\
&\quad \text{let} \quad Wait = Recurse(waitevents, Wait) \\
&\qquad\qquad \Box \\
&\qquad\qquad mower\_MowManager\_obstacle.in \longrightarrow Avoid \\
&\\
&\qquad\qquad Avoid = avoidCall \longrightarrow avoidRet \longrightarrow Wait \\
&\quad \text{within} \quad Wait \;|||\; RUN()
\end{aligned}
$$

where $waitevents = \{|mower\_MowManager\_boundary.in, mower\_MowManager\_fullPower.in, mower\_MowManager\_lowPower.in, enableCutterCall, enableCutterRet, moveForwardsCall, moveForwardsRet, avoidCall, avoidRet, turnCall, turnRet|\}$. The *propertyAvoidObstacle* defines a CSP process that accepts all of the events from the MowManager machine given in Figure 2.11. All of the events that do not relate to the avoidance property of interest are given by *waitevents*. When any of the *waitevents* occur the process recurses and waits for another event. On a *mower_MowManager_obstacle.in* obstacle event occurring, the avoid operation should be called and subsequently return, signified by *avoidCall* and *avoidRet*. After the avoid operation returns, the process goes back to accepting all events from the MowManager machine. The assertion can be now be specified as a refinement check *propertyAvoidObstacle* against the MowManager machine that can be expressed using the RoboChart assertion language as:

assertion MM_9: mower::MowManager **refines**
propertyAvoidObstacle **in the traces model**

The complete assertions file used for verifying the lawnmower robot and untimed results can be found in Appendix B.

In summary this section has provided a brief introduction to some of RoboChart's core modelling and verification features. The next section concludes the chapter by documenting our considerations on modelling robotic systems and their architecture using RoboChart.

2.4 Final Considerations

This chapter has surveyed some robotics specific architectures and DSLs. It has found that RoboChart stands out from other DSLs for its formally defined semantics and support for automated and semi-automated verification. Coupled with support for simulation and modelling the timed and probabilistic behaviour of robotic systems, RoboChart provides a basis for their improved verification utilising formal techniques.

Also evident is the use of layers in robotics architectures to separate high-level planning and decision making from the lower-level functional layer. The layered structure of robotics architectures vary in number and composition. Furthermore, different control techniques are used by the functional layer's of robotics architectures. Out of the architectures reviewed, either a service-based or a behavioural-control technique was used by the functional layer. It needs to be possible to model these different functional layer control techniques using RoboChart, to ensure real robotic systems can be modelled and verified.

Case studies will be used to evaluate RoboChart's support for modelling. To make RoboChart accessible to developers guidelines will be provided covering the use of layers in RoboChart models, as well as, the different control techniques of the functional layer. Our work will facilitate developers model their system and subsequently verify it, taking advantage formal techniques which are typically considered the preserve of those with expertise in formal methods.

The next chapter defines some initial guidelines based upon the modelling work carried out so far.

3 Guidelines

This chapter comprises of the RoboChart guidelines for developers. These guidelines have been created from the modelling experience obtained while using RoboChart, in particular the autonomous vehicle prototype system that is documented in Chapter 4.

The guidelines are separated into two distinct categories: usage and modelling. The usage guidelines cover the structure and naming of RoboChart models and can be found in Section 3.1. The modelling guidelines cover best practices in relation to the elements of RoboChart models and can be found in Section 3.2. Finally Section 3.3 evaluates the guidelines with respect to aims and objectives set out in section 1.3.

Some guidelines also have a positive effect on the use of RoboTool in particular, and we comment on this when it is the case.

3.1 Usage

This section covers the structure and naming of RoboChart models.

3.1.1 Naming

Guideline 1:1:1 An element's name should not provide any indication to its type in the metamodel.

Rationale:
RoboTools graphical notation differentiates each element of the metamodel using distinct symbols and structure, therefore, this information does not need to be included in element names. For example, using prefixes such as 'ctrl' for all controllers and 'stm' for all state machines of a model is not recommended.

Guideline 1:1:2 A naming convention should be used for each of the RoboChart elements.

Rationale:
Adopting a consistent style for different RoboChart elements improves the clarity of the model by making the elements types readily identifiable. See Tables 3.1 and 3.2 for a suggested naming convention.

Table 3.1: Naming conventions for RoboChart elements

Element	**Style**	**Example**
Packages	Lowercase with words separated using underscores.	movement_control
Controllers, Data types, Enumerations, Functions, Interfaces, Operations, Primitives, State Machines, Robotic Platform, States	Upper camel case.	CheckForMovement
Events, Variables	Lower camel case.	requestedDemand
Constants	Uppercase with words separated using underscores.	TURN_ RADIUS
Transitions	[source]To[destination] where source and destination are states in the state machine. [1]	StoppedToMoving.

[1] For states that have multiple transitions to the same state it is necessary to append an extra descriptive identifier after the destination.

Table 3.2: Naming conventions for RoboTool files

File	Style	Example
RoboChart (.rct)	*[sub-element].rct* where sub-element is the name of a member from a package.	ObstacleAvoidance.rct
Representation (.aird) for packages	*[package name].aird*	movement_control.aird

3.1.2 Project Structure

Guideline 1:2:1 Each controller, state machine, and operation should have its own RoboChart file.

Rationale:
Using separate files for RoboChart models means that each file contains information relating to only to one particular aspect of the system; therefore, larger models are easier to develop and maintain. Additionally, reusability is improved because different parts of the model can be taken isolation.

Guideline 1:2:2 Packages should be used to group functionally related elements of a model together. As a minimum, each controller should have its own package.

Rationale:
Packages reduce the likelihood of naming conflicts occurring and act to encapsulate elements, and therefore, facilitate reusability.

Guideline 1:2:3 Each package should have its own folder containing its state machines, operations, and types.

Rationale:
Organising the model into separate folders makes it easier to maintain, because related parts of the model are grouped together.

3.2 Modelling

This section covers the best practices in relation to the elements of RoboChart models.

3.2.1 General

Guideline 2:1:1 Operations, variables, and events should be independent.

Rationale:
Operations provided by the robotic platform model the interface between the software controllers and the hardware of the robot. This means that operations have side effects. Consequently, if multiple operations that have overlapping areas of control are called at the same time, the resulting robot behaviour may be unexpected. For example, calling simultaneously an operation to increase speed and another to set speed to a specified value, can have an effect that depends on how the robot's Application Programming Interface (API) for dealing with the motors is implemented.

3.2.2 Data Types

Guideline 2:2:1 RoboChart primitives should be used to model primary data types of the robotic system.

Rationale:
To avoid early over-specification, data types should be kept as abstract as possible when modelling a robotic system in RoboChart. The RoboChart primitives can be mapped across to types which are suitable for implementation in further iterations of the model.

3.2.3 Robotic Platform

Guideline 2:3:1 A robotic platform's events should not imply a specific underlying technology or implementation.

Rationale:
The events of a robotic platform should be general, describing the principle communications and their associated types. For example, a robotic system that uses LIDAR sensors should use an event named something like PointCloud3d. This is to promote the interchangeability and reuse of controllers which make up the robotic system.

3.2.4 Variables

Guideline 2:4:1 The use of shared variables should be avoided.

Rationale:
When accessed concurrently shared variables can introduce race conditions and synchronisation issues. In most cases events should be used to communicate the the required information among controllers, state machines, and the robotic platform.

3.3 Evaluation

The guidelines defined so far cover the general use of RoboChart, so that, the created models are clear and avoid constructs that are difficult to verify. However, the guidelines do not yet fully cover the modelling of robotic systems. Therefore, further work is necessary to expand the guidelines presented here based upon future case studies.

The next chapter documents the first case study of an autonomous vehicle prototype that will be used for evaluating RoboChart.

4 Autonomous Vehicle Case Study

This chapter documents the autonomous vehicle case study. We use RoboChart to model UK Connected Places Catapult's[1]

self-driving pod[2] prototype control software from their proprietary source code. The chapter has four sections: Section 4.1 provides a detailed description of the system and its components, Section 4.2 documents the development and structure of the RoboChart model, Section 4.3 documents the verification of properties of the system using the RoboChart model, and finally Section 4.4 evaluates the case study in relation to the objectives.

4.1 System Overview

This section introduces the Basic Autonomous Control System (B-ACS) of the pod prototype and its components. The control software is implemented using C++ and the ROS middleware and targets the LUTZ Pathfinder pod vehicle[3].

Our analysis has been made using the following resources from Connected Places Catapult:

i. LUTZ Pathfinder - systems design specification [75]

ii. LUTZ Pathfinder - electrical system specification [76]

iii. B-ACS ROS node diagram [77]

iv. TSC_ACS-Master source code [78]

The next subsections provide an overview of the system, details of the controller software, and properties of the software that are important to verify.

[1] Formerly, Transport Systems Catapult

[2] https://ts.catapult.org.uk/innovation-centre/cav/cav-projects-at-the-tsc/self-driving-pods/

[3] https://ori.ox.ac.uk/projects/lutz-self-driving-pods/

4.1.1 Overview

The B-ACS directs a pod vehicle around a predetermined route, specified by a sequence of latitude and longitude points. The control software is responsible for ensuring the vehicle follows the route, and keeps within the speed limit for the current location.

The system requires a safety driver for reacting to any obstacles that may be present and for the pre-emption and mitigation of hazardous situations. The safety driver can override automated control of the system by limiting the pod's speed or stopping the pod altogether.

The primary aim of prototype system is for use as a data collection platform to enable the evaluation of sensors in a fully autonomous system. Therefore, the prototype system is not truly autonomous.

Figure 4.1: Inputs and outputs of the autonomous pod control software.

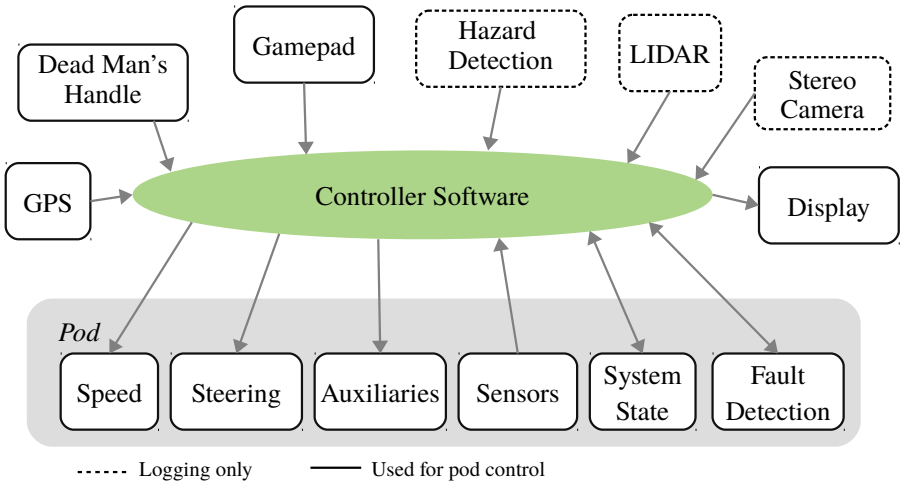


Figure 4.1 provides an overview of all the inputs and outputs available to the software controller of the pod prototype system. LUTZ Pathfinder pod vehicles provide a drive-by-wire interface, enabling software control of the vehicle through its onboard sensors and actuators. These inputs and outputs are shown in Figure 4.1 grouped together inside the shaded rectangle. In addition to the pod vehicle's onboard sensors, a range of other sensors are necessary for the prototype pod system's functionality

and are added to the LUTZ Pathfinder.

GPS The current latitude and longitude of the pod.

Dead Man's Handle The safety driver's speed limiting control.

Gamepad Manual commands for pod control.

Pod Speed The speed the pod currently maintains.

Pod Steering The steering angle the pod currently maintains.

Pod Auxiliaries The state the pod's peripheral devices are set to, for example, indicators and horn.

Pod Sensors The state of the pod's various internal sensors.

Pod System State The state id of the pod's embedded software, another state can be requested.

Pod Fault Detection A request-response handshake has to be maintained by the controller software with the pod's embedded software.

Hazard Detection Early hazard warnings.

LIDAR 3D points and object classifications of the environment.

Stereo Camera Stereo images of the environment.

Display Provides instructions to the user of the autonomous pod system.

The inputs outside of the grey box, shown in Figure 4.1, represent the sensors that are not part of the pod vehicle. Some of the additional sensors are not used for control of the vehicle and are represented in Figure 4.1 as inputs with dashed borders. The inputs that are not used to control the pod vehicle are instead recorded and logged for later analysis of the sensors' performance.

The controller software runs on a computer that has an Intel® Core™ processor, a solid state disk drive, and Ubuntu 16 Linux Operating System (OS). A Linux based OS is required because of the software controller's use of ROS.

4.1.2 Controller Software

The use of ROS middleware strongly influences the structure of the controller software; it is made up of many modules. Each module is executed concurrently as an individual process known as a ROS node. The ROS nodes typically communicate with each other using asynchronous messages via a publish and subscribe mechanism provided by ROS. The data structure of the messages communicated are specified using the ROS message definition specification format [79] which lists the data fields and their name.

Table 4.1: Namespaces of the self-driving pod system software controller.

Namespace	**Description**
b_acs	Contains the ROS nodes, message definitions, and classes used to generate the autonomous pod demand from the pod's location.
cavlab_core	Contains ROS nodes, message definitions, and classes used to process demands. The demand processing consists of switching, limiting, and age checking.
lutz	Contains ROS nodes, message definitions, and classes used to interface with the pod vehicle, and receive and manage the pod state, translating demands into control messages, and managing the state of the pod.
cavlab_hardware	Contains ROS nodes, message definitions, and classes used to interface with the hardware.

The namespaces used for the system provide an indication as to the top level structure of the controller software. Table 4.1 shows the namespaces used. They can be grouped into three categories: those relating to demand creation and processing (b_acs and cavlab_core), those handling the pod and its states (lutz), and those providing interfaces to hardware (cavlab_hardware).

The behaviour of the pod system's controller is determined by the function of each ROS node and the messages communicated among them. Therefore, the source code for each ROS node needs to be analysed to ascertain its function so that an equivalent RoboChart model can be created. Appendix C.1 lists the ROS nodes for each namespace and provides a description of their behaviour. The ROS node diagram [77] graphically shows how these nodes are connected and separates them

into three groups: b_acs, lutz, and nodes that receive sensor data not used for control of the pod vehicle. For each of the ROS node groups, the following Figures 4.2 to 4.4 present the nodes that form the group and the messages communicated among them.

Figure 4.2: The ROS nodes of the lutz group, adapted from [77] [78]

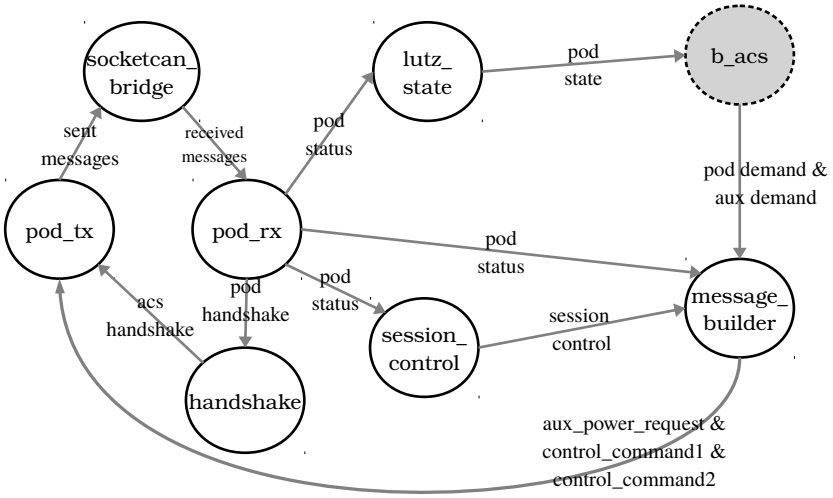


Figure 4.3: The ROS nodes of the data_logging group, adapted from [77]

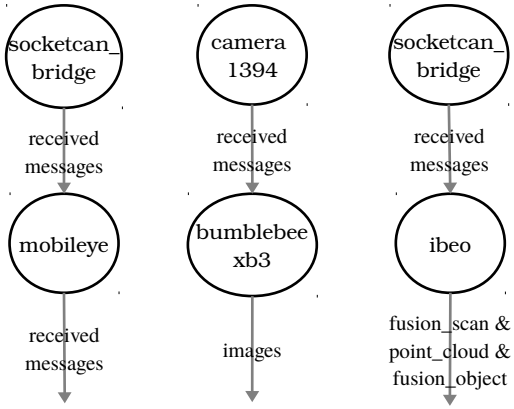
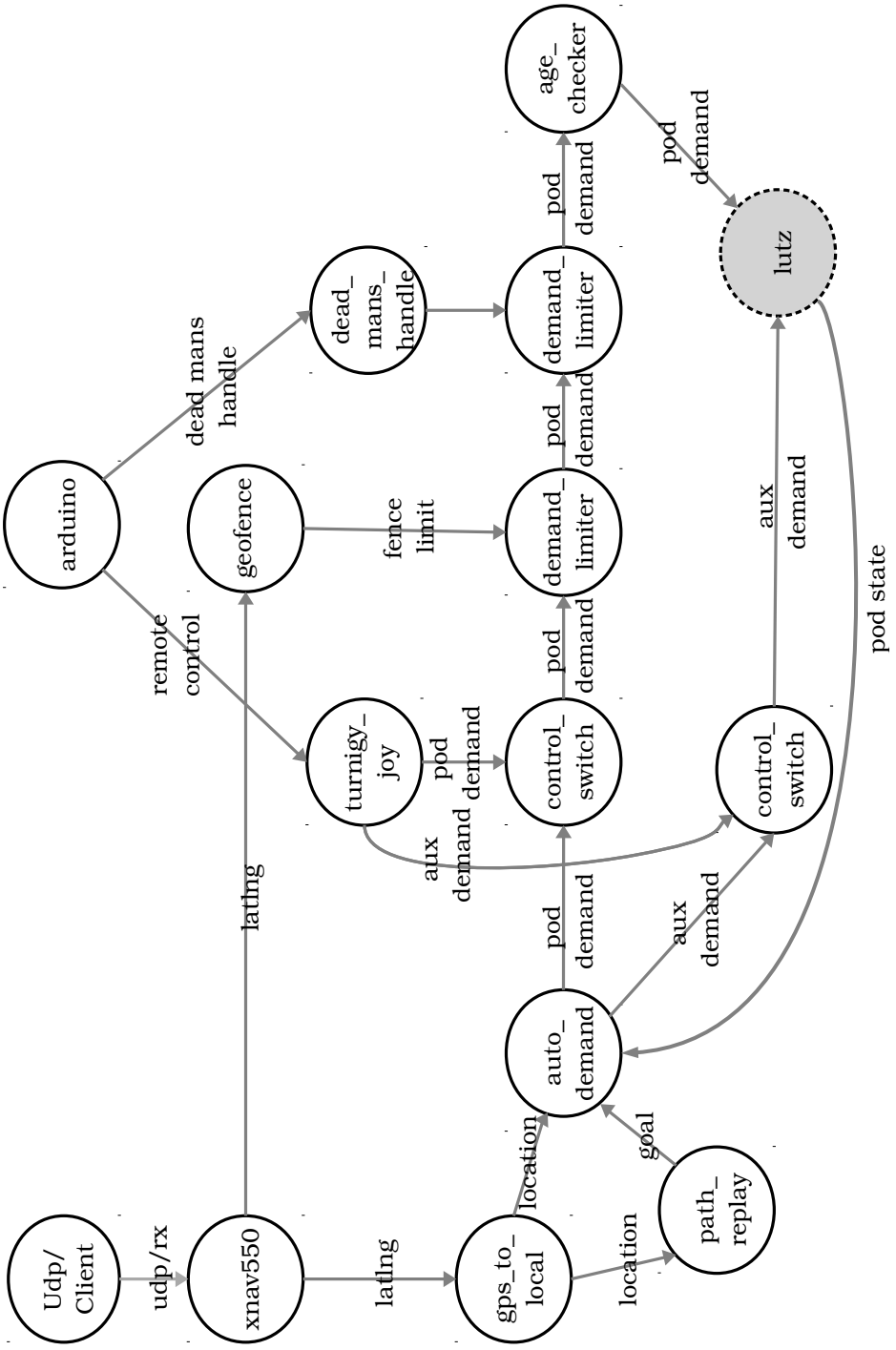


Figure 4.4: The ROS nodes of the b_acs group, adapted from [77] [78]



4.1.3 Important Properties

As already explained, RoboChart models enable the automatic generation of Communicating Sequential Processes (CSP) source code for verifying important properties of the modelled system. These properties can be separated into two main categories: timed properties and untimed properties. The untimed properties that can be verified include: termination, deadlock freedom, divergence freedom, determinism, and state reachability [19]. The timed properties that can be verified include: timelock freedom, zeno freedom, and all of the untimed properties with the exception of deadlock freedom [19].

Deadlock, divergence, and determinism have the potential to affect the reliable operation of the developed system. The resulting issue and the severity of its undesirable behaviour depends on the particular area of software where one of these properties occurs. For example, if the dead_mans_handle node were to diverge and stop responding to the safety drivers input, then it would reduce the safety drivers ability to control the pod in a hazardous situation. Termination may or may not be a desirable property depending on the software processes function. Reachability is an indicator of redundant functionality that cannot be used and may mislead the developers that maintain the system, therefore all states should be reachable.

The next section documents the development of the RoboChart model for the pod system.

4.2 Implementation Model - Development

This section presents the driverless pod RoboChart model. Section 4.2.1 outlines the overall structure of the model. The remaining subsections present the robotic platform and controllers of the system: Section 4.2.2 covers the robotic platform, Sections 4.2.3, 4.2.4, and 4.2.5 cover the controllers and their state machines. The data types used to model the pod system reflect those used in the provided source code [78]; Appendix C.2 documents all of the types defined by the RoboChart model.

4.2.1 Overall Structure

As already said, the robotic platform of the RoboChart model must capture the observable interactions between the robot and its environment. In doing so, it defines the variables, events, and operations available to

the controllers. For the autonomous pod, the robotic platform represents the pod vehicle including all of the additional sensors for control and evaluation as shown in Figure 4.1. The sensors and actuators of the autonomous pod system directly relate to the interactions available to it, therefore, they indicate the operations and events the robotic platform provides. Section 4.2.2 details the analysis of the inputs and outputs for the autonomous pod controller software and defines the RoboChart robotic platform.

Because RoboChart controllers are able to describe concurrent behaviour, they can represent either the individual ROS nodes of the autonomous pod system, or higher level functionality provided by groups of ROS nodes selected from Figures 4.2 to 4.4.

Representing each ROS node as an individual controller would mean that there are more controllers at the top level of the RoboChart model. This makes it more difficult to understand the resulting behaviour as the number of ROS nodes increases.

Representing functionally related groups of ROS nodes as controllers not only reduces the number of controllers at the top level, but also emphasises the coupling between related areas of functionality and, therefore, facilitates understanding of the resulting behaviour. However, this means that ROS nodes are represented as RoboChart state machines, so their behaviour is predominately sequential.

The three groups of ROS nodes given by Figures 4.2 to 4.4 will be used to form the controllers: b_acs, for the autonomous control nodes; lutz, for the vehicle specific control nodes; and data_logging for sensors that are not used for controlling the pod. This grouping also corresponds to the namespaces that contain the ROS nodes, with the b_acs and cavlab_core namespaces mapping to the b_acs group, the lutz namespace mapping to the lutz group, and the cavlab_hardware namespace mapping to both the b_acs and the data_logging groups.

The mapping between node groups and namespaces is not one-to-one because each provides a slightly different view of the system. The ROS node diagram shows only the structural ROS elements and relationships, compared to, the namespaces which are concerned with organising the implementation. Because namespaces organise the implementation they have a lower level view of the system, and therefore they are less abstract and contain more detail. For instance, the b_acs group is divided into two namespaces b_acs and cavlab_core, there is no discernable reason for this structure in the source code and no rationale is given by the provided documentation. Therefore, this structure may have been used

by developers to facilitate development.

The design of each controller is discussed in detail in the next sections.

4.2.2 Robotic Platform

The sensors of the system are modelled as inputs and the actuators as outputs of the controller software; Figure 4.1 shows these inputs and outputs. Table 4.2 and Table 4.3 assign names to corresponding elements that are to be used in the RoboChart model.

Table 4.2: Mapping from the pod system's inputs to the RoboChart robotic platform.

System Input	Name in the Model
Pod - Sensors	status epsOutboard parkBrake battery powerTrain eps indicator epsInboard
Pod - Fault Detection	faultDetectRequest faultDetectResponse
Pod - System State	controlCommand1
GPS	location
Dead Man's Handle	safetyDriverInput
Gamepad	remoteControl
Hazard Detection	earlyWarning
LIDAR	environmentPointCoud environmentObjects
Stereo Camera	environmentImage

Continued on next page

Table 4.2 – continued from previous page

System Input	Name in the Model
Configuration Files	configGoal configController configAutonomousDemand configGeofence configDeadMansController

Table 4.3: Mapping from the pod system's outputs to the RoboChart robotic platform.

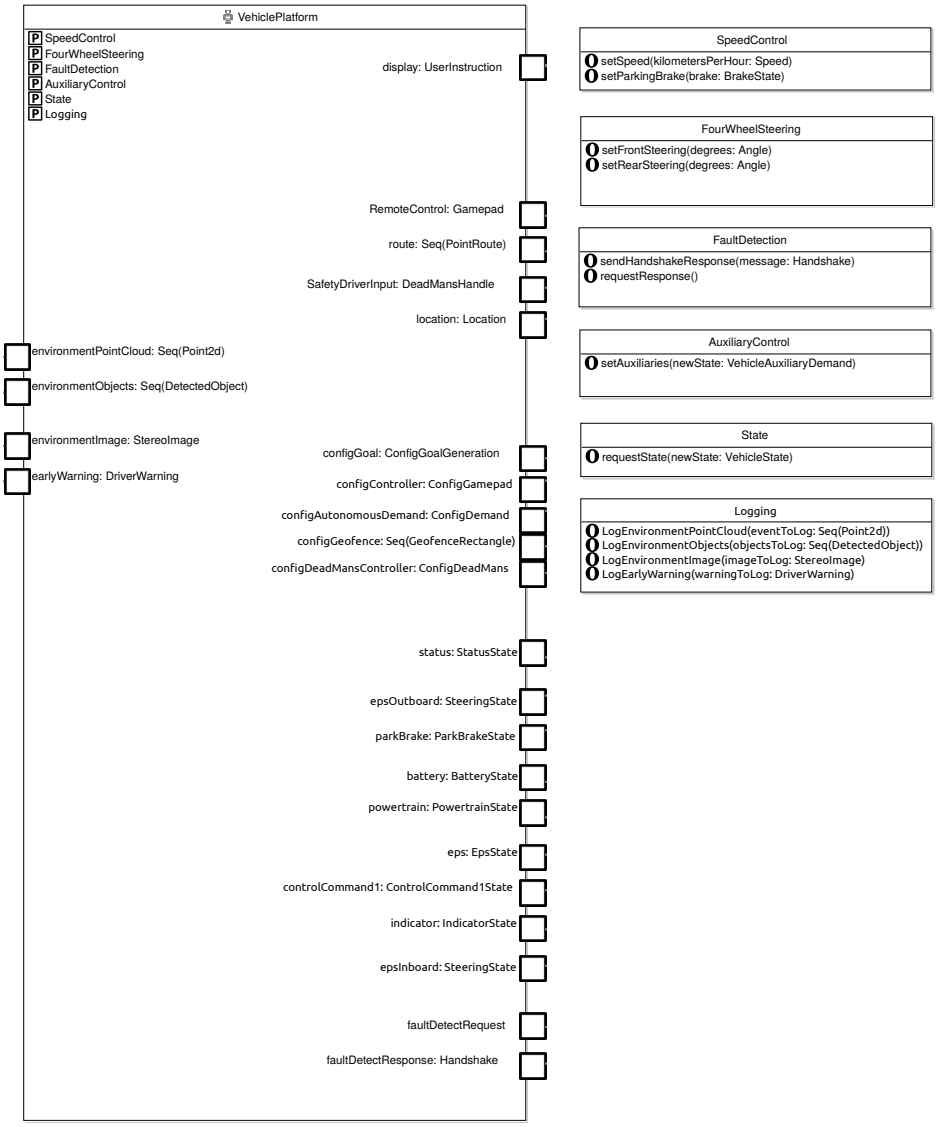
System Output	Name in the Model
Pod - Speed	setSpeed setParkingBrake
Pod - Steering	setFrontSteering setRearSteering
Pod - Auxiliaries	setAuxiliaries
Pod - System State	requestState
Pod - Fault Detection	sendHandshakeResponse requestResponse
Display	display

The inputs given in Table 4.2 provide information about the robot's state or environment, therefore, the element names can be mapped onto events.

The pod's outputs given in Table 4.3 alter the state of the system, this means that the element names can be mapped onto events or operations. For this case study outputs that significantly affect state of the system, for example, the movement of the pod will be modelled as operations grouped into interfaces. Because the display does not affect the state of the system it will be modelled as an event.

Figure 4.5 shows the robotic platform of the pod system and the definitions for each of its provided interfaces.

Figure 4.5: The robotic platform of the RoboChart model and its provided interfaces.



4.2.3 Controller - b_acs

To create the b_acs RoboChart controller, the ROS nodes relating to the b_acs group have to be represented using state machines. Some of the ROS nodes provide low-level connectivity to hardware devices. These nodes do not need to be modelled because of the abstraction provided by the robotic platform. Table C.5 lists all of the nodes in the b_acs group from Figure 4.4 and how each is represented in the b_acs controller model.

The UDP/client node is not modelled because it provides low-level connectivity to network devices abstracted by the robotic platform. The xnav550 and arduino nodes translate low-level sensor data into ROS messages; these nodes are modelled as part of the robotic platform as input events. All other nodes contribute towards the functionality of the controller and therefore are modelled as state machines.

In order to create the state machines that represent the ROS nodes, the message topics that each ROS node publishes and subscribes to, and the states for each state machine must be identified. This is because the messages are the primary means of communication among the ROS nodes. Table C.6 identifies and lists the ROS messages and their source and destination nodes. The ROS message data structure for communication over the ROS topics can be modelled directly using RoboChart data types and fields. For example, the LatLngHeadingFix message consists of three doubles representing a latitude, longitude, and a heading this can be represented using a RoboChart data type with three fields of type real. The behaviour of the ROS publish-subscribe messaging can be modelled using RoboChart asynchronous events with an associated data type to hold the message information.

Figures 4.6 to 4.8 show the resulting autonomous control RoboChart controller; each of the state machines can be found in the appendices from Figures C.3 to C.12. The events connecting the state machines represent the ROS messages given by Table C.6.

The states for each RoboChart state machine are determined by control flow analysis of the autonomous vehicles ROS nodes source code. Tables C.7 to C.15 detail the methods for each of the modelled ROS nodes as part of the analysis.

Figure 4.6: The b_acs controller; part 1 of 3

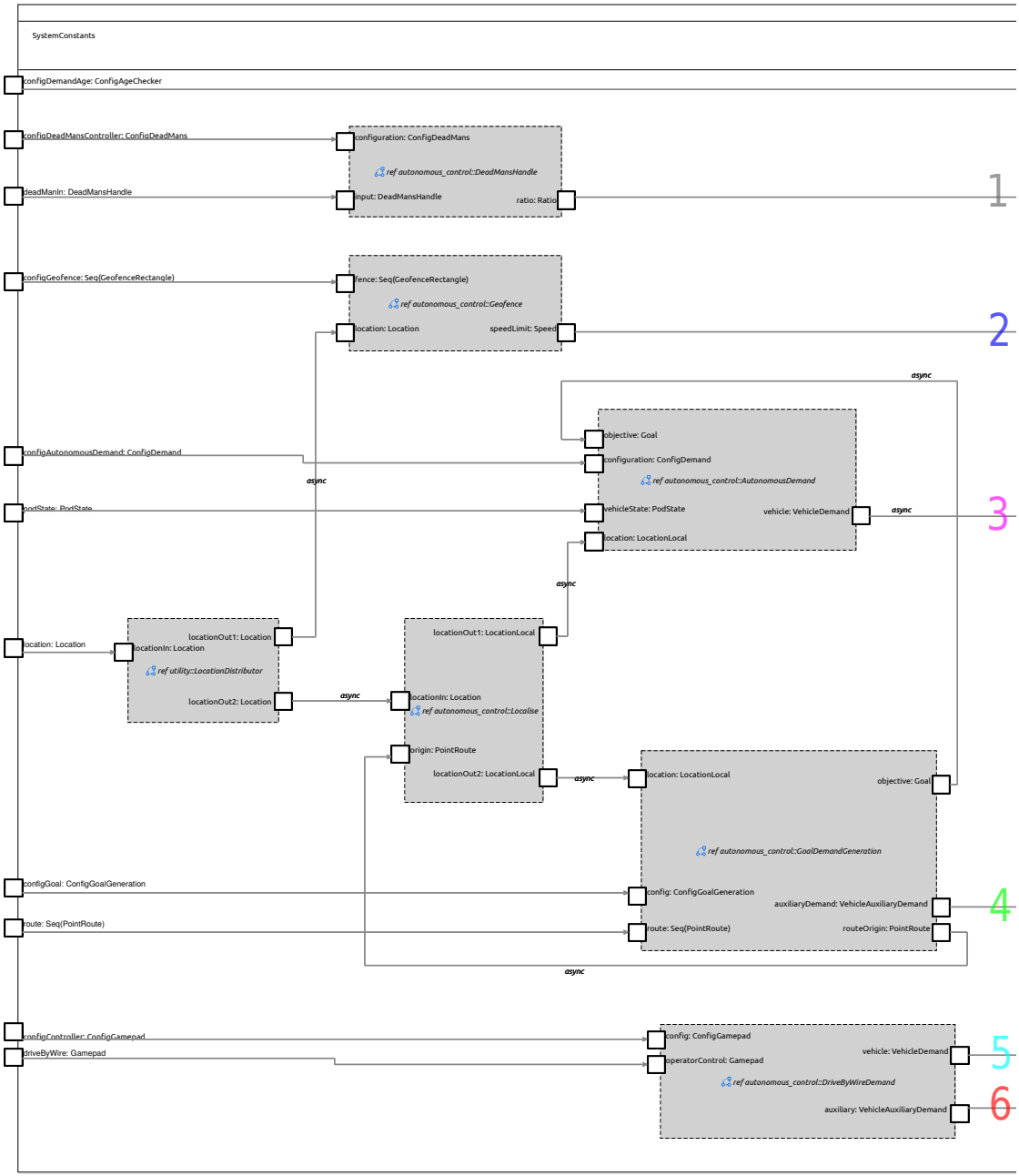


Figure 4.7: The b_acs controller; part 2 of 3

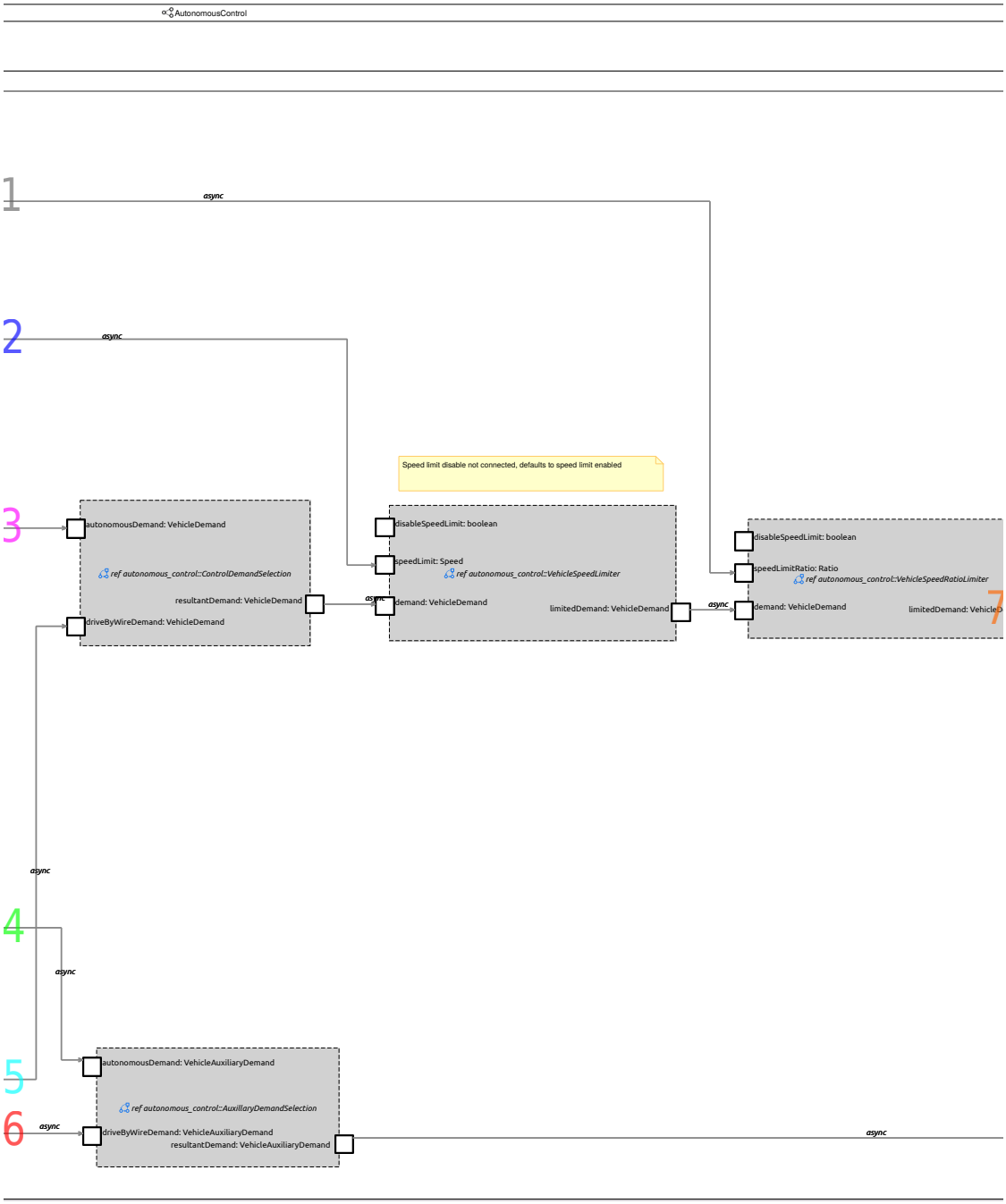
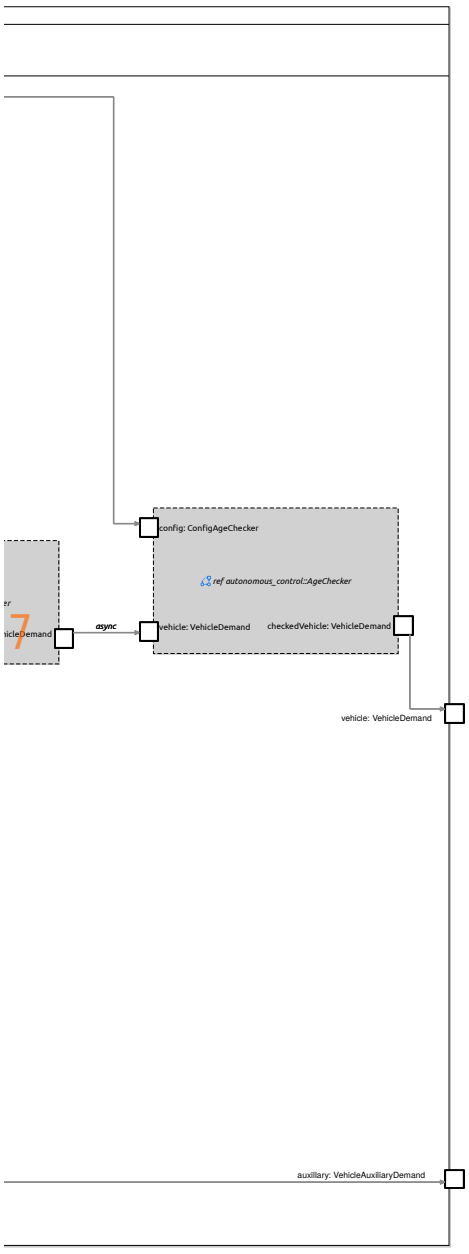


Figure 4.8: The b_acs controller; part 3 of 3



Listing 4.1: Pseudo code for a ROS node that normalises a data value.

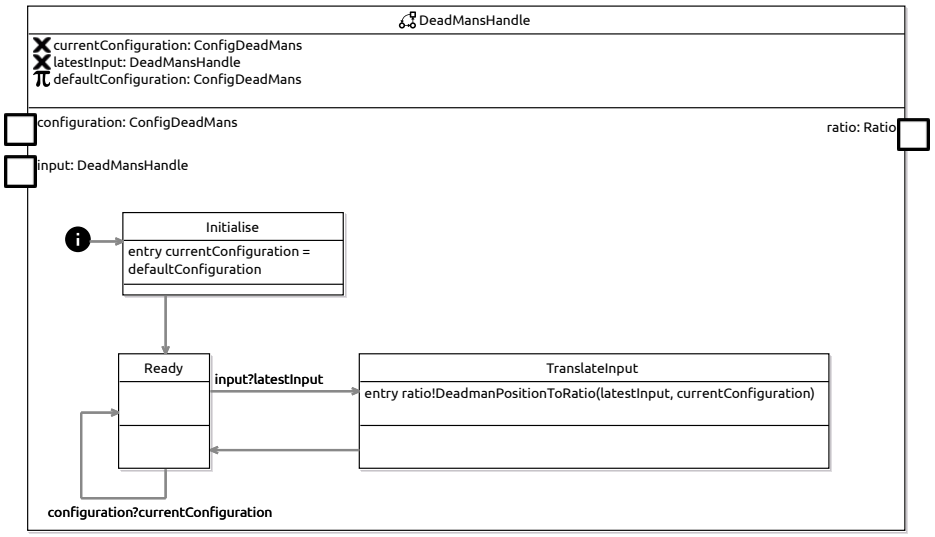
```
1  minValue 100
2  maxValue 500
3
4  subscriberControllerDataIn
5  publisherNormalisedValueOut
6
7  joystickNode(){
8      nodeParameters.getParam("minValue", minValue);
9      nodeParameters.getParam("maxValue", maxValue);
10
11     subscriberControllerDataIn.subscribe("input_topic", 25,
       controllerDataCallback, this)
12     publisherNormalisedValueOut.advertise("ratio_topic", 25)
13 }
14
15 controllerDataCallback( controllerReading & message ){
16     outputValue
17
18     if(msg.value < minValue) {
19         outputValue = 0
20     } elseif(msg.value > minValue AND msg.value < maxValue){
21         outputValue = (msg.value - minValue) / (maxValue-
       minValue)
22     } else {
23         outputValue = 1
24     }
25
26     publisherNormalisedValueOut.publish(outputValue)
27 }
28
29 spin(){
30     ros_spin()
31 }
```

An example of the code flow analysis performed on the ROS nodes for determining their states is presented next. Listing 4.1 provides pseudocode that is representative of the function of the dead_mans_handle node. The `joystickNode()` method is the node's constructor; it gets configuration parameters from the ROS parameter sever and subscribes and advertises to the topics the node requires for its operation. In the RoboChart model the configuration parameters have been represented as an event configuration, and the subscribed and published ROS topics can be represented by the events input and ratio. When the `joystickNode()` node constructor is executing (lines 7-13), the node can be considered to be in an initialising state. Therefore, the first state in the Robochart model

is the Initialise state. On entry to the Initialise state, default configuration parameter values are assigned to a variable currentConfiguration.

After the initialisation, the `spin()` method is invoked regularly which services the callbacks for the subscribed topics. In the RoboChart model this state is modelled as the Ready state. On receiving an `input_topic` message the `controllerDataCallback()` is called. In the RoboChart model this is represented by a transition leaving the Ready state with a transition action assigning the data from the received input event to a local variable latestInput.

Figure 4.9: The b_acs controller.



During the `controllerDataCallback()` (lines 15-27) the input value received in the message is normalised and then published to the `ratio_topic` topic. In the RoboChart model this can be represented as a TranslateInput state. During the TranslateInput state, the normalisation of the message value received can be modelled using a function taking the latestInput and the minimum and maximum configuration values as parameters. The normalised value returned by the function is then output via the ratio event and the state machine transitions back to the Ready state.

Figure 4.9 shows the resulting RoboChart state machine for the dead_mans_handle node.

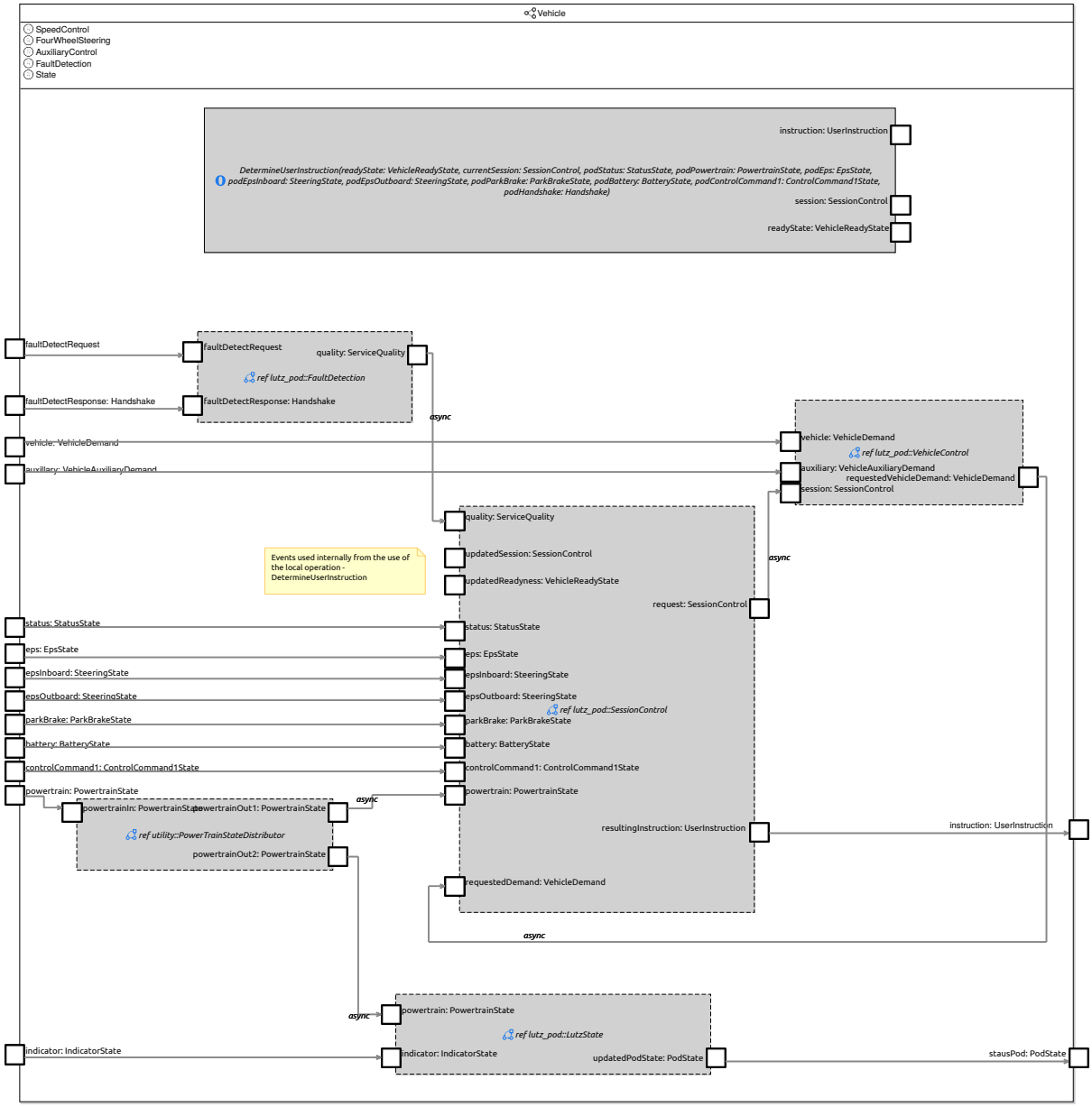
4.2.4 Controller - lutz

To create the lutz RoboChart controller, the ROS nodes relating to the lutz group have been represented using state machines. Some of the ROS nodes provide low-level connectivity to hardware devices so as before with the b_acs model these nodes will not be modelled. Table C.4 lists all of the nodes in the lutz controller model.

In order to create the state machines that represent the ROS nodes, the message topics that each ROS node publishes and subscribes to, and the states for each state machine must be identified. This is because the messages are the primary means of communication among the ROS nodes. Table C.17 identifies and lists the ROS messages and their source and destination nodes. The ROS message data structure for communication over the ROS topics will be modelled directly using RoboChart data types and fields as done for the b_acs model. Again the the behaviour of the ROS publish-subscribe messaging will be modelled using RoboChart asynchronous events with an associated data type to hold the message information.

Figure 4.10 shows the resulting lutz RoboChart controller; each of the state machines can be found in the appendices from Figures C.14 to C.19. The events connecting the state machines represent the ROS messages given by Table C.17.

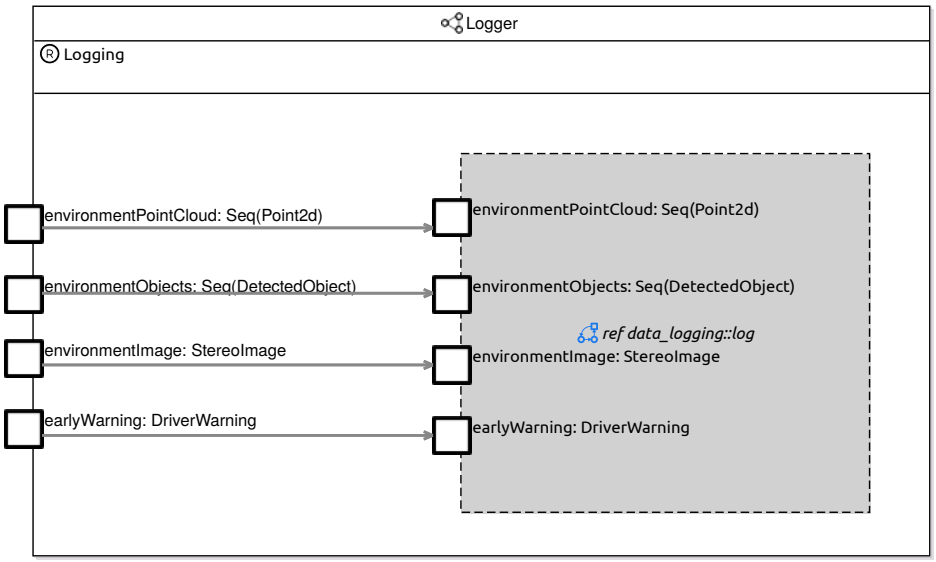
Figure 4.10: The lutz controller



4.2.5 Controller - data_logging

The driverless pod system uses the built in functionality provided by ROS to log sensor data for later evaluation. The ROS nodes associated with the sensors to be logged are shown in Figure 4.3. These nodes represent low-level hardware connectivity so they will not be modelled. The ROS messages that the sensor nodes output will be modelled as events in the robotic platform.

Figure 4.11: Logger Controller

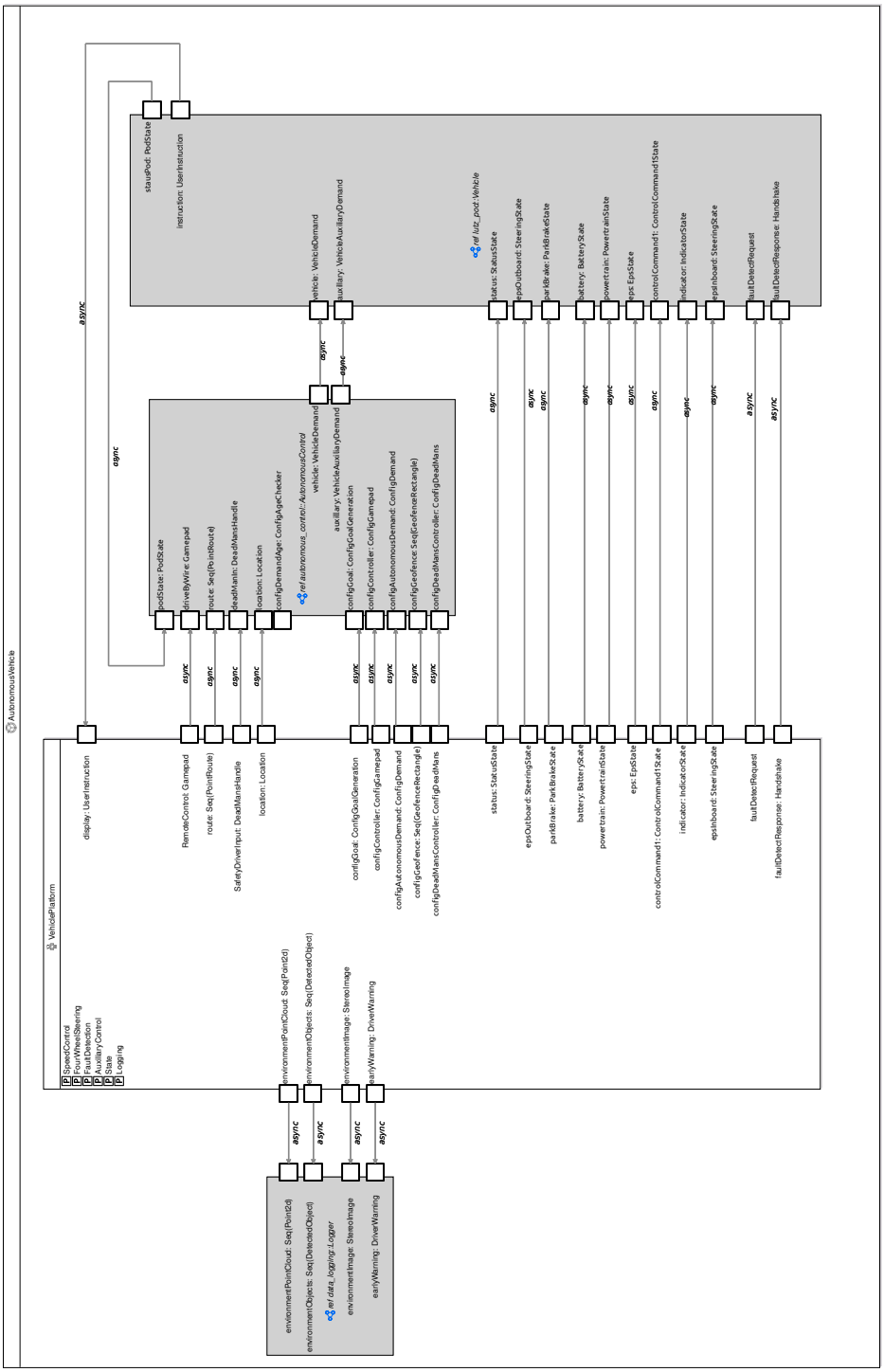


The data-logging functionality of ROS for the sensors being evaluated can be modelled using a RoboChart controller that accepts the sensors events. Figure 4.11 shows the controller; the state machine can be found in appendices Figure C.20.

4.2.6 Module - The Pod System

To create the RoboChart module of the pod system, the robotic platform and the three controllers that were previously defined are referenced, and the connections between the events are added. Figure 4.12 shows the resulting module and the connection between the controllers and the robotic platform.

Figure 4.12: The autonomous control module



4.3 Implementation Model - Verification

This section presents the verification results of the driverless pod RoboChart model.

For verifying the properties of the driverless pod RoboChart model FDR 4.2.3 was used running on a server with: one terabyte of memory and a AMD EPYC™ 7501 dual processor, giving a total 64 available cores. The verification of each state machine was given three hours of server execution time to complete. If a state machine could not be verified in three hours the result was recorded as 'did not complete' and the verification terminated.

4.3.1 Controller - b_acs

Table 4.4 and Table 4.5 summarise the verification results for the basic properties of each state machines from the b_acs controller. The associated results generated by RoboTool can be found in Appendix C.6.

Table 4.4: The untimed verification results summary for the state machines of the b_acs controller.

Result Table	State Machine	Property					Note
		①	②	③	④	⑤	
C.23	DeadMansHandle	✓	✓	✓	✓	✓	
C.25	Geofence	✓	✓	✓	✓	✓	Sequence restricted to size 1
C.27	AutonomousDemand	-	-	-	-	-	Did not complete
C.28	Localise	✓	✓	✓	✓	✓	
C.30	GoalDemandGeneration	-	-	-	-	-	FDR initialisation fails
C.31	DriveByWireDemand	-	-	-	-	-	Did not complete
C.32	ControlDemandSelection	✗	✓	✗	✓	✓	
C.33	VehicleSpeedLimiter	✓	✓	✓	✓	✓	
C.35	VehicleSpeedRatioLimiter	✓	✓	✓	✓	✓	
C.37	AgeChecker	-	-	-	-	-	Did not complete
C.38	AuxiliaryDemandSelection	✗	✓	✓	✓	✓	
C.40	LocationDistributor	✓	✓	✓	✓	✓	

Legend: ① Deterministic, ② Divergence freedom, ③ Deadlock freedom, ④ Does not terminate, ⑤ All states are reachable.

Table 4.5: The timed verification results summary for the state machines of the b_acs controller.

Result Table	State Machine	Property					Note
		①	②	③	④	⑤	
C.24	DeadMansHandle	✓	✓	✓	✓	✓	
C.26	Geofence	✓	✓	✓	✓	✓	Sequence restricted to size 1
-	AutonomousDemand	.	.	.	.	.	Did not test
C.29	Localise	✓	✓	✓	✓	✓	
-	GoalDemandGeneration	.	.	.	.	.	Did not test
-	DriveByWireDemand	.	.	.	.	.	Did not test
-	ControlDemandSelection	.	.	.	.	.	Did not test
C.34	VehicleSpeedLimiter	✓	✓	✓	✓	✓	
C.36	VehicleSpeedRatioLimiter	✓	✓	✓	✓	✓	
-	AgeChecker	.	.	.	.	.	Did not test
C.39	AuxiliaryDemandSelection	✗	✓	✓	✓	✓	
C.41	LocationDistributor	✓	✓	✓	✓	✓	

Legend: ① Deterministic, ② Divergence freedom, ③ Deadlock freedom, ④ Does not terminate, ⑤ All states are reachable.

From the untimed results one-third of the state machines could not be verified. For one of these, the GoalDemandGeneration state machine FDR failed the initial evaluation step. This is because the GoalDemandGeneration state machine input event types have a large number of possible permutations, resulting in a too many branches for FDR to evaluate. For the remaining three state machines, AutonomousDemand, DriveByWireDemand, and AgeChecker, could not be verified in the time given. This is because FDR did not complete constructing the labelled transition systems, which it uses for the verification of these state machines, due to their resulting size.

Checking the timed properties of the b_acs model increases the size of the labelled transition system that FDR must construct. The average increase in states of the labelled transition system for the timed RoboChart model was seventeen times larger than the untimed model. The average increase in transitions of the labelled transition system for the timed model was five times larger than the untimed model.

Because verifying the timed properties of a RoboChart model result in

FDR creating a larger labelled transition system compared to the untimed properties, more computing resources are required to verify the timed properties of a model. Therefore, timed properties of state machines were not tested for state machines whose untimed properties could not be verified, since they would similarly not complete with the same amount of computing resource available.

4.4 Evaluation

Verification of the basic properties of the b_acs controller of the driverless pod RoboChart model has shown that state explosion is a problem, with verification not being possible to complete on a server which has one terabyte of memory and sixty-four physical processor cores.

The size of types used in the RoboChart model has a big impact on number of states in the FDR labelled transition system used for verification, and therefore whether or not it is possible to verify properties of the RoboChart model. For compound types such as RoboChart DataTypes the number of permutations they have in the generated CSP model increases exponentially with each field added. Similarly the number of permutations for sequences for every item added also increases exponentially. Therefore, sequences of DataTypes can quickly reach a size that causes FDR to exhaust the available computing resource during verification or be too large for FDR to evaluate when types are modelled directly from the implementation.

Not being able to verify some of the individual state machines from the driverless pod RoboChart model, means that there is a need for additional techniques and methods that enable larger models of real systems to be verified.

5 Evaluation and Plan

This chapter of the report evaluates the progress made to date. It provides plans for the next two years of upcoming work, including an insight into its envisioned use. Finally, the chapter addresses ethical and data management issues and concludes with a summary of the primary risks that affect our work. The chapter consists of the following sections: Section 5.1 evaluates the progress made, Sections 5.2 to 5.3 present the plans for upcoming work and envisioned use, Section 5.4 considers the ethical issues, and finally Section 5.5 identifies the primary risks.

5.1 Evaluation of Progress

The overall goal of our work is to contribute to the advancement of software verification for robotic systems, by facilitating the development of future robotic systems that need to be robust and safe whilst operating in more unstructured environments. To attain this goal the objectives are to: contribute to the pragmatic aspects of modelling and verification of RoboChart, and to define guidelines for the developers of robotic systems on how robotics software architectures can be modelled and how verification can take advantage of the architectures.

In order to understand the use of software architecture in the robotics domain and justify the use of RoboChart, I have reviewed a selection of robotics architectures and DSLs. I found that for robotics software architectures, there is no single widely used architecture, but, layers are commonly used to structure them. In terms of DSLs, RoboChart offers compelling support for formal methods as well as automatic and semi-automatic verification. A related area that would benefit from additional study is the verification techniques that have been used for robotics systems.

I have defined dome some preliminary guidelines covering the general use of RoboChart, which are recorded in Chapter 3. However, the guidelines do not yet cover the modelling of robotic systems, because further case studies for evaluation are necessary.

The autonomous vehicle case study prototype that a I modelled, documented in Chapter 4, is one the largest RoboChart models developed. It demonstrates the need for additional techniques and methods because of the state space explosion problem associated with model checking. The development of the autonomous vehicle prototype case study has enabled the primary research methods and techniques to be exercised, namely, the use of case studies, modelling and property checking using RoboChart, and familiarisation with CSP.

The following two sections provide a plan of work for next two years.

5.2 Plan for Year 2

Table 5.1 outlines the tasks, allocated time, and deliverables for year two. The structure of work for the second year is separated into three tasks: architectural pattern identification (T1), developing reasoning techniques for robotic systems (T2), and communicating research (T3).

Table 5.1: Year two plan structure and deliverables

ID	Task	Months Allocated	Deliverables
T1	Architectural software pattern identification	6	Guidelines Extensions to RoboChart
T2	Develop reasoning techniques for robotic systems	4	Reasoning techniques
T3	Communicating research	2	Research paper Second year report

Task T1 will identify how architectural patterns used in the development of robotic systems can be adopted and described in RoboChart models. The patterns will be identified by using the concepts found from the review of architectures completed during the first year of work, and from the analysis of case studies. The case studies will be of realistic robotic systems so that RoboChart's support for modelling them following the identified architectural patterns can be evaluated. The first planned case study will be an automated mail delivery robot, based on the example given in [10]. Different architectural concepts will be evaluated for each case study, for example, architectures with different layer structures.

The autonomous vehicle case study from the first year of work, will

be revisited and used as another case study for the evaluation. The deliverables for this phase will be guidelines and patterns, along with any necessary extension to RoboChart in order to support the modelling of robotic systems following the identified patterns. The guidelines will act as a reference for developers and provide guidance on how to model and verify a robotic system's software using RoboChart. The scope of the extension to RoboChart will depend on the result of the evaluation and could range from no extension required, to the development of a new features for RoboChart.

Task T2 will develop reasoning techniques that enable properties of robotic systems to be verified taking advantage of the modular structure of the patterns. The techniques will be developed using the results and deliverables from T1 utilising formal methods such as model checking and theorem proving. The deliverable for this phase will be the developed reasoning techniques.

Task T3 will cover the report writing aspect of the work for year two. The two deliverables for this task are a research paper and a second year report. The research paper will cover any novel methods or techniques investigated as part of the T2. The second year report will document the progress made toward the university objectives [1] for the second year.

The next section provides a plan for the third year of work.

5.3 Plan for Year 3

Table 5.1 outlines the tasks, allocated time, and deliverables for year three. The structure of work for the third year is separated into two tasks: compositionality of techniques (T1) and communicating research (T2).

Table 5.2: Year three plan structure and deliverables

ID	**Task**	**Months Allocated**	**Deliverables**
T1	Compositionality of techniques	6	Definitions
T2	Communicating research	6	Research Paper Thesis

For task T1 depending on the performance of the verification approaches studied in year 2, which will be based on existing techniques, we will pursue novel techniques. They will involve the study of compositional reasoning that can take advantage of our architectural patterns. We

will design the techniques, and establish their soundness by a proof of compositionality. The deliverable will be definitions that ascribe meaning from the developed techniques to the robotic system.

Task T2 will cover the report writing aspect of the work for year three. The two deliverables for this task are a research paper and a thesis. The research paper will cover the compositionality aspects investigated during task T1. The thesis will conclude our work consolidating its findings.

5.4 Ethics and Data Management

The development of robotic systems raises many ethical concerns particularly with regard to autonomous and intelligent systems. Because of these systems increasing capabilities, prevalence, and closer interaction with people, there is a greater risk of them causing harm. Therefore, it is important that these systems behave in a moral and unbiased way that minimises the potential harm that they can cause. There are also wider societal impacts from the use autonomous and intelligent robotics systems, for instance, the potential obsolescence of large numbers of jobs forcing people into alternative careers. Consequently it is important that these ethical concerns are considered when developing technology that contributes to the development of robotic systems.

The technology developed by this project intends to contribute to the verification of software for robotic systems, and it does not directly contribute to the advancement of robotic systems high-level 'intelligence' or autonomy. Therefore, there are no concerns over moral agency relating to the technology developed as part of this project. The primary intention of the technology developed is to improve the safety of robotic systems by assisting developers in identifying and correcting difficult to detect problems at design time. As a result, the likelihood of failures resulting in harm occurring during runtime, due to software defects, is reduced. With regards to the societal impacts of robotic systems, the potential improvement in safety provides a net benefit to society, in comparison to, the systems still being developed using existing tools.

The project will be carried out in an ethical manner and in accordance with the University of York's ethics and data management policies [80] [81]. No human participants or animals are required as part of the work, and there are no anticipated impacts on the wider community or the environment. No personal data will be collected or stored as part of

the project, however, sensitive data in the form of proprietary source code and documentation may be used. Proprietary source code and documentation will only be kept on password protected local systems with access limited to only those who require it.

5.5 Risk Analysis

This section reviews the most significant risks associated with the project and the measures that will be taken to mitigate them.

The risks can be separated into two categories: internal risks and external risks. The internal risks are those that can be controlled by the project, and the external risks are those that are outside the control of project.

The first internal risk is overly-ambitious ideas resulting in delays and overrun tasks, for example, selecting and modelling an overly complex case study. This risk will be minimised by reviewing the tractability of initial ideas with supervisors. Additionally, during the tasks, the time spent versus time remaining will be periodically reviewed to check progress and provide a warning as to potential over-ambition.

The second internal risk is unexpected delays from the required internally developed tool features not being available, for example, RoboChart because it is under active development. This risk will be mitigated by effective communication with the tool developers, notifying them through the bug tracking system of any missing features or problems in advance of the task starting.

The third risk is inexperience in the application of formal methods. This risk will be mitigated by being vigilant of the allocated time versus the time taken to identify situations where insufficient progress is being made. Where insufficient progress is being made, wider expertise from within the research group will be utilised for guidance on ways to progress.

The fourth internal risk is ineffective communication resulting in outcomes not being clearly understood by stakeholders and missed opportunities, for example, poor grammar in reports. This risk will be mitigated by taking relevant training provided by the university on writing and presentation skill, as well as, taking advantage of other opportunities where experience can be gained.

The primary external risk is data loss or inaccessibility due to provider closure, for example, the private repositories used to store models suddenly ceasing operation. This risk will be mitigated by regularly backing

up the externally hosted repositories to university managed storage.

Overall the majority of significant project risks are internal, therefore, they can be easily monitored and the planned mitigation applied. The ability to monitor the risks also reduces the likelihood of unexpected issues affecting the project schedule. The consequences of the only significant external risk can be sufficiently reduced that it would have little to no impact on the project.

References

[1] University of York. (c2016). Phd: Students registered after 31 august 2016, the university of york, [Online]. Available: https://www.cs.york.ac.uk/staff/researchstudy/phdincomputerscience/phdstartedafter31aug2016/ (visited on 17/06/2019).

[2] A. M. Zanchettin, E. Croft, H. Ding and M. Li, "Collaborative robots in the workplace", *IEEE Robot. Autom. Mag.*, vol. 25, no. 2, pp. 16–17, 2018.

[3] T. Vandemeulebroucke, B. Dierckx de Casterlé and C. Gastmans, "The use of care robots in aged care: A systematic review of argument-based ethics literature", en, *Arch. Gerontol. Geriatr.*, vol. 74, pp. 15–25, Jan. 2018.

[4] N. Research. (2019). Navigant research leaderboard: Automated driving vehicles, [Online]. Available: https://www.navigantresearch.com/reports/navigant-research-leaderboard-automated-driving-vehicles (visited on 04/2019).

[5] T. Hoffmann and G. Prause, "On the regulatory framework for Last-Mile delivery robots", en, *Machines*, vol. 6, no. 3, p. 33, Aug. 2018.

[6] S. A. Redfield and M. L. Seto, "Verification challenges for autonomous systems", in *Autonomy and Artificial Intelligence: A Threat or Savior?*, W. F. Lawless, R. Mittu, D. Sofge and S. Russell, Eds., Cham: Springer International Publishing, 2017, pp. 103–127.

[7] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, en. Pearson Education, Sep. 2012.

[8] E. W. Dijkstra, "The structure of the "THE"-multiprogramming system", *Commun. ACM*, vol. 11, no. 5, pp. 341–346, May 1968.

[9] AUTOSAR, *AUTOSAR*, https://www.autosar.org/, Accessed: 2019-4-26, Apr. 2019.

[10] M. W. Achtelik *et al.*, *Springer Handbook of Robotics*, 2nd ed., B. Siciliano and O. Khatib, Eds. Gewerbestrasse 11, 6330 Cham, Switzerland: Springer International Publishing, 2016.

[11] J. Södling, R. Ekbom, P. Thorngren and H. Burden, "From model to rig – an automotive case study", in *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*, 2016.

[12] T. Franz, D. Lüdtke, O. Maibaum and A. Gerndt, "Model-based software engineering for an optical navigation system for spacecraft", *CEAS Space Journal*, vol. 10, no. 2, pp. 147–156, 2018.

[13] A. Nordmann, N. Hochgeschwender, D. Wigand and S. Wrede, "A survey on domain-specific modeling and languages in robotics", en, *Journal of Software Engineering for Robotics*, vol. 7, no. 1, pp. 75–99, Jul. 2016.

[14] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi and M. Ziane, "RobotML, a domain-specific language to design, simulate and deploy robotic applications", in *3rd International Conference on Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR 2012*, vol. 7628 LNAI, Tsukuba, 2012, pp. 149–160.

[15] C. Schlegel, A. Steck and A. Lotz, "Robotic software systems: From Code-Driven to Model-Driven software development", in *Robotic Systems - Applications, Control and Programming*, A. Dutta, Ed., In-Tech, Feb. 2012.

[16] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi and D. Brugali, "The BRICS component model: A model-based development paradigm for complex robotics software systems", in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ACM, Mar. 2013, pp. 1758–1764.

[17] M. Luckcuck, M. Farrell, L. Dennis, C. Dixon and M. Fisher, "Formal Specification and Verification of Autonomous Robotic Systems: A Survey", Jun. 2018. arXiv: 1807.00048 [cs.FL].

[18] G. O'Regan, *Concise Guide to Formal Methods: Theory, Fundamentals and Industry Applications*, en. Springer, Aug. 2017.

[19] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis and J. Woodcock, "RoboChart: Modelling and verification of the functional behaviour of robotic applications", *Software & Systems Modeling*, Jan. 2019.

[20] P. Ribeiro, *RoboStar - case studies*, https://www.cs.york.ac.uk/robostar/case_studies/, Accessed: 2019-5-21, May 2019.

[21] J. Nembrini, "Minimalist coherent swarming of wireless networked autonomous mobile robots", PhD thesis, University of the West of England, Jan. 2005.

[22] J. Chen, M. Gauci and R. Groß, "A strategy for transporting tall objects with a swarm of miniature mobile robots", in *2013 IEEE International Conference on Robotics and Automation*, May 2013, pp. 863–869.

[23] *What is your definition of software architecture?*, 2010. [Online]. Available: https://resources.sei.cmu.edu/asset_files/FactSheet/2010_010_001_513810.pdf.

[24] "ISO/IEC/IEEE systems and software engineering – architecture description", *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pp. 1–46, 2011. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2011.6129467.

[25] R. Brooks, "A robust layered control system for a mobile robot", *IEEE Journal on Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.

[26] D. L. Wigand, P. Mohammadi, E. M. Hoffman, N. G. Tsagarakis, J. J. Steil and S. Wrede, "An open-source architecture for simulation, execution and analysis of real-time robotics systems", in *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2018, pp. 93–100.

[27] *The intelligent robotics system architecture applied to robotics testbeds and research platforms*, vol. 2018-March, IEEE Computer Society, 2018.

[28] S. García, C. Menghi, P. Pelliccione, T. Berger and R. Wohlrab, "An architecture for decentralized, collaborative, and autonomous robots", in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 75–7509.

[29] T. Huntsberger and G. Woodward, "Intelligent autonomy for unmanned surface and underwater vehicles", in *OCEANS'11 MTS/IEEE KONA*, 2011, pp. 1–10.

[30] J. L. Sanchez-Lopez, M. Molina, H. Bavle, C. Sampedro, R. A. Suárez Fernández and P. Campoy, "A Multi-Layered Component-Based approach for the development of aerial robotic systems: The aerostack framework", *J. Intell. Rob. Syst.*, vol. 88, no. 2, pp. 683–709, 2017.

[31] B. Álvarez, P. Sánchez-Palma, J. A. Pastor and F. Ortiz, "An architectural framework for modeling teleoperated service robots", *Robotica*, vol. 24, no. 4, pp. 411–418, 2006.

[32] B. Sellner, F. W. Heger, L. M. Hiatt, R. Simmons and S. Singh, "Coordinated multiagent teams and sliding autonomy for large-scale assembly", *Proc. IEEE*, vol. 94, no. 7, pp. 1425–1443, 2006.

[33] P. Corke, P. Sikka, J. M. Roberts and E. Duff, "DDX : A distributed software architecture for robotic systems", in *Proceedings of the 2004 Australasian Conference on Robotics & Automation*, N. Barnes and D. Austin, Eds., Australian National University Canberra: Australian Robotics & Automation Association, 2004.

[34] *The CLARAty architecture for robotic autonomy*, vol. 1, Big Sky, MT, 2001.

[35] D. Luzeaux and A. Dalgalarrondo, "HARPIC, an hybrid architecture based on representations, perceptions, and intelligent control: A way to provide autonomy to robots", in *Computational Science - ICCS 2001*, Springer Berlin Heidelberg, 2001, pp. 327–336.

[36] R. Alami, R. Chatila, S. Fleury, M. Ghallab and F. Ingrand, "An architecture for autonomy", *Int. J. Rob. Res.*, vol. 17, no. 4, pp. 315–337, 1998.

[37] N. Muscettola, P. P. Nayak, B. Pell and B. C. Williams, "Remote agent: To boldly go where no AI system has gone before", *Artif. Intell.*, vol. 103, no. 1-2, pp. 5–47, 1998.

[38] J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon and N. Turro, "The ORCCAD architecture", *Int. J. Rob. Res.*, vol. 17, no. 4, pp. 338–359, 1998.

[39] R. Peter Bonasso, R. James Firby, E. Gat, D. Kortenkamp, D. P. Miller and M. G. Slack, "Experiences with an architecture for intelligent, reactive agents", *J. Exp. Theor. Artif. Intell.*, vol. 9, no. 2-3, pp. 237–256, 1997.

[40] D. M. Lyons and A. J. Hendriks, "Planning as incremental adaptation of a reactive system", *Rob. Auton. Syst.*, vol. 14, no. 4, pp. 255–288, 1995.

[41] S. T. Yu, M. G. Slack and D. P. Miller, "A streamlined software environment for situated skills", en, ser. volume 1, Houston TX: NASA, 1994, pp. 233–239.

[42] D. J. Musliner, E. H. Durfee and K. G. Shin, "CIRCA: A cooperative intelligent real-time control architecture", *IEEE Trans. Syst. Man Cybern.*, vol. 23, no. 6, pp. 1561–1574, 1993.

[43] E. Gat, "Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots", in *Proceedings of the Tenth National Conference on Artificial Intelligence*, ser. AAAI'92, San Jose, California: AAAI Press, 1992, pp. 809–815.

[44] R. P. Bonasso, "Integrating reaction plans and layered competences through synchronous control", in *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 2*, Morgan Kaufmann Publishers Inc., 1991, pp. 1225–1231.

[45] R. C. Arkin, "Motor schema — based mobile robot navigation", *Int. J. Rob. Res.*, vol. 8, no. 4, pp. 92–112, 1989.

[46] J. S. Albus, R. Lumia, J. Fiala and A. J. Wavering, "NASREM – the NASA/NBS standard reference model for telerobot control system architecture", in *Industrial Robots*, 1989.

[47] R. C. Arkin, "Towards cosmopolitan robots: Intelligent navigation in extended Man-Made environments", PhD thesis, University of Massachusetts, 1987.

[48] R. Chatila, S. Lacroix, T. Simeon and M. Herrb, "Planetary exploration by a mobile robot: Mission teleprogramming and autonomous navigation", *Auton. Robots*, vol. 2, no. 4, pp. 333–344, 1995.

[49] R. Alami, S. Fleury, M. Herrb, F. Ingrand and F. Robert, "Multi-robot cooperation in the MARTHA project", *IEEE Robot. Autom. Mag.*, vol. 5, no. 1, pp. 36–47, 1998.

[50] S. Bensalem, L. de Silva, F. Ingrand and R. Yan, *A verifiable and Correct-by-Construction controller for robot functional levels*, 2013.

[51] I. A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin and W. S. Kim, "CLARAty: An architecture for reusable robotic software", in *Unmanned Ground Vehicle Technology V*, ser. SPIE, vol. 5083, Florida, 2003.

[52] I. A. D. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diazcalderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I.-H. Shu and D. Apfelbaum, "CLARAty: Challenges and steps toward reusable robotic software", *Int. J. Adv. Rob. Syst.*, vol. 3, no. 1, pp. 023–030, 2006.

[53] T. Huntsberger, P. Pirjanian, A. Trebi-Ollennu, H. D. Nayar, H. Aghazarian, A. J. Ganino, M. Garrett, S. S. Joshi and P. S. Schenker, "CAMPOUT: A control architecture for tightly coupled coordination of multirobot systems for planetary surface exploration", *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 33, no. 5, pp. 550–559, 2003.

[54] S. Chien, R. Knight, A. Stechert, R. Sherwood and G. Rabideau, "Using iterative repair to improve the responsiveness of planning and scheduling", in *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, AAAI Press, 2000, pp. 300–307.

[55] J. Hsu, *U.S. navy's drone boat swarm practices harbor defense - IEEE spectrum*, https://spectrum.ieee.org/automaton/robotics/military-robots/navy-drone-boat-swarm-practices-harbor-defense, Accessed: 2019-5-16, 2016.

[56] J. Kramer and J. Magee, "Self-Managed systems: An architectural challenge", in *Future of Software Engineering (FOSE '07)*, 2007, pp. 259–268.

[57] M. Brambilla, *Model-driven software engineering in practice*, eng, ser. Synthesis lectures on software engineering. San Rafael, Calif.: Morgan & Claypool Publishers, 2012.

[58] *OMG® Unified Modeling Language® (OMG UML®)*, 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/.

[59] "IEEE standard VHDL language reference manual", *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009.

[60] A. W. Roscoe, *The Theory and Practice of Concurrency*, en. Prentice Hall, 1997.

[61] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti and J. Timmis, "Automatic property checking of robotic applications", in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 3869–3876.

[62] C. A. R. Hoare and H. Jifeng, *Unifying Theories of Programming*, en. London; New York: Prentice Hall, 1998.

[63] M. Kwiatkowska, G. Norman and D. Parker, "PRISM 4.0: Verification of probabilistic Real-Time systems", in *Computer Aided Verification*, Springer Berlin Heidelberg, 2011, pp. 585–591.

[64] J. Woodcock, P. G. Larsen, J. Bicarregui and J. Fitzgerald, "Formal methods: Practice and experience", *ACM Comput. Surv.*, vol. 41, no. 4, 19:1–19:36, Oct. 2009.

[65] S. Dhouib, N. Du Lac, J.-L. Farges, S. Gerard, M. Hemaissia-Jeannin, J. Lahera-Perez, S. Millet, B. Patin and S. Stinckwich, "Control architecture concepts and properties of an ontology devoted to exchanges in mobile robotics", in *6th National Conference on Control Architectures of Robots*, 2011, 24–p.

[66] C. Schlegel, A. Steck, D. Brugali and A. Knoll, "Design abstraction and processes in robotics: From Code-Driven to Model-Driven engineering", in *Lecture Notes in Computer Science*, 2010, pp. 324–335.

[67] A. Steck and C. Schlegel, "SmartTCL: An execution language for conditional reactive task execution in a three layer architecture for service robots", in *International Workshop on Dynamic languages for RObotic and Sensors systems (DYROS)*, Darmstadt, 2010, pp. 274–277.

[68] D. Stampfer and C. Schlegel, "Dynamic state charts: Composition and coordination of complex robot behavior and reuse of action plots", *Intelligent Service Robotics*, vol. 7, no. 2, pp. 53–65, 2014.

[69] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[70] M. Radestock and S. Eisenbach, "Coordination in evolving systems", in *Trends in Distributed Systems CORBA and Beyond*, Springer Berlin Heidelberg, 1996, pp. 162–176.

[71] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor and B. Álvarez, "V3CMM: A 3-view component meta-model for model-driven robotic software development", *Journal of Software Engineering for Robotics*, vol. 1, no. 1, 2010.

[72] A. Miyazawa, A. Cavalcanti, P. Ribeiro, W. Li, J. Woodcock and J. Timmis, *RoboChart and RoboTool: Modelling, verification and simulation for robotics*, https://www.cs.york.ac.uk/circus/publications/techreports/reports/robochart-reference.pdf, Accessed: 2019-6-14.

[73] ——, *RoboChart reference manual*, University of York, 2019.

[74] E. Ábrahám and K. Havelund, Eds., *FDR3 — A Modern Refinement Checker for CSP*, vol. 8413, Lecture Notes in Computer Science, 2014, pp. 187–201.

[75] Transport Systems Catapult, "Lutz pathfinder - system design specification", 2015.

[76] ——, "Lutz pathfinder - electrical system specification", 2016.

[77] ——, "B-acs ros nodes", 2017.

[78] ——, "Tscācs-master source code ", [Source code].

[79] ROS. (Jan. 2019). Msg - ros wiki, [Online]. Available: http://wiki.ros.org/msg (visited on 01/07/2019).

[80] University of York. (Jul. 2017). Code of practice and principles for good ethical governance, the University of York, [Online]. Available: https://www.york.ac.uk/staff/research/governance/research-policies/ethics-code/ (visited on 19/06/2019).

[81] ——, (Nov. 2017). Research data management policy, the University of York, [Online]. Available: https://www.york.ac.uk/about/departments/support-and-admin/information-services/information-policy/index/research-data-management-policy/ (visited on 19/06/2019).

Appendices

A Project

A.1 Training Record

The following log for William Barnett was output from the University of York SkillsForge system https://www.skillsforge.york.ac.uk.

Development Summary

Training and Courses

Date	Title	Status	Attendance	Points
22/10/2018	RDT201439 - How to survive your PhD (and enjoy it!)	Finished	**1 session(s):** Attended 1	5
06/11/2018	RDT201440 - PhD Writing Support Suite: Stage 1 Lecture Series	Finished	**4 session(s):** Attended 3 **1 not processed yet**	15
09/11/2018	RDT2065 - Making the Most of Your Voice	Finished	**1 session(s):** Attended 1	5

Other Development Activities

Date	Activity	Type	Category	Points	
30 Jul 2019 at 12:00	**CS Poster Session 2019**	**Other Activity**	**D: Engagement, influence and impact**		**Delete Print**
23 Jul 2019 at 16:00	**Using RoboChart to Model an Autonomous Vehicle System**	**Other Activity**	**D: Engagement, influence and impact**		**Delete Print**
12 Jun 2019 at 15:00	WLSC Group Session - Academic Vocabulary And Grammar	Other Training	A: Knowledge and intellectual abilities		Delete Print
19 Mar 2019 at 09:30	CyPhyAssure Spring School	Other Training	A: Knowledge and intellectual abilities		Delete Print
14 Jan 2019 at 12:00	CSAV Module	Other Training	A: Knowledge and intellectual abilities		Delete Print
20 Dec 2018 at 14:30	**25-minute Seminar**	**Other Activity**	**D: Engagement, influence and impact**		**Delete Print**
01 Oct 2018 at 12:00	DOSA Module Attendance	Other Training	A: Knowledge and intellectual abilities		Delete Print

B Lawn-Mowing System

B.1 Assertions

```
1  assertion MM_1: mower::MowManager is deterministic
2  assertion MM_2: mower::MowManager is divergence-free
3  assertion MM_3: mower::MowManager is deadlock-free
4  assertion MM_4: mower::MowManager does not terminate
5  assertion MM_5: mower::MowManager::Charging is reachable in
   mower::MowManager
6  assertion MM_6: mower::MowManager::Mowing is reachable in
   mower::MowManager
7  assertion MM_7: mower::MowManager::AvoidingObstacle is
   reachable in mower::MowManager
8  assertion MM_8: mower::MowManager::Turning is reachable in
   mower::MowManager
9  assertion MM_9: mower::MowManager refines
   propertyAvoidObstacle  in the traces model
10 assertion MM_10: mower::MowManager refines
   propertyTurnAtBoundAtBoundary in the traces model
11
12 assertion M_1: mower::Mower is deterministic
13 assertion M_2: mower::Mower is divergence-free
14 assertion M_3: mower::Mower is deadlock-free
15 assertion M_4: mower::Mower does not terminate
16
17 assertion LMS_1: lawnmower_system::Lawnmower is
   deterministic
18 assertion LMS_2: lawnmower_system::Lawnmower is
   divergence-free
19 assertion LMS_3: lawnmower_system::Lawnmower is
   deadlock-free
20
21 // MM_9
22 csp propertyAvoidObstacle csp-begin
23   propertyAvoidObstacle =
24     let Wait = Recurse({| mower_MowManager_boundary.in,
   mower_MowManager_fullPower.in, mower_MowManager_lowPower.in,
   enableCutterCall, enableCutterRet, moveForwardsCall,
   moveForwardsRet, avoidCall, avoidRet, turnCall, turnRet |},
   Wait)
25           []
```

```
26              mower_MowManager_obstacle.in -> Avoid
27
28          Avoid = avoidCall ->  avoidRet -> Wait
29      within
30          Wait ||| RUN({})
31  csp-end
32
33  //MM_10
34  csp propertyTurnAtBoundAtBoundary csp-begin
35    propertyTurnAtBoundAtBoundary =
36      let Wait = Recurse({| mower_MowManager_obstacle.in,
    mower_MowManager_fullPower.in, mower_MowManager_lowPower.in,
    enableCutterCall, enableCutterRet, moveForwardsCall,
    moveForwardsRet, avoidCall, avoidRet, turnCall, turnRet |},
    Wait)
37              []
38              mower_MowManager_boundary.in -> Turn
39
40          Turn = turnCall ->  turnRet -> Wait
41      within
42          Wait ||| RUN({})
43  csp-end
44
```

B.2 Results

Table B.1: The untimed results for the lawn-mowing system

Assertion	States	Transitions	Result
mower_MowManager is deterministic (MM_1) [failures divergences model]	14	17	true
mower_MowManager is divergence free (MM_2) [failures divergences model]	14	17	true
mower_MowManager is deadlock free (MM_3) [failures divergences model]	14	17	true
mower_MowManager does not terminate (MM_4)	14	17	true
mower_MowManager_Charging is reachable in mower_MowManager (MM_5)	6	6	true
mower_MowManager_Mowing is reachable in mower_MowManager (MM_6)	15	15	true
mower_MowManager_AvoidingObstacle is reachable in mower_MowManager (MM_7)	30	32	true
mower_MowManager_Turning is reachable in mower_MowManager (MM_8)	30	32	true
mower_MowManager is refined by propertyAvoidObstacle (MM_9) [traces model]	14	17	true
mower_MowManager is refined by propertyTurnAtBoundAtBoundary (MM_10) [traces model]	14	17	true
mower_Mower is deterministic (M_1) [failures divergences model]	14	17	true
mower_Mower is divergence free (M_2) [failures divergences model]	14	17	true
mower_Mower is deadlock free (M_3) [failures divergences model]	14	17	true
mower_Mower does not terminate (M_4)	14	17	true
lawnmower_system_Lawnmower is deterministic (LMS_1) [failures divergences model]	14	17	true
lawnmower_system_Lawnmower is divergence free (LMS_2) [failures divergences model]	14	17	true
lawnmower_system_Lawnmower is deadlock free (LMS_3) [failures divergences model]	14	17	true

C Case Study Autonomous Vehicle

C.1 ROS Nodes by Namespace

Table C.1: cavlab_hw ROS nodes

Node	Description
Udp/Client	Translates incoming packets from a UDP connection into ROS messages and publishes them.
xnav550	Receives UDP messages from the location sensor, and publishes various formats of the location message including latitude longitude and heading.
arduino	Translates incoming output from an Arduino microcontroller into two separate published ROS messages, remote control in, and dead man's handle.
turnigy_joy	Receives remote control in messages and converts them into published pod demand, and auxiliary demand messages.
dead_mans_handle	Receives dead man's handle messages, and converts them into a published speed limit message.

Table C.2: b_acs ROS nodes

Node	Description
gps_to_local	Receives the pod's global location sensor information and translates it into a published local 2d location message.
path_replay	Receives the pod's local location and creates and publishes an intermediary goal towards the route's destination. The goal contains the target local location, curvature, maximum speed, and duration describing the path to take. A corresponding auxiliary demand message is created and published. The route to the destination is defined using a file containing a sequence of global locations and auxiliary state.
auto_demand	Receives a goal, the pod's local location, and speed from the pod state to create and publish a demand message. The demand message contains the specific steer and speed values to meet the goal.

Table C.3: cavlab_core ROS nodes

Node	**Description**
control_switch	Receives multiple pod demand inputs and publishes only the highest priority non-abdicating input. The highest priority input can abdicate, allowing lower priority inputs to be published.
geofence	Receives the global location of the pod and publishes the speed limit for current location. The speed limit for rectangular areas are configured using a file containing a sequence of two pairs of latitude and longitude points, each with an associated speed limit.
demand_limiter	Receives a pod demand and a speed limit, and publishes the demand with the appropriately adjusted speed. The speed limit applied can either be ratio or speed based.
age_checker	Receives a pod demand and checks its age; demand messages that are older than a defined time have their speed reduced. The speed reduction increases the older the demand message is, until the speed is zero. The age checked demand with appropriately adjusted speed is published.

Table C.4: lutz ROS nodes

Node	**Description**
message_builder	Receives a pod demand, an auxiliary demand, the pod state to request, and the pod's state to create a message that meets the pod's CAN message specification for control of the pod. The created aux power request, control command 1, and control command 2 messages are published.
session_control	Receives the state of the pod and publishes a user instruction and session control message. The node uses a state machine to determine the instruction the user must follow to enter autonomous mode and the pod state that must be requested.
handshake	Receives a handshake message from the pod and calculates the correct response and publishes a handshake response.
lutz_state	Receives pod indicator and powertrain messages and consolidates them into a single pod state message that is then published.
pod_rx	Receives CAN messages from the pod and decodes them into a corresponding status message that is then published.
pod_tx	Receives control command 1, control command 2, auxiliary power request, handshake, and session id messages and converts them into the CAN message that is published.
socketcan_bridge	Sends and receives CAN messages to the pod.
console_ui	Receives the user instruction code and displays the corresponding user instruction.
on_axle_stands	Receives the pod demand and sets its speed to zero and publishes the resulting demand message.
text2speech	Receives the user instruction code and plays a sound corresponding to the received instruction.

C.2 Data Types

Figure C.1: System Type Definitions

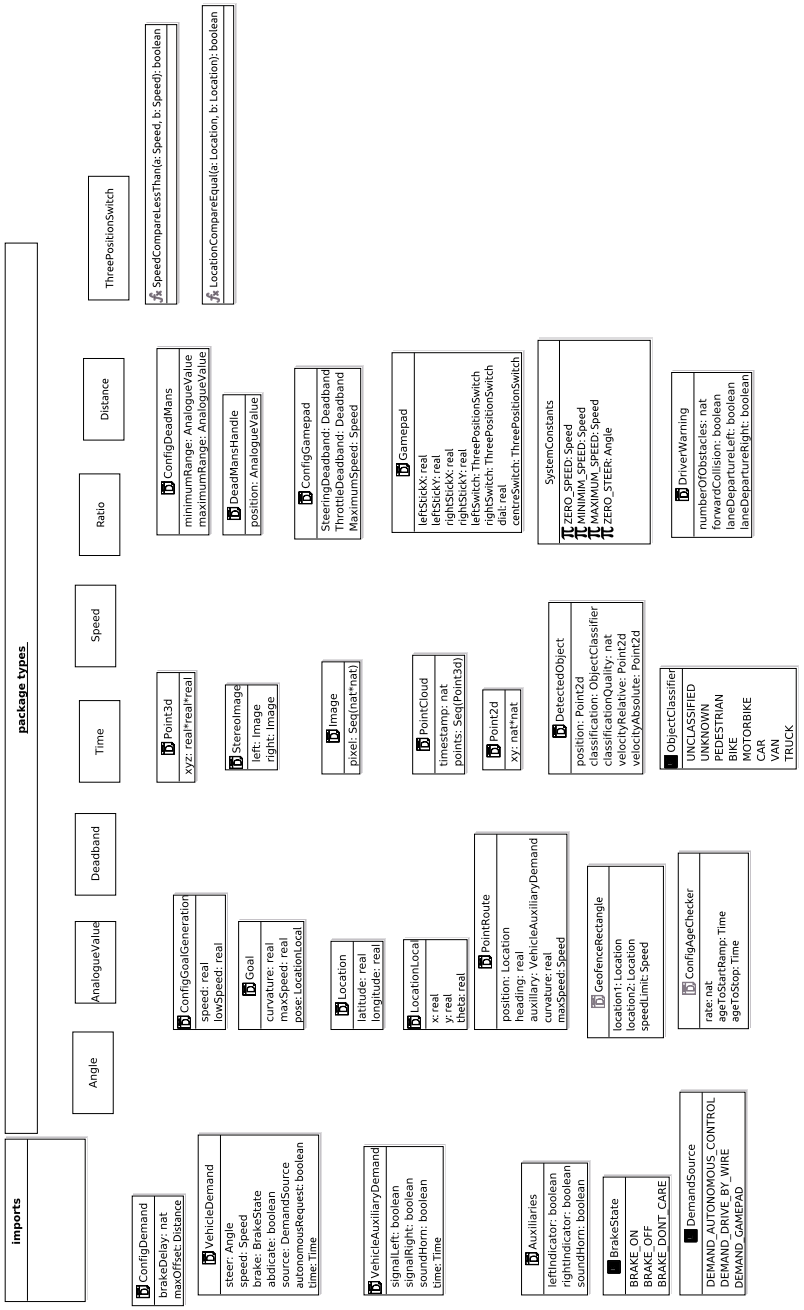
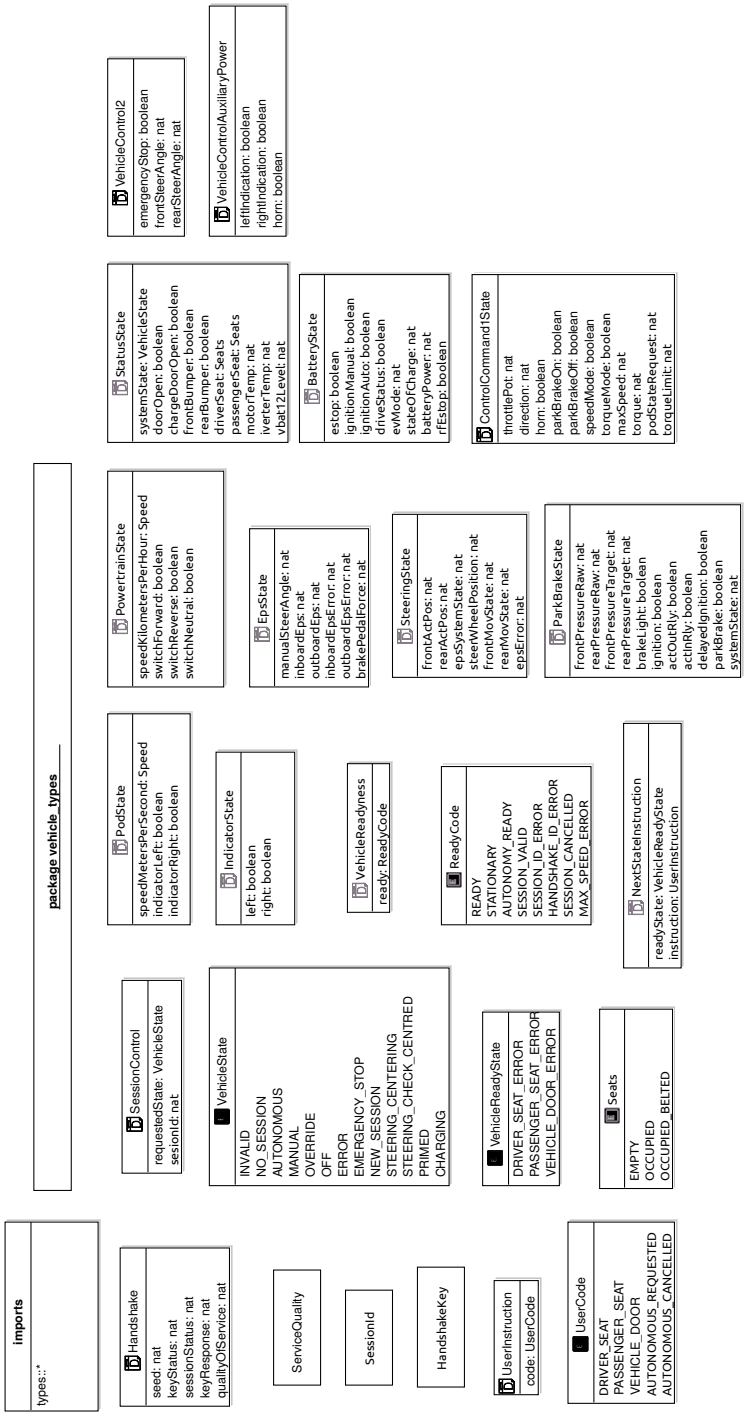


Figure C.2: Vehicle Type Definitions



C.3 Controller - Autonomous Control System

Table C.5: b_acs ROS nodes

Node	**RoboChart Model**
Udp/Client	Not modelled
xnav550	Robotic platform, Location
arduino	Robotic platform, safety driver input event and remote control event
dead_mans_ handle	State machine
gps_to_local	State machine
path_replay	State machine
auto_demand	State machine
turnigy_joy	State machine
control_switch (pod demand)	State machine
control_switch (auxiliary demand)	State machine
geofence	State machine
demand_limiter (geofence)	State machine
demand_limiter (speed)	State machine
age_checker	State machine

Table C.6: ACS ROS messages

Message	**Nodes**	**RoboChart Type**
LatLngHeadingFix	xnav550, out gps_to_local, in geofence, in	Location; distributed via the LocationDistributor state machine.
Pose2DStamped	gps_to_local, out path_replay, in auto_demand, in	LocationLocal
Goal	path_replay, out auto_demand, in	Goal
PodDemandSource	auto_demand, out control_switch, in (auto) control_switch, in (turnigy)	VehicleDemand
PodDemand	control_switch, out demand_limiter, out age_checker, out demand_limiter, in age_checker, in	VehicleDemand
SpeedLimit	geofence, out dead_mans_handle, out demand_limiter, in	Speed
AuxiliaryDemand	path_replay, out turnigy_joy, out auxiliary_switch	VehicleAuxiliaryDemand
PodState	auto_demand, in	PodState
Header (path reset)	turnigy_joy, out path_replay, in	(not modelled)
Path	path_replay, out	(not modelled)
AutoDemandCfg	auto_demand, out	(not modelled)
PathReplayCfg	path_replay, out	(not modelled)

Table C.7: ACS gps_to_local node methods

Method	Return	Parameters	Description
GpsToLocalNode	-	-	Constructor for the node. Initialises publish, subscribe node communications.
gpsCallback	-	LatLngHeadingFix	Receives latitude longitude and heading messages and converts them to local 2d coordinates. The local 2D coordinates are published to the location topic.

Table C.8: ACS path_replay node methods

Method	Return	Parameters	Description
GpsPathReplayNode	-	-	Constructor for the node. Checks the parameters provided at start-up for the gps path file, origin, and whether to loop the route. Reads the path file and sets the origin (finding the closest point if an origin isn't provided). Initialises publish, subscribe node communications.
ReadPathFile	-	file	Reads the path file and if the origin and start point if they weren't previously specified.
setOriginParam	-	-	Sets the origin location parameters from the path file that was previously read in.
setStartParam	-	-	Sets the start location parameters from the path file that was previously read in.
publishPath	-	-	Publishes the complete path to the allPath topic.

Continued on next page

Table C.8 – continued from previous page

Method	Return	Parameters	Description
calculateLook AheadDistance	double	Goal	Determines the look ahead distance based on the maximum speed for the goal.
locationCallback	-	Pose2DStamped	Receives local 2D coordinates. If the distance remaining to the current goal is less that the calculated lookahead distance, a new goal is published to the goal topic, and the associated auxiliary demand is published to the auxiliary_demand topic. If the goal has been reset the path is searched for the next closest point within range and is used for the next goal and clears the reset flag.
resetCallback	-	Header	Sets a local flag on reception of a path reset message for later handling.
configCallback	-	PathReplayConfig	Receives path replay configuration values and updates the node, dynamically reconfiguring it. The configuration values include: minimum and maximum speeds, de-acceleration, and look ahead distances.
handleCommand	bool	request, response	Handles ROS service requests to either replay route once, or replay route looped. In each case the path file is reloaded and the reset flag set.
makeCsvPath	string	string	Adds .csv file extension to a filename.

Path_replay also offers replay route service.

Table C.9: ACS auto_demand node methods

Method	Return	Parameters	Description
AutoDemandNode	-	-	Constructor for the node. Initialises publish, subscribe node communications and sets initial configuration values.
configCallback	-	AutoDemandConfig, uint32_t	Receives the auto demand configuration values and updates the node, dynamically re-configuring it.
goalCallback	-	Goal	Receives the goal message and stores it in a member variable.
locationCallback	-	Pose2DStamped	Receives the local 2D location and determines the steering and speed required to achieve the goal. If the speed is zero applies the brake after a delay. The resulting autonomous demand is published to the demand topic.
podStateCallback	-	PodState	Receives the pod state message and stores it in a member variable.

Table C.10: ACS turnigy_joy node methods

Method	**Return**	**Parameters**	**Description**
TurnigyJoyNode	-	-	Constructor for the node. Checks for any provided parameters at start-up (deadzones, minimum and maximum speeds) setting them as specified or uses default values. Initialises publish, subscribe node communications.
threePosition Switch	uint8_t	uint16_t	Converts an analogue reading to a position 1, 2, or 3.
rxCallback	-	rcReader	Receives the controller button readings and translates them to pod demand, auxiliary demand and reset path messages. The messages are then published to the respective turnigy_pod_demand, turnigy_auxiliary_demand, and path_reset topics.

Table C.11: ACS control_switch node methods

Method	Return	Parameters	Description
ControlSwitch Node	-	-	Constructor for the node. Parameters at start-up are used to configure the input demand sources (ROS topics) to switch between. Initialises publish, subscribe node communications.
messageReceived	-	PodDemandSource, int	Publishes the first input source (highest priority) which isn't abdicating. If all sources are abdicating publishes an output with zero speed and steer demand.
DemandSubscriber		string, int	Constructor for input demand subscribers.
DemandSubscriber		DemandSubscriber	Constructor for copying another input demand subscriber.
subscribe	-	-	Subscribes a DemandSubscriber to its ROS topic.
handler	-	PodDemandSource	Handles the reception of a DemandSubscriber's topic.
abdicated	bool	-	Returns a sources abdication state, old messages are are assumed not to be abdicated.

Control switch take parameters which allow multiple sources to be switched.

Table C.12: ACS geofence node methods

Method	Return	Parameters	Description
GeoFenceNode:: GeoFenceNode	-	-	Constructor for the node. Parameters at start-up are used to specify the maximum speed when not in defined area, and a geofence file defining areas and their speed limit. Initialises publish, subscribe node communications and publishes the geofence area for visualisation.
GeoFenceNode:: PublishVisualisation	-	NodeHandle	Converts the provided latitude and longitude into local points using the origin from the passed in parameters as the centre-point. The local points are then used to define shapes for visualising the geofence area which are published to the geofence_rect topic.
GeoFenceNode:: gpsCallback	-	LatLngHeadingFix	Receives the current location (latitude, longitude, and heading) and looks up the speed limit for the area publishing it to the geofence_limit topic.
GeoFenceNode:: ReadFile	-	string	Reads a comma separated file of serialised GeoRectangles and de-serialises them.
GeoFenceNode:: LocalMaxSpeed	double	double, double	Looks up and returns the speed limit for the passed in latitude and longitude based on the area.
GeoRectangle:: GeoRectangle	-	double, double, double, double, double	Constructor for a rectangle, taking two latitude and longitude points and the associated speed limit for the rectangle.
GeoRectangle:: Contains	bool	double, double	Determines if the passed in latitude and longitude is within the rectangles area, and returns true if it is.

Table C.13: ACS demand_limiter_speed and demand_limiter_geofence node methods

Method	Return	Parameters	Description
demandLimiterNode::demandLimiterNode	-	-	Constructor for the node. Initialises publish, subscribe node communications and the speed limit to 0. Parameters at start-up are used to configure the node for speed based or ratio based speed limiting.
demandLimiterNode::limitCallback	-	SpeedLimit	Receives a speed limit storing it in a member variable.
demandLimiterNode::demandCallback	-	PodDemand	Receives a pod demand applying either speed based or ratio based speed limiting, and publishes the resulting demand to the pod_demand_limited topic.

Table C.14: ACS dead_mans_handle node methods

Method	Return	Parameters	Description
deadMansHandleNode::deadMansHandleNode	-	-	Constructor for the node. Initialises publish, subscribe node communications. The minimum and maximum analogue reading ranges are set to default values, or to values provided by start-up parameters.
deadMansHandleNode::rxCallback	-	deadMansHandleReading	Receives a reading from the dead man's handle and publishes it to the speed_limit topic.

Table C.15: ACS age_checker node methods

Method	Return	Parameters	Description
ageCheckerNode:: ageCheckerNode	-	-	Constructor for the node. Initialises publish, subscribe node communications. The age to stop and age to start ramp limits are set to default values, or to values provided by parameters at start-up.
ageCheckerNode:: demandCallback	-	PodDemand	Receives a pod demand if it is recent and below the age to start ramping then the demand is published to the pod_demand_age_checked topic.
ageCheckerNode:: checkAge	-	-	Determines the age of the demand using the ROS header. The demand is allowed if its age is within the stop time limit. The speed of allowed demands is scaled in proportion to the messages age, so that the older the demand the lower the speed; with the demand being ramped down to zero by the stop time. If the demand exceeds the stop time limit, it is stopped by setting its speed and steer to zero. The resulting demand is published to the pod_demand_age_checked topic.
ageCheckerNode:: loop	-	-	The main loop of the node, calls checkAge.

Figure C.3: Auxiliary Demand Selection State Machine

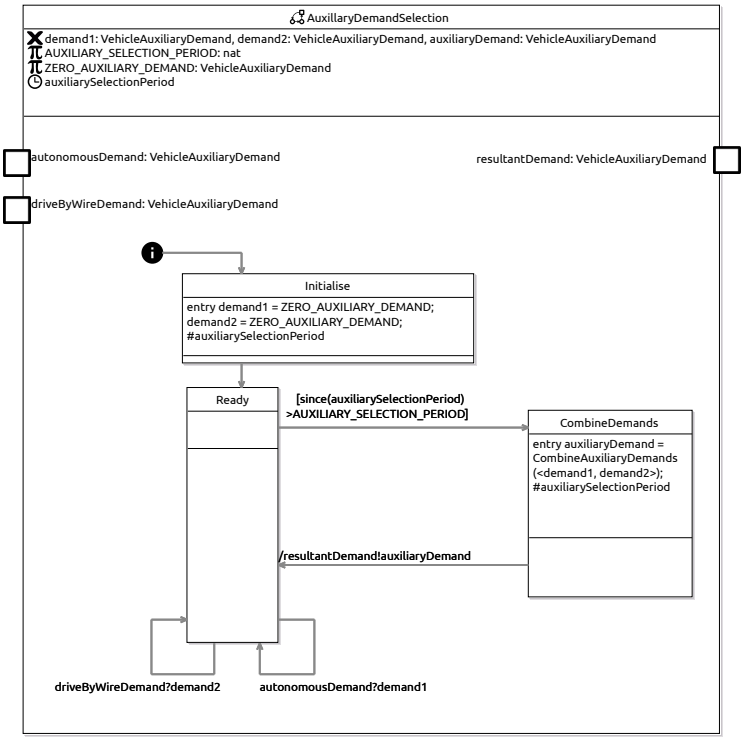


Figure C.4: Drive By Wire Demand State Machine

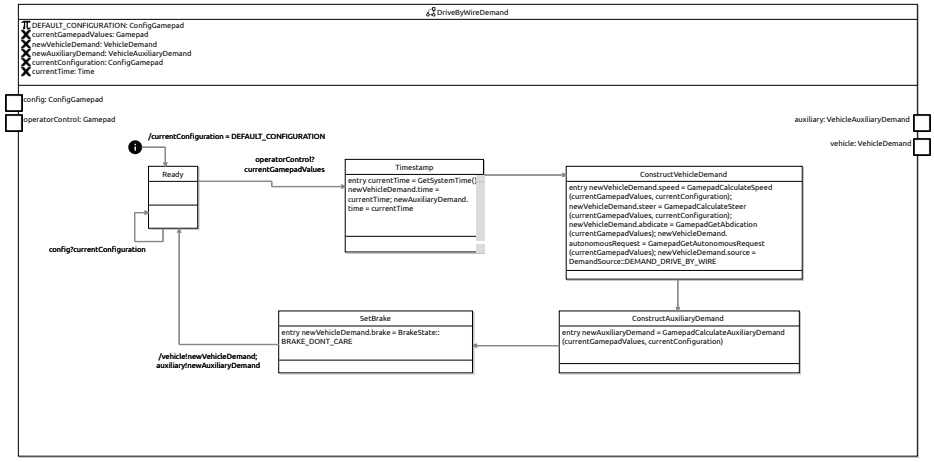


Figure C.5: Control Demand Selection State Machine

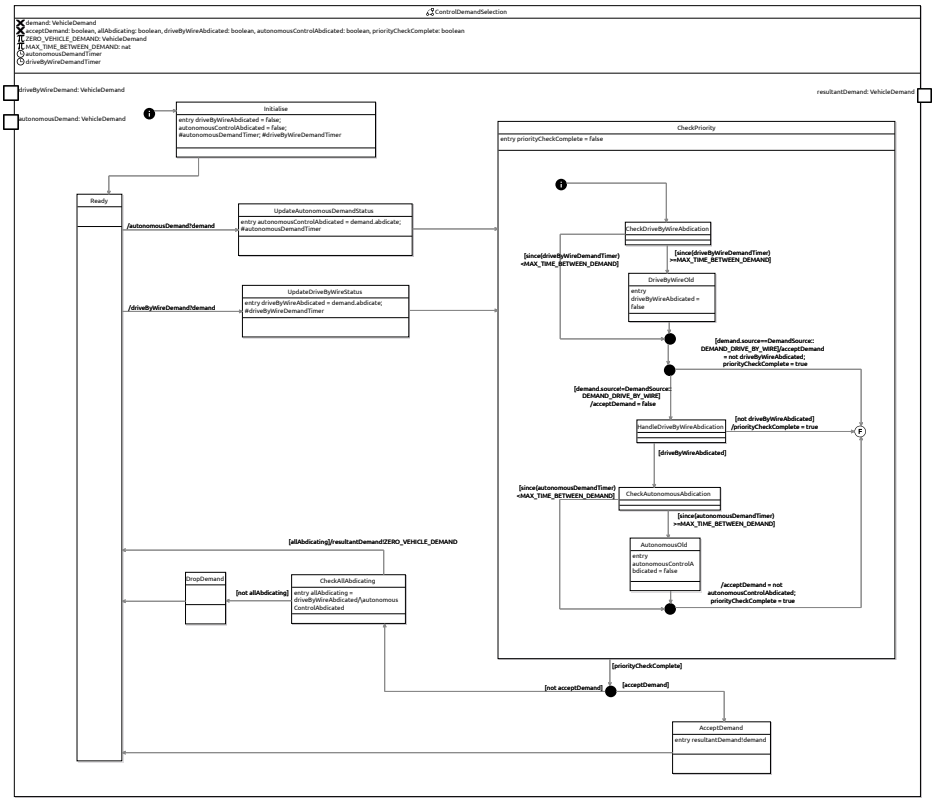


Figure C.6: Geofence State Machine

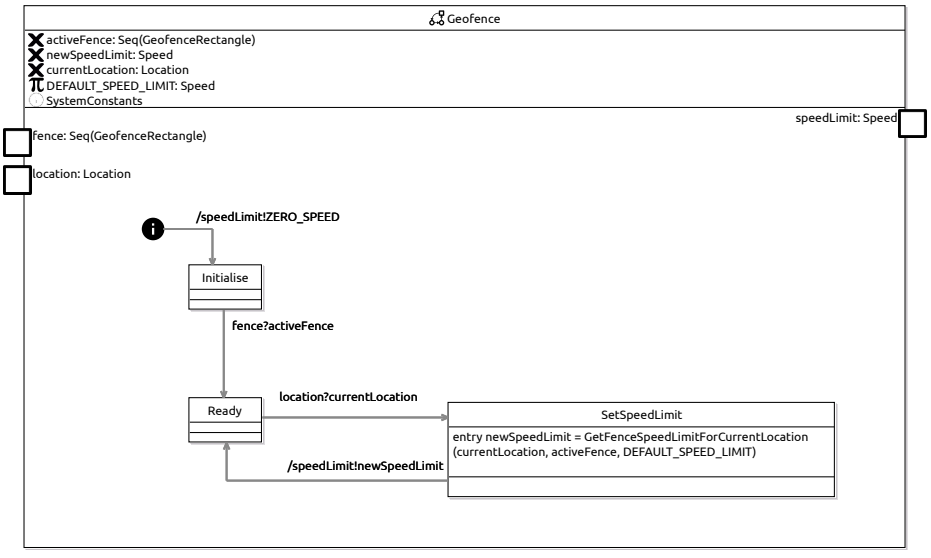


Figure C.7: Goal Demand Generation State Machine

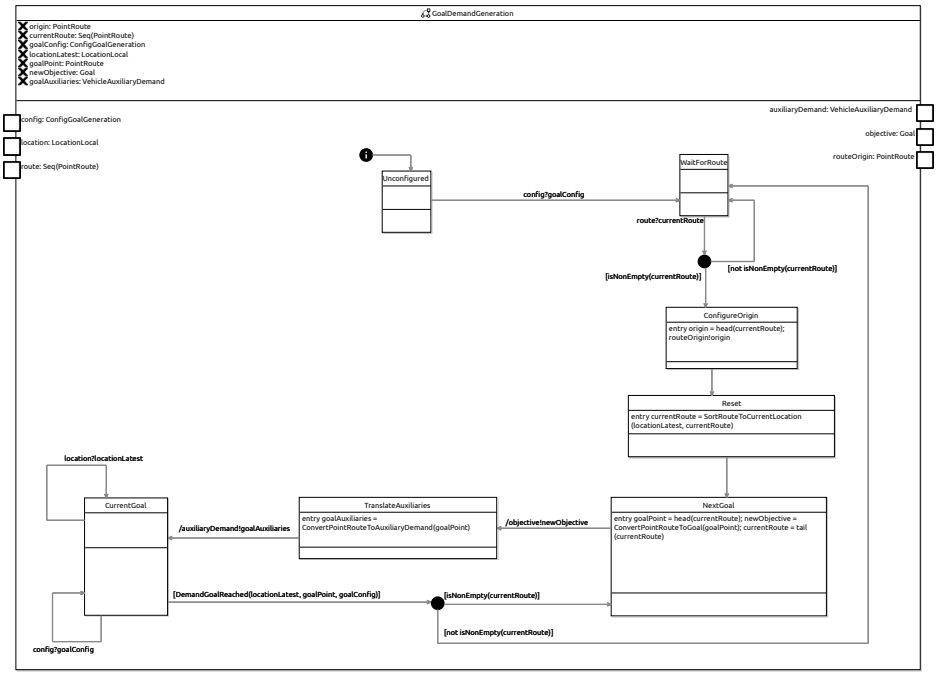


Figure C.8: Localise State Machine

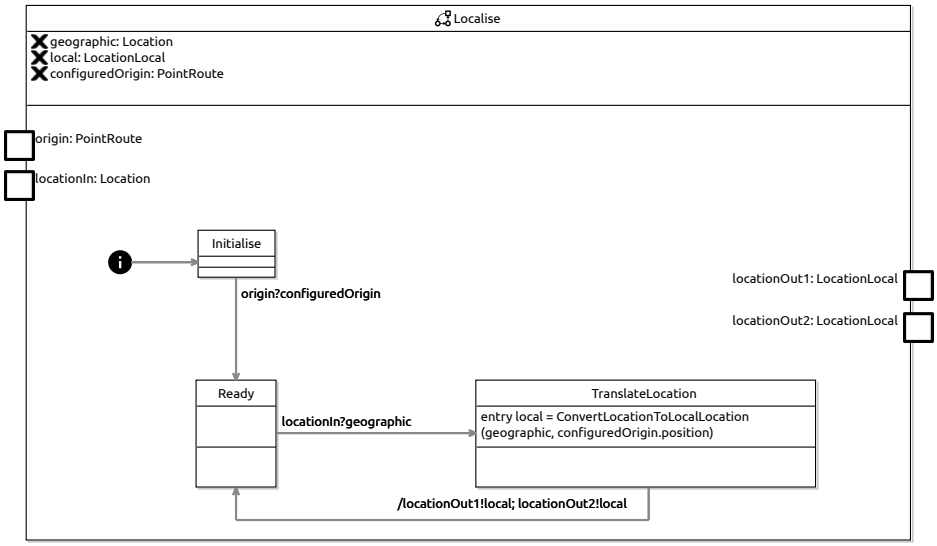


Figure C.9: Speed Limiter State Machine

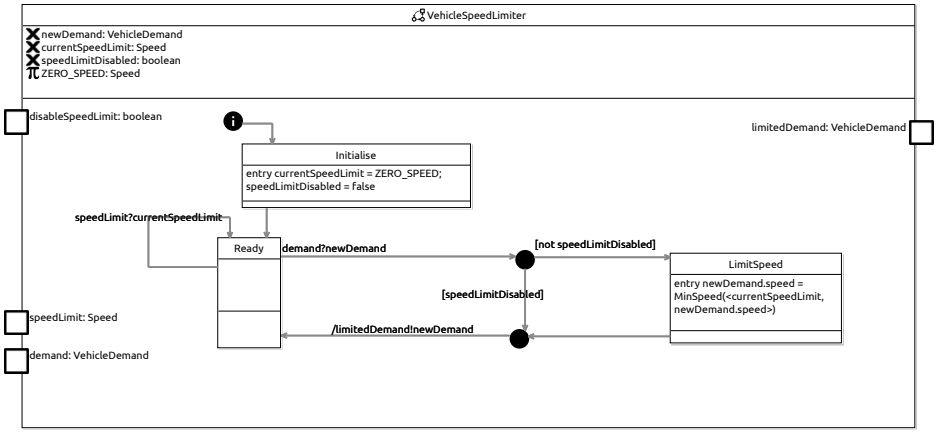


Figure C.10: Ratio Based Speed Limiter State Machine

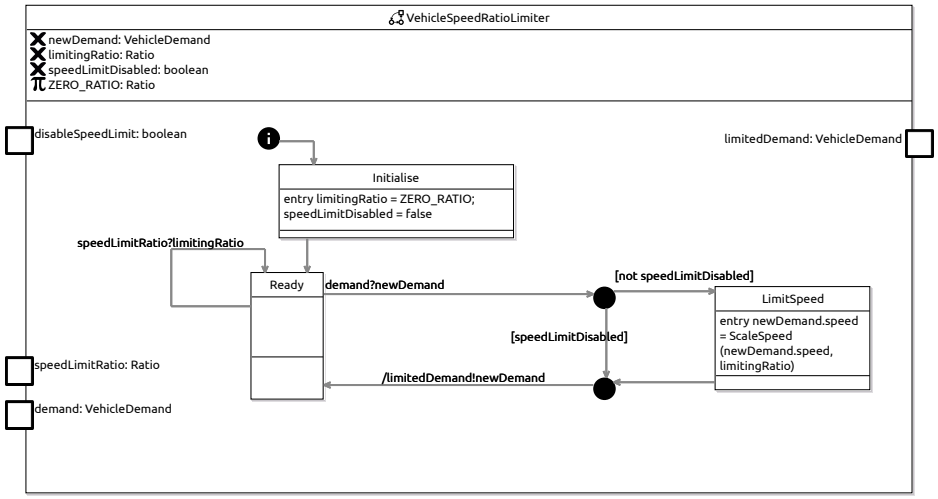


Figure C.11: Location Distributor State Machine

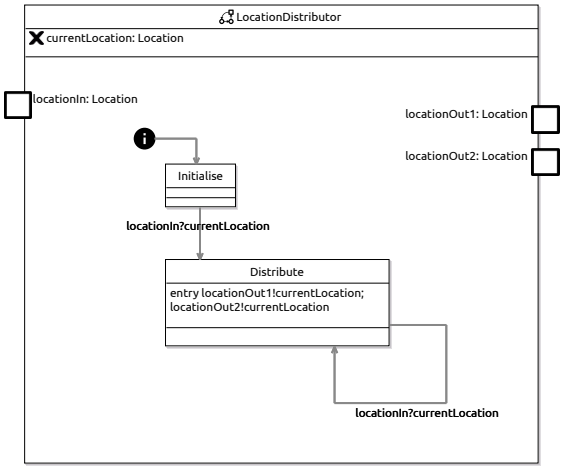


Figure C.12: Autonomous Demand State Machine

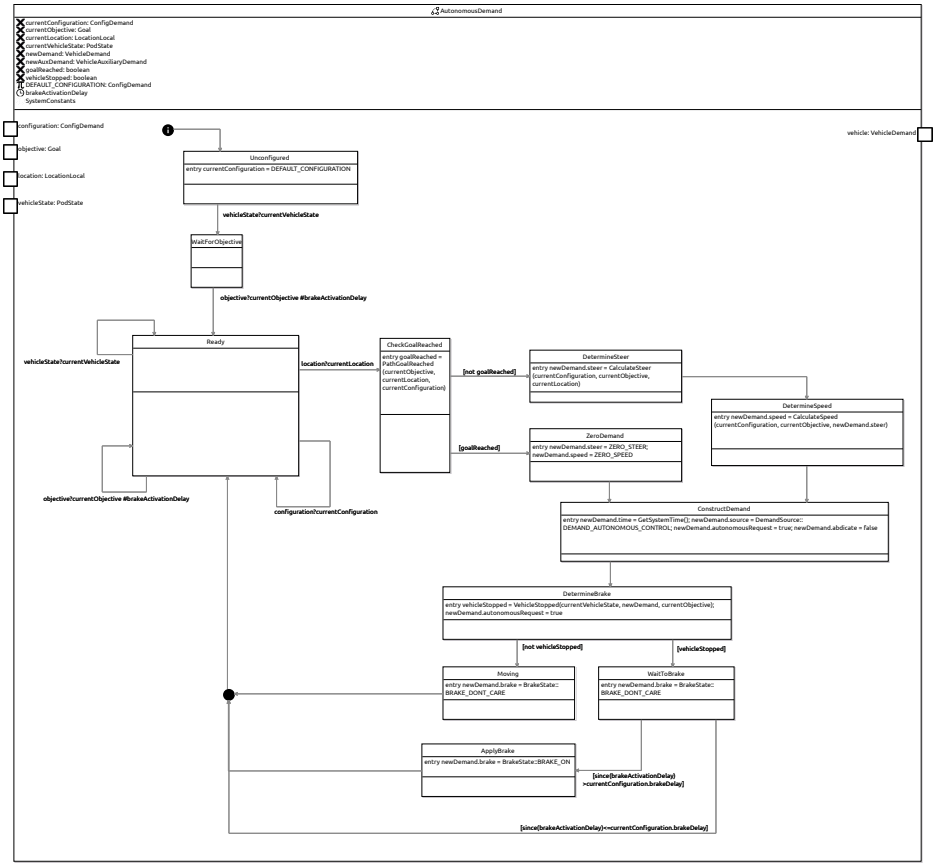
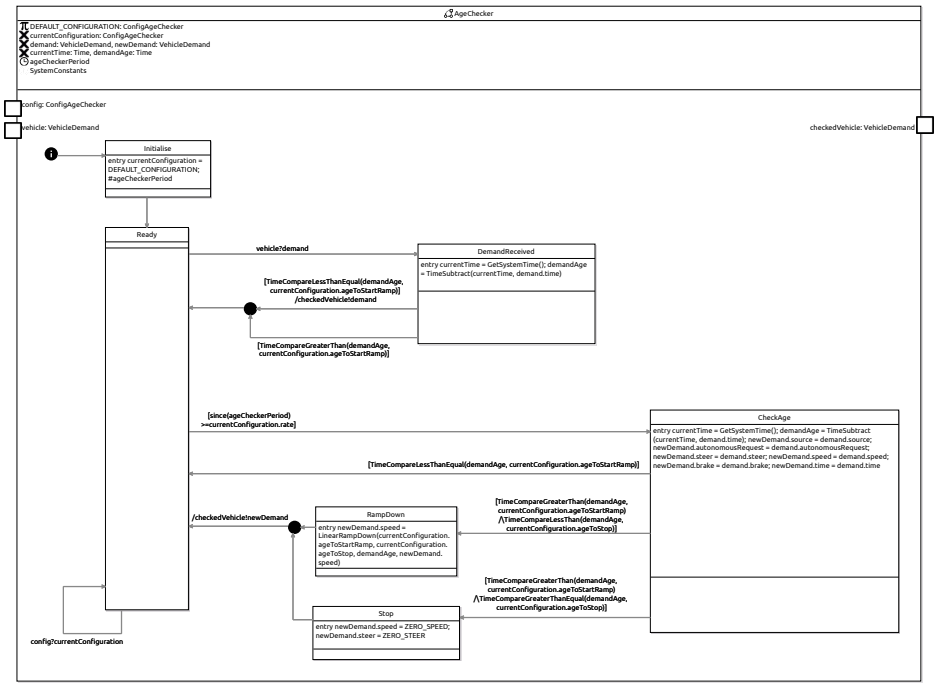


Figure C.13: Age Checker State Machine



C.4 Controller - Lutz Pod

Table C.16: Lutz pod ROS nodes

Node	**RoboChart Model**
message_builder	State machine
session_control	State machine
handshake	Statemachine
lutz_state	State machine
pod_rx	Robotic platfrom
pod_tx	Robotic platform
socketcan_bridge	Not modelled
console_ui	Not modelled
on_axle_stands	Not modelled
text2speech	Not modelled

Table C.17: Lutz vehicle ROS messages

Message	**Nodes**	**RoboChart Type**
PodDemand	message_builder, in	VehicleDemand
AuxiliaryDemand	message_builder, in	VehicleAuxiliaryDemand
PodState	pod_state, out	PodState
Status	pod_rx, out message_builder, in session_control, in	StatusState
SessionControl	session_control, out message_builder, in	SessionControl
ControlCommand1	message_builder, out session_control, in pod_tx, in	(not modelled)
ControlCommand2	message_builder, out pod_tx, in	(not modelled)
AuxPowerRequest	message_builder, out pod_tx, in	(not modelled)

Table C.17 – continued from previous page

Message	Nodes	RoboChart Type
Handshake	pod_rx, out handshake, out session_control, in handshake, in pod_tx, in	Handshake
UICode	session_control, out	UserInstruction
Battery	pod_rx, out session_control, in	BatteryState
EPS	pod_rx, out session_control, in	EpsState
Steer (eps inboard)	pod_rx, out session_control, in	SteeringState
Steer (eps outboard)	SteeringState pod_rx, out session_control, in	
Powertrain	pod_rx, out session_control, in pod_state, in	PowertrainState
ParkBrake	pod_rx, out session_control, in	ParkBrakeState
Indicator	pod_rx, out pod_state, in	IndicatorState
pod_can_rx	pod_rx, in	(not modelled)
pod_can_tx	pod_tx, out	(not modelled)

Table C.18: Lutz pod message_builder node methods

Method	Return	Parameters	Description
MessageBuilderNode::MessageBuilderNode	-	-	Constructor for the node. Initialises publish, subscribe node communications. The maximum speed and ticks per radian are set to default values, or to values provided by parameters at start-up.
MessageBuilderNode::rxSessionControl	-	SessionControl	Receives a session control request and stores the pod state request in a member variable.
MessageBuilderNode:: rxPodDemand	-	PodDemand	Receives a pod demand and stores it in a member variable.
MessageBuilderNode:: rxAuxDemand	-	AuxiliaryDemand	Receives an auxiliary demand and stores it in a member variable.
MessageBuilderNode::handlePOD_STATUS	-	Status	Receives the pod status and stores it in a member variable.
MessageBuilderNode:: buildMessages	-	-	Creates the control and command messages 1 and 2, and the auxiliary power request messages using the received pod demand, auxiliary demand, pod status, requested state. The created messages are stored in the respective member variables.
MessageBuilderNode:: publishMessages	-	-	Publishes control command messages 1 and 2, and the auxiliry power request from member variables to the relevant topics.
MessageBuilderNode:: loop	-	-	The main loop of the node, calls buildMessages and then publishMessages.

Table C.19: Lutz pod session_control node methods

Method	Return	Parameters	Description
SessionControlNode::SessionControlNode	-	-	Constructor for the node. Initialises publish, subscribe node communications.
SessionControlNode::makeAutonomous	-	-	If there has been a change in: the pod state, or autonomy request, or a ready state change pending handling; then the ready state handler is run or the session is cancelled.
SessionControlNode::generateUserInstruction	-	uint8_t, bool	Creates a user instruction code message using the passed in user instruction code and flag, along with the current pod state. The created message is then published to the ui_code topic.
SessionControlNode::rxCallback	PodDemand		Receives a pod demand and stores the last drive by wire request in a local variable. Outputs a user instruction indicating the status of the autonomous demand. If the last drive by wire request is true, then the drive by wire user instruction is output; if the last drive by wire request is false, then the cancel drive by wire user instruction is output. Finally, make autonomous is called to update the node's state.
SessionControlNode::handlePOD_STATUS	-	Status	Receives a pod status from them pod and stores it in a member variable including handling of the bump strips. If there has been a change in: the pod state, or autonomy request, or a ready state change pending handling; then the ready state handler is run or the session is cancelled.
SessionControlNode::handlePOD_HANDSHAKE	-	Handshake	Receives a handshake response from the pod and stores it in a member variable.

Continued on next page

Table C.19 – continued from previous page

Method	Return	Parameters	Description
SessionControlNode::handlePOD_BATTERY	-	Battery	Receives battery status from the pod and stores it in a member variable.
SessionControlNode::handlePOD_EPS	-	EPS	Receives EPS status from the pod and stores it in a member variable.
SessionControlNode::handlePOD_EPS_INBOARD	-	Steer	Receives EPS inboard status from the pod and stores it in a member variable.
SessionControlNode::handlePOD_EPS_OUTBOARD	-	Steer	Receives EPS outboard status from the pod and stores it in a member variable.
SessionControlNode::handlePOD_POWERTRAIN	-	Powertrain	Receives powertrain status from the pod and stores it in a member variable.
SessionControlNode::handlePOD_PARK_BRAKE	-	ParkBrake	Receives parking brake status from the pod and stores it in a member variable.
SessionControlNode::handleACS_CTRL_CMD1	-	ControlCommand1	Receives command control 1 status and stores it in a member variable.
SessionControlNode::handleBumpStrips	-	-	Manages flags for the detection of bump strip events by latching the appropriate flag based on changes in the pod status message.
SessionControlNode::handlePublisher	-	-	Publishes session control to the session_control topic using the state request and session id.
SessionControlNode::loop			The main loop of the node, calls handlePublisher.
SessionControlNode::EnterAutonomous	-	-	Handles the current pod state by calling the appropriate state handling method, see Table C.20 for the state handling methods.

Table C.20: Lutz pod session_control node methods for state handling

Method	**Return**	**Parameters**
SessionControlNode::MakeKeyManual	bool	-
SessionControlNode::MakeKeyAuto	bool	-
SessionControlNode::MakeSessiono	-	-
SessionControlNode::GetSessionValidState	ReadyState	-
SessionControlNode::MakeSessionValid	-	ReadyState
SessionControlNode::GetPrimedValidState	ReadyState	-
SessionControlNode::MakePrimedValid	-	ReadyState
SessionControlNode::GetAutonomousValidState	ReadyState	-
SessionControlNode::MakeAutonomousValid	-	ReadyState
SessionControlNode::MakeSessionCancelled	-	-
SessionControlNode::GetRevokeTrueState	ReadyState	-
SessionControlNode::GetRevokeReason	-	ReadyState
SessionControlNode::GetOverrideTrueState	ReadyState	-
SessionControlNode::GetOverrideTrueReason	-	ReadyState
SessionControlNode::MakeOverrideFalse	-	ReadyState
SessionControlNode::GetErrorTrueState	ReadyState	-
SessionControlNode::MakeErrorFalse	-	ReadyState
SessionControlNode::GetEstopState	ReadyState	-
SessionControlNode::GetEstopReason	-	ReadyState
SessionControlNode::MakeEstopFalse	-	ReadyState
SessionControlNode::GetPodReadyState	ReadyState	-
SessionControlNode::MakePodReady	-	ReadyState
SessionControlNode::GetStationaryState	ReadyState	-
SessionControlNode::MakePodStationary	-	ReadyState
SessionControlNode::GetHandshakeGoodState	ReadyState	-
SessionControlNode::MakeHandshakeGood	-	ReadyState
SessionControlNode::GetAutonomyReadyState	ReadyState	-
SessionControlNode::MakeAutonomyReady	-	ReadyState

Table C.21: Lutz pod handshake node methods

Method	Return	Parameters	Description
HandshakeNode:: HandshakeNode	-	-	Constructor for the node. Initialises publish, subscribe node communications and sets the quality of service message timeout to a default value
HandshakeNode:: rxCallback	-	Handshake	Receives a handshake response from the pod and verifies matches the expected seed value. If the response if correct the quality of service is increased.
HandshakeNode:: Handshake	-	-	Decreases the quality of service if the time since the last correct handshake message exceeds the quality of service message timeout. Publishes the next handshake message to the acs_handshake topic.
HandshakeNode:: loop	-	-	The main loop of the node, calls Handshake.

Table C.22: Lutz pod state node methods

Method	Return	Parameters	Description
LutzStateNode:: LutzStateNode	-	-	Constructor for the node, initialises publish, subscribe node communications.
LutzStateNode:: indicatorCallback	-	Indicator	Receives indicator status from the pod stores it in a member variable.
LutzStateNode:: powertrainCallback	-	Powertrain	Receives powertrain status from the pod and combines it with the indicator status previously received. Publishes the combined status to the pod_state topic.

Figure C.14: Fault Detection State Machine

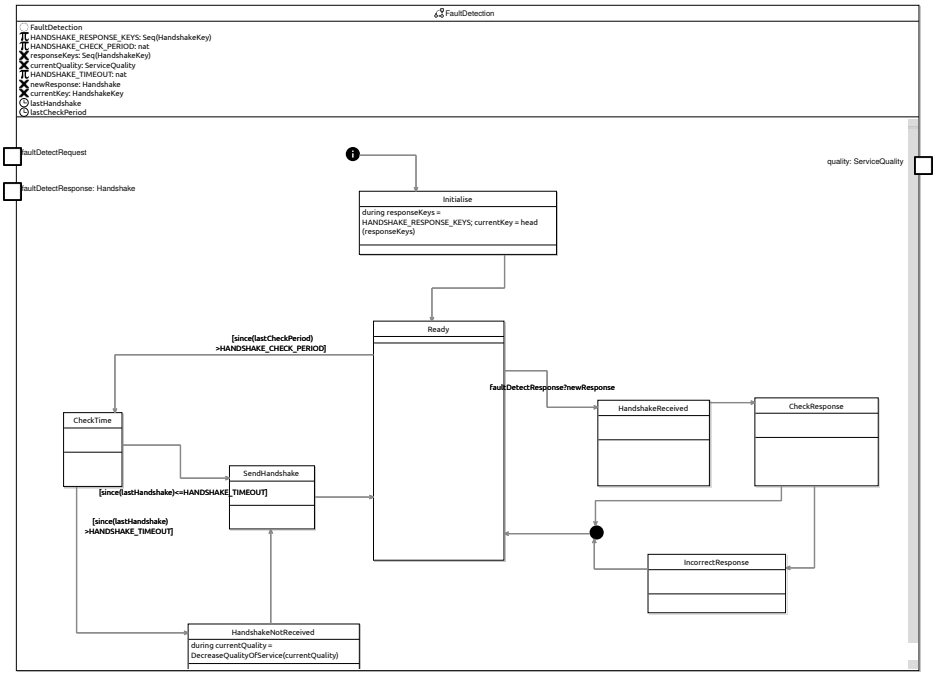


Figure C.15: Lutz State State Machine

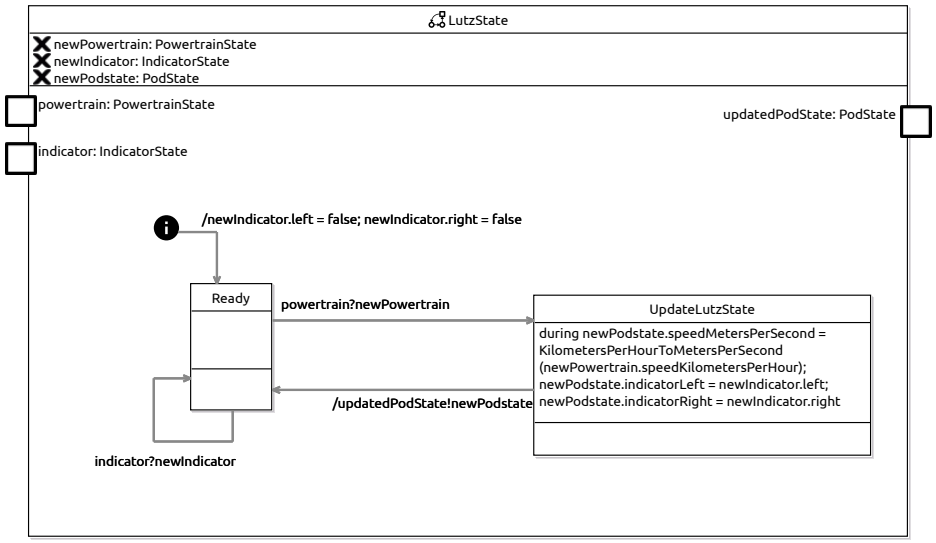


Figure C.16: Vehicle Control State Machine

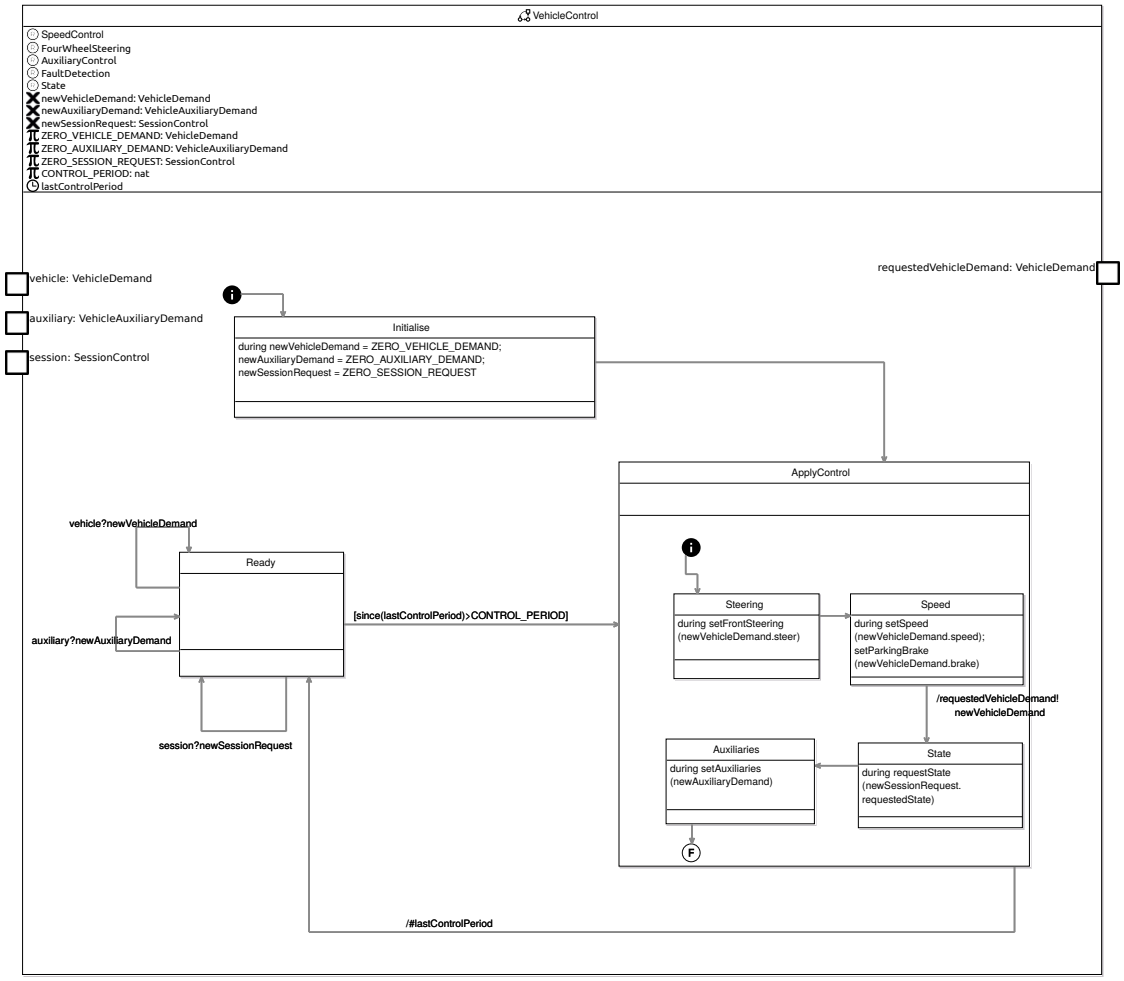


Figure C.17: Session Control State Machine

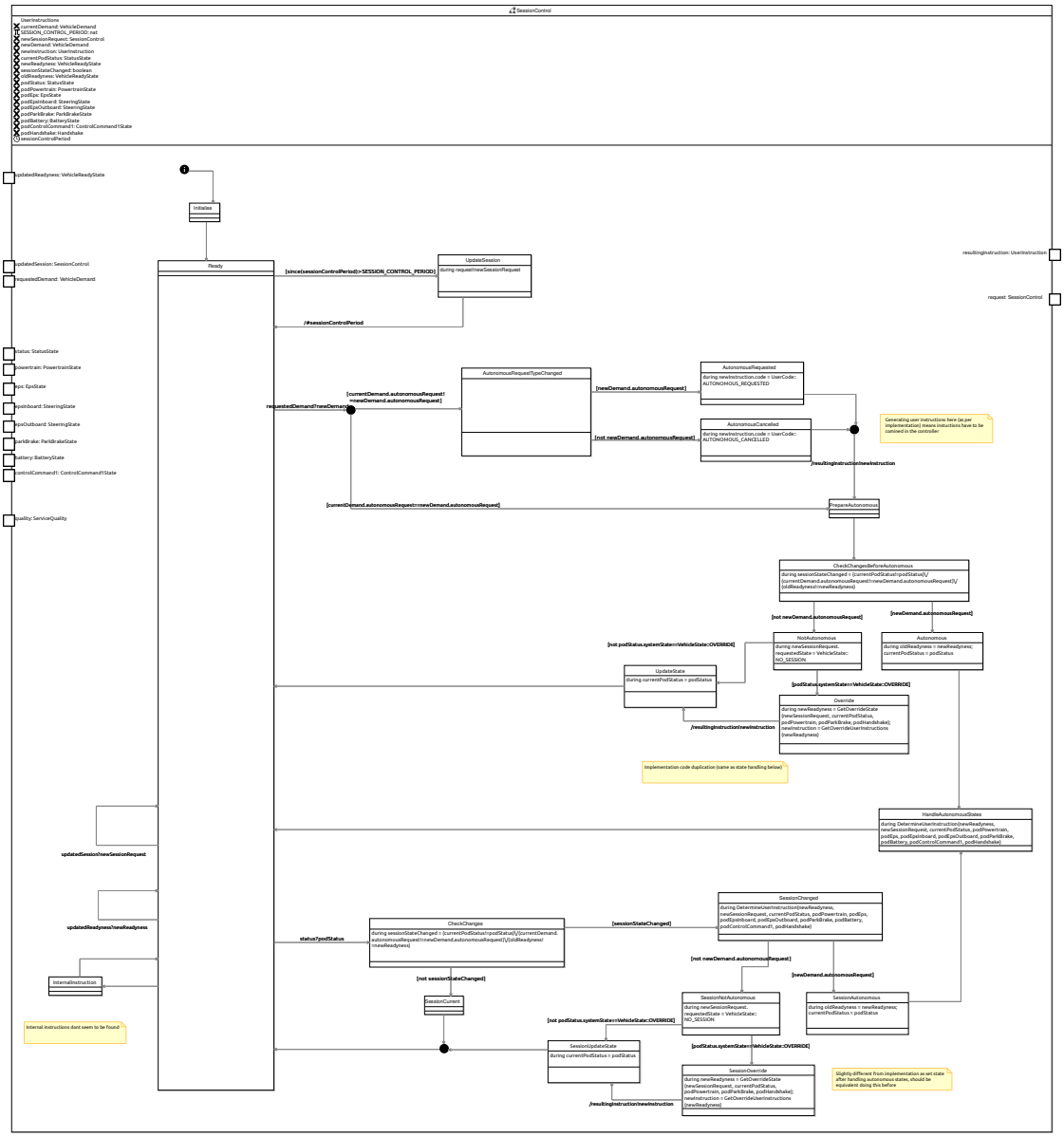


Figure C.18: Determine User Instruction Operation

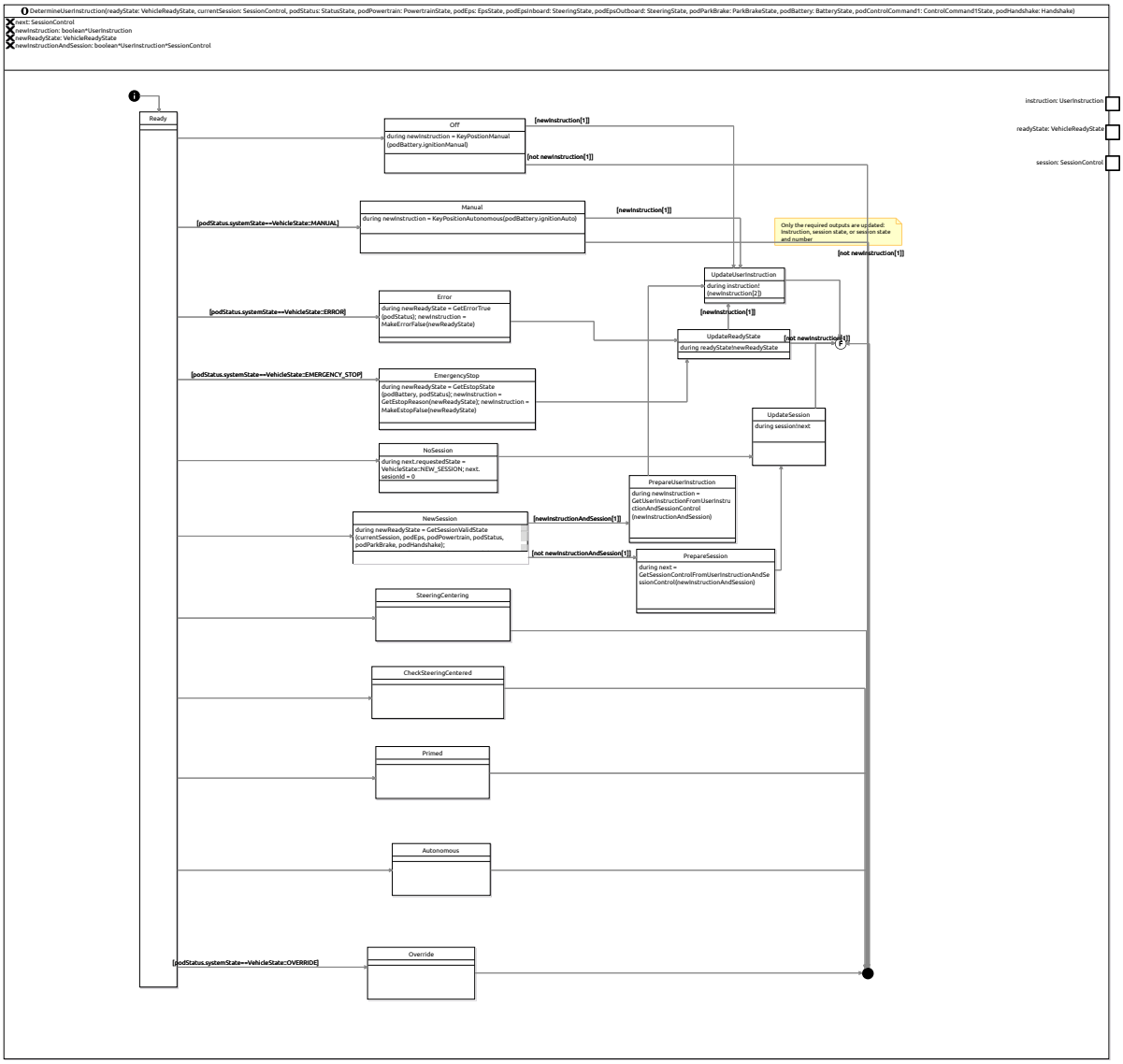
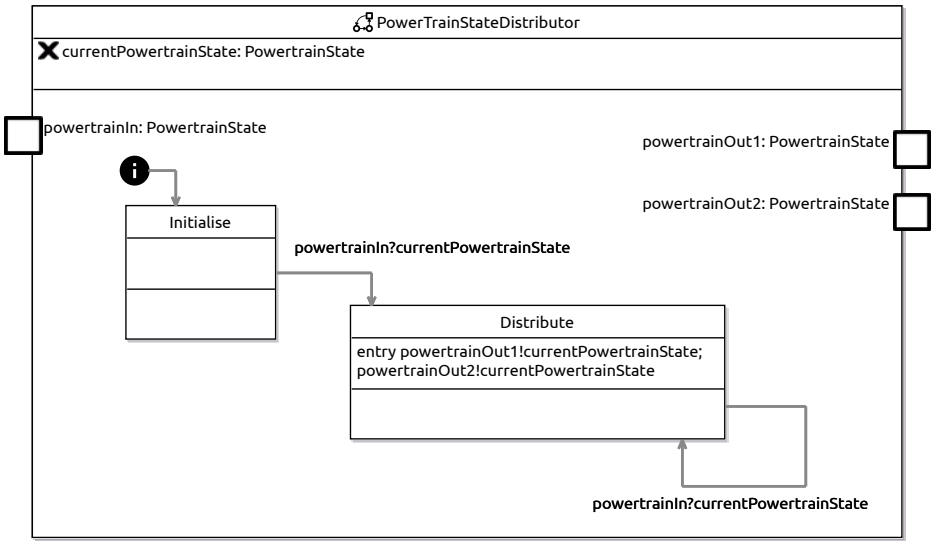
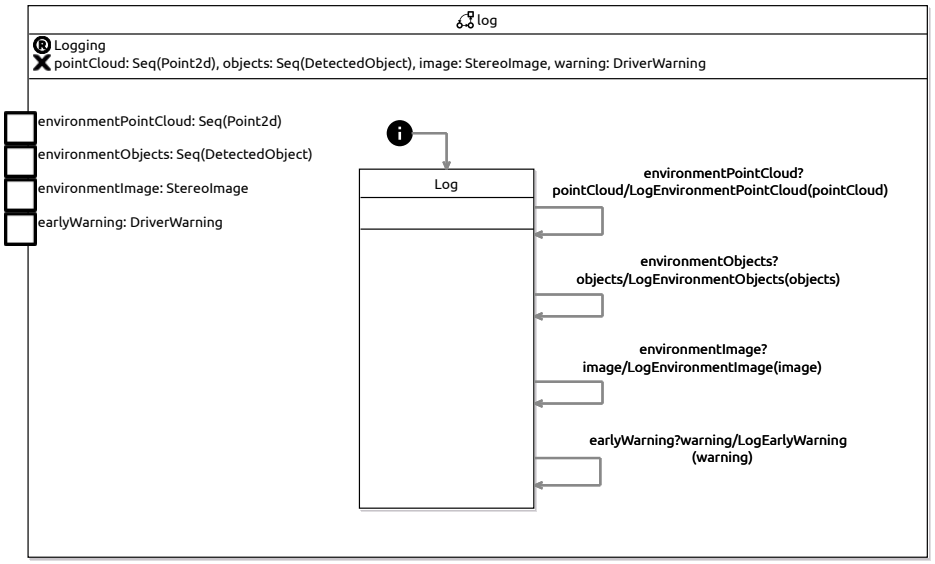


Figure C.19: Powertrain State Distributor State Machine



C.5 Controller - Logger

Figure C.20: Log State Machine



C.6 Verification Results - Autonomous Control System

Table C.23: The untimed results for the ACS DeadMansHandle state machine

Assertion	States	Transitions	Result
untimed autonomous_control_DeadMansHandle is deterministic (DMH_1) [failures divergences model]	4	16	true
untimed autonomous_control_DeadMansHandle is divergence free (DMH_2) [failures divergences model]	4	16	true
untimed autonomous_control_DeadMansHandle is deadlock free (DMH_3) [failures divergences model]	4	16	true
untimed autonomous_control_DeadMansHandle does not terminate (DMH_4)	4	16	true
untimed autonomous_control_DeadMansHandle _Initialise is reachable in autonomous_control_DeadMansHandle (DMH_5)	2	2	true
untimed autonomous_control_DeadMansHandle _Ready is reachable in autonomous_control_DeadMansHandle (DMH_6)	6	6	true
untimed autonomous_control_DeadMansHandle _TranslateInput is reachable in autonomous_control_DeadMansHandle (DMH_7)	22	33	true

Table C.24: The timed results for the ACS DeadMansHandle state machine

Assertion	States	Transitions	Result
autonomous_control_DeadMansHandle is deterministic (DMH_1) [failures divergences model]	522	874	true
autonomous_control_DeadMansHandle is divergence free (DMH_2) [failures divergences model]	522	874	true
autonomous_control_DeadMansHandle is deadlock free (DMH_3) [failures divergences model]	522	874	true
autonomous_control_DeadMansHandle does not terminate (DMH_4)	522	874	true
autonomous_control_DeadMansHandle_Initialise is reachable in autonomous_control_DeadMansHandle (DMH_5)	4	4	true
autonomous_control_DeadMansHandle_Ready is reachable in autonomous_control_DeadMansHandle (DMH_6)	9	9	true
autonomous_control_DeadMansHandle _TranslateInput is reachable in autonomous_control_DeadMansHandle (DMH_7)	135	244	true

Table C.25: The untimed results for the ACS Geofence state machine for a single sequence

Assertion	States	Transitions	Result
autonomous_control_Geofence is deterministic (GEO_1) [failures divergences model]	124	328	true
autonomous_control_Geofence is divergence free (GEO_2) [failures divergences model]	124	328	true
autonomous_control_Geofence is deadlock free (GEO_3) [failures divergences model]	124	328	true
autonomous_control_Geofence does not terminate (GEO_4)	124	328	true
autonomous_control_Geofence_Initialise is reachable in autonomous_control_Geofence (GEO_5)	3	3	true
autonomous_control_Geofence_Ready is reachable in autonomous_control_Geofence (GEO_6)	168	208	true
autonomous_control_Geofence_SetSpeedLimit is reachable in autonomous_control_Geofence (GEO_7)	533	656	true

Table C.26: The timed results for the ACS Geofence state machine for a single sequence

Assertion	States	Transitions	Result
autonomous_control_Geofence is deterministic (GEO_1) [failures divergences model]	9280	11279	true
autonomous_control_Geofence is divergence free (GEO_2) [failures divergences model]	9280	11279	true
autonomous_control_Geofence is deadlock free (GEO_3) [failures divergences model]	9280	11279	true
autonomous_control_Geofence does not terminate (GEO_4)	9280	11279	true
autonomous_control_Geofence_Initialise is reachable in autonomous_control_Geofence (GEO_5)	4	5	true
autonomous_control_Geofence_Ready is reachable in autonomous_control_Geofence (GEO_6)	410	412	true
autonomous_control_Geofence_SetSpeedLimit is reachable in autonomous_control_Geofence (GEO_7)	3083	3166	true

Table C.27: The untimed results for the ACS AutonomousDemand state machine

Did not complete, terminated after 3 hours - last line from log: Constructed 1,490,106,449 states and 2,561,000,000 transitions

Table C.28: The untimed results for the ACS Localise state machine

Assertion	States	Transitions	Result
autonomous_control_Localise is deterministic (LOC_1) [failures divergences model]	10	1293	true
autonomous_control_Localise is divergence free (LOC_2) [failures divergences model]	10	1293	true
autonomous_control_Localise is deadlock free (LOC_3) [failures divergences model]	10	1293	true
autonomous_control_Localise does not terminate (LOC_4)	10	1293	true
autonomous_control_Localise_Ready is reachable in autonomous_control_Localise (LOC_5)	7	1286	true
autonomous_control_Localise_TranslateLocation is reachable in autonomous_control_Localise (LOC_6)	24	1303	true

Table C.29: The timed results for the ACS Localise state machine

Assertion	States	Transitions	Result
autonomous_control_Localise is deterministic (LOC_1) [failures divergences model]	206084	238085	true
autonomous_control_Localise is divergence free (LOC_2) [failures divergences model]	206084	238085	true
autonomous_control_Localise is deadlock free (LOC_3) [failures divergences model]	206084	238085	true
autonomous_control_Localise does not terminate (LOC_4)	206084	238085	true
autonomous_control_Localise_Ready is reachable in autonomous_control_Localise (LOC_5)	6404	6405	true
autonomous_control_Localise_TranslateLocation is reachable in autonomous_control_Localise (LOC_6)	48644	49925	true

Table C.30: The untimed results for the ACS GoalDemandGeneration state machine

FDR unable to complete initialisation - no log trace given

Table C.31: The untimed results for the ACS DriveByWireDemand state machine

Did not complete, terminated after 3 hours - last line from log: Found 179,676,000 processes including 582 names

Table C.32: The untimed results for the ACS ControlDemandSelection state machine

Assertion	States	Transitions	Result
untimed autonomous_control_ControlDemand Selection is deterministic (CDS_1) [failures divergences model]	1	1	false
untimed autonomous_control_ControlDemand Selection is divergence free (CDS_2) [failures divergences model]	2	2161	true
untimed autonomous_control_ControlDemand Selection is deadlock free (CDS_3) [failures divergences model]	2	2161	false
untimed autonomous_control_ControlDemand Selection does not terminate (CDS_4)	2	2161	true
untimed autonomous_control_ControlDemand Selection_Initialise is reachable in autonomous_control_ControlDemand Selection (CDS_5)	2	2	true
untimed autonomous_control_ControlDemand Selection_Ready is reachable in autonomous_control_ControlDemand Selection (CDS_6)	6	6	true
untimed autonomous_control_ControlDemand Selection_UpdateAutonomousDemandStatus is reachable in autonomous_control_ControlDemand Selection (CDS_7)	21	2179	true
untimed autonomous_control_ControlDemand Selection_CheckPriority is reachable in autonomous_control_ControlDemand Selection (CDS_8)	43	2206	true

Continued on next page

Table C.32 – continued from previous page

Assertion	States	Transitions	Result
untimed autonomous_control_ControlDemand Selection_CheckPriority_CheckDriveByWireAbdication is reachable in autonomous_control_ControlDemand Selection (CDS_9)	40	2197	true
untimed autonomous_control_ControlDemand Selection_CheckPriority_DriveBy WireOld is reachable in autonomous_control_ControlDemand Selection (CDS_10)	71	2253	true
untimed autonomous_control_ControlDemand Selection_CheckPriority_HandleDriveBy WireAbdication is reachable in autonomous_control_ControlDemandSelection (CDS_11)	71	2253	true
untimed autonomous_control_ControlDemand Selection_CheckPriority_CheckAutonomous Abdication is reachable in autonomous_control_ControlDemand Selection (CDS_12)	99	2292	true
untimed autonomous_control_ControlDemand Selection_CheckPriority_AutonomousOld is reachable in autonomous_control_ControlDemand Selection (CDS_13)	111	2310	true
untimed autonomous_control_ControlDemand Selection_AcceptDemand is reachable in autonomous_control_ControlDemandSelection (CDS_14)	116	2318	false
untimed autonomous_control_ControlDemand Selection_CheckAllAbdicating is reachable in autonomous_control_ControlDemandSelection (CDS_15)	116	2318	false

Continued on next page

Table C.32 – continued from previous page

Assertion	States	Transitions	Result
untimed autonomous_control_ControlDemand Selection_DropDemand is reachable in autonomous_control_ControlDemand Selection (CDS_16)	116	2318	false
untimed autonomous_control_ControlDemand Selection_UpdateDriveByWireStatus is reachable in autonomous_control_ControlDemand Selection (CDS_17)	21	2178	true

Table C.33: The untimed results for the ACS VehicleSpeedLimiter state machine

Assertion	States	Transitions	Result
untimed autonomous_control_VehicleSpeed Limiter is deterministic (VSL_1) [failures divergences model]	3245	8666	true
untimed autonomous_control_VehicleSpeed Limiter is divergence free (VSL_2) [failures divergences model]	3245	8666	true
untimed autonomous_control_VehicleSpeed Limiter is deadlock free (VSL_3) [failures divergences model]	3245	8666	true
untimed autonomous_control_VehicleSpeed Limiter does not terminate (VSL_4)	3245	8666	true
untimed autonomous_control_VehicleSpeed Limiter_Initialise is reachable in autonomous_control_VehicleSpeed Limiter (VSL_5)	2	2	true
untimed autonomous_control_VehicleSpeed Limiter_Ready is reachable in autonomous_control_VehicleSpeed Limiter (VSL_6)	6	6	true
untimed autonomous_control_VehicleSpeed Limiter_LimitSpeed is reachable in autonomous_control_VehicleSpeed Limiter (VSL_7)	889	1758	true

Table C.34: The timed results for the ACS VehicleSpeedLimiter state machine

Assertion	States	Transitions	Result
autonomous_control_VehicleSpeedLimiter is deterministic (VSL_1) [failures divergences model]	104770	5967011	true
autonomous_control_VehicleSpeedLimiter is divergence free (VSL_2) [failures divergences model]	104770	5967011	true
autonomous_control_VehicleSpeedLimiter is deadlock free (VSL_3) [failures divergences model]	104770	5967011	true
autonomous_control_VehicleSpeedLimiter does not terminate (VSL_4)	104770	5967011	true
autonomous_control_VehicleSpeedLimiter _Initialise is reachable in autonomous_control_VehicleSpeedLimiter (VSL_5)	5	5	true
autonomous_control_VehicleSpeedLimiter _Ready is reachable in autonomous_control_VehicleSpeedLimiter (VSL_6)	10	10	true
autonomous_control_VehicleSpeedLimiter _LimitSpeed is reachable in autonomous_control_VehicleSpeedLimiter (VSL_7)	21856	27066	true

Table C.35: The untimed results for the ACS VehicleSpeedRatioLimiter state machine

Assertion	States	Transitions	Result
untimed autonomous_control_VehicleSpeedRatioLimiter is deterministic (VRL_1) [failures divergences model]	2163	5410	true
untimed autonomous_control_VehicleSpeedRatioLimiter is divergence free (VRL_2) [failures divergences model]	2163	5410	true
untimed autonomous_control_VehicleSpeedRatioLimiter is deadlock free (VRL_3) [failures divergences model]	2163	5410	true
untimed autonomous_control_VehicleSpeedRatioLimiter does not terminate (VRL_4)	2163	5410	true
untimed autonomous_control_VehicleSpeedRatioLimiter_Initialise is reachable in autonomous_control_VehicleSpeedRatioLimiter (VRL_5)	2	2	true
untimed autonomous_control_VehicleSpeedRatioLimiter_Ready is reachable in autonomous_control_VehicleSpeedRatioLimiter (VRL_6)	6	6	true
untimed autonomous_control_VehicleSpeedRatioLimiter_LimitSpeed is reachable in autonomous_control_VehicleSpeedRatioLimiter (VRL_7)	881	1748	true

Table C.36: The timed results for the ACS VehicleSpeedRatioLimiter state machine

Assertion	States	Transitions	Result
autonomous_control_VehicleSpeedRatioLimiter is deterministic (VRL_1) [failures divergences model]	64810	3575891	true
autonomous_control_VehicleSpeedRatioLimiter is divergence free (VRL_2) [failures divergences model]	64810	3575891	true
autonomous_control_VehicleSpeedRatioLimiter is deadlock free (VRL_3) [failures divergences model]	64810	3575891	true
autonomous_control_VehicleSpeedRatioLimiter does not terminate (VRL_4)	64810	3575891	true
autonomous_control_VehicleSpeedRatioLimiter _Initialise is reachable in autonomous_control_VehicleSpeedRatioLimiter (VRL_5)	5	5	true
autonomous_control_VehicleSpeedRatioLimiter _Ready is reachable in autonomous_control_VehicleSpeedRatioLimiter (VRL_6)	10	10	true
autonomous_control_VehicleSpeedRatioLimiter _LimitSpeed is reachable in autonomous_control_VehicleSpeedRatioLimiter (VRL_7)	15364	18398	true

Table C.37: The untimed results for the ACS AgeChecker state machine

Did not complete, terminated after 3 hours - last line from log: Constructed 11,299,000 states

Table C.38: The untimed results for the ACS AuxiliaryDemandSelection state machine

Assertion	States	Transitions	Result
autonomous_control_AuxillaryDemandSelection is deterministic (ADS_1) [failures divergences model]	1	1	false
autonomous_control_AuxillaryDemandSelection is divergence free (ADS_2) [failures divergences model]	64	2113	true
autonomous_control_AuxillaryDemandSelection is deadlock free (ADS_3) [failures divergences model]	64	2113	true
autonomous_control_AuxillaryDemandSelection does not terminate (ADS_4)	64	2113	true
autonomous_control_AuxillaryDemandSelection_Initialise is reachable in autonomous_control_AuxillaryDemandSelection (ADS_5)	2	2	true
autonomous_control_AuxillaryDemandSelection_Ready is reachable in autonomous_control_AuxillaryDemandSelection (ADS_6)	6	6	true
autonomous_control_AuxillaryDemandSelection_CombineDemands is reachable in autonomous_control_AuxillaryDemandSelection (ADS_7)	54	86	true

Table C.39: The timed results for the ACS AuxiliaryDemandSelection state machine

Assertion	States	Transitions	Result
autonomous_control_AuxillaryDemandSelection is deterministic (ADS_1) [failures divergences model]	124	263	false
autonomous_control_AuxillaryDemandSelection is divergence free (ADS_2) [failures divergences model]	83738	409126	true
autonomous_control_AuxillaryDemandSelection is deadlock free (ADS_3) [failures divergences model]	83738	409126	true
autonomous_control_AuxillaryDemandSelection does not terminate (ADS_4)	83738	409126	true
autonomous_control_AuxillaryDemandSelection _Initialise is reachable in autonomous_control_AuxillaryDemandSelection (ADS_5)	15	21	true
autonomous_control_AuxillaryDemandSelection _Ready is reachable in autonomous_control_AuxillaryDemandSelection (ADS_6)	25	36	true
autonomous_control_AuxillaryDemandSelection _CombineDemands is reachable in autonomous_control_AuxillaryDemandSelection (ADS_7)	4081	10232	true
clock autonomous_control_AuxillaryDemand Selection _auxiliarySelectionPeriod is initialised (ADS_8) [failures divergences model]	83738	409126	true

Table C.40: The untimed results for the LOD LocationDistributor state machine

Assertion	States	Transitions	Result
utility_LocationDistributor is deterministic (LOD_1) [failures divergences model]	9	13	true
utility_LocationDistributor is divergence free (LOD_2) [failures divergences model]	9	13	true
utility_LocationDistributor is deadlock free (LOD_3) [failures divergences model]	9	13	true
utility_LocationDistributor does not terminate (LOD_4)	9	13	true
utility_LocationDistributor_Initialise is reachable in utility_LocationDistributor (LOD_5)	2	2	true
utility_LocationDistributor_Distribute is reachable in utility_LocationDistributor (LOD_6)	24	27	true

Table C.41: The timed results for the LOD LocationDistributor state machine

Assertion	States	Transitions	Result
utility_LocationDistributor is deterministic (LOD_1) [failures divergences model]	80	117	true
utility_LocationDistributor is divergence free (LOD_2) [failures divergences model]	80	117	true
utility_LocationDistributor is deadlock free (LOD_3) [failures divergences model]	80	117	true
utility_LocationDistributor does not terminate (LOD_4)	80	117	true
utility_LocationDistributor_Initialise is reachable in utility_LocationDistributor (LOD_5)	3	3	true
utility_LocationDistributor_Distribute is reachable in utility_LocationDistributor (LOD_6)	40	49	true