RoboArch: Architectural Modelling for Robotic Applications

William Barnett

PhD

University of York
Computer Science

September 2022

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 in other areas so applying them to the field of robotics can contribute to the goal of building reliable and safe systems.

In this thesis we present RoboArch, a notation for describing the architectures and patterns of robotic systems software supported by the formally defined semantics of RoboChart. RoboChart is a DSL for modelling the behaviour of robot software controllers using state machines.

We describe RoboArch from the top-down. First, we examine the role of robotics software architectures in the development of robotic systems by reviewing five robotics architectures, and five DSLs. Next, for the layered architectural pattern, the RoboArch notation is introduced; we provide a metamodel, well-formedness conditions, and transformation rules to RoboChart. Further, we characterise two patterns: reactive skills and subsumption, which can be used by a layer.

Finally, we discuss a tool and its implementation for the evaluation of RoboArch and automation of the rules as model transformations. We use a case study of a small obstacle avoidance system to demonstrate: the application of the reactive skills pattern using RoboArch and the expected properties of the architecture that can be proven using the generated RoboChart model CSP semantics.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisors, Ana Cavalcanti and Alvaro Miyazawa, for their dedication and guidance throughout this journey. Furthermore, I would like to thank my colleagues in the RoboStar group and examiners Leandro Soares Indrusiak and Andrew Butterfield. I would also like to express gratitude to the UK's Engineering and Physical Sciences Research Council (EPSRC) and the Royal Academy of Engineering (RAEng) for the funding they have provided, without which this work would not have been possible.

Last but by no means least, my friends and family for their continuous support each in their own individual ways.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author, except where otherwise stated. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as references. The material from Chapters 2, 3, and 4 has been accepted for publication as the following work.

[1] W. Barnett, A. Cavalcanti and A. Miyazawa, "Architectural modelling for robotics: RoboArch and the CorteX example," Frontiers in Robotics and AI, vol. 9, 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/frobt.2022.991637

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 of wide interest in the robotics community. It is based on novel domain-specific notation that we call RoboArch. It embeds architectural concepts and enables the generation of RoboChart model sketches through model transformations.

This chapter details the motivation and objectives of our work and is structured as follows. Section 1.1 explains the motivation behind our work, Section 1.2 specifies the main objectives of our work, and finally Section 1.3 lays out the structure of this document.

1.1 Motivation

Advances in technology are enabling the development of robotic systems for an increasingly diverse range of applications. 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.

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 becomes increasingly complex.

Verifying complex robotic systems at design time for all possible scenarios is a significant challenge, as the system may encounter situations that have not been considered during design [6]. Complete design time verification of these types of systems is not feasible using existing methods alone [6]. To enable the successful realisation of forthcoming robotic systems, techniques and 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, a beneficial technique has been the definition of a system's architecture during its design. The description of a system's architecture provides a structural representation 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].

From experience, practitioners have identified structures and relationships within system architectures that solve recurring problems. These solutions have been generalised as architectural patterns that are reuseable in the design of new systems. Some examples of well-known architectural patterns include the client-server pattern used in internet applications, and the layered pattern [8, p. 31] used in embedded systems. There has also been the development of standardised domain-specific architectural patterns to ease the integration of system components and promote reuse of outputs. One notable example of a standardised architectural pattern is Autosar from the automotive industry [9].

For the robotics domain, many projects still choose to design a custom architecture. This means that there have been many proposals, but there is no single widely adopted architectural pattern. On the other hand, 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.

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 artefact(s) 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. In this way 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, a single language that is general enough to describe all of the required 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: BCM [14], RoboML [15], and SmartSoft [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], and so they can 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 and mathematical descriptions. Additionally, RoboChart supports automatic verification of properties such as deadlock and livelock freedom 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 satisfies key properties of interest.

To date, RoboChart has been used to model nineteen proof-of-concept case studies, which include an autonomous vehicle implemented in ROS [20]. 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. None of them adopt or describe an elaborate software architecture, beyond straightforward use of parallelism.

So that RoboChart can have a significant impact on to the wider verification of robotic systems, it must be widely accepted by developers. Therefore, RoboChart and its associated tools must be able to effectively support the modelling of software controllers of real robotic systems. For larger robotic systems, support for modelling taking advantage of commonly used architectural patterns can enable explicit modelling of the structure of systems with potential impact on reuse and compositional design and reasoning.

The RoboChart component model does not directly include concepts from the robotics architectural patterns. A key advantage of DSLs is their close correspondence to terminology and concepts of the target domain and the ability to be able to generate code instructions in a selected language from the domain concepts specified by the DSL [21, p. 3].

This motivates the primary research question for our work.

> What are the architectural patterns commonly used in robotics and how can they be used in behavioural models of robotic systems for verification?

In the following section the main objectives for the proposed work are specified.

1.2 Objectives and Contributions

The goal of our work is to contribute to the advancement of software modelling and verification for robotics, to support the creation of robotic

systems that are safe and robust. In order to achieve this goal, we have the following objectives:

- to identify architectural patterns that are used by robotic systems software;
- to contribute to the practice of modelling using RoboChart by providing automated support and guidance for modelling using common architectural patterns;
- to enrich the facilities for architectural modelling in RoboChart via the design of a new language that can be used to describe the architecture of control software in terms of concepts adopted by the robotics community;
- to describe precisely the design of the new language: metamodel, well-formedness conditions, and RoboChart semantics;
- to enable the generation of partial RoboChart models via model transformation.

Via a comprehensive study of the literature, the commonly used layered architectural pattern has been identified as a common option in the robotics community. We have also identified a number of architectural patterns used in the design of one or more layers. We describe the main concepts in all these architectural patterns.

To facilitate the modelling of systems that use these architectures, we have designed an architectural language: RoboArch. The definition of RoboArch formalises the concepts adopted in the various architectures. This provides guidance for the design of systems.

Additional architectures can be identified and characterised via extensions of RoboArch. Here we give a precise characterisation of an architectural pattern that can be used for the design of a particular layer. Others can be captured in a similar way. The pattern's characterisation incudes a description of its fundamental concepts, a metamodel that forms the foundation of the RoboArch notation, and well-formedness conditions.

A RoboArch model is an instance of one or more architectural patterns that are defined by the RoboArch metamodel and well-formedness conditions. The RoboArch semantics define how such an instance can be described directly in RoboChart. The semantics is given by a set

of model-transformation rules. The automation of the model transformations provide a mechanism that can be used to the advantage of practitioners. They can describe software controllers in RoboArch. This amounts to describing an instance of an architectural pattern formalised by RoboArch. From that, they can obtain a partial RoboChart model that can be completed and used as a basis for verification and for generation of simulations and tests.

The experience gained using RoboChart has been reflected in the models that RoboArch generates helping to establish the pragmatics of RoboChart and assist developers in its use.

1.3 Document Structure

Figure 1.1 shows the structure of the document its chapters, main subsections, and the relationships between them. The document is structured as follows.

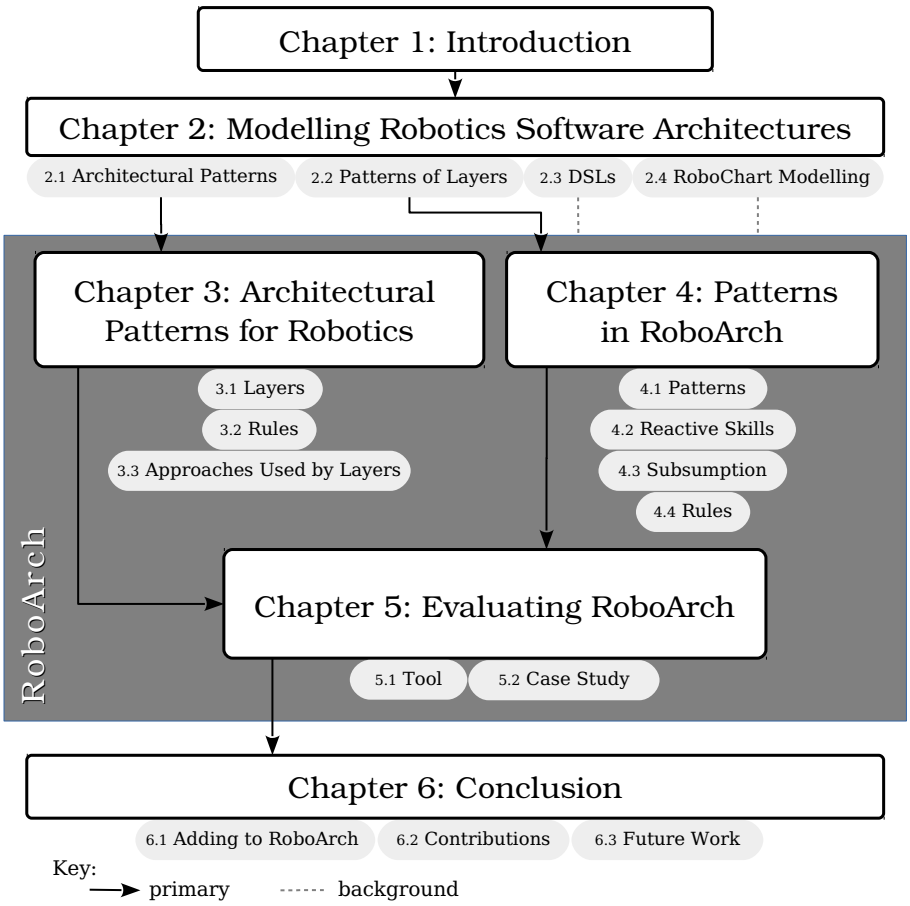
Chapter 2 surveys the role of software architecture in robotic systems development, and domain-specific languages (DSL) for robotics. The chapter ends by introducing RoboChart's graphical notation and verification facilities, and demonstrating how robotic systems can be modelled using RoboChart.

Chapter 3 outlines the approach used to characterise layered robotics architectural patterns as a metamodel and well-formedness conditions that define the RoboArch notation. We also discuss a selection of patterns that can be used by each of the layers. In this chapter we also define the metanotation used to specify the rules for translating from RoboArch to RoboChart. Finally, we give the core transformation rules for the layered architectural pattern.

Chapter 4 presents an analysis of the structural components of the control-layer reactive-skills pattern and how it can be modelled using RoboChart. We define the pattern's fundamental concepts, via the metamodel and well-formedness conditions for the elements of RoboArch that capture the pattern. We also formalise the pattern by specifying how its instances can be transformed using the rule definitions.

Chapter 5 evaluates RoboArch demonstrating how the architectural structure of robotic systems can be described and sketches of RoboChart

Figure 1.1: Document overview



models automatically generated. We present a tool that automates our transformation rules and enables their automatic application. We describe its architecture, and justify its role in evaluating RoboArch and its semantics as defined by the rules. Finally, we present a case study on the use of the reactive skills pattern. The verification of its RoboChart model provides evidence that RoboArch captures key properties of the reactive-skills patterns.

Chapter 6 concludes with a discussion of the contributions made, including assumptions made and future work.

2 Modelling Robotics Software Architectures

DSLs for robotics that can describe the architecture of a system provide a mechanism for development of increasingly complex robotic systems.

There have been many definitions for architecture put forward over the last thirty years [22]. We adopt the definition given by ISO 42010:2011 since it summarises the common themes of structure and relationships:

> "**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." [23]

The evolving definition has contributed to discrepancies between related terminology, namely, architectural patterns and styles. These terms are closely related in that they describe a general solution to a recurring problem and so they have been used interchangeably [24].

Clements et al. distinguish these terms, stating patterns additionally must include details on the problem and its context [25]. This distinction means architectural patterns mirror object-oriented software design patterns [26] albeit their concern is the higher system level of abstraction [8, p. 12]. Here, we adopt the term architectural patterns to describe general solutions. Our work with RoboArch formalises some patterns and provides a framework for the formalisation of others.

Achtelik et al. use the term architectural styles when discussing robotics system architecture in the handbook of robotics [10, Ch. 12] indicating that there is scope to refine styles into more precise and well-defined patterns.

In the robotics community, there are no widely accepted architectural patterns. Many different robotics architectures have been proposed with no single architecture fitting all applications [10, p. 283]. Instead, domain experts propose architectures that solve problems for their robotic system of interest.

In order to evaluate robotics DSLs support for modelling architectural patterns it is necessary to understand the structures and relationships

used in robotics software architectures. 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 those in which robotics has a role to play, is a particular challenge in its own right and one which we are not going to address.

Instead, we cover several architectural patterns, and contribute an approach that can be used to consider others in the future. Section 2.1 presents robotics architectures that have been used over the last twenty years. In particular, we focus on those that have been put forward as a solution to general problems, and, therefore, we recognise them as patterns. Section 2.2 gives a detailed account of some patterns, for each the identifies the concepts for each. Section 2.3 reviews a selection of DSLs for robotics. In doing so, we justify the suitability of RoboChart for our work and consider its wider applicability to other notations. 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. Section 2.4 presents an example of how a robot can be modelled using RoboChart. We conclude in Section 2.5, with some final considerations.

2.1 Robotics Software Architectural Patterns

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 architectural patterns for robotic systems have been developed.

Historical architectural patterns include Sense Plan Act (SPA) [10, p. 285] and subsumption [27]. SPA is an example of a pattern 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 architectural pattern that is reactive, where the environment is constantly sensed and used to directly shape the robot's actions. A robot using subsumption responds rapidly to a changing world; however, complex actions are difficult to achieve.

More recent hybrid architectural patterns combine the principles from SPA and subsumption to benefit from both the deliberative and reactive properties.

In order to determine the important architectural patterns 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-two architectural patterns used by robotics systems have been identified from the literature; these are listed in Table 2.1. Five

Table 2.1: The patterns identified from the literature.

Pattern	**Focus**	**Year**
CoSiMA [28]	Safe real-time robots	2018
◆ IRSA [29]	Autonomous robots	2018
◆ SERA [30]	Decentralised teams	2018
◆ CARACaS [31]	Autonomous robots	2011
Aerostack [32]	Autonomous unmanned aerial systems	2017
EFTCoR [33]	Service robot control	2006
Syndicate [34]	Autonomous teams	2006
DDX [35]	Distributed robot controllers	2004
◆ CLARAty [36]	Autonomous robots	2001
HARPIC [37]	Autonomous robots	2001
◆ LAAS [38]	Autonomous robots	1998
Remote Agent [39]	Autonomous robots	1998
ORCCAD [40]	Robot control	1998
Planner Reactor [41]	Autonomous robots	1995
Reactive Skills [42]	Autonomous robots	1994
CIRCA [43]	Real-time intelligent robots	1993
ATLANTIS [44]	Autonomous robots	1992
Layered Competencies [45]	Autonomous robots	1991
Motor Schema [46]	Robot control	1989
NASREM [47]	Autonomous robots	1989
AuRA [48]	Autonomous robots	1987
Subsumption [27]	Autonomous robots	1986

Legend: ◆ Selected for further discussion.

have been selected for discussion based upon evidence of application, reuse, and activity of development. The collective publications that focus on an architectural pattern have been used to find evidence of application, with the scale of any documented application used to give preference

patterns that have been used in large deployments in the real world. The number of publications where an architectural pattern was used in a new application has been used to assess reuse. Preference has been given to patterns with recent activity, determined by the date and frequency of publications where the pattern has been used.

Sections 2.1.1 to 2.1.5 presents the selected architectural patterns, and provides a review of the discussed important aspects. Finally Section 2.1.6, evaluates the state of architectural patterns for robotics and their structure.

2.1.1 LAAS

The LAAS architectural pattern was developed at LAAS[1] in 1998 for autonomous robots. A fundamental goal of LAAS is to provide both deliberative and reactive capabilities required for autonomy [38].

The LAAS pattern 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 determined by the decision layer.

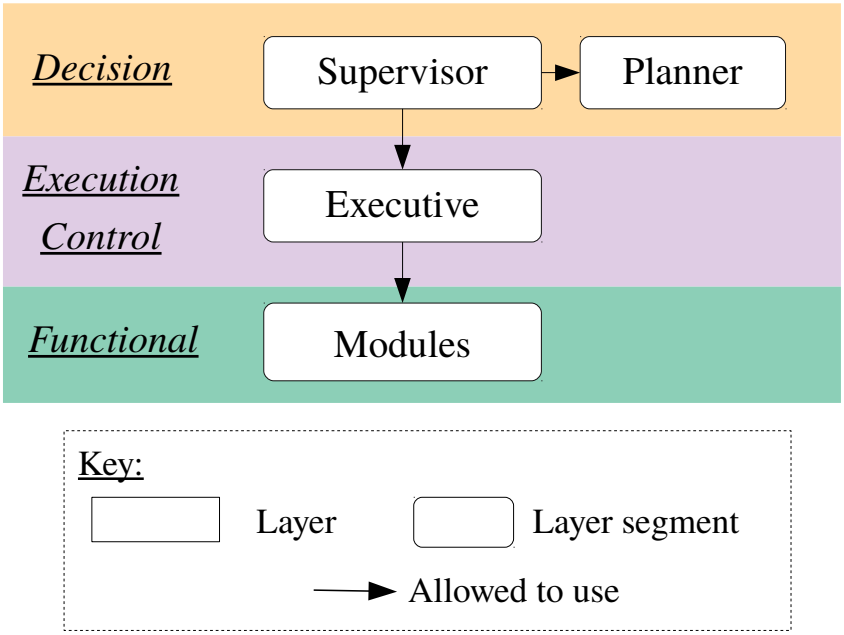
Decision Layer plans the sequence of actions necessary to achieve mission goals and supervises the execution of the plans.

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 [38]. An example of a data resource is 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 (G^{en}oM) can be used to generate module source code. To generate a module's source code, G^{en}oM combines a formalised description of the module along with pieces of code (codels) describing the module's algorithm.

The services provided by the 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

[1] Laboratory for Analysis and Architecture of Systems CNRS

Figure 2.1: The LAAS architectural pattern layer diagram.



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 triggers 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, and the associated response would be 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 has to notify the supervisor and obtain an updated plan.

These 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 [38].

LAAS has been used in the implementation of the ADAM rough terrain planetary exploration rover [49], and of three Hilare autonomous environment exploration robots as part of the MARTHA European project [50].

More recently, Behaviour Interaction Priority (BIP) models have been used to verify the functional layer of the LAAS pattern [51]. Functional layers described using GenoM 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

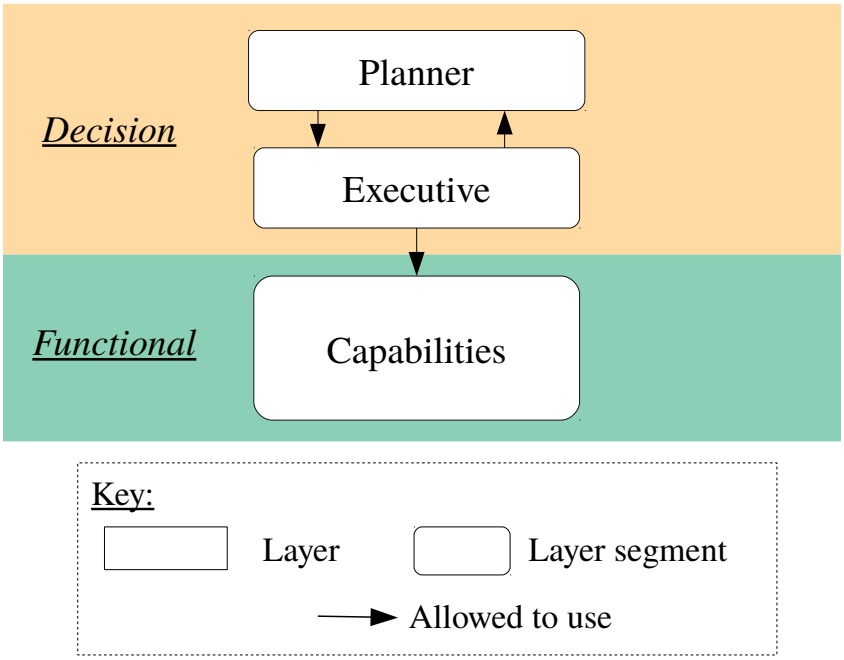
Coupled Layer Architecture for Robotic Autonomy (CLARAty) was developed at JPL in 2001 for planetary surface-exploration rovers. CLARAty is designed to be reusable and to support multiple robot platforms; it consists of two-layers formed by combining the planning and executive layers from a three-layer architecture [36]. A key concept defined by the CLARAty architectural pattern is granularity, which reflects the varying levels of deliberativeness available to the robotic system.

The CLARAty pattern is made up of the following layers:

Functional Layer is the interface to the system's hardware capabilities.

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

Figure 2.2: The CLARAty architectural pattern layer diagram.



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 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 pattern.

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 [52]. The structure of the functional layer means that the decision layer has a choice between selecting more reactive or more deliberative functions from the available capabilities. Use of the more reactive functions means that 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 pattern has been used for a variety of robot platforms: Rocky 8, FIDO, ROCKY 7, K9 Rovers, and ATRV Jr. COTS platform [53]. The different platforms have a variety of deployment architectures, from a single processor requiring hard real-time scheduling, and distributed microprocessors using soft real-time scheduling.

2.1.3 CARACaS

Control Architecture for Robotic Agent Command and Sensing (CARACaS) is an architectural pattern developed at JPL[2] in 2011 for control of autonomous underwater vehicles (AUV), and autonomous surface vehicles (ASV) [31]. 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

[2]NASA Jet Propulsion Laboratory

conditions and mission goals.

The five main elements of the CARACaS pattern are as follows:

Actuators interfaces the actuators of the vehicle.

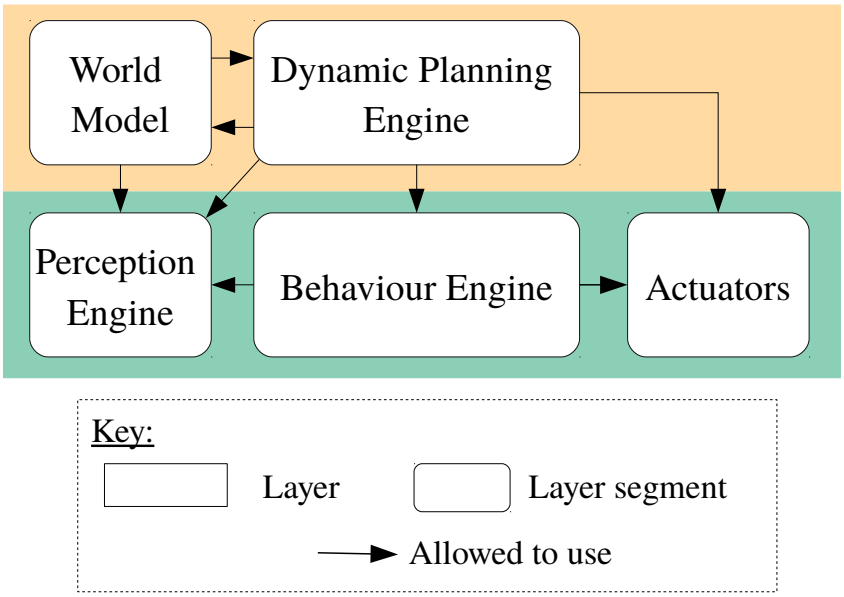
Behaviour Engine coordinates and enables the composition of behaviours acting on the vehicles actuators.

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.

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

Figure 2.3: The CARACaS architectural pattern layer diagram.



The Behaviour Engine makes use of behaviour composition and coordination methods developed as part of a previous multi-agent control architectural pattern CAMPOUT [54]. 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) [55]. 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 [31]. 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 [31]; 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 [31] 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 a pattern 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 [56]. 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 [29]. The IRSA architectural pattern uses concepts from the other patterns: CARACaS and CLARAty.

The main elements of the IRSA pattern are as follows:

Primitive provides 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.

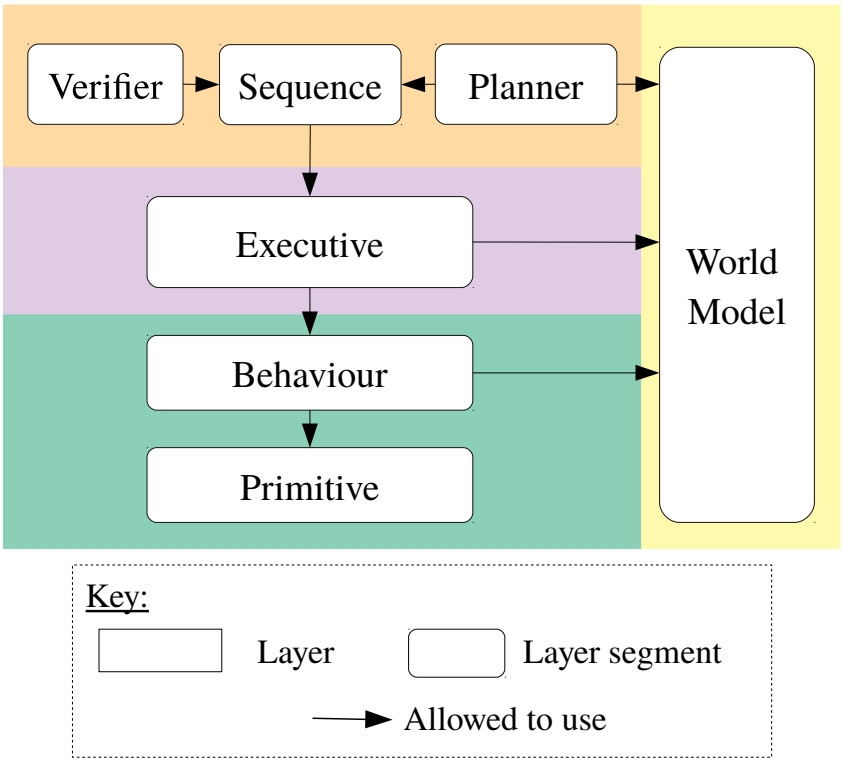
Planner uses the system state from the world model to produce the sequence of command instructions to be executed.

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 of local and global state information.

Figure 2.4: The IRSA architectural pattern layer diagram.



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

The IRSA pattern 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 primitive. Both the behaviour and the primitive elements provide control over the robot; therefore, these two elements can be placed in the bottom layer, 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, 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 architectural pattern 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 [29].

2.1.5 SERA

The Self-adaptive dEcentralised Robotic Architecture (SERA) has been developed at the Chalmers University of Technology in 2018 [30]. SERA's

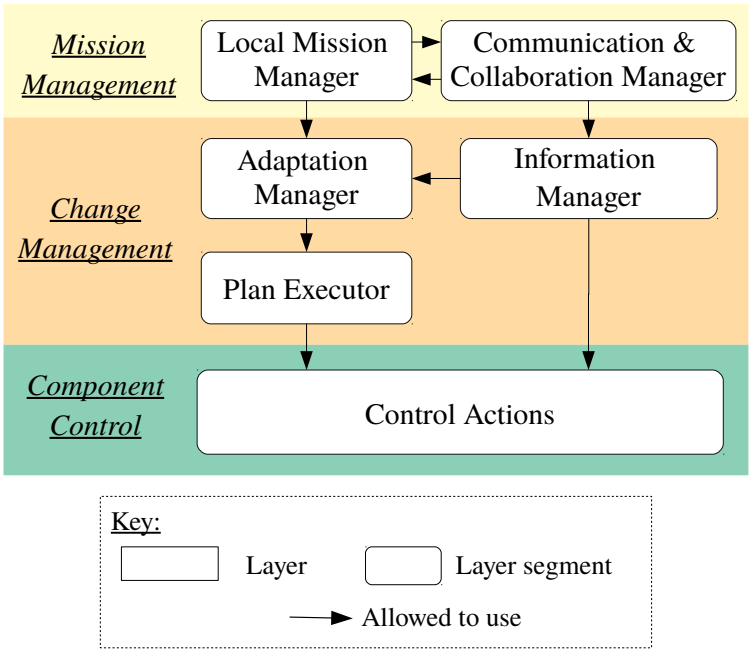
primary goal is to support decentralised self-adaptive collaboration between robots or humans, and it is based on the 3-layer self-management architectural pattern [57]. SERA has been evaluated in collaboration with industrial partners participating in the Co4Robots H2020 EU project [30]. The layers of the SERA pattern 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 to synchronise and achieve the global mission.

Figure 2.5: The SERA architectural pattern 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 pattern 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 pattern 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 architectural patterns. Generally no particular pattern or selection of patterns are widely used across different robotic systems. There is a tendency for each project to establish its own pattern. Between research groups, however, there is some reuse of patterns; for example, IRSA is being used for a variety of space testbeds and the RoboSimian robot.

Layers are a common theme among many of the recent architectural patterns. Even when layers have not been explicitly specified, the elements of a pattern are structured such that they can be mapped onto a layered architectural pattern. All patterns have a functional layer that in-

Table 2.2: Robotics architectural pattern summary

Pattern	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 architectural pattern.

teracts 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 architectural patterns because every robot requires a means to sense and interact with its environment. From the patterns surveyed, two categories of control approach are used by the functional layer: service or behavioural. Both of the terms functional and behavioural have had varying definitions in the context of robotics software. For our purposes here, that is, the identification and modelling of patterns, we distinguish them as follows. A service-based control approach provides an interface with a close correspondence to the services provided by the robotic platform. A behavioural-based control approach provides an interface that offers services that can be composed of other services. CLARAty, LAAS and SERA are all examples of patterns that have a service-based functional layer, whereas, CARACaS and IRSA have behavioural-based functional layers.

Patterns 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 layers. 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.

Patterns 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 pattern supports three techniques to arbitrate behaviours, whereas IRSA does not specify any arbitration mechanisms, leaving it open for developers decide for each project.

Examples of behavioural control patterns that can be used for functional layer include subsumption [27] and reactive skills as used by the control layer of 3T [58].

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.

Patterns 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 architectural pattern is placed.

Some patterns 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 pattern lends itself to the addition of new layers for extending the level of system capability.

The review of architectural patterns discussed in this section, provides details on the structure and control techniques used in robotics software. This insight will be used as a foundation to guide the design of RoboArch

to support use of RoboChart to model robotic systems.

The power of architectural patterns is their ability to be reused and applied as required to solve similar problems in the design of systems. The widespread use of layers is very clear. All of the layered patterns surveyed, however, have been proposed without, offering much insight or flexibility into alternative pattern choices for the layers. In contrast, Achtelik et al. [10, Sec. 12.3] describe an almost compositional view where patterns can be mixed and matched. We take this view and build upon it in RoboArch to enable the precise descriptions of patterns used by the robotics community and facilitate the use of different combinations of these patterns.

The next section examines some patterns in detail and describes their concepts and relationships.

2.2 Patterns of Robotics Software Layers

The review of robotics software architectural patterns in the previous section and its focus on the structure of software elements and relationships identified the common use of the layered pattern. In order to provide support for modelling systems that use these patterns, we must examine patterns in greater detail and precisely identify and describe the concepts and relationships of each pattern.

Section 2.2.1 and Section 2.2.2 each describe and provide an overview including the history and key concepts for the reactive skills and subsumption patterns, respectively.

2.2.1 Reactive Skills

The reactive skills pattern is used by the control layer of a three layer architecture [58] and is part of eight years of work by P. Bonasso et. al. that resulted in the architecture known as 3T. Reactive skills combine deliberation and reactivity to improve the robustness of tasks carried out by robotic systems. The pattern has been used for the control layer of a variety of different applications: a robot to identify people and approach them [59], a trash collecting robot [60], a robot that navigates a building [61], and in the automation of remote manipulation system procedures for the space shuttle [62].

The reactive skills pattern does not have its own programming language. Instead there is a framework [42] that allows skills to be implemen-

ted using other languages: C, C++, Pascal, LISP and REX. The framework allows skills implemented in different languages to be integrated and executed together.

No advice on development processes for developing a control layer using reactive skills has been published; it is left to the system designer to determine the skills that their system requires.

For our purposes here, that is, modelling, the reactive skills pattern is characterised by the following two concepts.

Skill Performs an operation using its input values, which can be from sensors or the outputs of other skills. A set of skills can be used together to accomplish a task that achieves the robot's current goal. An example of a skill is moving a hand towards a target [58].

Skills Manager Coordinates communication between skills and provides an interface for the executive layer to: activate the skills that are required to achieve the robot's current goal, receive notifications from monitored events, and set and get parameter values of skills.

Skills can be categorised as one of two types: C-Skill or D-Skill [42]. D-Skills interface physical devices, such as sensors and actuators, with the other skills of the control layer; their input values are actuation commands and their output values are sensor data. C-Skills compute a computational transform using the skill's inputs to determine the skill's outputs.

By the monitoring of skills outputs, the skills manager triggers events on desired conditions becoming true.

Table 2.3 summarises the primary architectural concepts of the reactive skills pattern and identifies related concepts using *italicised* text. The related concepts of the reactive skills concepts are defined in Table 2.4.

Table 2.3: The primary concepts of the reactive skills pattern.

Concept	**Description**
Skill	Made up of *inputs*, *outputs*, and a *computational transform*. Every skill has an *enable function* and a *disable function* that provide a means for the skill manager to enable and disable the skill. An *initialisation routine* runs at system startup that prepares the skill for first use. There are two categories of skill: C-Skill and D-Skill.
Skill manager	Interfaces the executive layer by providing the communications required to coordinate the skills and capability for event monitoring.

Table 2.4: The related concepts of reactive skills.

Element	**Description**
Initialisation routine	When the system starts the skill initialises itself, for example, set up communication ports.
Startup	When a skill is enabled it performs any required startup procedures each time it is enabled.
Reply	Response from the control layer to the executive layer.
Cleanup	When a skill is disabled cleanup actions are performed.
Parameter	A variable that allows a skill's behaviour to be adjusted for different contexts by the executive layer.
Input	Receives the value of a data type
Output	A resulting value that contributes to the robot's behaviour.
Computational transform	Once enabled the skill continually computes its outputs from its inputs.
Enable function	Allows a skill to be enabled in the appropriate context causing its computational transform to be continuously computed.
Disable function	Allows a skill to be disabled when it is no longer required causing the computation of its computational transform to cease.

2.2.2 Subsumption

The subsumption pattern described in [27] was one of the first approaches for the control of robots to promote the concept of reactivity using minimal state to reduce lengthy deliberation time. It has been used in the control of complete robotic systems, for example, a six legged walking robot [63], a three wheeled robot with an arm that navigates an office collecting cans [64], and a garbage collection robot that operates in a swarm [65]. The key concepts of the subsumption pattern have been used as part of the control layer of layered architectures to arbitrate between behaviours [31].

The Behaviour language [66] is a programming language that adopts the subsumption pattern to make the state machines that define behaviours easier to manage.

The layered structure of the subsumption pattern is designed to enable developers to use an incremental development process with each layer building upon the functionality of the last until the required system behaviour is achieved. These layers, however, do not relate to the layers of the three-layer architecture.

To facilitate the development of robotic systems according to the subsumption patterns, Brooks defines an informal numbering called competence which is based on his own experiences in developing mobile robots; it classifies the desired behaviours of the robot in its environment [27]. For example the lowest level 0: avoid objects, up to the highest level 7: reason about the behaviour of objects and modify plans accordingly. Each layer of the system is then designed to accomplish each level of competence creating a correspondece between layer and competence level.

For our purposes here, that is, modelling, the subsumption pattern is characterised by the following concepts.

Modules Comprised of a finite state machine that has the ability to hold data structures in variables. Modules also have inputs and outputs that enable communication among modules. All modules have a have a special input that resets a module to its initial state. Optionally modules can have an interface to the sensors and actuators of the robot enabling the finite state machine to utilise sensor values and set the actuators. We note here that the term module does not refer to a RoboChart module.

Inhibitors Comprised of two inputs and an output. When a message

is received on the first input it is relayed to the output. When a message is received on the second input, the relaying of messages is prevented for a period of time. Inhibitors do not modify the inputs provided to them, therefore, they relate to the communication of messages that could be provided by a middleware.

Suppressors Comprised of two inputs and an output. The messages received on each input are relayed to the output. However, messages received on the second input are given priority and there is a minimum time delay that following the reception of a message on the second input no messages from the first input are relayed. Suppressors do not modify the inputs provided to them, therefore, they relate to the communication of messages that could be provided by a middleware.

Layers Comprised of the modules, inhibitors, suppressors, and the connections among them that specify the source and destination of messages. The collection of the components that the layer contains specifies a subset of the required robot's behaviour.

Table 2.5: The primary concepts of subsumption.

Concept	Description
Layer	Formed of modules, inhibitors, suppressors, and connections.
Module	Contain a *finite state machine* and *instance variables*. To communicate modules can have *input lines* and *output lines*. All modules have a *reset* input. Modules that receive sensor data or set actuators use a *robot communication* interface that controls the robot's physical devices.
Inhibitor	Consist of two inputs and an output. The signal from the first input is relayed to the output until a signal is received on the second input, when for a predetermined amount of time no signal is relayed to the output.

Continued on next page

Table 2.5 – *Continued from previous page*

Concept	Description
Suppressor	Consist of two inputs and an output. The signal from the first input is relayed to the output until a signal is received on the second input, when for a predetermined amount of time only the second signal is relayed to the output.

Table 2.5 summarises the primary concepts of subsumption and identifies related concepts using *italicised* text. The related concepts are defined in Table 2.6.

RoboArch enables the description of a layer's pattern using the terminology of subsumption. We extend the control layer of the mail delivery system that was presented in Chapter 3 to include the structure of behaviours as subsumption. Because subsumption is being used for only the control layer of the mail delivery system, the higher competence levels are fulfilled by the approaches used by the executive and planning layers.

For the movement control aspect of the system, we define the competence levels as follows:

Level 0 – Stop when an object becomes too close.

Level 1 – Move down a hallway while avoiding obstacles.

Level 2 – Identify the current location of the delivery robot, move to a specified location, find office numbers and doors.

Table 2.6: The related concepts of subsumption.

Element	Description
Finite State Machine	Finite State Machine
Reset	of a module, special input that sets module to its initial state.
Output line	Sends a message
Input line	Receives a message and has a single element buffer. Latches that indicate the arrival of a new message.

Continued on next page

Table 2.6 – *Continued from previous page*

Element	Description
Robot Communication	Modules can, if required, communicate with the physical robot to get sensor values or set actuator values. The actual communication mechanism is unspecified.
Signal	The message as it travels along connections between modules.
Message	A data value that is communicated between modules.
Instance Variable	Variable of module that hold data structures.
Input Buffer	Holds the most recently arrived message from an input line.
Event	The occurrence of a message on an input line.
Delay	Time delay
Connections	Output lines from one module can be connected to one or many module input lines. (Brooks mentions these can be thought of as wires.)

The competence levels defined for the office delivery system have been specified by decomposing the task of 'moving to different locations while searching for an office number' into behaviours that the executive layer can use. A standard robotics ontology [67] was used to identify the concepts of each behaviour. The behaviours were placed into levels so that each provided part of the system's behaviour and the number of concepts a behaviour used was considered.

As a guide the number of concepts increases for higher levels of competence: level 0 requires only the concept of distance to accomplish its behaviour, level 1 the concepts of distance and velocity, and level 2 distance, velocity, location and office number.

2.2.3 Evaluation

The descriptions given for the reactive skills and subsumption patterns is a step towards their formalisation in order to provide support for modelling systems that use them.

The next section provides a review of DSLs for robotics and justifies the use of RoboChart for our work.

2.3 Domain Specific Languages

Modelling languages can either be general purpose or domain specific [68, 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) [69] 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 [68, p. 70], and the widely used domain-specific concepts can be represented in a common way [68, p. 70]. An example of a DSL is VHDL [70] for modelling electronic hardware.

There is an increasing number of DSLs for robotics [13], each offering features for different aspects of the robotics domain. We consider the features of interest provided by a selection of DSLs and present an evaluation of each. 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.

A documented metamodel provides a means to analyse and compare the syntactic structure of 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.

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. Seven DSLs have been selected for discussion: six based upon having a documented metamodel, providing support for code generation, and evidence of ongoing support, the seventh, WRIGHT, for being related work but for general software

architecture not specific to robotics.

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 2.3.1 to 2.3.5 present the selected DSLs and provide a review of the discussed important aspects. Finally section Section 2.3.8 evaluates the DSLs.

2.3.1 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 [71]. 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 [71].

The main structural elements of RoboChart are as follows:

Module is the top-level component representing the robot control software, 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 [72]. 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 of large systems, helping multiple developers work on different areas simultaneously.

CSP has a Unified Theory of Programming [73] making it possible for RoboChart to support reasoning about additional aspects of robotic systems, such as, 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 [74], 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 [75].

2.3.2 RobotML

RobotML is a notation that can be used to address the challenge of interoperability of robotics software to improve reusability [15]. 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 [76] 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 is 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:

[3]Plateforme pour la Robotique Organisant les Transferts Entre Utilisateurs et Scientifiques - https://anr.fr/Project-ANR-09-SEGI-0010

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

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).

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 behaviour, 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.

2.3.3 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 [16]. To enable software components to be integrated in a compositional manner, strictly defined communication patterns are enforced by the component model [77]. 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 that 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 behaviour 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) [78]. Dynamic reconfiguration at runtime is supported by dynamic state charts [79], an extension to state charts [80].

2.3.4 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 [14]. 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 [81] (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 is 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.

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

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 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.

2.3.5 V^{3}CMM

The three-view component metamodel (V^{3}CMM) addresses the need for improved processes and tools for the development of robotic systems with increasing levels of functionality, while reducing overall development time and cost [82]. 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 V3CMM 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.

V3CMM focuses on modelling platform-independent behaviour of a robotic system and does not provide views that require additional platform-specific details such as tasks [82]. 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 [82].

2.3.6 RsaML

The Robotic Software Architecture Modelling Language (RsaML)[83] is a modelling language for robotics software architecture to promote software engineering methods and techniques in robotics. RsaML is based on Monthe et al. review of robotics software architectures from the literature. The concepts of the architectures surveyed are consolidated and captured as a metamodel that defines RsaML. To date there are no notations available for RsaML, instead it makes use of Eclipse Modeling Framework (EMF) utilities for the creation of models from its metamodel. However, RsaML defines well-formedness conditions using the Object Constraint Language (OCL), which enables some validation of the created models.

The key structural concepts of RsaML are as follows:

System is the top-level concept that represents a robotic system that has both software and hardware elements.

Layers are the level in the hierarchy of the architecture and are made up of modules.

Modules represent a software component and they communicate through ports to other modules.

Functions implement the processing of a module as a set of actions.

A notable feature of RsaML is its comprehensiveness covering goals to time constraints and scheduling. This also means that there is direct representation for concepts of robotics architectural patterns, for example, layers and suppression and inhibition of communications from subsumption [27].

RsaML promotes separation of the robotic platform from the architecture by having the system composed of software and hardware. It helps the reuse of architectures with different applications. There is no support for modelling behaviour so only partial source code could be generated from an RsaML model.

2.3.7 WRIGHT

WRIGHT is a notation for describing software architecture in a precise and unambiguous way in order to address the informal approaches typically used to specify software architecture [84]. WRIGHT's semantics is defined using CSP. It formalises component and connector communication allowing software structure to be modelled. Because WRIGHT extends CSP, behaviour of components can be specified using CSP processes. This means that properties of the software given by a description can be verified. There is support for automatic verification of some properties, for example, consistency and deadlock freedom.

The main elements of WRIGHT are as follows:

Style defines shared properties of configurations.

Configurations are a collection of component instances combined via connectors.

Components describe a localised independent computation. They are composed of an interface and a computation that is the behaviour of the component.

Connectors represent interactions among collections of components.

The interface of a component defines the component's ports that it uses to communicate. The computation of a component specifies the components

behaviour and how the component makes use of its interface ports. A configuration defines a system's architecture and can reference styles to reuse their definitions.

A notable feature of WRIGHT is its ability to be able to describe architectural patterns using styles. Styles are made up of parametrised components and connectors plus a set of constraints that all members of the style must satisfy.

There is no support for time so it is not possible to verify properties that include time constraints.

2.3.8 Evaluation

Table 2.7 summarises the features of the DSLs discussed in the previous sections. All of the DSLs reviewed are robotics specific with the exception of WRIGHT which is for general purpose software development. All of the DSLs reviewed use a component-based approach that can be used 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, with the exception of WRIGHT.

Table 2.7: Feature comparison of robotics DSLs

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

Each robotics 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 behaviour of controllers with respect to services provided by a robotic platform. RobotML which is based on an ontology and RsaML models both capture a complete system. SmartMARS models capture software components structure and the communications patterns used between components. BCM models capture the software components structure. Finally, V^{3}CMM and WRIGHT models capture the software components structure including their behaviour; additionally, WRIGHT can describe the more general structure of components and connections as patterns. As a result of these approaches, distinct features are offered that we now consider.

Three of the DSLs do not model the behaviour of the software components. For both SmartMARS and BCM the source code provided by developers instead defines the components behaviours , whereas, for RsaML behaviour cannot be defined beyond the outline of functions. The other four DSLs do model the behaviour of components, all providing state machines for this purpose except WRIGHT which relies on behaviour defined as CSP processes. DSLs that do model component behaviour can generate more complete source code.

RoboChart is prominent in that it is the only robotics 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.

The WRIGHT DSL does have some similar characteristics to RoboChart for automatic verification and being formally specified. However, WRIGHT does not support the specification of time or probabilistic properties which are important for modelling robotic systems.

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.

2.4 Modelling Robotic Systems Using RoboChart

This section introduces RoboChart's graphical notation through a simple example of a robotic lawnmower. Section 2.4.1 explains the structure and elements of a RoboChart model for the robot lawnmower, and Section 2.4.2 demonstrates the verification of some properties of the robot lawnmower using the RoboChart model.

2.4.1 Modelling

The diverse and often specialised designs 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 hardware and associated embedded software 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, via during actions in 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: 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 can communicate values, so they can have an associated type.

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.

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 including: natural numbers, strings, integers, booleans, and real numbers. Custom primitive types can be created. Other types supported are enumerations, product types, and datatypes that are made up of fields of types. Finally sets and sequences are also supported.

Related events, operations, and variables can be grouped into an interface. RoboChart platforms, controllers and machines can declare 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. Similarly required interfaces can contain only variables and operations. Defined interfaces indicate the events and variables that an element uses to perform its function.

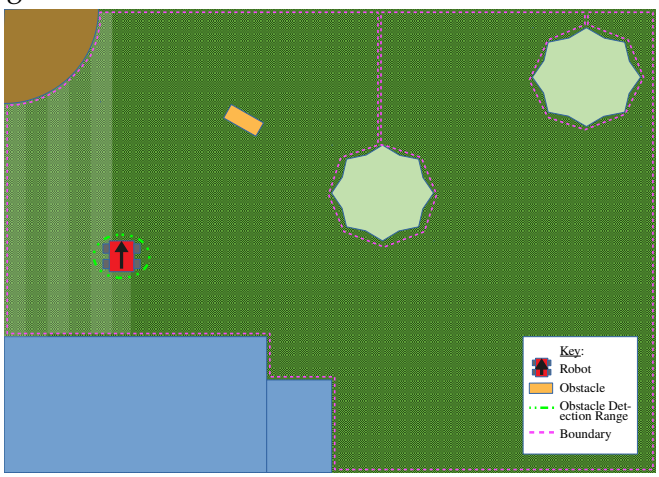
The support RoboChart provides for interfaces makes the services a controller requires from the platform in order to operate clearly visible. Additionally, the support provided by defined interfaces to group together related events and variables further contributes to the structure of RoboChart models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

The lawn-mower 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. To enable autonomous operation in its environment the robot features two types of sensor: a Very Low Frequency (VLF) sensor for detecting the boundary wire, and ultrasonic sensors for obstacle detection.

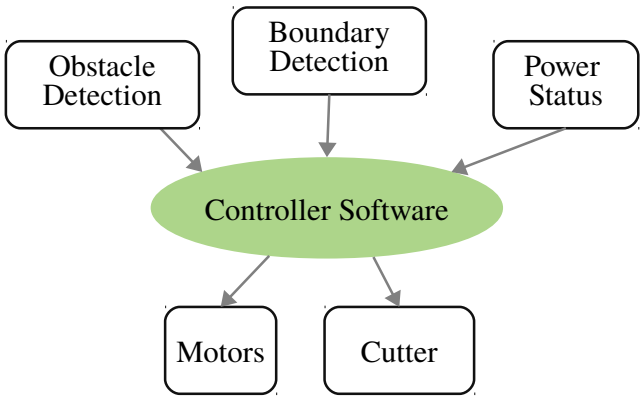
The software controllable inputs and outputs of the lawnmower robot

Figure 2.6: The environment of the lawnmower robot.



are shown in Figure 2.7.

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



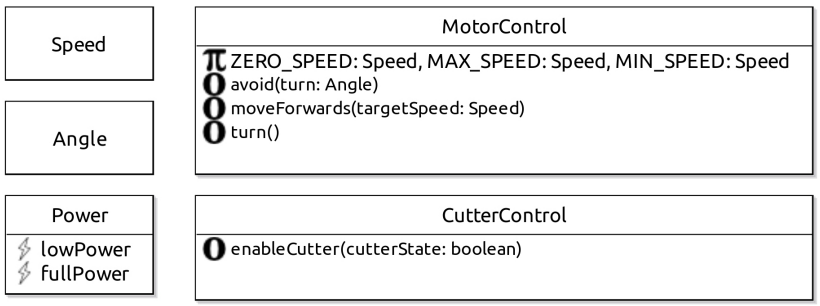
Inputs to the robot's controller software originate from two places: either from the environment via the robot's sensors, or internal state information provided by the robot's hardware. All of the inputs to the controller software can be modelled using events.

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 are defined together in an interface.

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. Events are depicted within interfaces as the event name prefixed with a lightning symbol (ϟ). Operations and constants are similarly shown, but with operations prefixed with an *O* symbol (**O**) and constants prefixed with a pi symbol (**π**).

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



To minimise the cost of the lawnmower it has a single embedded microcontroller. Therefore, it can be modelled as a single controller which we 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.

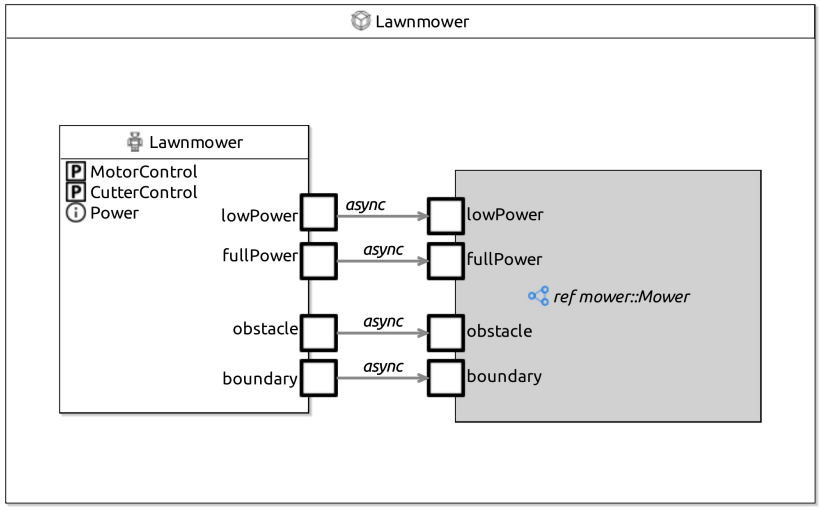
Figure 2.9 shows the RoboChart module for the lawnmower system; it is made up of the Lawnmower robotic platform and the single controller Mower. Events of the controllers and platforms are depicted as square boxes on their boundary labelled with the event name.

The flow of events is depicted by the connections between the robotic platform, controllers, and state machines. This is useful when modelling more complex systems, particularly where there are multiple controllers and state machines. RoboChart supports both synchronous and asynchronous connections that follow the commonly used communica-

tion approaches of the robotics domain. Asynchronous connections are labelled as 'async' whereas synchronous connections are unlabelled.

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.

Figure 2.9: The module of the lawnmower robot.



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. 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 [85, Sec. 2.1.6]. States can also be composite and so contain a state machine that is executed when that state is entered.

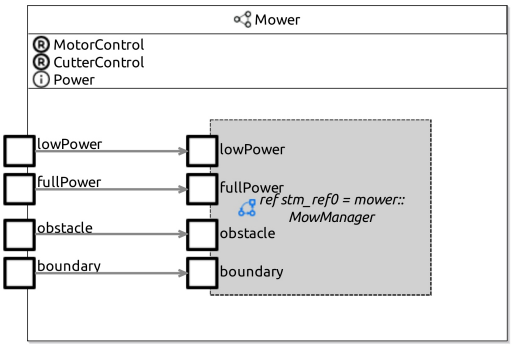
Transitions connect states and junctions and they can have any combin-

ation of triggers, guards, or action statements. Triggers cause a transition to be taken on the occurrence of a particular event. Optional deadlines can be given to triggers supporting the specification of time properties of the system. The guards are boolean expressions that only allow a transition to be taken when it evaluates to true. The action statement enables any required actions to be executed on the occurrence of a transition.

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.

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 2.10: The controller of the lawnmower system.



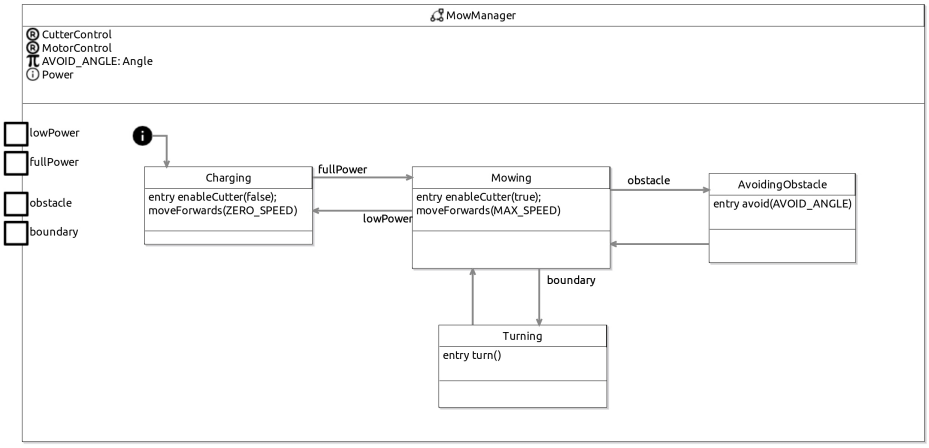
The MowManager machine requires all of the interfaces and must handle all of the events of the system, because, a single controller and a single state machine are being used to model the system. The behaviour of the lawnmower robot can be modelled using four states: Charging, Mowing, AvoidingObstacle, 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 named

AVOID_ANGLE 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 with an i. 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 transitions to the Mowing state when a fullPower event occurs. Similarly, when in the Mowing state and a lowPower event is received, the machine transitions back into the Charging state. The complete behaviour of the MowManager machine can be summarised as follows:

- The robot is 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 is entered, the cutter is enabled and the robot begins moving forward.
- If the robot is in the Mowing state and a boundary event is triggered, the robot enters the Turning state. When the turn has completed the Mowing state is entered.
- If the robot is in the Mowing state and an obstacle event is triggered, the robot enters the AvoidingObstacle state. Once the obstacle has

been avoided the Mowing state is entered.

- If the charge runs low when in the Mowing state then the Charging state is entered, the cutters are switched off and the robot stops.

The next section discusses the proof of some of the properties that can be verified using the RoboChart model of the lawnmower.

2.4.2 Verification

The RoboTool IDE automatically generates a CSP model from the RoboChart model enabling verification of the software's properties. Basic properties can be automatically verified and other properties of interest can be included in the verification following their specification in CSP. For the lawn-mowing software 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.

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

Divergence Freedom A module, controller or state machine does not enter an infinite sequence of internal actions.

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

Termination 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 of 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 that 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 [85, Sec. 6].

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 the model checker FDR [86].

In addition to the basic properties, a property of interest can be verified by specifying it as a CSP process, and then using refinement checking to verify the CSP process against the CSP model. For example, the lawnmower should avoid an obstacle when it is detected; this can be expressed by the following process named Property Avoid Obstacle (PAO).

$$
\begin{aligned}
PAO = \Box\ & ev : waitevents \bullet (ev \longrightarrow PAO) \\
& \Box \\
& Mower :: MowManager :: obstacle.in \longrightarrow \\
& \quad Mower :: MowManager :: avoidCall \longrightarrow PAO
\end{aligned}
$$

PAO is 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 property of interest are in the set *waitevents*. When any of the *waitevents* occur the process recurses and waits for another event. On a *mower_MowManager_obstacle.in* CSP event occurring, representing the RoboChart event obstacle, the avoid operation should be called, indicated by the CSP events *avoidCall*. Afterwards, the process goes back to accepting all events from the MowManager machine.

The assertion can be now be specified as a refinement check against the MowManager machine that can be expressed using the assertion language as:

assertion MM_9: mower::MowManager **refines**
PAO **in the failures model**

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

2.5 Final Considerations

This chapter has surveyed some robotics specific architectures and DSLs. It has found that RoboChart stands out for its formally defined semantics

and support for automated and semi-automated verification. Coupled with support for simulation and modelling timed and probabilistic behaviours of robotic systems, RoboChart provides a basis for their improved verification utilising formal techniques.

From our study of architectural patterns used within the robotics community, the use of the layered pattern is evident. The layers 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 is 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.

The next chapter outlines an approach for characterising layered robotics architectural patterns as a metamodel and well-formedness conditions that define RoboArch. It examines a selection of patterns that can be used by each of the layers. Finally, it introduces rules for translating from RoboArch to RoboChart.

3 Architectural Patterns for Robotics

The preceding chapter has identified the common use of layered architectures in robotics and analyses the structure and function of each layer. In this chapter we introduce RoboArch, a notation for describing architectures, and examine how architectural patterns can be characterised as a metamodel and well-formedness conditions. For the layered pattern we define its key concepts and present a metamodel and transformation rules from RoboArch to RoboChart.

The presented metamodel and transformation rules give a precise characterisation of the layered architecture and identifies several patterns that we study. These have, so far, been given just an informal account in the literature, with, in some cases, several variations considered by different authors. Our metamodel and rules form the basis for our tool to automatically generate sketches of RoboChart models from an architectural description of layers.

Section 3.1 provides an overview of the RoboArch notation and how it can be used to model a layered architectural pattern. It introduces the robotic mail delivery system used as a running example. Section 3.2 presents the transformation rules and the resulting RoboChart model structure. Section 3.3 provides a detailed analysis of the approaches for the control layer and how they can be modelled. We conclude in Section 3.4, with some final considerations.

3.1 RoboArch: Layers

Layers provide structural organisation to software. In robotics architectures they have been used to help manage the large volumes of data produced by sensors and actuators while being able to reason about and plan for dynamic environments. First, we show how a layered pattern can be described using RoboArch without specifying the patterns used for each of the layers. Later, in Chapter 4, an example of a pattern for the control layer is given demonstrating how a pattern for a layer can be characterised.

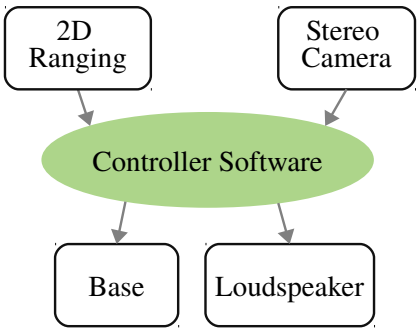
3.1.1 Overview

RoboArch enables the description of the architectural structure and patterns used by a robotic system to be captured. We will use an office delivery robot example from the handbook of robotics [10, Sec. 12.3] to introduce RoboArch's notation.

The goal of the robot is to deliver items within a typical office building, transporting items of post from a central mailroom to each of the offices within the building. To achieve its goal the robot must safely navigate along the corridors of the building while avoiding any obstacles such as people and furniture. The robot is given a map of the building so that it does not have to construct a map by exploring the environment.

For our example we assume a robot that comprises of the following sensors and actuators: a two-wheel differential driven base, a loudspeaker, a forward-facing 2D ranging sensor, and a stereoscopic imaging sensor. Figure 3.1 shows the software controllable inputs and outputs of the robot. The inputs originating from the environment via the robot's sensors can be modelled as events. The outputs from the software API of the base and loudspeaker can be modelled as operations. However, before defining the robotic platform, the system and its types for these events and operations must first be declared.

Figure 3.1: The inputs and outputs of the mail delivery robot's controller software.



The `system` clause of RoboArch gives a name to a model that describes the architectural elements of a system, namely layers and a robotic platform. For our example, Listing 3.1 declares the `system` with the name `MailDelivery`.

RoboArch adopts the type system of RoboChart, which itself is based on that of Z [87]. RoboChart's libraries provide support for core types including: real, nat, boolean, and string and types from the Z mathematical toolkit like sequences. Listing 3.2 adds the type declarations to the system from lines 3 to 33.

Listing 3.1: The system declaration

```
1 system MailDelivery
```

Listing 3.2: A system and its type declarations.

```
1  system MailDelivery
2
3  type Velocity
4  type Coordinate
5
6  datatype Office {
7      number:nat
8      location:Coordinate
9  }
10
11 datatype Velocities {
12     linear:Velocity
13     angular:Velocity
14 }
15
16 type Time
17 type ColourPoint
18
19 datatype PointImage {
20     header: Time
21     width: nat
22     height: nat
23     points: Seq( ColourPoint )
24 }
25
26 datatype Scan {
27     header: Time
28     startAngle: Angle
29     stopAngle: Angle
30     timeBetweenMeasurements: Time
31     Ranges: Seq( Range )
32     Intensities: Seq( Intensity )
33 }
```

Examples of abstract type definitions, characterised just by its name, include `Velocity` on line 3 and `Coordinate` on line 4. One example of a record type is `Office` on line 6 that has two fields. The first field, `number`, of type `nat`, and the second field, `location`, of type `Coordinate`.

The robotic platform determines the possible interactions with the environment, therefore, the capabilities of the robotic platform are often an important consideration made by developers early on in a system's design. RoboArch supports modelling of the platform by adopting RoboChart's robotic platform concept that abstracts the services the physical robot's software provides as events, operations, and variables.

The robotic platform for our example is given in Listing 3.3. The `robotic platform` clause on line 53 declares a platform with name `DeliveryRobot` that references interfaces named `Base`, `Audio`, `PointCloud`, and `EnvColourPoints` which are declared in Listing 3.4. The interfaces in our example logically group platform services related by sub-module capability of the robot's hardware.

Listing 3.3: Platform declaration

```
1  system MailDelivery

...  ... /* Type Declarations */

...  ... /* Interface Declarations */

53 robotic platform DeliveryRobot {
54     provides Base
55     provides Audio
56     uses PointCloud
57     uses ColourPointCloud
58 }
```

The hardware module relating to `Base` is the core that the robot is built around and it is the means for moving around the environment. The module relating to `Audio` can produce sound for announcing its arrival notifying recipients of their mail. The module relating to `PointCloud` senses the environment providing information on the position of obstacles as series of points. The module relating to `EnvColourPoints` senses the environment providing images that are used for locating and navigating the robot.

The interfaces are referenced using `provides` and `uses` clauses. The `provides` clauses on lines 54 and 55 reference provided interfaces `Base` and `Audio` that contain operations and variables. The `uses` clauses on lines 56 and 57 reference defined interfaces `PointCloud` and `ColourPointCloud` that contain events.

The `Base` interface declared on line 35 of Listing 3.4 models the interactions that control movement. There is one operation `Move` and two constants. The `Move` operation has one argument `request` of type `Velocity` and when called, results in the linear and angular velocities of the robot being adjusted to match the passed in parameters. The two constants `ZERO_LINEAR_VELOCITY` and `ZERO_ANGULAR_VELOCITY` define the zero named values for readability and maintainability.

Listing 3.4: Interface declarations for the platform.

```
1  system MailDelivery

...  ... /* Type Declarations */

35 interface Base {
36     Move (  request : Velocities )
37     const ZERO_LINEAR_VELOCITY : Velocity
38     const ZERO_ANGULAR_VELOCITY : Velocity
39 }
40
41 interface Audio {
42     Announce (  request : String )
43 }
44
45 interface PointCloud {
46     event envPoints : Scan
47 }
48
49 interface ColourPointCloud {
50     event envColourPoints : PointImage
51 }
```

The `Audio` interface declared on line 41 models interactions that provide feedback to users; it has one operation `Announce`. The `Announce` operation has one argument of type `string` and when called, the passed in string is converted into audio and played via a loudspeaker.

The `PointCloud` interface declared on line 45 models obstacle sensor data, for instance, from a type of laser scanning device. The interface contains one event `envPoints` with a value of type `Scan` that is a record containing a sequence of ranges to obstacles from a scan. For the delivery robot the value of an `envPoints` event represents the current obstacles surrounding the robot; it is used to safely stop if the planned path is obstructed.

The `ColourPointCloud` interface declared on line 49 models vision sensor data, for instance, from a colour camera. The interface contains

one event `envColourPoints` with a value of type `PointImage` that is a record containing a sequence of colour points. For the delivery robot the value of an `envColourPoints` event represents a colour image of the environment; it is used for object recognition and verifying progress of actions.

Following the platform in a RoboArch model the next important concept is the layers. For our example here, we specify a typical three-layer architecture with planning, executive, and control layers. Layers model the architectural structure of a system's software. Every layer must have a unique descriptive name that identifies it within the system. Optionally layers can have a type, a pattern, inputs and outputs.

The three specific layer types that can be specified using RoboArch are `ControlLayer`, `ExecutiveLayer`, and `PlanningLayer`. Providing a type enables the greatest level of validation of the patterns used by a system's layers. On the other hand, the possibility of not providing a type means a more customised architectural structure, rather than the standard three-layer architecture, can be used.

The input and output events that layers define make a layer's services accessible to others without specifying the behaviour of the layer. Instead, the behaviour is left to be defined by a pattern and by customising the generated RoboChart model.

Later in Section 3.3 we explore specific patterns that can be used for each layer. The planning layer for our example is given in Listing 3.5. The `layer` clause on line 60 declares a layer with name `Pln` and type `PlanningLayer`.

Listing 3.5: A custom planning layer.

```
1  system MailDelivery

... /* Type Declarations */

... /* Interface Declarations */

... /* Platform Declaration */

60 layer Pln: PlanningLayer {
61
62     inputs= deliveryComplete,
63             pickupFailed;
64
65     outputs= deliverMail: Office ;
66 } ;
```

There is one output `deliverMail` with value of type `Office` that requests the number of the office to which mail is currently being delivered. There are two inputs `deliveryComplete` and `pickupFailed` that have no associated value type; their occurrence indicates an outcome of the currently requested delivery. The occurrence of a `deliveryComplete` input event indicates that delivery to the specified office has been successful. The occurrence of a `pickupFailed` input event indicates that pickup of a delivery has failed. The `Pln` layer `inputs` and `outputs` are used to communicate with another layer; in our example, an executive layer type.

The executive layer for our example is given in Listing 3.6. The `layer` clause on line 69 declares a layer with name `Exe` and type `ExecutiveLayer`. The inputs and outputs communicate either with the planning layer or control layer. The connections are defined later in the RoboArch model.

Listing 3.6: A custom executive layer.

```
1  system MailDelivery

... ... /* Type Declarations */

... ... /* Interface Declarations */

... ... /* Platform Declaration */

... ... /* Planning Layer Declaration */

69 layer Exe: ExecutiveLayer {
70
71     inputs= deliverMail:Office,
72             destinationReached,
73             location:Coordinate,
74             doorFound,
75             mailPickedUp;
76
77     outputs= deliveryComplete,
78              pickupFailed,
79              destination:Coordinate,
80              doorToFind:nat,
81              getLocation,
82              speak:string;
83 } ;
```

The Exe layer declares six outputs. One of which is destination with value of type Coordinate that requests the control layer to move the delivery robot to the location given by the coordinate while avoiding obstacles. The doorToFind output with associated nat type requests the control layer to search for the given door number within the robot's visual field. The getLocation output with no value type requests the control layer to return the current location the robot is determined to be. The speak output requests the control layer to announce the given text string using the robot's loudspeaker.

The remaining outputs deliveryComplete and pickupFailed inform the Pln layer of the status of the delivery. A deliveryComplete event is triggered once the mail has been picked up by the recipient and a pickupFailed event is triggered when the mail is not collected.

The Exe layer declares five inputs. One of which is deliverMail with value of type Office that informs the layer of the given office to deliver mail to. The other inputs provide feedback on the delivery it is executing. The occurrence of a destinationReached input event indicates that the robot has reached the requested destination. The occurrence of a location with value of type Coordinate gives the current location of the robot. The occurrence of a doorFound input event indicates that the requested door is visible. The occurrence of a mailPickedUp input event indicates the mail item has been taken. The executive layer uses services of the control layer.

A layer that is of a control type shares all of the aforementioned features of layers. In addition, control layers directly communicate with a robotic platform, therefore, they reference platform interfaces. The control layer for our example is given in Listing 3.7. The layer clause on line 86 declares a layer with name Ctl and type ControlLayer. The inputs and outputs between lines 93 to 101 communicate with the executive layer. The requires and uses clauses between lines 88 to 91 reference interfaces that are used by the control layer.

The requires clauses on lines 88 and 89 reference required interfaces Base and Audio that contain operations and variables. The uses clauses on lines 90 and 91 reference defined interfaces PointCloud and ColourPointCloud that contain events.

The Ctl layer declares four inputs. The first is destination with value of type Coordinate that informs the layer of the given coordinate to navigate to. The doorToFind with value of type nat that informs the layer of the office door number to look for in the field of view. The occurrence of a getLocation input event is a request for the layer to

Listing 3.7: A custom control layer.

```
1   system MailDelivery

... /* Type Declarations */

... /* Interface Declarations */

... /* Platform Declaration */

... /* Planning Layer Declaration */

... /* Executive Layer Declaration */

86  layer Ctl: ControlLayer {
87
88      requires Base
89      requires Audio
90      uses PointCloud
91      uses ColourPointCloud
92
93      inputs= destination:Coordinate,
94              doorToFind:nat,
95              getLocation,
96              speak: string;
97
98      outputs= location:Coordinate,
99               destinationReached,
100              doorFound,
101              mailPickedUp;
102 } ;
```

return the current location by triggering an output event with its value set to the coordinate of the current location. The `speak` with value of type `string` takes the input that is communicated to the platform to be announced via loudspeaker.

The `Ctl` layer declares four outputs. The first is `location` with value of type `Coordinate` that provides the current coordinate of the robot within the building in response to a `getLocation` input. The `destinationReached` occurs when the robot has reached its requested destination. The `doorFound` occurs when the requested door number is within the visual field of the robot. The `mailPickedUp` occurs when the mail item being delivered has been collected by the recipient.

The inputs `destination`, `doorToFind`, `getLocation`, and `speak` from the executive layer use services of the platform to produce designed

behaviours. For example, `destination` input requests to move the robot to the given location with the implemented behaviour of the control layer invoking the `Move()` operation of the platform declared in the `Base` interface.

The outputs `location`, `destinationReached`, `doorFound`, and `mailPickedUp` provide feedback to the executive layer on the status of its requests to the control layer.

Listing 3.8 declares the connections for our mail delivery example.

Listing 3.8: Connection declarations

```
1   system MailDelivery

... ... /* Type Declarations */

... ... /* Interface Declarations */

... ... /* Platform Declaration */

... ... /* Planning Layer Declaration */

... ... /* Executive Layer Declaration */

... ... /* Control Layer Declaration */

105 connections=
106     Pln on deliverMail to Exe on deliverMail,
107     Exe on deliveryComplete to Pln on deliveryComplete,
108     Exe on pickupFailed to Pln on pickupFailed,
109
110     Exe on destination to Ctl on destination,
111     Exe on doorToFind to Ctl on doorToFind,
112     Exe on getLocation to Ctl on getLocation,
113     Exe on speak to Ctl on speak,
114
115     Ctl on location to Exe on location,
116     Ctl on destinationReached to Exe on
        destinationReached,
117     Ctl on doorFound to Exe on doorFound,
118     Ctl on mailPickedUp to Exe on mailPickedUp,
119
120     DeliveryRobot on envPoints to Ctl on envPoints,
121     DeliveryRobot on envColourPoints Ctl on
        envColourPoints
122  ;
```

The connections between the layers and robotic platform connection nodes are defined under a system's `connections` clause. Each connection is unidirectional and connects an input or output `on` a node (layer or platform) `to` another node's output or input. Lines 105 to 121 of

Line 106 declares a connection from the `Pln` layer's `deliverMail` output event to the `Exe` layer's `deliverMail` input event.

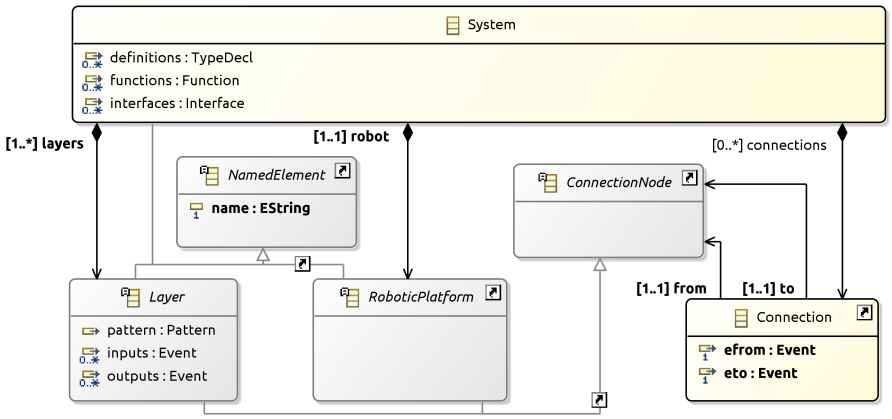
We have now introduced all parts of the textual notation for describing the layered structure of the mail delivery example. Next we describe the RoboArch metamodel.

3.1.2 Metamodel

The structure of RoboArch models is determined by a metamodel that specifies the available concepts and how they relate to one another. In this section, we focus on the top-level structure of RoboArch models: the systems, layers, robotic platforms, and connections.

A RoboArch model is organised in a system. Figure 3.2 defines a RoboArch System that can include declarations of layers, platforms, connections, definitions, functions, and interfaces.

Figure 3.2: System metamodel



The NamedElement parent means that Systems have a name attribute for uniquely identifying each System instance. Systems contain at least one layer and must have a robotic platform.

The robotic platform concept is reused from RoboChart. RoboticPlatforms have a name attribute for identification and they can declare events

and variables as well as reference interfaces that have operations or events for modelling the services of physical robots.

A Layer can have inputs and outputs that are, via connections, the means of interacting with a layer's services. They can optionally have a pattern that defines their behaviour using the pattern-specific terminology and concepts. A detailed characterisation of patterns and their metamodel are presented in the next chapter. The inputs and outputs are RoboChart events that can have a type and if present defines the values that can be communicated.

The structure of a system comprises of a number of connection nodes and connections. The ConnectionNode class characterises elements that can be connected through their events; they are Layers and RoboticPlatforms.

Connections are between two events, a source efrom and a target eto that belong to their respective to and from ConnectionNodes. The Connections of a System establish a relationship among its Layers and RoboticPlatform. The metamodel does not restrict the events and types that can be connected so not all of the possible connections are meaningful, for example, if the connected events are of different types. Well-formedness conditions, presented later, are used to restrict models to only those that are meaningful.

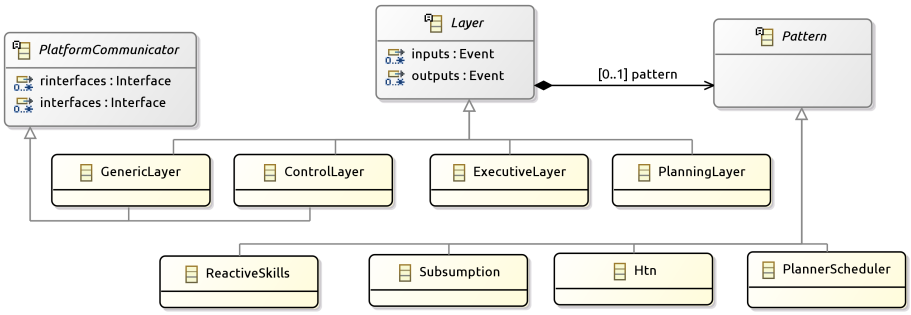
Systems declare type definitions, functions, and interfaces that are used for modelling layers, robotic platforms, and patterns. The collections of TypeDecl attribute named definitions, Function attribute named functions, and Interface attribute named interfaces are as in RoboChart.

Subclasses of TypeDecl define: an abstract type without definition that only has a name, an enumeration type that contains a collection of literals which associate a name with a type, and record type called data type similar to UML and Z schema types. RoboArch supports the use of the RoboChart's library that provides a set of core types: for example, real, nat, int, boolean, char, and string. The Z mathematical toolkit from RoboChart is also supported, it provides data types for sequences, functions, and relations.

Layer is an abstract class that is further specialised; Figure 3.3 shows the metamodel of layers. GenericLayer is the most general kind of layer and represents a layer without a type, offering flexibility to model systems with minimal restrictions. GenericLayers are able to communicate with the RoboticPlatform; therefore, they are a subclass of PlatformCommunicator.

The required and defined interface attributes of PlatformCommunicator are rinterfaces and interfaces, respectively. They enable the platform services a layer requires to be specified.

Figure 3.3: Layers metamodel



The three other kinds of layers, ControlLayer, ExecutiveLayer, and PlanningLayer, allow models to be validated according to the layered pattern. The ControlLayer Layer can communicate with the RoboticPlatform therfore is a subclass of PlatformCommunicator.

Layers optionally can have a pattern that define its own set of concepts for modelling structure and behaviour. Examples of patterns of include: reactive skills, subsumption, hierarchical task network, and planner scheduler, each represented by a subclass of Pattern. In Chapter 4 we characterise and present the metamodel, well-formedness conditions, and rules for the reactive-skills pattern.

Not all models that can be created obeying the metamodel can be given a semantics translated to RoboChart. For example, it is not possible to give reasonable meaning to layers with connections between events of the same direction, input to input or output to output. Therefore, we use well-formedness conditions to ensure only valid models can be created.

3.1.3 Well-formedness Conditions

Well-formedness conditions restrict the models allowed by the RoboArch metamodel to only those that are meaningful. Here we focus on conditions relating to the top-level system.

S1 A System must have one or more connections that relate a single Layer to a RoboticPlatform or there must be a Layer that has at least one or more rinterfaces.

This ensures that it is possible for there to be some interaction with the platform which describes the services provided by the hardware. If there

was no interaction with the platform the robot would do nothing.

S2 For Systems with more than one Layer, each Layer must have at least one input or output event.

Because a layer must provide a service that forms part of the overall system behaviour and itself may rely on services of another layer they cannot exist in isolation. Therefore, a layer must provide a means for external interaction as input or output events.

S3 For Systems that have a ControlLayer, ExecutiveLayer, and a PlanningLayer, Connections must associate the ExecutiveLayer layer with both the ControlLayer and the PlanningLayer.

The three-layer robotics architecture defines control, executive and planning layers and establishes their responsibilities and the order in which the layers are allowed interact. If a system has all three layers, the executive layer must be the middle intermediate layer between the planning and control layers with no direct communication between them because the executive layer decomposes requests it receives from the planning layer.

S4 Connections efrom and eto event types must match.

This ensures that the communications carried out via the connections are well typed.

S5 Connections must only connect Layer inputs to outputs or vice versa.

This ensures that data is sent from an output and received by an input.

S6 Connections must only associate a Layer with at most two other Layers.

In the layered architecture a layer can use the services of one other layer and the same layer can provide its services to at most one other. Therefore, a layer can have up to two immediate neighbours that they can communicate with, shielding them from direct communication with any other layer.

S7 Connections involving the ControlLayer must only associate it with at most one other Layer.

Because the control layer uses the services of the platform, it means that it can only accept requests from one other layer for similar reasons to those described above for S6.

S8 The connections of a System must only associate events defined by interfaces of GenericLayers and ControlLayers with events of the RoboticPlatform.

The interfaces of RoboArch layers are only used to define communication between layers and a robotic platform's, therefore, their connections must only be made between that layer and the a robotic platform.

The description captured by the notation of RoboArch forms a model that allows the application of model-driven techniques to reason about and analyse the system being developed. In the next section we introduce the transformation rules that are used to generate RoboChart models.

3.2 Rules

From the RoboArch description of a model via model transformation a RoboChart model can be generated. RoboChart's formally defined semantics underpins RoboArch because of the rules that link them, and allow properties of the systems described using RoboArch to be verified.

Semi-formal transformation rules map concepts of RoboArch to RoboChart and consist of five parts presented in tabular form. A name ⓐ uniquely identifies the rule and broadly follows the convention of [*RoboArchConcept*]to[*RoboChartConcept*]. The parameter ⓑ are the RoboArch concepts that are the inputs to be transformed. The result ⓒ are the RoboChart concepts that will be created upon the rule being applied. The optional Precondition ⓓ that must be satisfied in order for the rule to be applicable. The definition ⓔ is the main body of the rule that uses a Z-like notation to define the RoboChart output result in terms of the RoboArch input parameter.

The top-level RoboArch element is the system that contains all other elements that model the architecture of the robotic system being developed. Therefore, the transformation rule that calls all others takes the System as its parameter and is named SystemToRCModule.

The SystemToRCModule Rule 3.2.1 takes a RoboArch System and produces a RoboChart RCModule along with a set of type declarations and a set of interfaces. This rule applies further rules to define a valid RoboChart model that can be used as a basis for verification.

RULE 3.2.1:

ⓐ	**Name**	SystemToRCModule
ⓑ	**Parameter** *name:type*	amsys:System
ⓒ	**Result** *name:type*	rcdefs: Set(TypeDecl), rcfuns: Set(Function) , rcifcs: Set(Interface), rcmod: RCModule
ⓔ	**Definition**	rcdefs = amsys.definitions ∪ ⋃ {lyr: amsys.layers • LayerToTypes(lyr)} rcfuns = amsys.functions ∪ ⋃ {lyr: amsys.layers • LayerToFunctions(lyr)} rcifcs = amsys.interfaces ∪ ⋃ {lyr: amsys.layers • LayerToInterface(lyr)} rcmod = ⦇ name = amsys.name, nodes = LayersToControllers(amsys.layers) ∪ roboticPlatform, connections = amsys.connections ⦈$_{RCModule}$ --- ***where*** cLayer = {lyr: amsys.layers ∣ lyr ∈ ControlLayer}

Continued on next page

RULE 3.2.1 – continued from previous page

	if ((# amsys.robot.operations > 0) ∨ (# amsys.robot.interfaces > 0) ∨ (# amsys.robot.pinterfaces > 0) ∨ (# amsys.robot.variableList > 0)) **then** roboticPlatform = amsys.robot **else** roboticPlatform = ControlLayerToRoboticPlatform(cLayer)

The resulting RoboChart type definitions **rcdefs** are the union of RoboArch system type definitions **amsys.definitions** and the generalised union of the result of the application of Rule B.1.1 LayerToTypes for each layer of the RoboArch layers **amsys.layers**. The RoboArch system definitions are adopted from RoboChart so directly map across to the RoboArch model with no modification therefore do not require additional rules. The results from applying the LayerToTypes rule add additional types that arise from a layer's pattern, the details of which will be given in Chapter 4 that looks at patterns in detail.

The specification of the RoboChart function definitions **rcfuns** follows the same structure as for **rcdefs**. Directly declared **amsys.functions** are included with no modification and a generalised union that calls a rule for each of the layers. However, this time it is LayerToFunctions Rule B.1.2 that adds additional functions that are required by a layer's pattern.

The resulting specification of the RoboChart interface definitions **rcifs** follows the same structure as **rcdefs**. Directly declared **amsys.interfaces** are included with no modification and a generalised union that calls the LayerToInterface Rule B.1.3 for each of the layers adds additional interfaces required by a layer's pattern.

The resulting RoboChart module definition **rcmod** is specified by a binding (object or record definition) using the notation $\langle\!|$ S $|\!\rangle_T$ where T indicates the type of the element being defined. This binding notation is used throughout the rules to describe elements of the generated model. Each attribute of the type is defined by statements S. For the module the attributes defined are name, nodes and connections. The name of the module is simply the RoboArch system name **amsys.name**. The nodes of the module are controllers that result from calling LayersToControllers Rule 3.2.2 taking the system's layers as a parameter and a **roboticPlatform** as defined by the where clause. The connections of the module are the

connections defined directly by **amsys.connections**.

RULE 3.2.2:

ⓐ	**Name**	LayersToControllers
ⓑ	**Parameter** *name:type*	amlyrs: Set(Layer)
ⓒ	**Result** *name:type*	rcctl: Set(Controller)
ⓔ	**Definition**	rcctl = { l:amlyrs • ⦇ name= l.name, events= l.inputs ∪ l.outputs, machines= PatternToMachinesAndConnections(l.pattern).1, connections= PatternToMachinesAndConnections(l.pattern).2, **if** (l.pattern ∈ GenericLayer ∨ l.pattern ∈ ControlLayer) **then** interfaces= l.interfaces, rinterfaces= l.rinterfaces **else** interfaces= ∅, rinterfaces= ∅ ⦈$_{Controller}$ }

The where clause of the SystemToRCModule rule defines the **roboticPlatform** and **cLayer**. If the robot attribute of a System has any operations, interfaces, or variables, the **roboticPlatform** is **amsys.robot**; this means that the platform is directly mapped to the RoboChart model because RoboArch adopts the platform concept of RoboChart and requires no transformation. If the robot operations, interfaces, or variables are all empty, the **roboticPlatform** is the result of calling the Rule B.1.4 ControlLayerToRoboticPlatform that takes the control layer **cLayer**. The **cLayer** is defined as a a set comprehension that filters the control layers from

the system's layers **amsys.layers**. This definition of the **roboticPlatform** allows the platform to be generated or defined by the developer offering multiple ways to approach modelling.

We will use the mail delivery example from Section 3.1 and apply the rules to demonstrate the resulting RoboChart model that is generated.

The structure of the generated RoboChart model so far from examining the SystemToRCModule rule will be a single module with the system name `MailDelivery`. See Figure 3.4 for the RoboChart representation. One of the module's nodes will be the robotic platform the others are defined by the by the LayersToControllers rule.

The LayersToControllers rule takes a set of layers and produces a set of controllers. The resulting set of controllers **rcctl** is given by a set comprehension that binds each RoboArch layer **l** given as input to the attributes of the type Controller: name, events, machines, connections, interfaces, and rinterfaces.

The name of each controller is the name of a layer **l.name**. The events of each controller are defined by union of the layers inputs and outputs. The machines and connections for each controller are the result of calling Rule 4.4.1 PatternToMachinesAndConnections. The respective machines or connections of the tuple are selected by the dot operator. The details of the PatternToMachinesAndConnections rule will be given in Chapter 4 where we analyse a pattern that is used in the control layer of robotic systems.

Only control layers and generic layers in RoboArch can communicate with the robotic platform and have interfaces. Interfaces they define are directly mapped across to the layer's corresponding controller. For the other executive and planning layers the interfaces of the resulting controller are defined as the empty set as a consequence of well-formedness condition S8.

Figure 3.4 shows the three controllers and a robotic platform for `MailDelivery`. There is a controller for each of the layers with matching names from the model `Pln`, `Exe`, and `Ctl` from Listings 3.5 to 3.7. The robotic platform has the name of the platform from the model `DeliveryRobot` from Listing 3.3. The connections between the layers of the model from Listing 3.8 map to the connections between the controllers in the RoboChart model. We will now examine the structure of each of the generated components in turn.

Figure 3.4: MailDelivery RoboChart Module

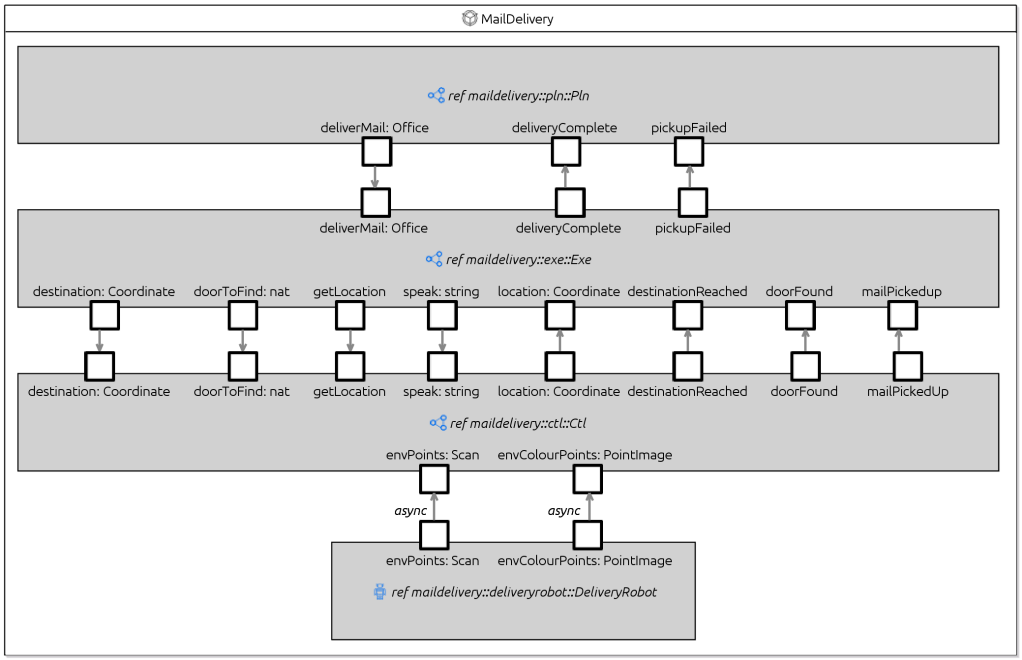
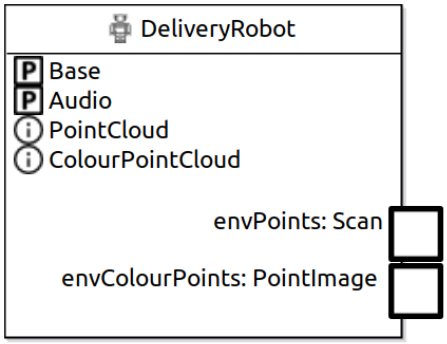


Figure 3.5: DeliveryRobot RoboChart Platform



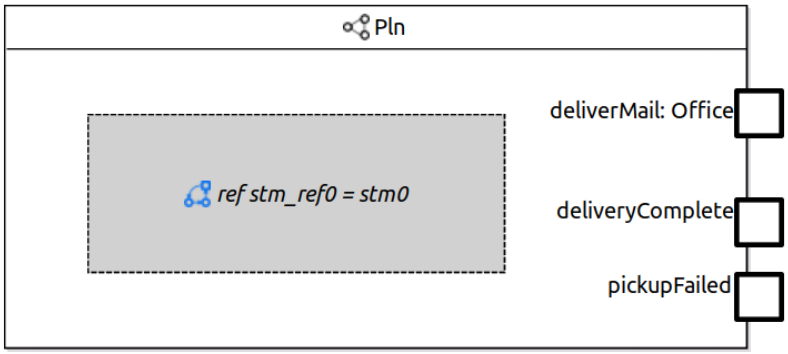
The platform of the RoboArch model as we have seen in the rules is directly reused in the RoboChart model. Figure 3.5 shows RoboTool's graphical representation of the platform of our mail delivery example. The types from the RoboArch model are based on RoboChart's so they can be directly reused by the generated RoboChart model. These types

are used throughout the RoboChart model's controllers, platforms, and state machines. RoboTool's graphical representation of the types can be found in Appendix C.1.

The provided interfaces Base and Audio and the defined interfaces PointCloud and ColourPointCloud are within the platform. The defined interface events are visible as the boxes on border of the platform.

Figure 3.6 shows the planning layer Pln. The inputs and outputs events appear along the border of the controller. Inside the controller is a single

Figure 3.6: Pln RoboChart Controller



minimal machine because in our mail delivery example no pattern is specified by the RoboArch description. The PatternToMachinesAndConnections Rule 4.4.1 defined in the next chapter characterises this machine. The minimal machine acts as a placeholder to be customised by the user to specify their required behaviour.

The minimal machine consists of a single initial junction, a state s0, and a transition that leads from the initial junction to s0. Figure 3.7 shows RoboTool's graphical representation of the minimal machine.

Figure 3.7: Minimal RoboChart State Machine

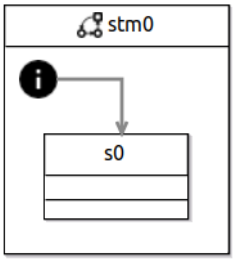
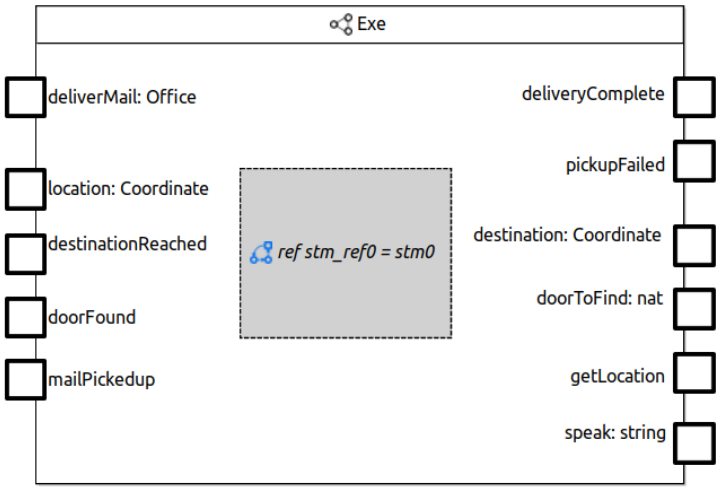


Figure 3.8 shows the executive layer Exe; it follows the same structure as the Pln layer. The input and outputs of the layer are along the border of the controller. Inside is the minimal state machine, to be elaborated with custom behaviour, because no pattern has been specified by the RoboArch description.

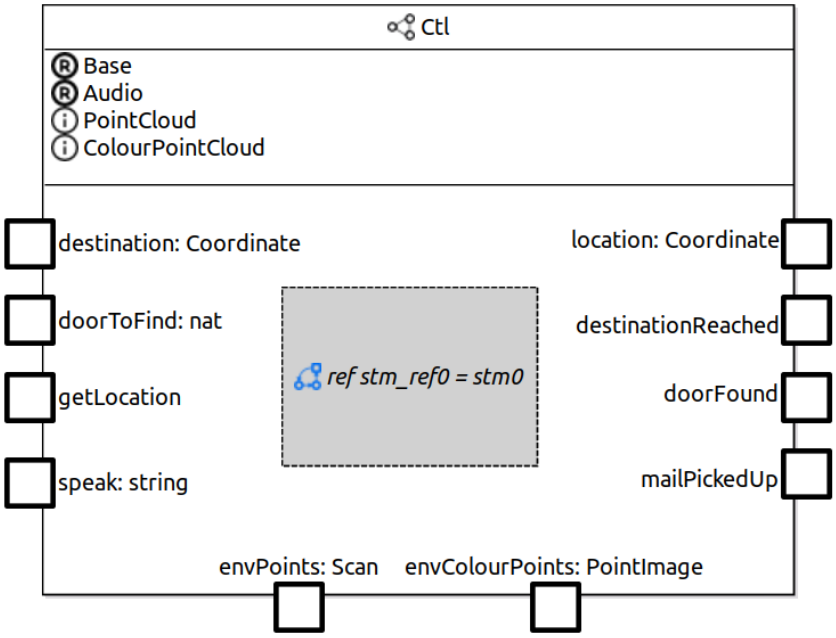
Figure 3.8: Exe RoboChart Controller



The final controller is the control layer Ctl shown in Figure 3.9. Again it follows the same structure as the previous two layers Pln and Exe with the inputs and outputs and the minimal machine as a placeholder. However, it additionally communicates with the robotic platform to make use of its services that ultimately allow the physical robot to be controlled.

Inside the top-half of the Ctl controller are the required and defined interfaces Base and Audio that declare services provided by a platform; for our example this is the DeliveryRobot. Events from the defined interfaces envPoints and envColourPoints are visible along the edges of the controller.

Figure 3.9: Ctl RoboChart Controller



We now have seen the most abstract RoboArch description of a mail delivery system that specifies a three-layer architecture without defining a layer's pattern or behaviour. Even without these, a RoboChart model is generated that a developer can use as a starting point for modelling their own custom behaviour. This abstract example demonstrates the top-level structure that is used for all RoboArch models, including those with layers when they use a pattern. We now give an overview of some patterns from the literature for each of the layers.

3.3 Approaches Used by Layers

Different approaches have been used to realise the functionality of the different kinds of layers resulting in a number of patterns that can be selected.

For control layers, examples of behavioural-based control patterns include reactive skills [42] and subsumption [27]. Examples of service based control patterns include CLARAty [36], and G^{en}oM [38].

For executive layers, the hierarchical task decomposition pattern is widely used [10, p. 294]. This is because hierarchical decomposition is effective at reducing complexity as each level in the hierarchy has a smaller number of activities available to it compared to no hierarchy. Executive languages such as RAPs [88], PRS [89], TDL [90], and PLEXIL [91] all choose to represent their tasks in this way. Architectural patterns vary in how much, if at all, their executive layer decomposes tasks.

For planning layers to determine the actions that are required to accomplish goals, hierarchical task networks and integrated planner-scheduler patterns have been used [10, Sec. 12.3.4]. The hierarchical task network pattern performs planning and scheduling as distinct phases, using a hierarchical structure that resembles the task hierarchy of the executive layer. However, the task hierarchy used by the planning layer contains tasks that are more abstract and, additionally, resource usage is taken into account. Examples of hierarchical task network planners include O-Plan [92] and HATP [93].

The integrated planner-scheduler pattern combines planning and scheduling so that additional properties such as time can be considered during planning, resulting in a plan that can better utilise the resources available. The integrated approach also enables detailed planning to be deferred or brought forward, therefore, reducing the overall computing resources required since an entire plan does not have to be found each time re-planning occurs. Examples of planner-schedulers include HSTS [94] and CASPER [55].

The different patterns for each layer outlined here can be accommodated in RoboArch as shown in the next chapter.

3.4 Final Considerations

We have introduced the textual notation of RoboArch for capturing descriptions of software architectures for robotics. The metamodel that underpins RoboArch models has been presented and the well-formedness conditions have been defined so that meaningless models can be flagged. Transformation rules from RoboArch to RoboChart give a semantics to RoboArch layers. For an architecture using layers that do not have a pattern, the resulting RoboChart model and its structure for the top-level rules has been demonstrated.

In the next chapter we examine the reactive skills layer pattern to demonstrate in detail how patterns can be used to add further structure and behaviour to layers.

4 Patterns in RoboArch

In this chapter, we present the RoboArch notation for patterns, their metamodel, and the transformation rules to RoboChart.

Section 4.1 introduces the notation f.2or patterns of a layer and the metamodel for the pattern types. Section 4.2 presents the notation for the reactive skills pattern and our characterisation as a metamodel and well-formedness conditions. Section 4.3 presents the notation for the subsumption pattern. Section 4.4 presents the transformation rules for the patterns and the generated RoboChart model for the mail delivery example. We conclude in Section 4.5 with some final considerations. First, we give the general notation that each pattern further defines.

4.1 RoboArch: Patterns

Patterns define further internal structure to the layers; they also define behaviours and contribute to the resulting software functionality. We now examine the structure of patterns and how they are modelled using RoboArch including the encoding of behaviour they define within transformation rules. The RoboArch model and rules enable the automatic generation of partial RoboChart models (or sketches) that include structural and behavioural components as specified by the pattern. The generated partial models can be completed by the developer to add the remaining implementation specific details for the verification of system properties.

Here we present the general RoboArch notation for a layer that is defined by a pattern. Listing 4.1 shows the structure of a RoboArch layer. Line 1 declares a layer named `lptn` of generic type. I, O, P_{Typ}, and P_{Def} are statements defined by the developer for their system. In Chapter 3 we have seen how the layers inputs I on Line 3 and outputs O on Line 5 can be defined.

The `pattern` clause on Line 7 declares the pattern of the layer by its type P_{Typ} that can be `ReactiveSkills`, `Subsumption`, `Htn`, or `PlannerScheduler`. Dealing with additional patterns is future work

but the examples shown here describe how this can be done.

Listing 4.1: Layer with pattern.

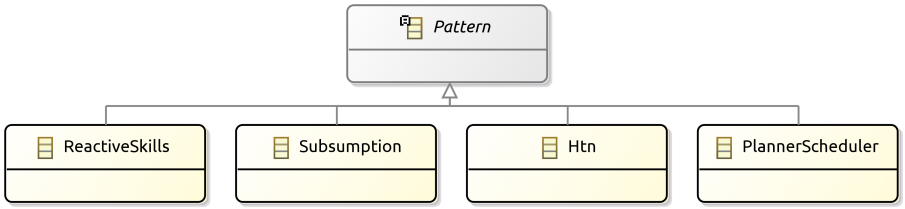
```
1  layer lptn {
2
3      inputs=  I;
4
5      outputs=  O;
6
7      pattern= P_Typ;
8
9      P_Def
10
11 } ;
```

The pattern definition that describes the structure of the modelled system in terms of its concepts are given by P_{Def} on Line 9. This form for layers that include a pattern provides the means to specify further details for a given pattern. We will now describe the metamodel for RoboArch pattern types.

4.1.1 Metamodel

Figure 4.1 defines a RoboArch pattern as an abstract class that is the parent class of all the supported patterns. Each pattern extends the

Figure 4.1: Pattern types metamodel



Pattern class with its own concepts that form the pattern's definition. This metamodel structure allows interchangeability of a layer's pattern giving developers the flexibility to select a pattern. We introduce no well-formedness conditions for the Patterns, for example, restricting the layer in which a pattern can be used. Instead, these conditions are included in the well-formedness conditions for the individual patterns.

In the next section we examine the reactive skills pattern and its corresponding ReactiveSkills class.

4.2 Reactive Skills

The reactive skills pattern is a pattern that can be used for the control layer and means that it interfaces the robotic platform with another layer. Here we provide our characterisation of the pattern and elaborate the mail delivery example of Chapter 3 showing how RoboArch can be used to model a layer that uses reactive skills.

RoboArch enables the description of a layer's pattern using the concepts of reactive skills. We extend the control layer of the mail delivery system that was presented in Chapter 3 to include the structure of behaviours as reactive skills. Our example for the reactive skills pattern models the behaviours responsible for moving the delivery robot to a given target location. Listing 4.2 shows the RoboArch model for our reactive skills example. The listing instantiates the pattern type P_{Typ} and pattern definition P_{Def} introduced earlier in this chapter.

The type of pattern a layer uses is specified by the `pattern` clause of RoboArch and it determines the subsequent clauses that can be declared in the pattern definition P_{Def}. For reactive skills the clauses available are `skills`, `connections`, and `monitors`. Line 1 of the listing declares P_{Typ} as a `pattern` type of `ReactiveSkills`. The remaining lines 3 to 35 define the pattern definition P_{Def}.

Skills are a fundamental concept to the pattern and the structure it provides. Every skill must have a unique descriptive name that identifies it within its layer. Optionally skills can have parameters, a priority, inputs, and outputs.

The `skills` clause on Line 3 declares the skills of the layer. There are separate clauses for defining each type of skill: `dskill` for D-Skills and `cskill` for C-Skills. In our example there are three D-Skills that have names `Move`, `ColourVision`, and `Proximity` declared on lines 4, 8, and 12, respectively. There are also two C-Skills that have names `MoveToLocation` and `DetermineLocation` declared on Lines 16 and 23, respectively.

In the reactive skills pattern D-Skills interface physical devices so the outputs of D-Skills are from sensors and inputs are to actuators. RoboArch `dskills` declare ouputs using the `outputs` clause and inputs using the `inputs` clause. In our example, the `Move` D-Skill has one input,

declared on Line 5, with name `velocity` and type `Velocities`.

Listing 4.2: Reactive skills movement

```
1  pattern= ReactiveSkills;
2
3  skills=
4    dskill Move {
5        inputs= velocity:Velocities;
6    },
7
8    dskill ColourVision {
9        outputs= envColourPoints:PointImage;
10   },
11
12   dskill Proximity {
13       outputs= envPoints:Scan;
14   },
15
16   cskill MoveToLocation {
17       parameters=  target: Coordinate;
18       inputs=  current:Coordinate,
19              hazards:Scan;
20       outputs= velocity:Velocities;
21   },
22
23   cskill DetermineLocation {
24       inputs= image:PointImage;
25       outputs= location:Coordinate;
26   };
27
28 connections=
29     ColourVision on envColourPoints to DetermineLocation
       on image,
30     Proximity on envPoints to MoveToLocation on hazards,
31     DetermineLocation on location to MoveToLocation on
      current,
32     MoveToLocation on velocity to Move on velocity ;
33
34 monitors= ( DestinationReached |
35   DetermineLocation::location == MoveToLocation::target
      );
```

A skill communicating a value to a `dskill`'s `input` results in the physical state of the device that the `dskill` represents being affected by

the given value. In our example, a value communicated to the `Move` skill's `velocity` input results in the velocities of the motors in the robot's base being set.

A skill receiving a value from a `dskill`'s `output` represents the state of the environment the robot is currently situated, as sensed by the device the `dskill` represents. In our example, a value received from the `Proximity` skill's `envPoints` output gives a set of ranges that are distances to surfaces in the delivery robot's field of view.

In the reactive skills pattern a C-Skill uses its inputs to compute its outputs resulting in behaviour that can be used to accomplish parts of a task. A `cskill`'s inputs and outputs in RoboArch are declared using `inputs` and `outputs` clauses.

In our example, the `DetermineLocation` skill takes a colour image of the environment and by an image based localisation technique calculates the coordinates the delivery robot is located. Therefore, to perform its function `DetermineLocation` has one `input` with name `image` of type `PointImage`, declared on Line 24, and one `output` with name `location` of type `Coordinate`, declared on Line 25. The computational transform that specifies the behaviour of C-Skills is left to be defined by customising the generated RoboChart model.

The `MoveToLocation` skill calculates the velocities required to move the delivery robot to the target location, stopping if any obstacles are encountered. The inputs the skill require are the current location of the delivery robot named `current` and the distance of objects in the field of view named `hazards` defined on Lines 18 and 19, respectively. The target location is provided as parameter `target` defined on Line 17 and is set by another layer. The calculated output from the given inputs is called `velocity` and defined on Line 20.

Skills can communicate among themselves via the skills manager. The source and destination of the skills' inputs and outputs are determined by connections. The connections between skills are defined in the `connections` clause. Each connection is unidirectional and connects an input of one skill to an output of another. Lines 28 to 32 declare skill connections for our mail delivery example.

Line 29 declares a connection from the `ColourVision` skill's `envColourPoints` output to the `DetermineLocation` skill's `image` input.

Layers that depend on a reactive skills layer may want to monitor for particular conditions becoming true. In order to minimise the frequency the dependant layer needs to check conditions, the reactive skills pattern

provides events that are asynchronously triggered to notify the dependant layer on the occurrence of any monitored conditions.

The `monitors` clause on Line 34 declares the monitors for the layer. Monitors must have a descriptive name for identification and define the logical condition that is to be monitored in terms of skill outputs and parameters. For our example, a condition that is useful to monitor is the arrival of the delivery robot at the target location. Line 34 and 35 define a condition named `DestinationReached` whose condition evaluates to true when the `location` output of the `DetermineLocation` skill is equal to the `target` parameter of `MoveToLocation` skill.

We have now introduced all parts of the textual notation for describing the movement of the mail delivery example using the reactive skills pattern. Next we describe the RoboArch metamodel for reactive skills.

4.2.1 Metamodel

The structure of the RoboArch model for a pattern is determined by a metamodel that specifies the available concepts and how they relate to one another. In this section we focus on the reactive skills pattern of RoboArch models: the skills, skills manager, skill connections, and monitors.

The reactive skill pattern ReactiveSkills is naturally a subclass of the RoboArch Pattern and extends it to include the concepts of reactive skills. Figure 4.2 defines a RoboArch ReactiveSkills pattern.

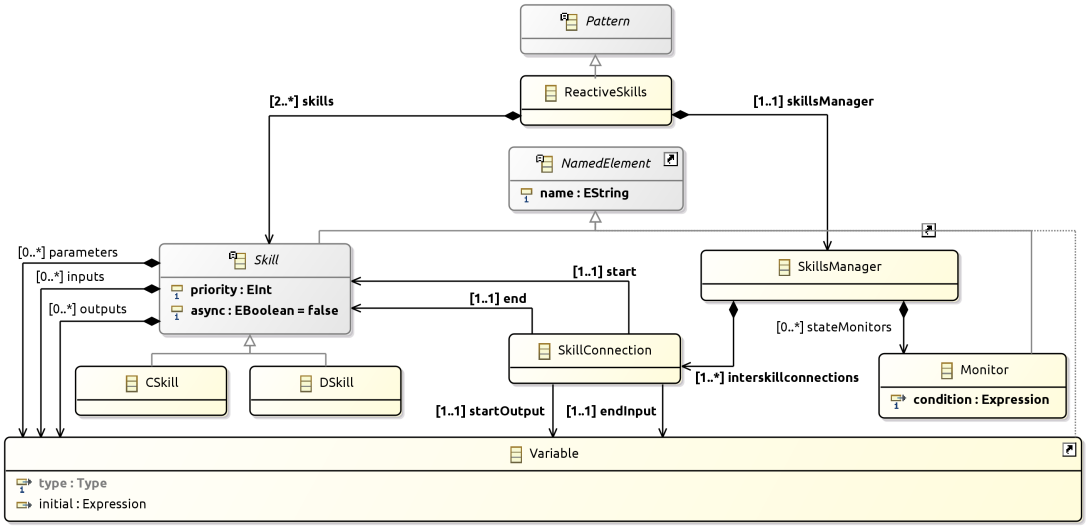
There must be at least two Skills and exactly one skills manager for the model to be meaningful. Additionally, out of the skills there must be at least one C-Skill and at least one D-Skill which is enforced by well-formedness conditions in the Section 4.2.2.

Skills can have inputs, outputs, and parameters that they use to perform their operation all represented by Variables. Skills can be asynchronous and have a priority value indicated by a boolean attribute async and an integer attribute priority.

Skill is an abstract class and its subclasses are CSkill and DSkill. They do not further specialise their parent Skill because they are structurally similar. However, CSkills and DSkills have distinct behaviours and therefore their own well-formedness conditions and so must be distinguishable.

A SkillsManager manages inter-skill connections and event monitoring so they can have Monitors and at least one SkillConnections. Monitors have a name for identification and a condition that when satisfied a corresponding event alerting the dependant layer is triggered. The condition

Figure 4.2: Reactive skills metamodel



attribute is a RoboChart expression.

Lastly, SkillConnections determine the source of the input values for Skills that the skills manager coordinates. SkillConnections are between two Variables, a source startOutput and a target endInput that belong to the respective start and end Skills. The metamodel does not restrict the Variable and types that can be connected so not all of the possible connections are meaningful, for example, if the connected Variables are both inputs. Well-formedness conditions are used to restrict models to only those that are meaningful.

4.2.2 Well-Formedness

Well-formedness conditions restrict the models allowed by the RoboArch metamodel to only those that are meaningful. Here we focus on conditions relating to the reactive skills pattern.

RS1 A Layer that has a pattern of type ReactiveSkills must be a GenericLayer or ControlLayer.

This is because the reactive skill pattern provides the fundamental behaviours that interact with the sensors and actuators via the services of robotic platform, which other layers depend on to carry out the robot's

tasks. Therefore, the reactive skill pattern can only be used by layers that can communicate with the robotic platform.

RS2 For a Layer with pattern type ReactiveSkills, at least one of the System's connections is from that layer to a RoboticPlatform or that layer has at least one rinterface.

The reactive skill pattern provides behaviours to other layers that they use to control the robot in order to carry out their tasks. Therefore, a layer using the reactive skills pattern must communicate in some way with the robotic platform either by events, variables, or operations.

RS3 For each event of ReactiveSkills Layer's interfaces, there must be a DSkill input with a matching name.

Establishes a relationship between the input of a D-Skill and events of the robotic platform that have the same names. This means that connections between the D-Skills and events of the robotic platform can be defined in the transformation rules to RoboChart.

RS4 ReactiveSkills must contain a CSkill and a DSkill.

A C-Skill or D-Skill in isolation can perform no meaningful function that alters the state of the robot or its environment. A C-Skill requires a D-Skill in order to interact with a sensor or actuator via the services of the robotic platform.

RS5 A CSkill must have at least one output.

A C-Skill that does not provide an output does not contribute to the behaviour of the system and is therefore redundant and can be removed.

RS6 A DSkill must have at least one output *or* input.

A D-Skill interfaces the other skills with a robot's sensors and actuators via the services of the robotic platform. Therefore, a D-Skill with no inputs or outputs cannot communicate values to or from any skills to be able to use the platform and any of the services it provides.

RS7 The start and end Skill of a SkillConnection must be distinct.

A skill's output must communicate values to another skill and not to itself. A skill should use its own variables for any of its intermediate values.

RS8 The startOutput of a SkillConnection must be an output of its start Skill.

This is because the skill connection's start attribute specifies the context of allowable variable for the skill connection's startOutput.

RS9 The endInput of a SkillConnection must be an input of its end Skill.

This is because the skills connection's end attribute specifies the context of allowable variable for the skill connection's endInput. In combination with RS8 this ensures only an input can be connected to an output and vice versa.

RS10 The types of the startOutput and endInput of a SkillConnection must match.

This ensures that the communications carried out via skill connections are well typed.

RS11 A Monitor's condition must only refer to parameters, inputs, and outputs of the Skills.

Because expressions allow comparisons across variables of a system, the only conditions that can be monitored by a skills manager are those that include the attributes of skills.

In the next section we introduce subsumption pattern and its RoboArch notation.

4.3 Subsumption

The subsumption pattern is a pattern that can be used for the control layer and means that it interfaces the robotic platform with another layer. Here we provide an overview of the pattern and elaborate the mail delivery example of Chapter 3 showing how RoboArch can be used to model a layer that uses subsumption.

4.3.1 Overview

Listing 4.3 and Listing 4.4 show the RoboArch model for our subsumption example. The listing instantiates the pattern type P_{Typ} and pattern definition P_{Def} introduced earlier in this chapter.

Listing 4.3: Subsumption movement

```
pattern= Subsumption;

modules=
  module Move {
      inputs= velocity:Velocities, halt, go;
      states= Moving, Halted;
  },

  module DetermineLocation {
      inputs= image:envColourPoints;
      outputs= location:Coordinate;
  },

  module Collide {
      inputs= hazards:Scan;
      outputs= halt, go;

  },

  module Avoid {
      inputs= hazards:Scan, velocity:Velocities;
      outputs= adjustedVelocity:Velocities;
      states= NoObstacle, Avoiding
  },

  module MoveAlongCorridor {
      inputs= image:EnvColourPoints;
      outputs= velocity:Velocities;
      states= Following, NoCorridor;
  },

  module MoveToLocation {
      inputs= target: Coordinate, current:Coordinate;
      outputs= velocity:Velocities, arrived;
      states= Moving, Waiting;
  } ;
```

Listing 4.4: Subsumption movement connections

```
37 connections=
38     connect Collide on go to Move on go,
39     connect Collide on halt to Move on halt,
40     connect Avoid on adjustedVelocity to Move on
       velocity,
41
42     connect Ctl on envPoints to Collide on hazards Avoid
        on hazards,
43
44     connect Ctl on envColourPoints to MoveAongCorridor
       on image DetermineLocation on image,
45     connect MoveAlongCorridor on velocity to Avoid on
       velocity,
46     connect DetermineLocation on location to
       MoveToLocation on current,
47     connect MoveToLocation on velocity to (suppress
       Avoid on velocity 50),
48     connect Ctl on destination to MoveToLocation on
       target,
49     connect MoveToLocation on arrived to Ctl on
       destinationReached,
50 } ;
```

As mentioned the type of pattern a layer uses is specified by the `pattern` clause of RoboArch and it determines the subsequent clauses that can be declared in the pattern definition P_{Def}. For subsumption the clauses available are `modules` and `connections`. Line 1 of the listing declares a `pattern` type of `Subsumption`.

Modules are a fundamental concept to the pattern and the structure it provides. Every module must have a unique descriptive name that identifies it within its layer. Optionally modules can have inputs, outputs, variables, and states.

The `modules` clause on line 3 declares the modules of the layer. In our example there are six modules that have names `Move`, `Collide`, `Avoid`, `DetermineLocation`, `MoveAlongCorridor`, and `MoveToLocation` declared on lines 4, 9, 14, 20, 26, and 32, respectively.

Modules are state machines that apply functions to their inputs to produce outputs. In our example the `DetermineLocation` module has one input `image` and one output `location`. The function of the module is to, on receiving an `image` value, calculate the location via an image recognition function not specified here and output the coordinate via the

`location` output.

States can be defined for a module using the `states` clause. In our example the `Avoid` module declares two states `NoObstacle` and `Avoiding` on line 23 that make up the behaviour of the module for later elaboration by the developer.

In the subsumption pattern modules can communicate with the physical robot however no mechanism is defined to do this. RoboArch's characterisation of subsumption utilises the robotic platform so events that connect the robotic platform via the `Ctl` layer represent events from the robot's sensors and actuators.

The other means that a module can use to communicate with the robot is through operations and variables of platform via the controllers required interfaces in the module's state definitions.

Modules can communicate among themselves; connections determine the inputs and outputs that are connected. Additionally, connections specify the subsumption pattern's characteristic suppression and inhibition and shape the resulting behaviour of the combined skills. The connections between modules are defined under a layer's `connections` clause. Each connection is unidirectional and asynchronous and connects an input of one skill to the output of another. Lines 37 to 49 of Figure 4.4 declare connections for our mail delivery example.

Line 38 defines a connection from the source `go` output of the `Collide` module to the destination input `go` of the `Move` module. Connections can have multiple destination inputs where the source event output is received by all of the destinations. Line 42 is an example of a connection with two destinations `hazards` of the `Collide` module and `hazards` of the `Avoid` module.

The destinations of a connection can optionally specify suppression or inhibition to suppress or inhibit connections to a module for a given number of time units. In our example Line 47 specifies a suppressing connection of the `velocity` destination on the `Avoid` module for `50` time units following an event being output by the source.

The description captured by the notation of RoboArch forms a model that allows the application of model-driven techniques to reason about and analyse the system being developed. In the next section we introduce the transformation rules that are used to generate RoboChart models for the reactive skills pattern.

4.4 Rules

The transformation rules presented in this section extend the rules from Chapter 3 to include the patterns. They follow the same five part structure of name, parameters, result, optional precondition, and definition.

We have seen previously how each layer of a RoboArch model corresponds to a controller in the RoboChart model. Now we introduce the rule that transforms the different patterns into machines and connections which populate the controllers defining their behaviour.

4.4.1 Patterns

RoboArch supports a selection of patterns that can be used by its layers. The PatternToMachinesAndConnections Rule 4.4.1 is responsible for applying the rules that handle each of the patterns. The rule takes a set of RoboArch patterns containing no or just one pattern and produces a set of state machines and a set of connections.

Layers that have no pattern defined, result in **amptn** being empty and are the most abstract specification of a layer that RoboArch allows. For these layers, the resulting machines rcstms is a single minimal machine **minimalsm** and as such has no connections so **rccns** is empty.

The **minimalsm** is specified by the where clause of the rule definition as an initial junction named initial, state named s0, and transition named initialTos0 that transitions immediately from initial to **s**0. The minimal state machine serves as a placeholder that the user customises with their own behavioural specification.

RULE 4.4.1:

ⓐ	**Name**	PatternToMachinesAndConnections
ⓑ	**Parameter** *name:type*	amptn: Set(Pattern)
ⓒ	**Result** *name:type*	rcstms: Set(StateMachine), rccns: Set(Connections)

Continued on next page

RULE 4.4.1 – continued from previous page

ⓔ	**Definition**	**if** (#amptn = 0) **then**
		rcstms= { minimalsm }
		rccns= ∅
		else
		rcstms= ⋃ ⟪ stmscons ⟫.1
		rccns= ⋃ ⟪ stmscons ⟫.2
		where
		stmscons= ⋃ {
		{ p: amptn ∣ p ∈ ReactiveSkills •
		ReactiveSkillsPatternToMachinesAndConnections(p) },
		{ p: amptn ∣ p ∈ Subsumption •
		SubsumptionPatternToMachinesAndConnections(p) },
		{ p: amptn ∣ p ∈ Htn •
		HtnPatternToMachinesAndConnections(p)},
		{ p: amptn ∣ p ∈ PlannerScheduler •
		PlannerSchedulerPatternToMachinesAndConnections(p) }
		}
		minimalsm= ⟪
		nodes= { ⟪ name= "initial" ⟫$_{Initial}$,
		⟪ name= "s0" ⟫$_{State}$ }
		transitions= {
		⟪ name= "initialTos0",
		source= ref("initial", Initial) ,
		target= ref("s0", State)
		⟫$_{Transition}$ }
		⟫$_{StateMachine}$

For layers that have a defined pattern, the resulting machines, **rcstms**, are given by the generalised union of the first component of relational image of **stmscons**. The connections, **rccns**, are similarly defined but are given by the second component of the relational image of **stmscons**.

The where part of the rule definition specifies the machines and connections **stmscons** by applying for each type of pattern in **amptn** a rule of

the form [*PatternType*]PatternToMachinesAndConnections. For example, if the pattern is a ReactiveSkills type the ReactiveSkillsToMachinesAndConnections is applied with the pattern passed in as a parameter. These rules return a tuple containing a set of machines and a set of connections for the pattern specified by the RoboArch model.

In addition to the rules that specify the behaviour of patterns there are others responsible for providing supporting types, functions, and interfaces: Rule B.1.1 LayerToTypes, Rule B.1.2 LayerToFunctions, and Rule B.1.3 LayerToInterfaces which can all be found in Appendix B.

Rules B.1.1 to B.1.3 are applied by the top-level Rule 3.2.1 and are similar to Rule 4.4.1 because they apply rules specific to each pattern, however, for the supporting definitions they add. Because these rules simply apply others based on pattern type they will not be discussed in detail here, instead, the pattern specific rules they apply will be introduced in the upcoming sections for the pattern type. Next we take a detailed look at each the rules for the reactive skills pattern used for generating RoboChart models.

4.4.2 Reactive Skills

From a RoboArch description a RoboChart model can be generated via model transformation. The transformation rules presented in this section cover the reactive skills pattern. We will use the mail delivery example from Section 4.2 to demonstrate the resulting RoboChart model that is generated by application of the rules.

A summary of the concepts of the reactive skills pattern from Section 2.2 and how they map to concepts of RoboChart along with the respective rules is given in Tables 4.2 and 4.3. The rule that applies the rules from the tables is Rule 4.4.2 ReactiveSkillsPatternToMachinesAndConnections.

Table 4.2: Mapping of the primary architectural concepts of the reactive skills pattern to RoboChart.

Reactive Skills		*RoboChart*		
Concept		**Concept**	**Mapping**	**Rule**
Skill	$\rightarrow$	State machine	A state machine with - *Events:* for each of the inputs, outputs, and any parameter values that the skill uses. *Variables:* for each input (to hold received corresponding event values), parameter values, and any other necessary intermediary results as part of the skill's computational transform.	4.4.4
Skill manager	$\rightarrow$	State machine	A state machine that interfaces the skills to the executive layer with - *Events:* To manage the activation and deactivation of skills, receive parameter values, and communicate monitor-event and information replies. *Variables:* for each input and any other necessary for the coordination of skills execution.	4.4.7

Table 4.3: Mapping of the reactive skills pattern related concepts to RoboChart.

* = Modelled using a combination of RoboChart concepts.

Reactive Skills		*RoboChart*		
Element		**Concept**	**Mapping**	**Rule**
Initialisation routine	→	*	A state named initialise of a skill state machine that executes before all other states of the skill.	4.4.5 (C-Skill), 4.4.6 (D-Skill)
Startup	→	*	A state named startup of a skill state machine that is entered after the skill has been enabled.	4.4.5 (C-Skill), 4.4.6 (D-Skill)
Reply	→	*	An output event of the skill manager to the executive layer.	4.4.7
Cleanup	→	*	A state named cleanup of a skill state machine that is entered when the skill has been disabled.	4.4.5 (C-Skill), 4.4.6 (D-Skill)
Parameter	→	*	Event and a variable, whose value is received from the skills manager, of a state machine that represents a skill. The event is connected to the skill manager.	4.4.5 (C-Skill), 4.4.6 (D-Skill), 4.4.3
Input	→	*	Event and a variable, which is used to receive a value, of a state machine that represents a skill. The event is connected to the skill manager.	4.4.5 (C-Skill), 4.4.6 (D-Skill), 4.4.3
Output	→	Event	Event of a state machine that is used to output a value. The event is connected to the skill manager.	4.4.5 (C-Skill), 4.4.6 (D-Skill), 4.4.3

Continued on next page

Table 4.3 – *Continued from previous page*

Reactive Skills		*RoboChart*		
Element		**Concept**	**Mapping**	**Rule**
Computational transform	→	*	The combination of functions or operators that the enabled state applies to the input events resulting in the skill's output events.	4.4.5 (C-Skill)
Enable function	→	*	A RoboChart event of a state machine representing a skill that triggers the enabled state to be entered.	4.4.5 (C-Skill), 4.4.6 (D-Skill)
Disable function	→	*	A RoboChart event of a state machine representing a skill that triggers the disabled state to be entered.	4.4.5 (C-Skill), 4.4.6 (D-Skill)

RULE 4.4.2:

ⓐ	**Name**	ReactiveSkillsPatternToMachinesAndConnections
ⓑ	**Parameter** *name:type*	amptn: ReactiveSkills
ⓒ	**Result** *name:type*	rcstms: Set(StateMachine), rccns: Set(Connections)
ⓔ	**Definition**	rcstms = {skl: amptn.skills • SkillToStateMachine(skl)} ∪ {SkillsManagerToStateMachine(amptn.skillsManager, amptn.skills)} rccns = SkillsManagerToConnections(amptn.skillsManager, amptn.skills)

The ReactiveSkillsPatternToMachinesAndConnections rule takes a ReactiveSkills pattern **amptn** and produces a set of machines and a set of connections. The specification of the resulting set of machines **rcstms** is

given by the union of machine representations for each skill of **amptn** and the skills manager.

The machines representing the skills are defined by the set comprehension that applies the SkillToStateMachine rule for each skill of **amptn**. The machine representing the skills manager is defined by the application of the SkillsManagerToStateMachine rule taking the pattern's skills manager and skills as parameters.

The resulting set of connections **rccns** of the ReactiveSkillsPatternToMachinesAndConnections rule specify the connections among the machines. The specification of the rccns connections are given by applying the SkillsManagerToConnections rule taking the pattern's skills manager and skills as parameters.

The SkillsManagerToConnections Rule 4.4.3 specifies connections that connect the skills and skills manager machines. The rule returns a set of connections **rcsmc** that is specified as the generalised union of the set comprehension over each skill. The expression part of the comprehension applies the GetConnectionsForCommonInterface Function B.3.1 that takes four parameters: connection node *a* (target), connection node *b*, the name of the interface, the input events, and the output events.

The function returns a set of connections that connect the events of the named interface that each of the nodes share. The directionality of the connections is specified by the input and output events with respect to the target node *a*.

The input and output event parameters determine the direction of the connections to the events of the skills manager, **smstm**, target node. The where clause of the rule specifies **smstm** as the reference to the skills manager machine. This results in the rule connecting each skill and its interface to the corresponding interface of the skills manager and allows the skills manager to coordinate skills communication. The events of the skills interface are explained when the skill to machine rules are introduced.

The generated RoboChart controller for the mail delivery example given by Listing 4.2 that uses the reactive skills pattern is shown in Figure 4.3. The minimal machine of the `Ctl` layer from Chapter 3 where no pattern was specified is replaced with the machines and connections defined by Rule 4.4.2.

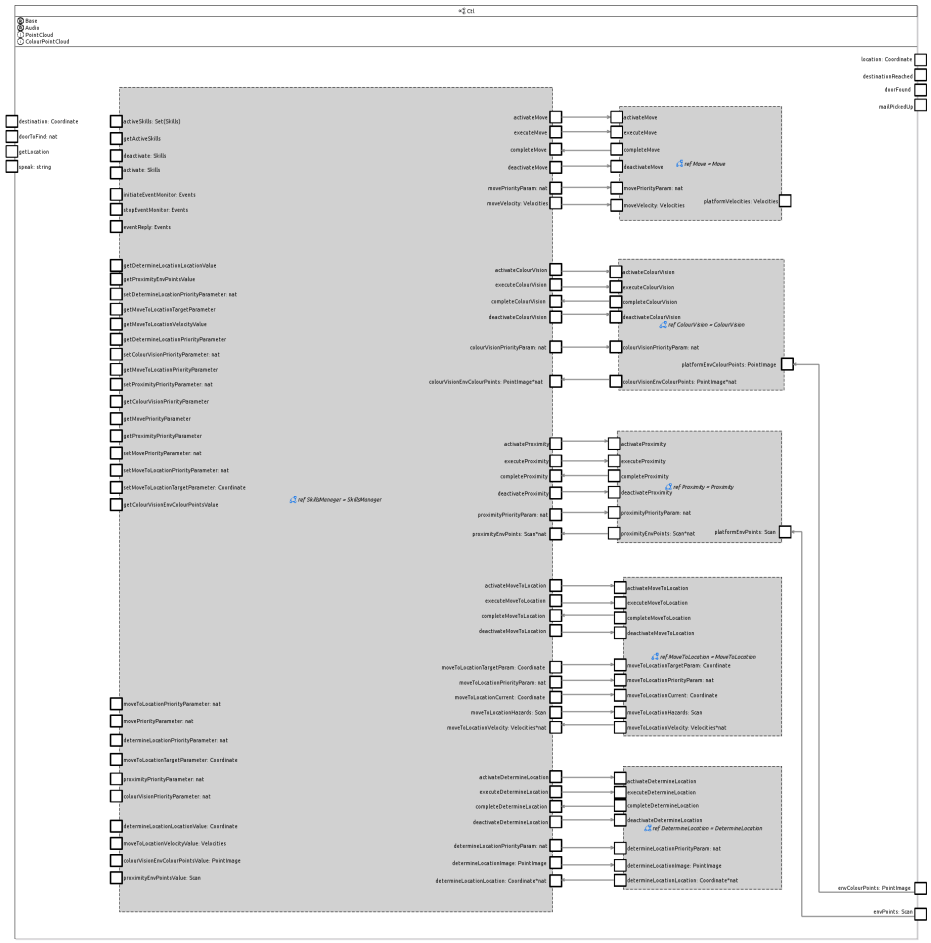
There are six machines created, one for the skills manager and each of the skills. The skills events are connected to the SkillsManager machine that coordinates communication among skills and are defined by the

RULE 4.4.3:

ⓐ	**Name**	SkillsManagerToConnections
ⓑ	**Parameter** *name:type*	amsm: SkillsManager, skls: Set(Skills)
ⓒ	**Result** *name:type*	rcsmc: Set(Connection)
ⓔ	**Definition**	rcsmc = ⋃ { sk : skls • GetConnectionsForCommonInterface(smstm, ref(sk.name, StateMachine), SkillInterfaceName(sk.name), ref(SkillCompleteEventName(sk.name), Event) ∪ { o: sk.outputs • ref(SkillEventName(sk.name, o.name), Event) }, { ref(SkillActivateEventName(sk.name), Event), ref(SkillDeactivateEventName(sk.name), Event), ref(SkillExecuteEventName(sk.name), Event) } ∪ { i: sk.inputs • ref(SkillEventName(sk.name, i.name), Event) } ∪ { p: sk.parameters • ref(SkillParameterEventName(sk.name, p.name), Event) }) } --- ***where*** smstm= ref(SkillsManagerName(), StateMachine)

SkillsManagerToConnections rule. The connections that connect `dskills` platform events to the controller platform events are specified by the LayersToControllers Rule 3.2.2. The connections between the layer Ctl con-

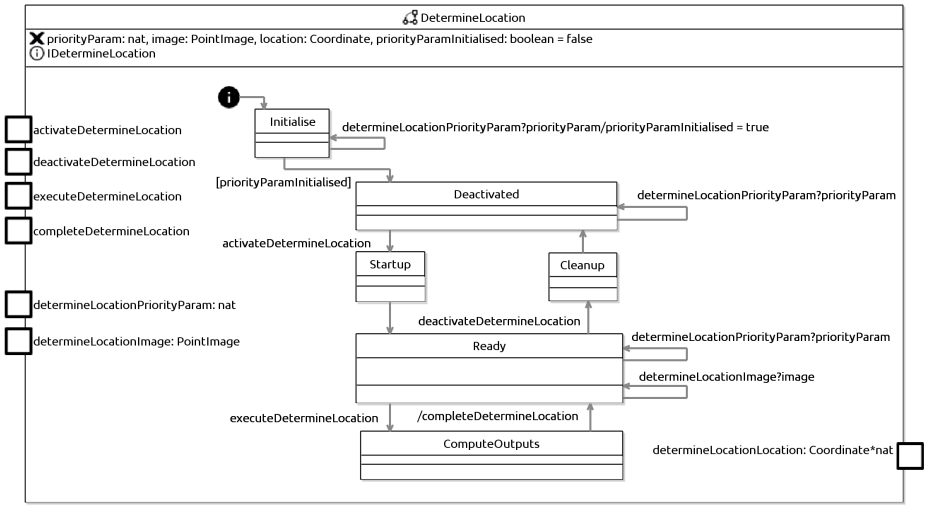
Figure 4.3: Ctl RoboChart Controller



troller and the SkillsManager machine must be completed by the designer to adapt the interface that they defined for the layer and the interface the layer's pattern specifies. Next we present the rules that specify the behaviour of the skills and skills manger machines.

Figure 4.4 shows the RoboChart machine for the mail delivery movement `DetermineLocation` C-Skill of Listing 4.2. A C-Skill state machine starts at the state Initialise, where it accepts a priority for the skill via an event: in our example, DetermineLocation. When that input is taken, the variable priorityParamInitialised is updated to record that. Once that variable has value true, a transition to the state Deactivated becomes en-

Figure 4.4: C-Skill Determine Location RoboChart Machine



abled, and is taken. In Deactivated, the priority can still be updated, until the skills manager raises an activate event (activateDetermineLocation in the example), when the machine moves to the state Startup.

Typically, the designer needs to enrich the state Startup to add the actions that the skill carries out at start up, perhaps via an entry action or a machine, making Startup a composite state. When those actions are completed, DetermineLocation moves to the state Ready.

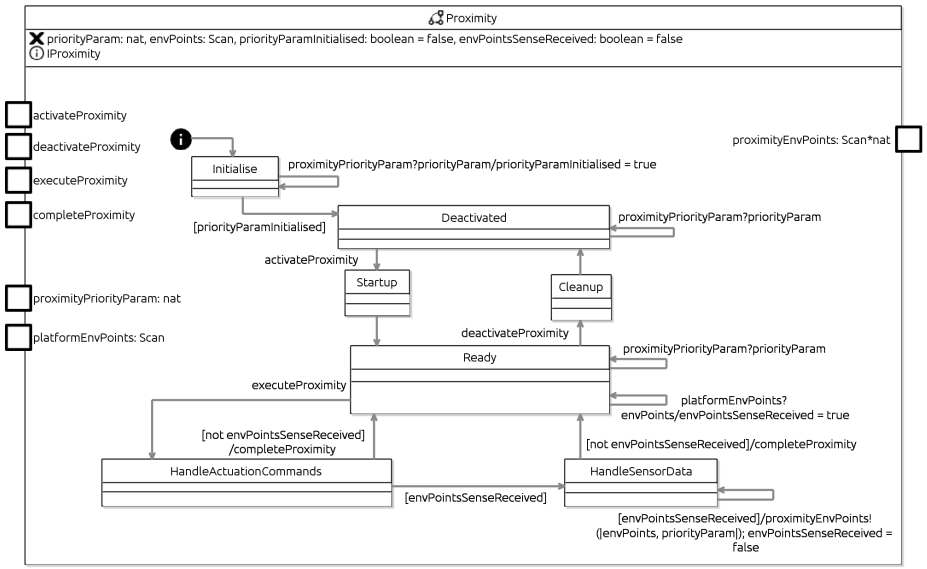
In Ready, a new priority and inputs from the skills manager may be received. In the example, the value image may be received from the skills manager via an event determineLocationImage. This can go on until the skills manager raises a deactivate or execute event (deactivateDetermine-Location or executeDetermineLocation, in the example).

On an execute event occurring, the machine moves to the state Compute-Outputs. Actions there, defined by the designer, specify the calculations to be carried out by the skill and communicates new output values to the skills manager. After the user defined actions, the machine flags that the skill has completed its task (completeProximity) and goes back to Ready.

On a deactivate event occurring the machine moves to the Cleanup state which similarly to the Startup state the designer can enrich to perform actions required to prepare for deactivation. When those actions are completed, DetermineLocation moves to the state Deactivated.

Figure 4.5 shows the generated `Proximity` D-Skill machine of the mail delivery movement example from Listing 4.2. Initially a D-Skill

Figure 4.5: D-Skill Proximity RoboChart Machine



behaves in the same way as a C-Skill until the Ready state is entered, where the skill can receive priority event values and platform event values. The transitions taken when an event is received from the platform include an action that sets a corresponding boolean variable to true. In the example, on a platformEnvPoints event occurring the received value is stored in the envPoints variable and the envPointsReceived variable is set to true.

This can go on until the skills manager raises an execute event (executeProximity, in the example) when the machine moves to the state HandleActuationCommands. Actions there, defined by the designer, might deal with buffering, for example. If, however, no input has been received (just not envPointsSenseReceived in Proximity), the machine flags that the skill has completed its task (completeProximity) and goes back to Ready.

If an input has been received, the machine moves to HandleSensorData. In general, HandleSensorData may deal with several pieces of data coming from the platform. All those that have been received may be communicated to another skill, together with its priority. In our example, we

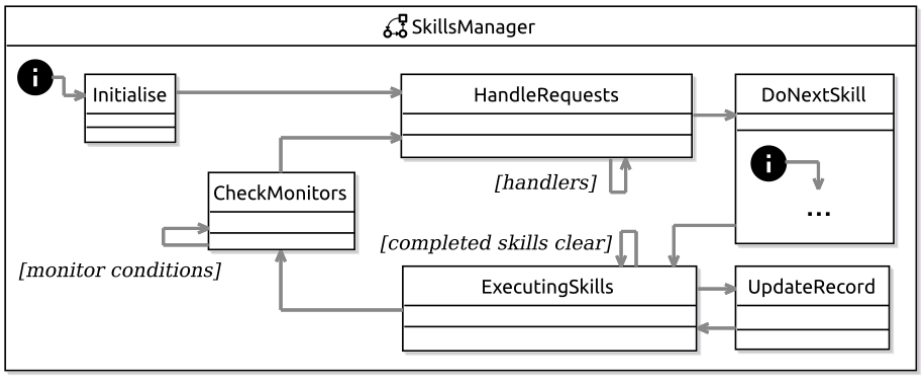
have just envPoints, which is output via proximityEnvPoints. When that happens, the value of envPointsSenseReceived is updated back to false. When all data has been communicated, a D-Skill machine moves back to Ready.

Variations of the D-Skill state-machine definition take into account D-Skills that can output to the platform, and also D-Skills that have several inputs or several outputs. The remaining Skill machines for the mail delivery movement example Move, ColourVision, and MoveToLocation can be found in Appendix C.3 to C.5.

The complete machine for our example `MailDelivery` system skills manager can be found in Appendix C.6.

Figure 4.6 sketches the machine for the skills manager. In the sketch, we show that a skills-manager machine starts in the state Initialise, in which it sets local variables, such as cycleSkills, recording the skills to execute in the next cycle. Afterwards, the skills-manager machine moves to HandleRequests.

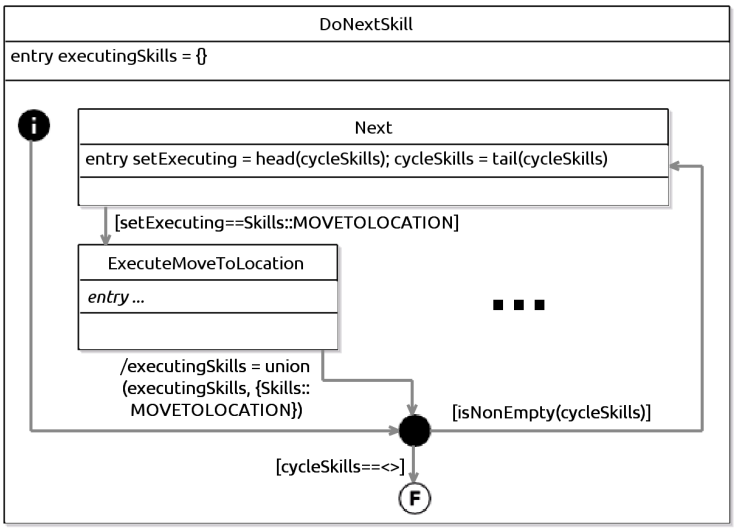
Figure 4.6: SkillsManager RoboChart overview



In the state HandleRequests, for each request, there is a transition triggered by an event that represents a request from the dependant layer, whose transition action provides the required information, or updates variables to record the request: activate or deactivate skills, initiate event monitor, stop event monitor, or set skill parameters. Once the amount of time defined by the cycle of the skill manager is past, the machine moves to DoNextSkill. The cycle time is defined by a constant, whose value can be defined by the designer or left unspecified (until simulation or code generation).

DoNextSkill, shown in the sketch of Figure 4.7, is a composite state that uses two variables. The first, cycleSkills used to start all skills that are to execute in the current cycle. The second, executingSkills is used to record skills that have been started. In the DoNextSkill machine, there is a state

Figure 4.7: SkillsManager DoNextSkill state RoboChart overview



for each skill named "Execute" + [*Skill Name*] that in its entry action sends the skill's input values and raises the event that starts its execution (such as executeDetermineLocation). After a skill has been started the transition to a junction is taken and its action updates executingSkills adding the skill's enumerated value to the set. When cycleSkills is not empty the transition to the Next state is taken. On entering Next, the machine updates the cycleSkills variable removing the next skill that is going to be executed. When all skills are started, cycleSkills gets empty, and the transition to ExecutingSkills is taken.

The state ExecutingSkills accepts the outputs of skills while they are executing. When an output is received, the machine moves to the state UpdateRecord, where the inputs to which the received output is connected are updated. This is done using an UpdateValue function (omitted here) that only updates the input if it is the first update of the cycle or if the new value comes from a skill with higher priority. After each update, the machine moves back to ExecutingSkills.

The state ExecutingSkills also accepts completion events from the skills (such as completeDetermineLocation), updating the executingSkills

variable after each such event. When all skills have completed execution, executingSkills is empty and the transition to CheckMonitors is taken.

In CheckMonitors, there is a transition for each monitor. If a monitor condition occurs, a corresponding event notifies the depending layer. When all monitors are checked, the machine moves back to the state HandleRequests, after reinitialising variables such as cycleSkills.

The SkillToStateMachine Rule 4.4.4 takes a skill amsk and produces a machine rcsk representing the skill. The rcsk machine is the result of applying the appropriate rule CSkillToStateMachine or DSkillToStateMachine for the given skills subtypes `cskill` or `dskill`.

RULE 4.4.4:

ⓐ	**Name**	SkillToStateMachine
ⓑ	**Parameter** *name:type*	amsk: Skill
ⓒ	**Result** *name:type*	rcsk: StateMachine
ⓔ	**Definition**	amskl ∈ CSkill ⇒ rcsk = CSkillToStateMachine(amsk) amskl ∈ DSkill ⇒ rcsk = DSkillToStateMachine(amsk)

The CSkillToStateMachine Rule 4.4.5 specifies interfaces, variables, nodes, and transitions that make up the C-Skill's machine. The name of the machine, **rccsk.name**, equals the name of the skill. The interfaces, **rccsk.interfaces**, that define the machine's events is the interface referenced by the ref function taking the name of the interface and the type to reference as parameters. The name of a skill's interface is given by the function SkillInterfaceName that takes the name of the skill as a parameter and is used in conjunction with **ref** to reference the skills interface without applying the rule directly.

The interface that is referenced is a result of applying the SkillToInterface Rule B.2.6. A skill's interface has events for its control and communication of its input, output, and parameter values. The events for the control of a skill include activate, deactivate, execute, and complete, each named as the skill's name prepended with the control action, for example, "activate"+[amskl.name]. For the communication of values each

input, output, and parameter has an event named as its source name prepended with the skill's name, for example, [amskl.name]+[inp.name].

RULE 4.4.5:

ⓐ	**Name**	CSkillToStateMachine
ⓑ	**Parameter** *name:type*	amcsk: CSkill
ⓒ	**Result** *name:type*	rccsk: StateMachine
ⓔ	**Definition**	rccsk.name = amcsk.name rccsk.interfaces = ref(SkillInterfaceName(amcsk.name) , Interface) rccsk.variableList = { ⟪ modifier = VariableModifier::VAR, vars= { par: amcsk.parameters • ⟪ name = SkillParamVariableName(par.name), type = par.type ⟫$_{Variable}$ } ∪ { inp: amcsk.inputs • ⟪ name = inp.name, type = inp.type ⟫$_{Variable}$ } ∪ { out: amcsk.outputs • ⟪ name = out.name, type = out.type ⟫$_{Variable}$ } ∪ { par: amcsk.parameters • ⟪ name = SkillParamGuardConditionName(par.name), type = ref(boolean, PrmitiveType), initial = BoolExp(false) ⟫$_{Variable}$ } ⟫$_{VariableList}$ }

Continued on next page

RULE 4.4.5 – continued from previous page

	rccsk.nodes = } ⟪ name = "Initial" ⟫$_{Initial}$, ⟪ name = "Initialise" ⟫$_{State}$, ⟪ name = "Deactivated" ⟫$_{State}$, ⟪ name = "Startup" ⟫$_{State}$, ⟪ name = "Ready" ⟫$_{State}$, ⟪ name = "ComputeOutputs" ⟫$_{State}$, ⟪ name = "Cleanup" ⟫$_{State}$ } rccsk.transitions = { ⟪ name = "InitialToInitialise", source = ref("Initial", Initial), target = ref(Initialise, State) ⟫$_{Transition}$, ⟪ name = "InitialiseToDeactivated", source = ref("Initialise", State), target = ref("Deactivated", State), condition = initGuardCondition ⟫$_{Transition}$, ⟪ name = "DeactivatedToStartup", source = ref("Deactivated", State), target = ref("Startup", State), trigger = ⟪ _type = TriggerType.SIMPLE, event = ref(SkillActivateEventName(amcsk.name) , Event) ⟫$_{Trigger}$ ⟫$_{Transition}$, ⟪ name = "StartupToReady", source = ref("Startup", State), target = ref("Ready", State) ⟫$_{Transition}$,

Continued on next page

RULE 4.4.5 – continued from previous page

```
        ⟨| name = "ReadyToComputeOutputs",
        source = ref("Ready", State),
        target = ref("ComputeOutputs", State),
        trigger = ⟨| _type = TriggerType.SIMPLE,
                     event = ref( SkillExecuteEventName(
                                          amcsk.name) , Event)
|⟩Trigger      |⟩Transition ,
    ⟨| name = "ComputeOutputsToReady",
      source = ref("ComputeOutputs", State),
      target = ref("Ready", State),
      action = ⟨| trigger =
                    ⟨| _type = TriggerType.SIMPLE,
                      event = ref( SkillCompleteEvent(
                                          amcsk.name),
                                   Event ) |⟩Trigger
                  |⟩SendEvent
    |⟩Transition ,
    ⟨| name = "ReadyToCleanup", source = ref("Ready",
State),
      target = ref("Cleanup", State),
      trigger = ⟨| _type = TriggerType.SIMPLE,
                     event = ref( SkillDeactivateEventName(
                                          amcsk.name) , Event)
|⟩Trigger
    |⟩Transition ,
    ⟨| name = "CleanupToDeactivated",
      source = ref("Cleanup", State),
      target = ref("Deactivated", State) |⟩Transition ,
}
∪
```

Continued on next page

RULE 4.4.5 – continued from previous page

```
{ inp: amcsk.inputs •
    ⟪ name = "ReadyReceive" + inp.name,
      source = ref("Ready", State),
      target = ref("Ready", State),
      trigger = ⟪ _type= TriggerType.INPUT,
                  event= ref( SkillEventName(amcsk.name,
                                             inp.name),
                              Event ),
                  parameter= ref( SkillVariableName(
                                             inp.name),
                                  Variable )
                ⟫Trigger
    ⟫Transition }
∪
{ par: amcsk.parameters •
    ⟪ name = "InitialiseReceive" + par.name + Param,
      source = ref("Initialise", State),
      target = ref("Initialise", State),
      trigger = ⟪ _type= TriggerType.INPUT,
                  event= ref( SkillParamEventName(
                                             amcsk.name,
par.name),
                              Event ),
                  parameter= ref( SkillParamVariableName(
                                             par.name),
                                  Variable )
                ⟫Trigger
       action = ⟪ left = ref( SkillParamGuardCondition-
Name(
                                             par.name),
                                  Variable ),
                  right = BoolExp(true) ⟫Assignment
    ⟫Transition }
∪
```

Continued on next page

RULE 4.4.5 – continued from previous page

```
{ par: amcsk.parameters •
    ⟪ name = "DeactivatedReceive" + par.name + Param,
      source = ref("Deactivated", State),
      target = ref("Deactivated", State),
      trigger = ⟪ _type= TriggerType.INPUT,
                  event= ref( SkillParamEventName(
                                                  amcsk.name,
par.name),
                                  Event ),
                  parameter= ref( SkillParamVariableName(
                                                  par.name),
                                  Variable )
                  ⟫Trigger
    ⟫Transition }
∪
{ par: amcsk.parameters •
    ⟪ name = "ReadyReceive" + par.name +
                                          SkillParamPostfix(),
      source = ref("Ready", State),
      target = ref("Ready", State),
      trigger = ⟪ _type= TriggerType.INPUT,
                  event= ref( SkillParamEventName(
                                                  amcsk.name,
par.name),
                                  Event ),
                  parameter= ref( SkillParamVariableName(
                                                  par.name),
                                  Variable )
                  ⟫Trigger
    ⟫Transition }
```

Continued on next page

RULE 4.4.5 – continued from previous page

	where
	#amcsk.parameters = 1 ⇒
	initGuardCondition =
	⟪ ref= ref(SkillParamGuardConditionName(
	(μ p: amcsk.parameters • p.name)),
	Variable) ⟫ *RefExp*
	#amcsk.parameters > 1 ⇒
	initGuardCondition = AndExp(
	seq({ par: amcsk.parameters •
	ref(SkillParamGuardConditionName(par),
	Variable) })
	)

To support the use of skills' priority for the resolution of conflicts over resources, the event type for a skill's outputs are a product type that includes a nat type in addition to the type of the output.

The variables that specify **rccsk.variableList** are used during the operation of the machine for storing the values received by a skills inputs; variables with a matching name are defined for each input and parameter. For convenience of specifying a skill's custom user behaviour in the RoboChart model, a variable for each of the outputs is similarly defined. The last category of variable defined is a Boolean type for recording the initialisation of Parameters; their name is given by the SkillParamter-GuardConditionName function taking the name of the parameter.

The nodes that specify **rccsk.nodes** are the states and junctions of the machine. There is an initial junction that is compulsory for all machines and six states a C-Skill has according to the reactive skills pattern: Initialise, Deactivated, Startup, Ready, Compute, Outputs, and Cleanup.

The transitions that specify **rccsk.transitions** make up the remainder of the rule's definition. They give the conditions that must occur for the transition to be taken and another state entered. The naming of transitions generally follows the convention [source state name]+"To"+[target state name]. The source and target nodes of the transitions are referenced using the ref function. Each transition can also have trigger and condition that describe when the transition takes place and an action that describes

the effect of the transition.

The transition named InitialiseToDeactivated is an example of a transition whose source is the Initialise state, target is the Deactivated state, and has a condition given by initGuardCondition. The initGuardCondition is specified by the where part of the rule and references the boolean parameter initialisation flags for each of a skill's parameters. When there is a single parameter, an expression referring to the unique object is obtained using the definite description μ-notation: $(\mu\, x : \alpha \bullet e)$ where x is the object, α is the set, and e is the expression involving the object x from α . When there are multiple parameters, a generalised conjunction expression is defined so that the InitialiseToDeactivated transition is only taken when all of a skill's parameters have been initialised.

The DSkillToStateMachine Rule 4.4.6 specifies the interfaces, variables, nodes, and transitions that make up the D-Skill's machine. Because D-Skills share many similarities with C-Skills we present the parts of the rule that are unique to D-Skills.

In addition to the skills interface specified by **rcdsm.interfaces** a D-Skill's machine representation has events for communicating sensor values from, and actuation values to, the platform. These make up the machines events, **rcdsm.events**, where for each D-Skill input and output a corresponding platform event is defined by a set comprehension over the union of the skill's inputs and outputs. The name of the platform events is given by the DSkillPlatformEvent name function that takes a name of the input or output and adds the platform prefix.

D-Skills receive sensor data values from the platform and must handle their communication to the skills manager. A D-Skill's machine uses a boolean variable for each received sensor value from the platform to indicate when new values have been received and enable them to be communicated when the skill is next executed. These variables are specified in **rcdsm.variableList** as the set comprehension over a skill's output sensor data amdsk.outputs that defines a boolean variable for each output whose initial value is false. The name of the boolean variables are given by the DSkillSenseGuardConditionName function that takes the name of the output.

Rather than computing values, D-Skills interface sensors and actuators, so instead of a ComputeOutput state a D-Skill machine has two states: one for handling actuation commands and another for handling sensor data. These states are part of the specification of **rcdsm.nodes** and are named HandleActuationCommands and HandleSensorData. The handle HandleActuationCommands state defines an action specified by

sendActuationCommands in the where clause. The action is an entry action that outputs the skill's input values via the corresponding platform events.

RULE 4.4.6:

ⓐ	**Name**	DSkillToStateMachine
ⓑ	**Parameter** *name:type*	amdsk: DSkill
ⓒ	**Result** *name:type*	rcdsm: StateMachine
ⓔ	**Definition**	rcdsm.name = amdsk.name rcdsm.interfaces = ref(SkillInterfaceName(amdsk.name), Interface) ∪ ref(DSkillActuationInterface(amdsk.name), Interface) ∪ ref(DSkillSensingInterface(amdsk.name) , Interface) , rcdsm.events = { io: (amdsk.inputs ∪ amdsk.outputs) • ⟨ name= DSkillPlatformEventName(io.name) type= io.type ⟩$_{Event}$ } , rcdsm.variableList = { ⟨ modifier = VariableModifier::VAR, vars= { par: amdsk.parameters • ⟨ name = SkillParamVariableName(par.name), type = par.type ⟩$_{Variable}$ } ∪ { sdt: amdsk.outputs • ⟨ name = sdt.name, type = sdt.type ⟩$_{Variable}$ } ∪

Continued on next page

RULE 4.4.6 – continued from previous page

```
                    { act: amdsk.inputs •
                            ⟪ name = act.name,
                              type = act.type ⟫Variable }
                  ∪
                  { par: amdsk.parameters •
                            ⟪ name =
                              SkillParamGuardConditionName(
                                                    par.name),
                              type = ref(boolean, PrmitiveType),
                              value = BoolExp(false)
                             ⟫Variable },
                  ∪
                  { sdt: amdsk.outputs •
                            ⟪ name =
                                  DSkillSenseGuardCondition-
Name(
                                                    sdt.name),
                              type = ref(boolean, PrmitiveType),
                              value = BoolExp(false)
                    ⟫Variable }

      ⟫VariableList }

rcdsm.nodes = {
    ⟪ name = "Initial" ⟫Initial ,

    ⟪ name = "Initialise" ⟫State ,

    ⟪ name = "Deactivated" ⟫State ,

    ⟪ name = "Startup" ⟫State ,

    ⟪ name = "Ready" ⟫State ,

    ⟪ name = "HandleActuationCommands",
      actions = { ⟪ action= sendActuationCommands
                    ⟫EntryAction }
    ⟫State,

    ⟪ name = "HandleSensorData" ⟫State,

    ⟪ name = "Cleanup" ⟫State }                          109
```

Continued on next page

RULE 4.4.6 – continued from previous page

```
rcdsm.transitions = {
    ⟨| name = "InitialToInitialise", source = ref("Initial",
Initial),
      target = ref(Initialise, State) |⟩Transition ,
    ⟨| name = "InitialiseToDeactivated",
      source = ref("Initialise", State),
      target = ref("Deactivated", State),
      condition = initGuardCondition |⟩Transition ,
    ⟨| name = "DeactivatedToStartup",
      source = ref("Deactivated", State),
      target = ref("Startup", State),
      trigger = ⟨| _type = TriggerType.SIMPLE,
                   event = ref( SkillActivateEventName(
                                        amdsk.name) , Event)
|⟩Trigger      |⟩Transition ,
    ⟨| name = "StartupToReady", source = ref("Startup",
State),
      target = ref("Ready", State) |⟩Transition ,
    ⟨| name = "ReadyToHandleActuationCommands",
      source = ref("Ready", State),
      target = ref("HandleActuationCommands", State),
      trigger = ⟨| _type = TriggerType.SIMPLE,
                   event = ref( SkillExecuteEventName(
                                        amdsk.name) , Event)
|⟩Trigger      |⟩Transition ,
    ⟨| name = "HandleActuationCommandsToHandle-
SensorData",
      source = ref("HandleActuationCommands", State),
      target = ref("HandleSensorData", State),
    |⟩Transition ,
```

Continued on next page

RULE 4.4.6 – continued from previous page

```
    ⟪ name = "HandleSensorDataToReady",
      source = ref("HandleSensorData", State),
      target = ref("Ready", State),
      condition = sensorDataGuardCondition
       action = ⟪ trigger =
                    ⟪ _type = TriggerType.SIMPLE,
                      event = ref( SkillCompleteEvent(
                                          amdsk.name),
                                  Event ) ⟫Trigger
                  ⟫SendEvent
    ⟫Transition ,
    ⟪ name = "ReadyToCleanup", source = ref("Ready",
State),
      target = ref("Cleanup", State),
      trigger = ⟪ _type = TriggerType.SIMPLE,
                  event = ref( SkillDeactivateEventName(
                                      amdsk.name) , Event)
⟫Trigger
    ⟫Transition ,
    ⟪ name = "CleanupToDeactivated",
      source = ref("Cleanup", State),
      target = ref("Deactivated", State) ⟫Transition ,
}
∪
{ sdt: amdsk.outputs •
    ⟪ name = "HandleSensorDataReceive" + sdt.name,
      source = ref("HandleSensorData", State),
      target = ref("HandleSensorData", State),
      trigger = ⟪ _type= TriggerType.INPUT,
                  event= ref( DSkillPlatformEventName(
                                      sdt.name),
                              Event ),
                  parameter= ref( SkillVariableName(
                                          sdt.name),
                                  Variable )
                ⟫Trigger
```

Continued on next page

RULE 4.4.6 – continued from previous page

```
        action =
            ⟪ statements =
                ⟨ ⟪ left = ref( DSkillSenseGuardCondition-
Name(
                                                    sdt.name),
                                        Variable ),

                    right = BoolExp(true) ⟫Assignment ,
                  ⟪ _type= TriggerType.OUTPUT,
                    event= ref( SkillEventName(
                          amdsk.name,
                                                      sdt.name),
                                    Event ),
                    parameter= ref( SkillVariableName(
                                                      sdt.name),
                                        Variable )
                  ⟫Trigger ⟩
            ⟫SeqStatement
    ⟫Transition }
∪
{ act: amdsk.inputs •
    ⟪ name = "ReadyReceive" + act.name,
      source = ref("Ready", State),
      target = ref("Ready", State),
      trigger = ⟪ _type= TriggerType.INPUT,
                  event= ref( DSkillPlatformEventName(
                                        act.name),
                                    Event ),
                  parameter= ref( SkillVariableName(
                                                      act.name),
                                        Variable )
                  ⟫Trigger
    ⟫Transition }
∪
```

Continued on next page

RULE 4.4.6 – continued from previous page

	{ par: amdsk.parameters • ⟪ name = "InitialiseReceive" + par.name + Param, source = ref("Initialise", State), target = ref("Initialise", State), trigger = ⟪ _type= TriggerType.INPUT, event= ref(SkillParamEventName(amdsk.name, par.name), Event), parameter= ref(SkillParamVariableName(par.name), Variable) ⟫$_{Trigger}$ action = ⟪ left = ref(SkillParamGuardCondition- Name(par.name), Variable), right = BoolExp(true) ⟫$_{Assignment}$ ⟫$_{Transition}$ } ∪

Continued on next page

RULE 4.4.6 – continued from previous page

	{ par: amdsk.parameters • ⟨| name = "DeactivatedReceive" + par.name + Param, source = ref("Deactivated", State), target = ref("Deactivated", State), trigger = ⟨| _type= TriggerType.INPUT, event= ref(SkillParamEventName(amdsk.name, par.name), Event), parameter= ref(SkillParamVariableName(par.name), Variable) |⟩*Trigger* |⟩*Transition* } ∪ { par: amdsk.parameters • ⟨| name = "ReadyReceive" + par.name + SkillParamPostfix(), source = ref("Ready", State), target = ref("Ready", State), trigger = ⟨| _type= TriggerType.INPUT, event= ref(SkillParamEventName(amdsk.name, par.name), Event), parameter= ref(SkillParamVariableName(par.name), Variable) |⟩*Trigger* |⟩*Transition* } --- ***where***

Continued on next page

RULE 4.4.6 – continued from previous page

```
#amdsk.inputs = 1 ⇒
   sendActuationCommands =
      ⟪ trigger=
         ⟪ _type= TriggerType.OUTPUT,
           event= ref( DSkillPlatformEventName(
            (μ s: amdsk.inputs • s.name) ),
                    Variable )
           value =
            ⟪ values =
                    {  ref(SkillVariableName(sdt.name),
Variable) }
                ∪
                { ref( SkillParamVariableName(
                        PriorityParamName() ) , Variable)
}
              ⟫TupleExp

         ⟫Trigger
      ⟫SendEvent
#amdsk.inputs > 1 ⇒
   sendActuationCommands = SeqStatement( seq(
     { act: amdsk.inputs •
      ⟪ trigger =
         ⟪ _type= TriggerType.OUTPUT,
           event= ref( DSkillPlatformEventName(
                                          act.name),
                                Event ),
          parameter= ref( SkillVariableName(act.name),
                                Variable )
           ⟫Trigger
      ⟫SendEvent
     }   )   )
#amdsk.parameters = 1 ⇒
   initGuardCondition =
      ⟪ ref= ref(SkillParamGuardConditionName(
                    (μ p: amdsk.parameters • p.name)
),
                    Variable) ⟫RefExp
```


Continued on next page

RULE 4.4.6 – continued from previous page

	#amdsk.parameters > 1 ⇒
	initGuardCondition = AndExp(
	seq({ par: amdsk.parameters •
	ref(SkillParamGuardConditionName(par),
	Variable) })
	)
	
	#amdsk.outputs = 1 ⇒
	sensorDataGuardCondition =
	⦇ ref= ref(DSkillSenseGuardConditionName(
	(μ s: amdsk.outputs • s.name)),
	Variable) ⦈ *RefExp*
	#amdsk.outputs > 1 ⇒
	sensorDataGuardCondition = AndExp(
	seq({ sdt: amdsk.outputs •
	ref(DSkillSenseGuardConditionName(sdt),
	Variable) })
	)

Finally the transitions specified by **amdsk.transitions** give the conditions that must occur before another state is entered.

The last node that makes up the controller of the RoboChart model for a reactive skills model is the skills manager machine. The SkillsManagerToStateMachine Rule 4.4.7 specifies the interfaces, variables, clocks, nodes, and transitions that make up the skills manager machine. The name of the machine, **rcskm.name**, equals the result of the function SkillsManagerName which returns the string "SkillsManager".

The interfaces **rcskm.interfaces** that define the machine's events are referenced by the ref function taking the name of the interface and type as parameters. There is an interface for controlling skills and an interface for each skill that the skills manager uses to coordinate skill communication. The referenced interfaces are specified as the union between the skills manager's given by the SkillsManagerToInterfaceName function and the skills' given by the set comprehension over each skill **skls** for SkillInterfaceName that takes the skills name as a parameter.

The referenced skills manager interface is the result of applying the

RULE 4.4.7:

ⓐ	**Name**	SkillsManagerToStateMachine
ⓑ	**Parameter** *name:type*	amskm: SkillsManager, skls: Set(Skills)
ⓒ	**Result** *name:type*	rcskm: StateMachine

ⓔ	**Definition**	rcskm.name= SkillsManagerName()
		
		rcskm.interfaces = ref(SkillsManagerInterfaceName(), Interface) ∪ { s: skls • ref(SkillInterfaceName(s.name), Interface) }
		
		rcskm.variableList = {
		⟨| modifier = VariableModifier::VAR,
		vars = {
		SkillsManagerBaseVariables()
		∪
		SkillsManagerNewSkillInputFlagVariables(skls)
		∪
		SkillsManagerParameterVariables(skls)
		∪
		SkillsManagerInputOutputVariables(skls)
		}
		|⟩$_{VariableList}$,
		⟨| modifier = VariableModifier::CONST,
		vars = SkillsManagerConstants()
		|⟩$_{VariableList}$
		}
		rcskm.clocks = { ⟨| name = SkillsManagerClockRequestHandlingName() |⟩$_{Clock}$ }

Continued on next page

RULE 4.4.7 – continued from previous page

	rcskm.nodes = SkillsManagerPatternStates(skls, amskm.interskillconnections, amskm.stateMonitors) ∪ { ⟨| name = SkillsManagerJunctionEmptyCycleSkillsName() |⟩$_{Junction}$ } rcskm.transitions = SkillsManagerPatternTransitions(skls)

ReactiveSkillsLayerToInterface Rule B.2.3. The rule specifies the interface as the union of the result of applying two further rules one for the inputs SkillsManagerInterfaceInputs Rule B.2.4 and one for the outputs SkillsManagerInterfaceOutputs Rule B.2.5 that both take the reactive skill layer as a parameter.

The skills manager control interface inputs include events for requesting: the activation and deactivation of skills, the currently active skills, initiation and stopping of monitoring conditions, setting and getting of parameters, and getting the latest value of a skill's output. The skill activation and deactivation events have a Skills enumeration value to indicate the skill to be modified. Similarly the monitoring condition initiation and stopping events have an Events enumeration value to indicate the conditions the skills manager will monitor. The parameter set request events have a value that is the new value to replace the current parameter value with.

The skills manager control interface outputs include events for replying to requests of: the currently active skills, each of a skill's parameter value, and the latest value for each of a skills output. Additionally, the interface has an event for notifying when one of the conditions being monitored occurs. The active skills reply has a value of type set of skills enumerations that indicates all of the currently active skills. The parameter and value reply event each have the corresponding type of the skill's parameter or output. The monitored event notification has an Events enumeration type that occurs when one of the actively monitored conditions becomes true.

The other referenced skills' interfaces have been previously discussed

with the skill rules. They are specified by the SkillToInterfaceRule Rule B.2.6. The skills manager uses the skills interfaces to coordinate communication between skill and complete control requests from the dependant layer.

The variables that specify **rcskm.variableList** are used during operation of the machine for storing the values received by skill control requests and for holding skills parameter, input, and output values for communication. The variables are defined as the generalised union over four functions that each return a set of variables. The functions group together variable that are loosely related by their usage.

The variables for the base operation of the machine are the result of the SkillsManagerBaseVariables Function B.3.2. The function defines variables of set types for Skills and Events that are used to keep track of the currently active skills and monitored conditions, respectively. Also defined, are variables of sequence types for recording the skills that are to be executed in the next cycle of the skills managers execution.

The variables for tracking the occurrence of new inputs are the result of the SkillsManagerNewSkillInputFlagVariables Function B.3.3. The function defines variables of boolean type that is set to true when a new input values of a skill's input has been received. These variables are named "new" + [skill name] + [skill input/output name].

The variables for received parameter values are the result of the SkillsManagerParameterVariables Function B.3.4. The function defines variables for each skill parameter with a matching type and named [skill name] + [parameter name].

The variables for received skill input and output values are the result of SkillsManagerInputOutputVariables Function B.3.5. The function defines variables for each skill input and output with a matching type and named "current" + [skill name] + [input/output name].

The constants that specify **rcskm.variableList** are used at the start of the machine's operation to initialise private variables to known starting values and are the result of SkillsManagerConstants Function B.3.6. The function defines two constants: the first defines the handling delay given to handle requests before executing skills named REQUEST_HANDLING_DELAY. The second is a sequence of elements whose type is Skills enumeration, that is, a list of the asynchronous skills named ASYNCHRONOUS_SKILLS.

The clock that specifies **rcskm.clocks** is used to control the time spent handling requests before the skills manager begins executing skills. Its name is the result of the function SkillsManagerClockRequestHandling-

Name Function B.3.10 and is "requestHandling".

The nodes that specify **rcskm.nodes** are the states and junctions of the machine defined as the union of the result of the SkillsManagerPatternStates Function B.3.7 and a junction. The function takes three parameters for specifying the behaviour of the states: a System's skills and the SkillsManager's interskillconnections and monitors. The six states the function defines are Initialise, HandleRequests, DoNextSkill, ExecutingSkills, UpdateRecord, and CheckMonitors.

The junction is named using the result of the SkillsManagerJunctionEmptyCycleSkillsName Function B.3.8, that is, "JunctionEmptyCycleSkills". It is used to allow the HandleRequests state to be re-entered on the condition that the current cycle skills is empty because there would be no skills to execute.

The transitions that specify **rcskm.transitions** make up the remainder of the rule's definition and are the result of the function SkillsManagerPatternTransitions Function B.3.9. The skills manager transition naming and structure follows the skill transitions previously described. An overview of the transitions can be seen in Figure 4.6 for all skills managers.

The rules that have been specified here are supported by a tool that automates the transformation of RoboArch models, as shown in the next Chapter.

4.5 Final Considerations

We have expanded the RoboArch notation to include pattern definitions and presented two patterns: reactive skills and subsumption. For both patterns, we have demonstrated the RoboArch notation for the mail delivery example. For the reactive skills pattern a further metamodel and well-formedness conditions have been presented. We concluded the chapter by presenting the transformation rules of RoboArch to RoboChart that give RoboArch its semantics. The presented rules include the top-level rule that applies further pattern specific rules and the complete set of rules for the reactive skills pattern. Finally, as part of the reactive skills rules presentation we saw the generated RoboChart model for the reactive skills mail delivery example.

In the next chapter we evaluate RoboArch and present a tool that automates the transformation of RoboArch models. Finally, we present a case study including its verification using the reactive skills pattern for a robot that explores its environment.

5 Evaluating RoboArch

Chapters 3 and 4 introduced RoboArch and its notation, metamodel, and well-formedness conditions, in this chapter we evaluate RoboArch through a tool implementation and a case study.

There are three objectives to our evaluation of RoboArch: can architectural patterns be modelled using the notation, do generated RoboChart models correspond to the semantics defined by the rules, and can the resulting RoboChart models be used to verify properties of architectural patterns.

For the evaluation a tool was developed developed that implements the RoboArch metamodel and well-formedness conditions and automates the generation of its RoboChart semantics. The tool enables experience of using RoboArch to be accumulated and the automation of transformations that facilitate further study. Section 5.1 presents the tool's design and architecture including measures such as unit testing to ensure its correctness and validate the transformation rules.

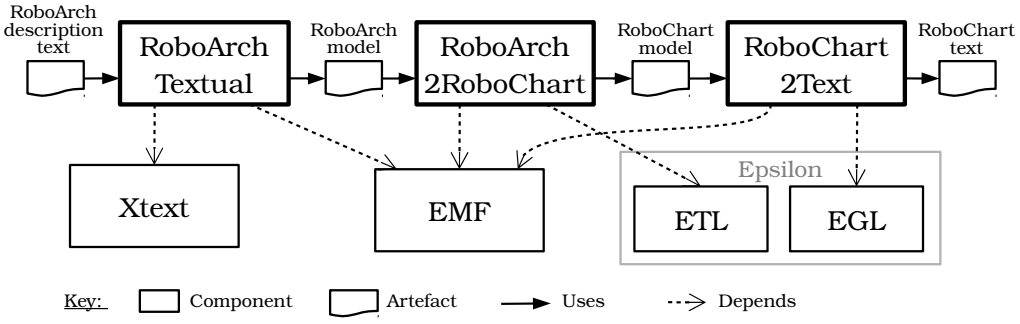
The case study of a robot that explores its environment while avoiding obstacles including the verification of its properties using the tool is presented in Section 5.2. To aid repeatability both the unit tests and examples are included in the tool's package which can be found on the following website[1]. We conclude in Section 5.3 with some final considerations.

5.1 Tool

Our tool has three components. One that is responsible for the RoboArch language and its textual notation including its grammar and parser, another for the model-to-model transformation from RoboArch to RoboChart, and the third for the model to text generation of RoboChart models. These components and their relationships to one another are shown in Figure 5.1 as RoboArchTextual, RoboArch2RoboChart, and RoboChart2Text.

[1] https://ghithub.com/UoY-RoboStar/roboarch-textual

Figure 5.1: Components of RoboArch and their technological dependencies.



This structure promotes a loose coupling between the main elements of the tool and allows access to the models and text artefacts for further processing, verification, or analysis. Furthermore, modularity of well defined components improves extensibility; for instance, a module that generates another notation from a RoboArch model can be created to support additional verification methods with no changes to the other modules. We now discuss the selection of key technologies for each of the components indicated by the dependencies in Figure 5.1.

There are two main approaches used for the development of DSLs. Internal DSLs that are defined in terms of a general purpose language and external DSLs that define their own language. Recently, language workbenches have emerged such as Xtext [95] and Meta Programming System (MPS) [96] that provide specialised tools and frameworks to support development of external DSLs.

For the RoboArchTextual component of the tool, Xtext was selected because of its use of model-based parse trees that allow the direct application of model-based techniques. The choice of Xtext means that RoboArch falls under the external category of DSL. Collective experience of the RoboStar[2] research group on Xtext's use in RoboTool and reuse of a subset of the RoboChart metamodel also motivated the selection of Xtext.

Xtext closely integrates with the well established eclipse IDE. Eclipse itself was first released in 2001 as an open source language-neutral platform for software application development [97]. Throughout this time support for programming languages via plug-ins on top of the original

[2] https://robostar.cs.york.ac.uk/

Java have been added; for example, there are plug-ins C, C++, Haskell, and Python. This means that there is substantial existing infrastructure that can be utilised for the development of the RoboArch tool.

The key parts that have been implemented in the RoboArchTextual component of the tool are metamodel, grammar, and checks for scoping and validation. The metamodel presented in Chapters 3 and 4 has been implemented using the EMF [98]. The resulting implementation is compiled into an executable library form; the classes it contains are used throughout the RoboArchTextual module implementation.

The grammar for the textual notation presented in Chapters 3 and 4 is defined using Xtext's grammar language which closely resembles Extended Backus-Naur Form (EBNF) [99]. The grammar establishes the relationship between the metamodel and text, and Xtext uses it to generate the parser.

As we have previously discussed the metamodel alone is not enough to ensure only meaningful models are created. The well-formedness conditions are implemented as scoping and validation checks and ensure that only meaningful models can be created. When a condition is violated, for example, connecting an input to another input, an error is raised to notify the user of the issue and that it needs to be corrected before any outputs are generated.

The RoboArch2RoboChart component contains the model-to-model transformation rules. Model-to-model transformations can be implemented either using a general purpose language such Java or using a DSL, for example, ETL [100], ATL [101], and QVT [102]. For our transformation from RoboArch to RoboChart, we selected the DSL approach because the structure of a DSL allows the implemented rules to have a close correspondence to the rules we defined, therefore, assisting in their implementation and validation of the rule definitions. As explained and illustrated in Section 5.1.1, our implementation is in direct correspondence with our specification of the rules in Chapters 3 and 4, and in Appendix B.

Epsilon Transformation Language (ETL) was selected because it is part of a wider family of languages like EUnit [103] that provide support for unit testing during Model Driven Development (MDD).

The last component of Figure 5.1 generates RoboChart text from an instance of the RoboChart metamodel. It is the least significant component technologically and uses a well known template based text generation approach. The templating language selected is Epsilon Generation Language (EGL) to ease integration because other Epsilon languages are

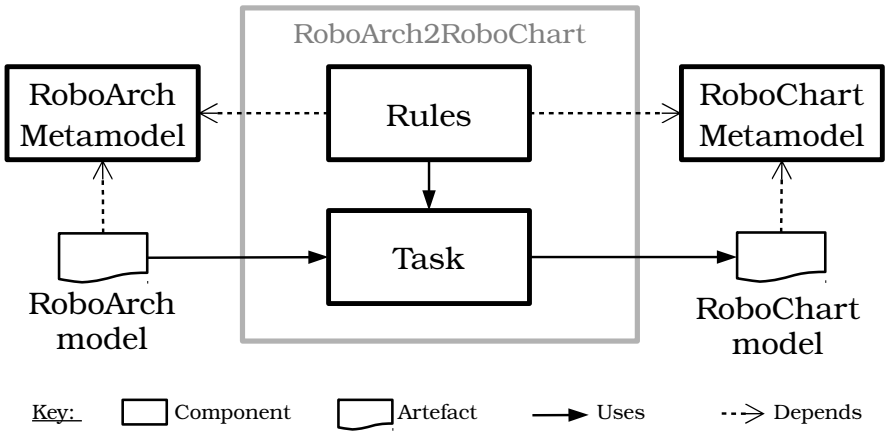
being used for the transformations in the RoboArch2RoboChart component.

Now we have introduced the tool and its modules, we will examine the rule implementation because of the pivotal role they have in giving RoboArch semantics.

5.1.1 Rule Implementation

As we have seen the RoboArch2RoboChart component is responsible for transforming a source RoboArch model, which has been parsed from a RoboArch description, to a target RoboChart model. This is achieved using model transformation rules implemented in ETL. Figure 5.2 shows the workflow of the transformations.

Figure 5.2: RoboArch to RoboChart rule transformation workflow.



The Rules are made up of the ETL implementation of the RoboArch rules presented in Chapters 3 and 4 they relate elements of the RoboArch metamodel to elements in the RoboChart metamodel. This means that the Rules have clear dependence on the source and target metamodels. The Task coordinates and executes the model transformation; it is responsible for loading the source model, initialising the target model, and executing the Rules that populate the target model. The Task is realised as an Ant [104] build build script. Next we present the ETL implementation for the SystemToRCModule rule.

Listings 5.1 to 5.3 show the rule implementation for the SystemToRCModule Rule 3.2.1 that was presented in Chapter 3. There are five

parts to RoboArch rules: name, parameters, result, precondition, and definition. The start of a rule and its name is declared using the `rule` clause. For our example, Line 1 of Listing 5.1 declares a rule with the name `SystemToRCModule`.

Listing 5.1: Rule implementation for SystemToRCModule: declaration and where.

```
1  rule SystemToRCModule
2      transform amsys: RoboArch!System
3      to rcsysPkg: RoboChart!RCPackage,
4          rcdefsPkg: RoboChart!RCPackage{
5
6      //Where
7      var cLayer = amsys.layers.select( lyr |
8                      lyr.isTypeOf(RoboArch!ControlLayer) );
9
10     if ( (amsys.robot.operations.size() > 0) or
11          (amsys.robot.interfaces.size() > 0) or
12          (amsys.robot.pinterfaces.size() > 0) or
13          (amsys.robot.variableList.size() > 0) ) {
14
15         var roboticPlatform = amsys.robot.equivalent();
16
17     } else {
18         var roboticPlatform = roboticPlatform.add(
19                 ControlLayerToRoboticPlatform(cLayer));
20     }
```

The second part of a rule is its parameters that specify the input elements of the RoboArch model to be transformed, and corresponds to the `transform` clause in the implementation. Line 2 specifies the System of a RoboArch model as the target class and is assigned the name `amsys`. Because the transformation language selected supports multiple metamodels, in the implemented rules the metamodel has to be specified when referring to each class from a metamodel in the format *[metamodel]![class]*.

The third part to a rule is its result that specifies the output elements of the RoboChart model, and corresponds to the `to` clause in the implementation. Lines 3 and 4 specify the result of the transformation as being two RoboChart packages `rcsysPkg` and `rcdefsPkg`. The use of packages

is one difference between the rules as defined and their implementation. Because a RoboChart package's only function is to contain elements of a RoboChart model, and they do not affect a model's meaning they are not specified in our rule definitions. However, packages need to be defined as part of the implementation to contain the results of the application of the rule. The `rcsysPkg` contains the module **rcmod** of the rule, and the `rcdefsPkg` contains the definitions, functions, and interfaces **rcdefs**, **rcfuns**, and **rcifcs**.

The fourth part of a rule is the precondition; the SystemToRCmodule rule does not have one, however one can be specified in the implementation using a `guard` clause in the format *guard: [boolean expression]*.

The final part of a rule is its definition that specifies the result in terms of the input parameters. An important aspect of model-to-model transformation is resolving target model element, or elements, for a given source model element. This resolution can involve creating any necessary child model elements and therefore calling additional transformation rules. Furthermore, what has already been transformed has to tracked so that duplicates in the target model are not created. For this purpose ETL provides the `equivalent` operation that when called for a given source element the target model elements are returned.

So far the structure of an implemented rule closely follows the rules definition, however a difference in the implementation is that terms introduced in a rule definition's **where** section have to be declared first. This is because the **where** terms are implemented as variables that must be declared before they can be used, compared to, the rule where they are specified at the end of the rule definition. In our example the **where** definitions are declared from Lines 7 to 20.

The rule's **where** section specifies two terms **cLayer** and **roboticPlatform**. The **cLayer** is a set comprehension with a predicate to select layers of type control layer from **amsys.layers**; Line 7 of our example implements this using the `select` operation of collections. The **roboticPlatform** term is defined as a conditional statement that can be directly implemented with only minor syntactic differences; see Lines 10 to 20. The most significant difference is the use of the `size` operation of a collection compared to the cardinality operator #.

The remaining part of a rule's definition are the results **rcdefs**, **rcfuns**, **rcifs**, and **rcmod**; their implementation is shown in Listing 5.2. The structure of the rule and the order of definition of the results is mirrored in the implementation. A variable is used to hold the result with a matching name until it is later added to the returned package. For our

example the rule definition's **rcdefs** implementation on Lines 22 to 26 declares a variable `rcdefs`.

Listing 5.2: Rule implementation for SystemToRCModule: definition.

```
21    //Definitions
22    var rcdefs = amsys.definitions.equivalent();
23
24    for (lyr: RoboArch!Layer in amsys.layers){
25        rcdefs.addAll( LayerToTypes(lyr) );
26    }
27
28    //Functions
29    var rcfuns = amsys.functions.equivalent();
30
31    for (lyr: RoboArch!Layer in amsys.layers){
32        rcfuns.addAll( LayerToFunctions(lyr) );
33    }
34
35    //Interfaces
36    var rcifs = amsys.interfaces.equivalent();
37
38    for (lyr: RoboArch!Layer in amsys.layers){
39        rcifs.addAll( LayerToInterface(lyr) );
40    }
41
42    //Module
43    var rcmod = new RoboChart!RCModule;
44
45    rcmod.name = amsys.name;
46
47    rcmod.nodes = LayersToControllers( amsys.layers );
48    rcmod.nodes.addAll(roboticPlatform);
49
50    rcmod.connections = amsys.connections.equivalent();
```

The variable is assigned the definitions as specified in the RoboArch description. Because RoboArch reuses RoboChart's types, the definitions are directly assigned in the rule. However, the rule implementation in ETL uses the object oriented paradigm where both the source RoboArch and

target RoboChart models are objects. This means that the implemented rules must clone the definitions in the RoboArch model and assign them to the RoboChart model so that there are no references between the source and target models. The cloning is achieved using the ETL `equivalent` function that returns the cloned RoboChart elements.

The remaining part of the implementation of the definition of **rcdefs** is the union of the set comprehensions that applies the **LayersToTypes** rule for every layer **lyr** in **amsys.lyr**. This is implemented as a `for` loop calling the `LayerToTypes` function.

ETL Rules cannot be directly called; however, we use a wrapper function that calls `equivalent` on the provided metamodel object with the rule name for the wrapper function passed in as a parameter.

The `addAll` operation of the collection type combines the collections in the `rcdefs` variable similar to the union in the rule.

The implementation of the specifications of **rcfuns** and **rcifs** in Lines 29 to 40 follow a very similar pattern to the implementation of the **rcdefs** definition but for type definitions and functions.

The **rcmod** result defined by the rule introduces the definition of a new element in the RoboChart model. Line 43 of the implementation uses the `new` keyword to create the `RCModule` that is then populated mirroring the structure and order of the rule.

The final part of our example rule implementation is shown in Listing 5.3. This part of the implementation is not present in the rule because it deals with populating the packages with the results produced earlier in the rule's definition. Lines 52 to 54 remove empty sets from `rcdefs`, `rcifs`, and `rcfuns`.

Listing 5.3: Rule implementation for SystemToRCModule: package.

```
51      //Populate types package
52      rcdefs.remove(null);
53      rcifs.remove(null);
54      rcfuns.remove(null);
55
56      rcdefsPkg.name = DefinitionsName();
57      rcdefsPkg.types = rcdefs;
58      rcdefsPkg.interfaces = rcifs;
59      rcdefsPkg.functions = rcfuns;
60
61      //Populate system package
62      rcsysPkg.name = amsys.name;
63      rcsysPkg.imports.add(
64          getPackageImport(DefinitionsName())
65      );
66
67      rcsysPkg.imports.add(
68          getPackageImport(PlatformName())
69      );
70
71      for(lyr: RoboArch!Layer in amsys.layers){
72          rcsysPkg.imports.add(
73              getPackageImport(
74                  lyr.name.toLowerCase().firstToUpperCase()
75              )
76          );
77      }
78
79      rcsysPkg.modules.add(rcmod);
80  }
```

Lines 56 to 59 populate the package types, interfaces, and function by direct assignment of `rcdefs`, `rcifs`, and `rcfuns` to the respective package attribute. Lines 62 to 79 populate the system's package by adding the module and specifying the package imports which the system package depends.

In the next section we, outline the test strategy used to verify the tool.

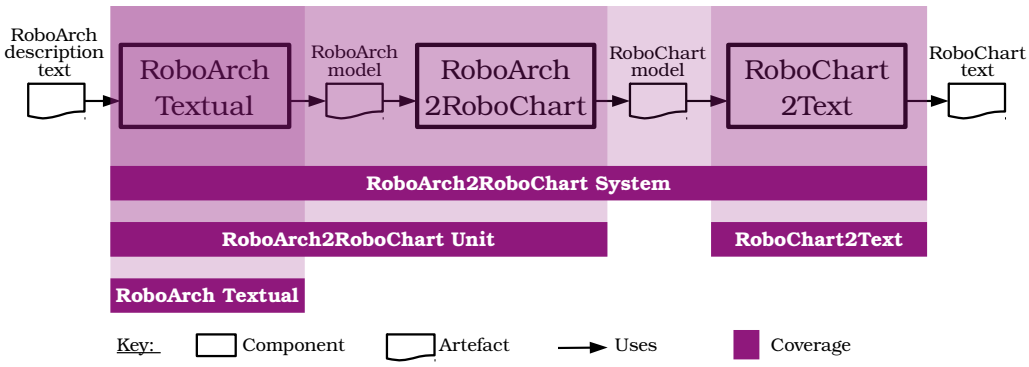
5.1.2 Verification

To demonstrate that our tool meets its specifications (RoboArch metamodel, well-formedness conditions, and translation to RoboChart rules) it is im-

portant that it is adequately tested. This section provides the details of the testing strategy used in the development.

Figure 5.3 shows the components of the tool and the scope of the tests described by the strategy. Each component is a step to get to a RoboChart textual model representation from a given RoboArch description of a system. The first step creates a model by parsing and validating the input

Figure 5.3: Component test coverage.



text. The second step, RoboArch2RoboChart, creates a RoboChart model by applying transformation rules to the input RoboArch model. The third step, RoboChart2Text, creates RoboChart text by using templates and the input RoboChart model.

RoboArchTextual

The aspects that are tested for the first component are the parsing and well-formedness conditions to verify features of the textual language. Unit tests cover the parsing of each class in the RoboArch metamodel, and, for every well-formedness condition, which when violated raises the appropriate error message.

In total, 20 tests have been implemented. They use JUnit supported by Xtext convenience functions that facilitate the parsing of text and they can be found at the following website[3].

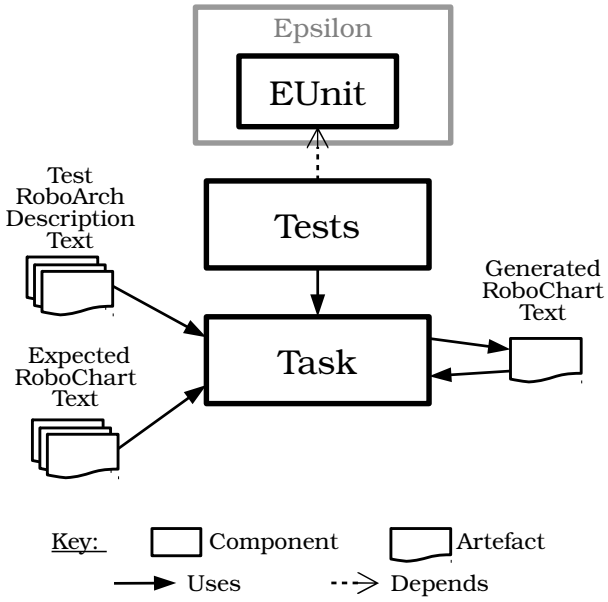
[3] https://github.com/UoY-RoboStar/roboarch-textual/tree/thesis/circus.robocalc.roboarch.textual.tests/src/circus/robocalc/roboarch/textual/tests

RoboArch2RoboChart

The transformation rules of RoboArch2RoboChart module are tested as system tests and unit tests. Wherever possible rules are tested as system tests in order to exercise the integrated toolchain. Rules producing RoboChart model fragments that cannot be tested by system tests are unit tested instead. The combination of system tests and unit test ensures the coverage of every rule.

Figure 5.4 shows the workflow of a system test. The Tests are defined using Epsilon EUnit [103] that aids model-based testing by providing a set of assertions for comparing models and support for model reuse against different tests. The Task coordinates and executes the Tests and is realised

Figure 5.4: Workflow of a system test.



as an Ant [104] build script; for each test it is responsible for parsing and loading the required models from the RoboArch description and the corresponding expected RoboChart text. After loading the models, the Task runs the test that includes executing the transformation rules and text generators of the RoboArch2RoboChart and RoboChart2Text components to create the RoboChart text for the test. Finally, the Test parses the generated RoboChart text and loads it as a model for evaluation of the test's assertion.

The unit tests of the RoboArch2RoboChart component follow a similar workflow to the one shown in Figure 5.4, but rather than the generated RoboChart text being parsed for evaluating the test's assertion, the model output by RoboArch2RoboChart is used for the evaluation instead.

There are two kinds of goals a test can have: verify that valid RoboChart models are generated for given RoboArch text and verify that domain knowledge concepts of robotics architectures are represented as expected in the RoboChart model.

The validity tests can be implemented as a system or unit test. The RoboChart model validity test as a system test takes a RoboArch model as input. The test first runs the RoboArch parser and validator for the input RoboArch text to obtain the RoboArch instance of the metamodel. Next, the test runs a RoboArch2RoboChart transformation rule on the result to generate a RoboChart instance of its metamodel. Finally, the test runs the RoboArch2Text generator to output the RoboChart text for the input RoboArch model. Using the RoboChart parser and validator the test verifies that the generated RoboChart text is free from errors if a complete model is expected as the rule's output, or for the presence of expected errors if the rule produces a partial model.

The RoboChart model validity as a unit test takes a RoboArch model and the expected RoboChart model fragment as its inputs. The test follows the first two steps of the system test running the RoboArch parser and validator for the input RoboArch text, followed by, the RoboArch2RoboChart transformation rule to output a RoboChart model. The test verifies that the generated RoboChart model matches the expected RoboChart model fragment.

The domain unit tests take a RoboArch text and run the parser and validator, followed by the RoboArch2RoboChart transformation rule for the target feature that is being tested. The test checks the output RoboChart model objects using assertions that capture the expected domain knowledge.

In total, 5 validity and 19 domain tests have been implemented. They can be found at the following website[4].

RoboChart2Text

The code templates of the RoboChart2Text are tested as unit tests, a test for each element of the RoboChart metamodel. There is one kind of

[4] https://github.com/UoY-RoboStar/roboarch2chart-epsilon/tree/thesis/circus.robocalc.roboarch2chart.tests/eunit

test goal: for a given input RoboChart model the expected RoboChart text is generated. These tests have a RoboChart model and the expected RoboChart text as inputs. A test runs the RoboChart2Text generator for the input RoboChart model and outputs the generated RoboChart text. The test verifies that the generated RoboChart text matches the expected RoboChart text.

The text comparison used is a relaxed comparison that ignores white-space characters to reduce the fragility of the code template tests. This means that some errors in the text generated by RoboChart2Text will not be detected. Validation tests that check generated RoboChart text using the RoboChart parser are completed as part of the RoboArch system level testing.

In total, 27 unit tests have been implemented. They use EUnit tests and are coordinated using the Ant execution language. The tests can be found at the following website[5]. While the unit tests verify the implementation of the tool they do not validate the RoboArch semantics; to do this we use a case study presented in the next section.

5.2 Obstacle Avoidance Case Study

For the obstacle avoidance case study, we use RoboArch to specify the software architecture of a puck robot whose goal is to move through its environment avoiding any obstacles that are present. We explore useful properties of the architecture that can be proven using the RoboChart model generated from the RoboArch description of the system.

The robot for the system is two wheeled and cylindrically shaped. For movement the robot has a pair of motors and for sensing its surroundings there is a forward facing proximity sensor. The sensor measures the distance between itself and objects that are in its line of sight.

The environment the robot is expected to operate within is any typical room indoors that has a flat surface. The behaviour of the robot is to move forwards until an obstacle is less than a predetermined fixed distance away and stop. After stopping, the robot rotates until it is no longer blocked by the obstacle and moves forwards in the new direction. Next we present the RoboArch description followed by the RoboChart model.

[5] https://github.com/UoY-RoboStar/robochart2text-epsilon/tree/thesis/circus.robocalc.robochart2text.gen/tests

5.2.1 RoboArch Description

For our obstacle avoidance robotic system we specify a two layer software architecture that consists of a control and an executive layer. The control layer uses the reactive skills pattern to define the robot's behaviours. Because the desired behaviour is purely reactive and it can be accomplished by the control layer alone, the function of the executive layer is to initialise the control layer and put it into its operating state exploring the environment. Listing 5.4 to 5.9 show the RoboArch description of this two layer structure.

Listing 5.4 defines the system, types, interfaces, and platform. The system named `ObstacleAvoidance` that holds the rest of the system's definitions is declared on Line 1. A record type named `Velocities` is

Listing 5.4: Obstacle avoidance definitions and platform declarations

```
1  system ObstacleAvoidance
2
3  datatype Velocities {
4      linear:real
5      angular:real
6  }
7
8  interface Motors {
9       event move: Velocities
10 }
11
12 interface Sense {
13     event proximity: int
14 }
15
16 robotic platform PuckRobot {
17     uses Motors
18     uses Sense
19 }
```

declared on Lines 3 to 6 it has two fields named `linear` and `angular` both of type `real`. The information held by `Velocities` represents a trajectory of the robot. The interfaces `Motors` and `Sense` are declared on Lines 8 to 14 they specify events `move` and `proximity`. The interfaces `Motors` and `Sense` are declared on Lines 8 to 14. They specify events `move` and `proximity`. Proximity events are used to communicate in-

tegers representing the distance of an obstacle from the robot's sensor. Move events are used to communicate the velocities the robot's motors should be set to. The robotic platform named `PuckRobot` is declared on Lines 16 to 19; it uses the `Motors` and `Sense` interfaces.

Listing 5.5 defines the layer named `Executive`; it is given no type because its function is to initialise the fully reactive control layer. The inputs and outputs defined on Lines 24 to 44 match those of the control layer but with inputs and outputs swapped for well-formedness of communication. No pattern for the `Executive` layer is given so the generated RoboChart model is a minimal machine and will be customised.

Listing 5.5: Obstacle avoidance executive layer

```
22 layer Executive {
23
24   inputs= eventReply:Events, activeSkills:Set(Skills),
25       ProximitygapValue:int, exploreVOutValue:Velocities,
26       explorePriorityParameter:nat,
27       proximityPriorityParameter:nat,
28       exploreMaxSpeedParameter:real,
29       exploreSafetyDistanceParameter:int,
30       movePriorityParameter:nat;
31
32   outputs= activate:Skills, deactivate:Skills,
33       getActiveSkills, initiateEventMonitor:Events,
34       stopEventMonitor:Events, getProximityGapValue,
35       getExploreVOutValue, getExplorePriorityParameter,
36       setExplorePriorityParameter:nat,
37       getProximityPriorityParameter,
38       setProximityPriorityParameter:nat,
39       getExploreMaxSpeedParameter,
40       setExploreMaxSpeedParameter:real,
41       getExploreSafetyDistanceParameter,
42       setExploreSafetyDistanceParameter:int,
43       getMovePriorityParameter,
44       setMovePriorityParameter:nat;
45 } ;
```

Listings 5.6 and 5.7 define the layer named `Control` that is of type `ControlLayer`. Because the layer interacts with the platform it uses the `Motors` and `Sense` interfaces as shown on Lines 50 and 51.

Listing 5.6: Obstacle avoidance control layer

```
48 layer Control: ControlLayer {
49
50   uses Motors
51   uses Sense
52
53   inputs= activate:Skills, deactivate:Skills,
54       getActiveSkills, initiateEventMonitor:Events,
55       stopEventMonitor:Events, getProximityGapValue,
56       getExploreVOutValue, getExplorePriorityParameter,
57       setExplorePriorityParameter:nat,
58       getProximityPriorityParameter,
59       setProximityPriorityParameter:nat,
60       getExploreMaxSpeedParameter,
61       setExploreMaxSpeedParameter:real,
62       getExploreSafetyDistanceParameter,
63       setExploreSafetyDistanceParameter:int,
64       getMovePriorityParameter,
65       setMovePriorityParameter:nat;
66
67   outputs= eventReply:Events, activeSkills:Set(Skills),
68       ProximitygapValue:int,
69       exploreVOutValue:Velocities,
70       explorePriorityParameter:nat,
71       proximityPriorityParameter:nat,
72       exploreMaxSpeedParameter:real,
73       exploreSafetyDistanceParameter:int,
74       movePriorityParameter:nat;
```

Line 76 declares the reactive skills pattern which can have skills, connections, and monitors. Lines 78 to 91 defines three skills: `Move`, `Proximity`, and `Explore`. `Move` is a D-Skill with one input `move` of type `Velocities` that sets the speed of the robot's motors enabling its direction to be controlled. `Proximity` is a D-Skill with an output `proximity`. `Explore` is a C-Skill that has parameters `safetyDistance` of type `int` and `maxSpeed` of type `real`, an input `obstacleDistance` of type `int`, and an output `vOut` of type `velocities`. When `proximity` values are greater than the `safetyDistance`, `Explore` outputs a non-zero linear velocity and zero angular velocity. When `proximity` values are lower than the `safetyDistance`, `Explore` outputs zero linear velocity and a non-zero angular velocity.

Listing 5.7: Obstacle avoidance control layer pattern

```
76 pattern= ReactiveSkills;
77
78 skills=
79   dskill Move {
80       inputs= vIn:Velocities;
81   },
82
83   dskill Proximity {
84       outputs= gap:int;
85   },
86
87   cskill Explore {
88       parameters=  safetyDistance: int, maxSpeed: real;
89       inputs= obstacleDistance: int;
90       outputs= vOut:Velocities;
91   } ;
92
93 connections=
94   Proximity on gap to Explore on obstacleDistance,
95   Explore on vOut to Move on vIn;
96
97 monitors= ( SafetyDistanceReached |
98     Proximity::gap < Explore::safetyDistance );
99
100 } ;
```

Lines 93 to 95 declare the connections that connect the inputs and outputs of the skills. They enable values from the platform to be used by the `Explore` C-Skill and the velocities it outputs to set the robot's velocity via the platform. There are two connections: one from `Proximity` to `Explore` and the other from `Explore` to `Move`.

A monitor named `SafetyDistanceReached` that checks for the condition `proximity` being less than the `safetyDistance` is defined on Line 97. On the condition occurring, the `Executive` layer will be notified.

Listings 5.8 and 5.9 declare the system's connections connecting the layers and platform. For our example the inputs and outputs between the `Control` and `Executive` layers that have matching names are connected and the two events of the platform are connected to their corresponding input or output events of the `Control` layer.

Listing 5.8: Obstacle avoidance layer connections executive to control

```
103 connections=
104   Executive on activate to Control on activate,
105   Executive on deactivate to Control on deactivate,
106   Executive on getActiveSkills to Control on
       getActiveSkills,
107   Executive on initiateEventMonitor to Control on
       initiateEventMonitor,
108   Executive on stopEventMonitor to Control on
       stopEventMonitor,
109   Executive on getProximityGapValue to Control on
       getProximityGapValue,
110   Executive on getExploreVOutValue to Control on
       getExploreVOutValue,
111   Executive on getExplorePriorityParameter to Control on
        getExplorePriorityParameter,
112   Executive on setExplorePriorityParameter to Control on
        setExplorePriorityParameter,
113   Executive on getProximityPriorityParameter to Control
       on getProximityPriorityParameter,
114   Executive on setProximityPriorityParameter to Control
       on setProximityPriorityParameter,
115   Executive on getExploreMaxSpeedParameter to Control on
        getExploreMaxSpeedParameter,
116   Executive on setExploreMaxSpeedParameter to Control on
        setExploreMaxSpeedParameter,
117   Executive on getExploreSafetyDistanceParameter to
       Control on getExploreSafetyDistanceParameter,
118   Executive on setExploreSafetyDistanceParameter to
       Control on setExploreSafetyDistanceParameter,
119   Executive on getMovePriorityParameter to Control on
       getMovePriorityParameter,
120   Executive on setMovePriorityParameter to Control on
       setMovePriorityParameter,
```

Listing 5.9: Obstacle avoidance layer connections control to executive and platform

```
122  Control on eventReply to Executive on eventReply,
123  Control on activeSkills to Executive on activeSkills,
124  Control on ProximitygapValue to Executive on
      ProximitygapValue,
125  Control on exploreVOutValue to Executive on
      exploreVOutValue,
126  Control on explorePriorityParameter to Executive on
      explorePriorityParameter,
127  Control on proximityPriorityParameter to Executive on
      proximityPriorityParameter,
128  Control on exploreMaxSpeedParameter to Executive on
      exploreMaxSpeedParameter,
129  Control on exploreSafetyDistanceParameter to Executive
       on exploreSafetyDistanceParameter,
130  Control on movePriorityParameter to Executive on
      movePriorityParameter,
131
132  Control on move to PuckRobot on move,
133  PuckRobot on proximity to Control on proximity;
```

The description of the obstacle avoidance system we have seen is used to generate a RoboChart model and verify some properties of the reactive skills pattern. In the next section we present the RoboChart model that is used for verifying properties of the reactive skills pattern.

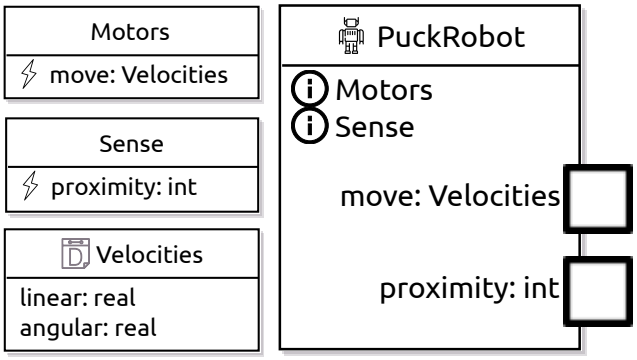
5.2.2 RoboChart model

Using the tool to apply the transformation rules from Chapters 3 and 4 to the RoboArch description of the obstacle avoidance case study, a sketch of a RoboChart model of the system is generated. Here we present the complete RoboChart model and highlight the customisations made to complete the sketch so that it can be used for verification of properties.

The SystemToRCModule Rule 3.2.1 previously discussed is the rule that calls the other rules and specifies the structure of the top-level RoboArch Module, Layers, and Platform.

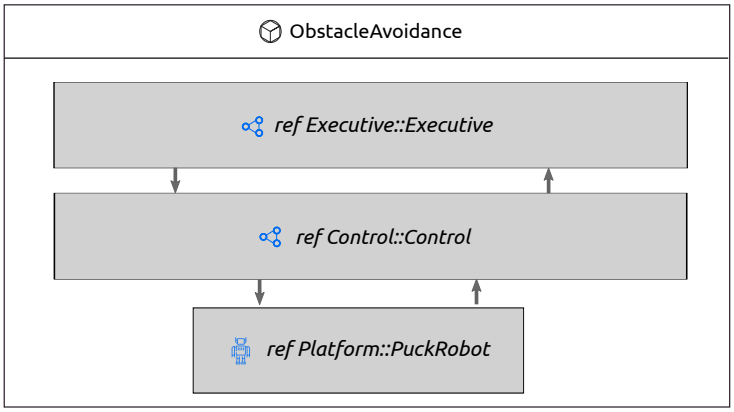
The `ObstacleAvoidance` system's definitions and interfaces are declared in the RoboArch description. These declarations are directly assigned to **rcdef**, **rcifcs**, and **rcmod** in the rule, therefore, they are reused in the RoboChart model. Figure 5.5 shows the type, interfaces, and platform declarations of the RoboChart model.

Figure 5.5: RoboChart Platform



The SystemToRCModule rule specifies the RoboChart module named `ObstacleAvoidance`. The nodes of the module in the RoboChart model are controllers created by the LayersToControllers Rule 3.2.2. This means that for our example there are two controllers named `Executive` and `Control`. Figure 5.6 sketches the generated RoboChart module; the complete module showing all connections can be found in Appendix D.1. The module contains the two controllers Executive and Control and the platform PuckRobot.

Figure 5.6: RoboChart Module overview



The connections of the module are declared in the RoboArch description and are directly assigned to the module's connections of the RoboChart model. The connections appear in Figure 5.6 between the control-

lers and the robotic platform.

For the detail of the layer `Control` the LayersToControllers Rule 3.2.2 specifies a controller with matching name. The controller's events are given by the layer's inputs and outputs. The layer's machines and connections are specified by the PatternToMachinesAndConnections Rule 4.4.1; because the layer's pattern is declared as `ReactiveSkills` Rule 4.4.2 is called. Figure 5.8 shows the details of the generated controller. There are four machines, one for each skill `Move`, `Proximity`, and `Explore` plus a machine for the skills manager.

The `dskill` machines follow the structure specified by Rule 4.4.6 and shown in Figure 4.5. The generated behaviour forwards values received from the platform sensor event, platformProximity, to the `Proximity` skill's output event proximity, and forwards the calculated values received from the `Explore` skill to the platform event platformMove.

The `cskill` machine follows the structure specified by Rule 4.4.5 and shown in Figure 4.4. The ComputeOutputs state was customised to model the obstacle avoidance behaviour on the distance dropping below the `safetyDistance`. Figure 5.7 is the customised state for our example. The avoidance behaviour is modelled using a conditional as an entry action.

Figure 5.7: Explore Skill Compute State

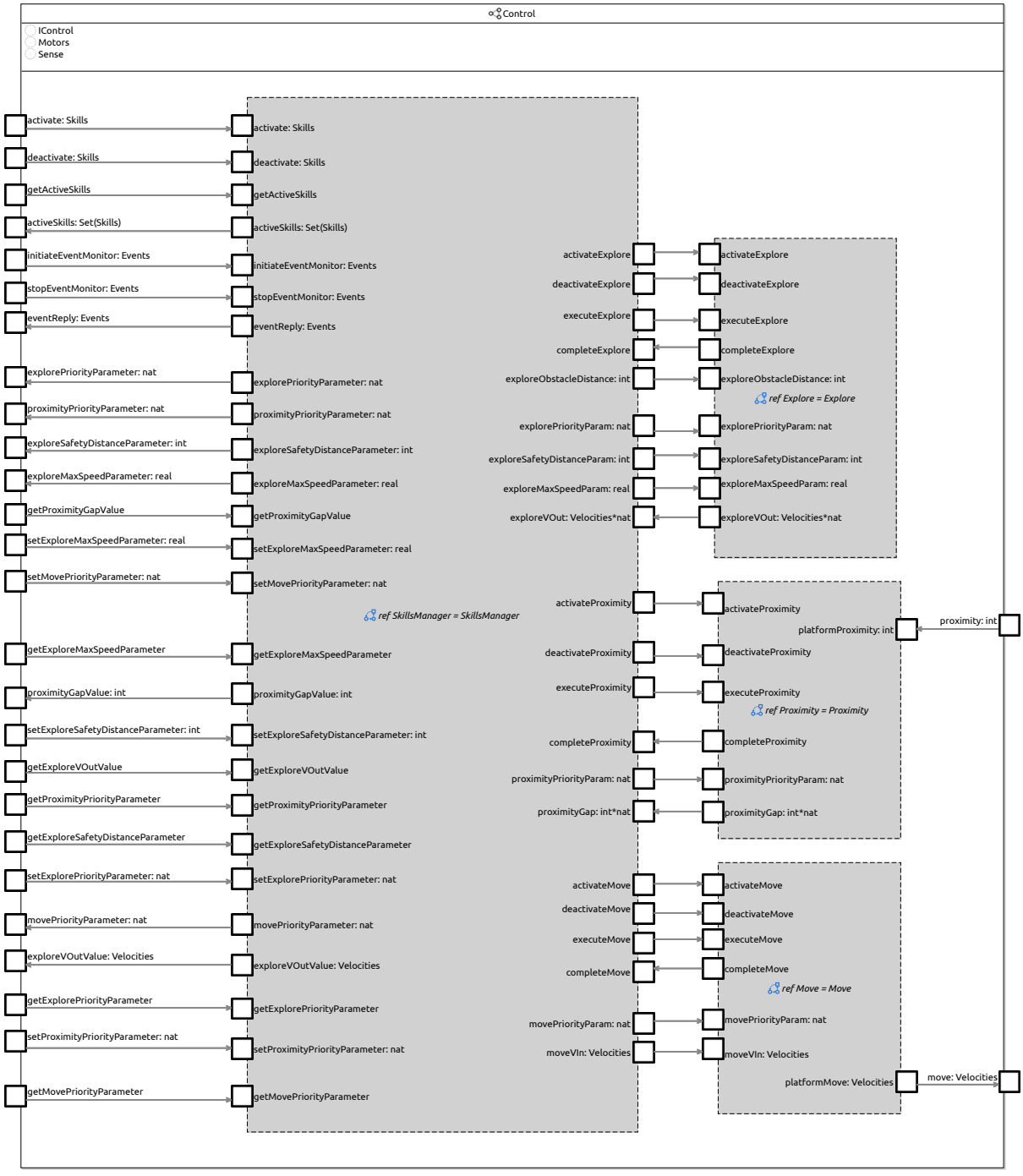
ComputeOutputs
entry if (obstacleDistance>safetyDistanceParam) then nextMotorsVelocity.linear = maxSpeedParam; nextMotorsVelocity.angular = 0; exploreVOut!(|nextMotorsVelocity, PriorityParam|) else nextMotorsVelocity. linear = 0; nextMotorsVelocity.angular = maxSpeedParam; exploreVOut!(|nextMotorsVelocity, PriorityParam|) end

The skills manager machine follows the structure specified by Rule 4.4.7 and shown in Figure 4.6. No customisation of the skills manager is necessary because its behaviour is established by the reactive skills pattern.

The last thing that had to be customised in the Control controller generated RoboChart model was the connections between the SkillsManager and the events on the boundary of the Control controller.

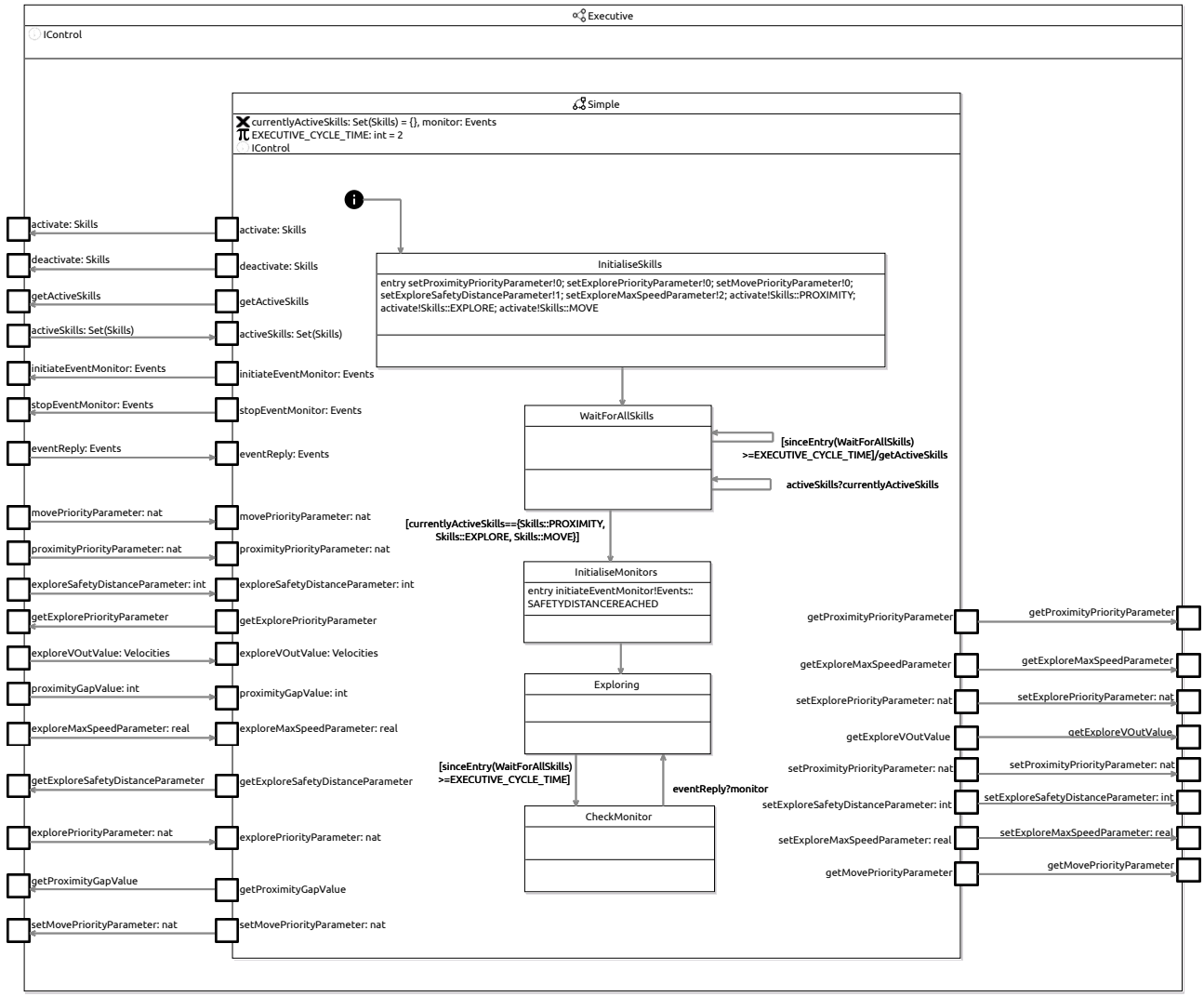
For the layer `Executive`, Rule 3.2.2 LayersToControllers specifies a Controller with matching name. The controllers events are given by the layer's inputs and outputs. The layer's machines and connections are specified by the PatternToMachinesAndConnections Rule 4.4.1.

Figure 5.8: RoboChart Control Layer



As no pattern for the `Executive` layer is declared, the PatternToMachinesAndConnections Rule4.4.1 defines a minimal machine and no connections in the generated model's Controller. We customise the `Executive` layer to model the initialisation of the Control layer. Figure 5.9 shows the customised Executive controller.

Figure 5.9: RoboChart Executive Layer



There is one machine called Simple that has five states: InitialiseSkills, WaitForAllSkills, InitialiseMonitors, Exploring, and CheckMonitors.

The machine enters the InitialiseSkills on startup and performs the entry action that sets the parameters to starting values, for example, `Explore`'s safetyDistance using the setExploreSafetyDistance event. The entry action also activates all three skills using the activateSkills event after which the transition to the WaitForAllSkills is taken and the state entered.

After an amount of time given by EXECUTIVE_CYCLE_TIME the transition back to WaitForAllSkills is taken. The entry action requests the currently active skills by triggering the getActiveSkill event. Next the WaitForAllSkills is re-entered. While in the WaitForAllSkills state, responses from the Control controller activeSkills event can be accepted and the value stored in the currentlyActiveSkillsVariable. When all the skills are reported as being active (currentlyActiveSkills) then the transition to InitialiseMonitors is taken and the state entered.

On entering InitialiseMonitors the entry action enables the `SafetyDistanceReached` monitor and the transition to Exploring, and the state is entered. After EXECUTIVE_CYCLE_TIME time units since entering the state have elapsed the transition to the CheckMonitor state is taken and the state is entered. On an eventReply being accepted, the value is stored in the monitor variable, and the transition to the Exploring state is taken.

In summary once the Control has been initialised the Executive periodically stores any received notifications of the safety distance being violated.

From the RoboChart model of the obstacle avoidance system, the CSP model can be generated enabling properties of reactive skills to be verified via model checking.

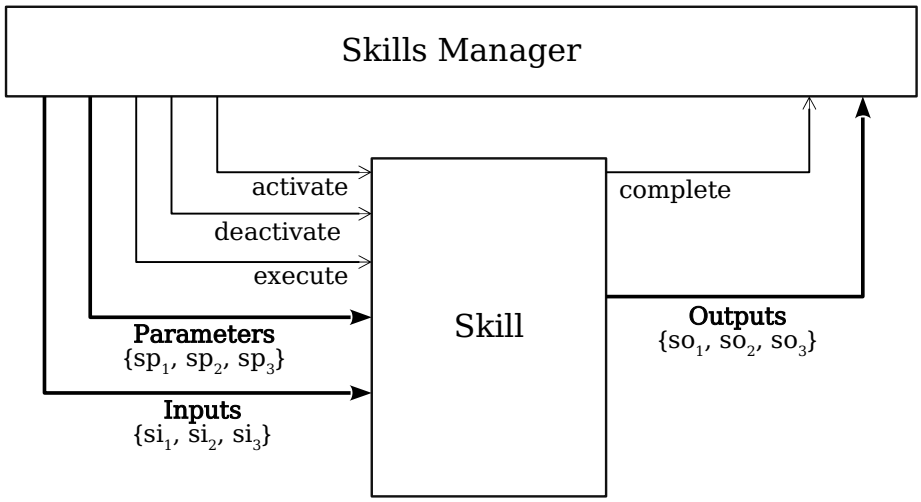
5.2.3 Properties

In this section we specify the properties that models using the reactive skills pattern have. Because the RoboChart semantics is formalised using CSP and models can be verified using a refinement model checker, the expected properties are specified as CSP processes. The corresponding CSP for each description can be found in Appendix E.

The two kinds of skill in the reactive skills pattern C-Skill and D-Skill share some common properties. Figure 5.10 shows the inputs and outputs of the CSP processes and defines sets of events that are used to specify the common properties in both their description and corresponding CSP.

The boxes in Figure 5.10 represent processes and the arrows channels. The direction of an arrow points to the destination process where events are input. Bold arrows show channels that can have a multiplicity greater than one and they are grouped by metamodel attribute; the exact number depends on the number of attributes defined in the RoboArch model.

Figure 5.10: Skill CSP process input and output events.



$$skillControlEvents = \{activate, deactivate, execute, complete\}$$
$$skillParameterEvents = \{|sp_1, sp_2, sp_3 ...|\}$$
$$skillInputEvents = \{|si_1, si_2, si_3 ...|\}$$
$$skillOutputEvents = \{|so_1, so_2, so_3 ...|\}$$

$$allSkillEvents = skillControlEvents \cup skillParameterEvents \cup skillInputEvents \cup skillOutputEvents$$

where sp_w, si_x, so_y are channels

External behaviour common to all skills is specified by the following properties.

EB-S1 A skill initially accepts a value for any of the parameters.

EB-S2 After accepting one of each parameter value the skill accepts an activate event and any of its parameter values.

EB-S3 After accepting an activate event a skill accepts any of its parameter values, input values, an execute event or deactivate event.

EB-S4 After accepting an execute event a skill must output a complete event before any of the other skill control events.

EB-S5 After accepting a deactivate event a skill can only accept its parameter values or an activate event.

EB-S6 No skill outputs are communicated until the skill's priority parameter event is accepted. After accepting a priority parameter event, the skill's output value always includes the most recently accepted priority value.

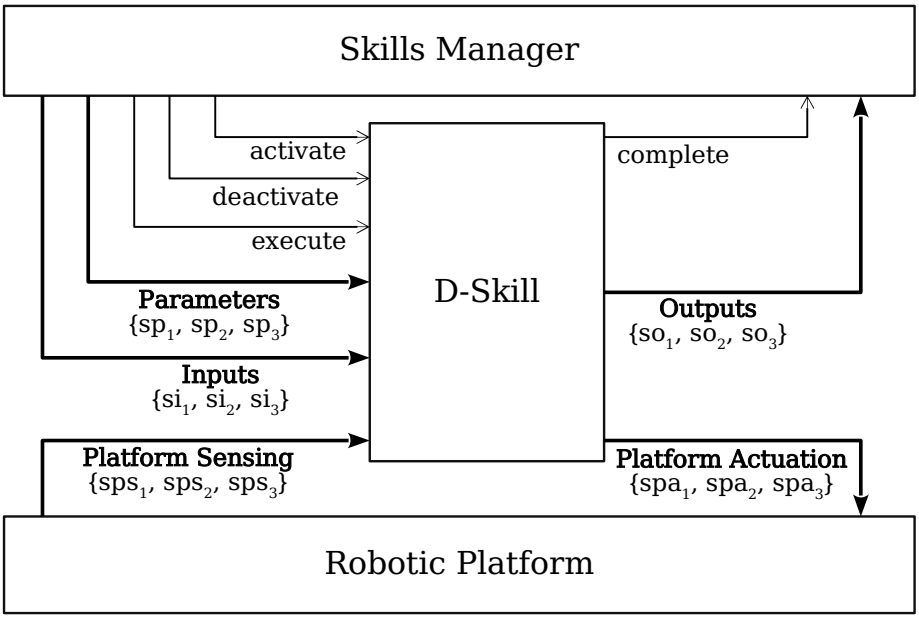
C-Skills compute values for their outputs from their input values and communicate the new values to the skill manager. They have an additional property:

EB-CS1 After accepting an execute event the skill communicates zero or more output value(s) before communicating a complete event.

C-Skill events and communication structure are the same as those defined for Skills.

D-Skills communicate with the robotic platform so they additionally can receive values as inputs from the platform's sensors and output values to the platform that alter the robot's state. Figure 5.11 shows the inputs and outputs of the CSP processes and defines sets of events that are used to specify D-Skill properties.

Figure 5.11: D-Skill CSP process input and output events.



$$skillControlEvents = \{activate, deactivate, execute, complete\}$$
$$skillParameterEvents = \{|sp_1, sp_2, sp_3 ...|\}$$
$$skillInputEvents = \{|si_1, si_2, si_3 ...|\}$$
$$skillOutputEvents = \{|so_1, so_2, so_3 ...|\}$$

$$skillPlatformActuationEvents = \{|spa_1, spa_2, spa_3 ...|\}$$
$$skillPlatformSensingEvents = \{|sps_1, sps_2, sps_3 ...|\}$$

$$allSkillEvents = skillControlEvents \cup skillParameterEvents \cup skillInputEvents \cup skillOutputEvents \cup skillPlatformActuationEvents \cup skillPlatformSensingEvents$$

$$sp_{priority} \equiv sp_1$$

where sp_u, si_v, so_w, spa_x, sps_y are channels

In addition to the common skill properties D-Skills have three further properties:

EB-DS1 After accepting an execute event and before communicating a complete event, a D-Skill can communicate zero or more platform actuation value(s) and zero or more output value(s).

EB-DS2 After accepting an activate or complete event followed by an execute event, the latest input event(s) values that were accepted since the last activate or complete event, if any, are communicated to the skill's corresponding platform actuation output.

EB-DS3 After accepting an activate or complete event followed by an execute event, the latest platform sensing event(s) values that were accepted since the last activate or complete event, if any, are communicated to the corresponding output(s).

The skills manager coordinates the communication between skills and provides an interface that enables the execution of skills to be controlled and monitored. Figure 5.12 shows the inputs and outputs of the CSP processes that are used to specify the Skills Manager properties. The complete skills manager set definitions can be found in Appendix E.4. The skill channels indicated by the '...' symbol in Figure 5.12 are given by the C-Skill and D-Skill from Figures 5.10 and 5.11

External behaviour for the Skills Manager is specified by the following properties.

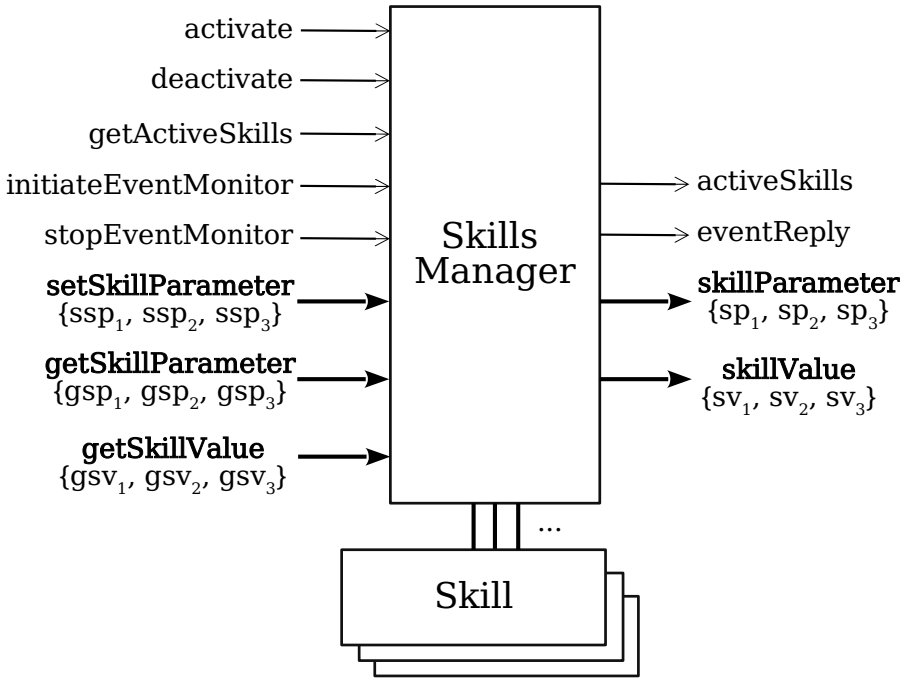
EB-SM1 After accepting a getActiveSkills handling request the response is communicated via an activeSkills event.

EB-SM2 After accepting a get⟨skill.name⟩⟨param.name⟩Parameter handling request the response is communicated via a ⟨skill.name⟩⟨param.name⟩Parameter event.

EB-SM3 After accepting a set⟨skill.name⟩⟨param.name⟩Parameter handling request the new parameter value is communicated via a ⟨skill.name⟩⟨param.name⟩Param event.

EB-SM4 No handling response can happen until a handling request has been accepted.

Figure 5.12: Skills Manager CSP Process input and output events.



This account of the skills manager properties is not exhaustive and further properties can be specified to increase the coverage of the verification of the reactive skills pattern.

In the next section, we summarise the results of verifying these properties for the obstacle avoidance model.

5.2.4 Verification Results

The fourteen reactive skills properties previously specified were checked using FDR to ensure that each property is refined by the RoboChart obstacle avoidance model from Section 5.2.2. Table 5.1 summarises the results of verification on the RoboStar test server that has two terabyes of memory and two physical CPUs each with 32 physical cores with a 2.0 GHz base clock.

The table shows the skills properties for EB-S1 to EB-S6, EB-CS1, and EB-DS1 to EB-DS3 all passed in the untimed and timed models with an average execution time of 6 seconds. However, the Skills Manager

properties EB-SM1 to EB-SM4 could not be verified because the resources of test machine were exhausted after running for 6 hours 42 minutes.

Table 5.1: Summary of verification results for the reactive skills properties.

Property	Untimed	Timed
EB-S1	✓	✓
EB-S2	✓	✓
EB-S3	✓	✓
EB-S4	✓	✓
EB-S5	✓	✓
EB-S6	✓	✓
EB-CS1	✓	✓
EB-DS1	✓	✓
EB-DS2	✓	✓
EB-DS3	✓	✓
EB-SM1	∗	–
EB-SM2	∗	–
EB-SM3	∗	–
EB-SM4	∗	–

key: ✓=Pass, ✗=Fail, –=Not tested,
∗=Ran out of resources

Table 5.2: Summary of verification results for the reactive skills properties simplified.

Property	Untimed	Timed
EB-SM1	✓	✓
EB-SM2	✓	✓
EB-SM3	✓	✓
EB-SM4	∗	–

key: - ✓=Pass, ✗=Fail, –=Not tested
∗=Ran out of resources

In order to prove some properties for the skills manager, the obstacle avoidance model was simplified to remove the proximity D-Skill. This meant the simplified model had one D-Skill Move and one C-Skill Explore. The explore D-Skill was modified to have one output that output a constant velocity to the move D-Skill. The results of verifying the simplified model are shown in Table 5.2.

For the untimed and timed models EB-SM1 to EB-SM3 could be verified

and had an average execution time of 342 seconds however EB-SM4, although executed, again exhausted the resources of the test machine so the property could not be verified.

The assertions used for for property verification can be found in Appendices D.2 to D.5.

5.3 Final Considerations

In this chapter, we have discussed the evaluation of RoboArch based on a supporting tool. We have presented the architecture of the tool and its verification. We have shown the close correspondence of the implementation of the rules and their definitions in Chapters 3 and 4 that enabled the rules correctness to be evaluated. This was achieved through the implementation of the rules and then running the transformations on test models so that errors, for instance in the usage of types, could be identified and corrected.

We have seen RoboArch used in the obstacle avoidance case study to describe a complete system's software architecture that is layered and uses the reactive skills pattern to achieve some simple target behaviours. The RoboChart model automatically generated from the RoboArch description and its customisation that specify remaining application specific details were presented.

Finally, some properties of the reactive skills pattern were specified and verified using the completed RoboChart model of the obstacle avoidance system. The successful verification of these properties demonstrates the expected behaviours of the system's model and contributes to the validation of RoboArch's semantics. However, there were issues of scalability; for the larger components like the skills manager not all properties could be verified by model checking. Simplifying the model of the case study helped but does not solve the problem of scalability. This highlights the need for additional approaches and techniques for verification.

In the next chapter, we conclude by identifying the contributions of RoboArch including its limitations and future work.

6 Conclusion

In this chapter we present the steps for adding patterns to RoboArch in Section 6.1, summarise our contributions in Section 6.2, and outline future work in Section 6.3.

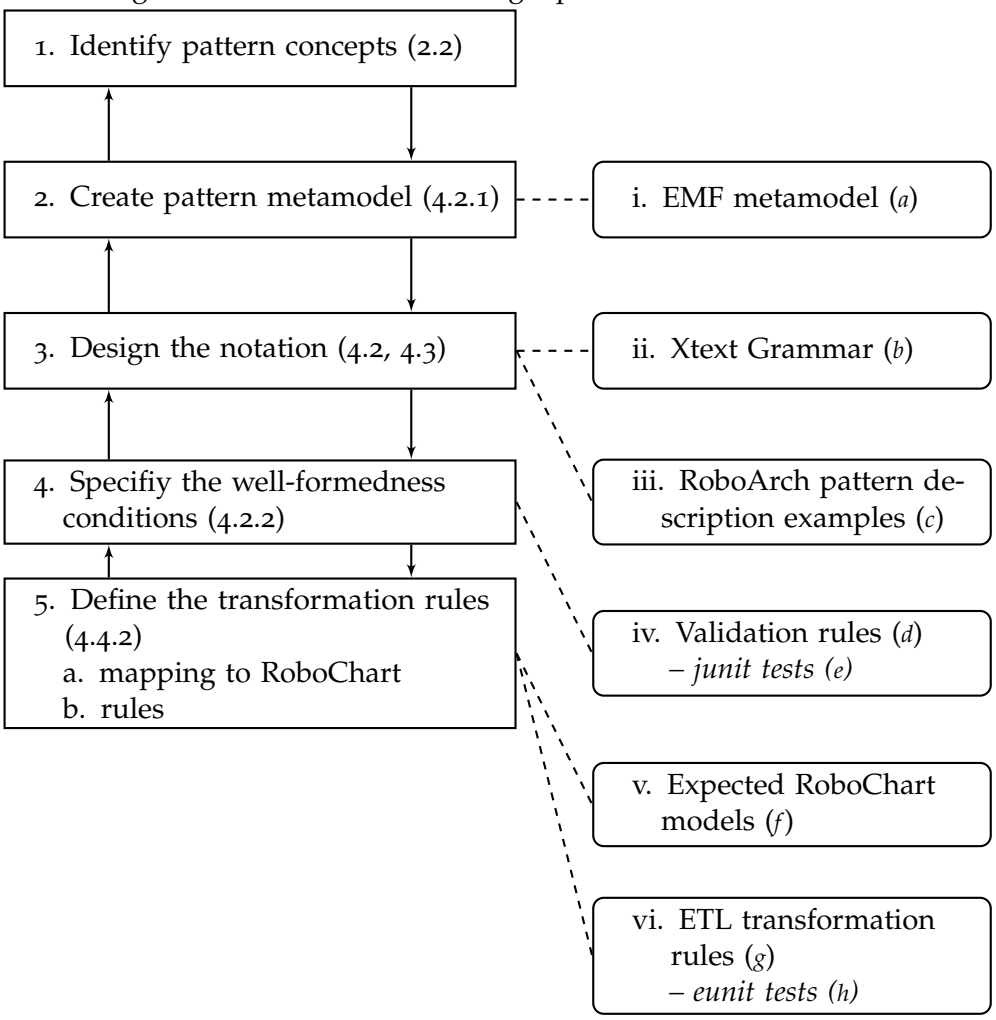
6.1 Adding Patterns to RoboArch

Figure 6.1 shows the steps to add a new pattern to RoboArch. The boxes with square corners indicate the tasks necessary for characterising a pattern. The boxes with rounded corners are the tasks for extending the RoboArch tool to support a pattern. Dashed lines from a characterisation task indicates the related tool extension tasks. The arrows between tasks indicate the next suggested task to complete in the workflow. As each task is completed, from the insights gained through further elaboration of the pattern, it may become apparent that an earlier assumption made from a previous task is no longer correct and needs to be updated; this is shown by arrows pointing to an earlier task.

Once the pattern to be characterised has been selected the first task is to identify the pattern's concepts from the available description. A starting point is to find keywords their meaning and determine what elements they contribute to the pattern. For each identified element try to answer: is it composed of other elements? how do each of the elements interact and what elements do they interact with? are there any restrictions on interaction? how does this element relate to other elements and are there any common categories? Record these elements in a table summarising the description defining each; these are the concepts. Finally, identify primary concepts of the pattern that depend on other concepts for their definition. Separate the primary concepts out into their own table. Examples of concept identification tables for reactive skills and subsumption can be found in Section 2.2.

The second task is to create the pattern metamodel. The concepts identified in the first task need to be translated into classes, attributes, and relationships of a metamodel. The understanding of the pattern from

Figure 6.1: Workflow for adding a pattern to RoboArch.



[a] https://github.com/UoY-RoboStar/roboarch-metamodel/blob/thesis/circus.robocalc.roboarch/model/roboarch.ecore

[b] https://github.com/UoY-RoboStar/roboarch-textual/blob/thesis/circus.robocalc.roboarch.textual/src/circus/robocalc/roboarch/textual/RoboArch.xtext

[c] https://github.com/UoY-RoboStar/roboarch2chart-epsilon/tree/thesis/RoboArch_Test_Text

[d] https://github.com/UoY-RoboStar/roboarch-textual/tree/thesis/circus.robocalc.roboarch.textual/src/circus/robocalc/roboarch/textual/validation]

[e] https://github.com/UoY-RoboStar/roboarch-textual/tree/thesis/circus.robocalc.roboarch.textual.tests/src/circus/robocalc/roboarch/textual/tests

[f] https://github.com/UoY-RoboStar/roboarch2chart-epsilon/tree/thesis/RoboChart_Test_Models

[g] https://github.com/UoY-RoboStar/roboarch2chart-epsilon/tree/thesis/circus.robocalc.roboarch2chart.erules/erules

[h] https://github.com/UoY-RoboStar/roboarch2chart-epsilon/tree/thesis/circus.robocalc.roboarch2chart.tests/eunit

completing the first task make this a modelling exercise. The related tool step *i*, is to create the EMF metamodel.

The third task is to design the pattern's RoboArch notation. There are two related tool implementation tasks *ii* to create the grammar and *iii* to create example RoboArch descriptions that are valid with respect to the grammar. The descriptions created at this stage can be used for testing later on in the tool development.

The fourth task is to specify the well-formedness conditions. These ensure that only meaningful models can be created. The related tool implementation task is to create the validation rules that enforce the well-formedness conditions when the pattern's RoboArch description is being parsed.

The fifth task is to define the transformation rules. This is divided into two parts: part one is to map the concepts from the first task to their RoboChart concept representation, and part two is to define transformation rules that precisely describe the mappings. There are two related implementation tasks: *v* to create a set of expected RoboChart models that correspond to the RoboArch descriptions from the notation design, and *vi* to create the transformation rules in ETL so that descriptions using the newly characterised pattern can be automatically generated.

6.2 Summary of Contributions

It is evident from our literature review of software architecture for robotics in Chapter 2 that there has been little reuse of architectural patterns with most projects tending to establish their own. However, one commonality is the use of layers. The patterns reviewed were all informally defined leading to ambiguity in the behaviour they describe. Therefore, there was a need for a more precise way to define patterns that support the creation of safe and robust robotic systems.

In the existing DSLs for robotics, the main architectural elements are the components themselves with few constraints on how they are used together. The use of architectural patterns as enabled and encouraged by our work has been useful in other areas some examples are communication protocols [105] and embedded systems, to abstract application functions from the hardware. Our results may be useful in the context of other notations, beyond RoboChart, and in other application areas. Contributions of the work presented in this thesis are in three main areas.

The first is the characterisation of patterns in Chapter 3. For each

pattern their concepts and definitions are recorded and a metamodel and well-formedness conditions are specified. These characterisations form the foundation of the RoboArch language.

The second contribution is the RoboArch language and its notation given in Chapter 3 and Chaper 4. They specify the format for RoboArch's textual description for describing the software architecture of robotic systems and the model transformation rules to RoboChart that give RoboArch its semantics.

Compared to the DSLs reviewed in Chapter 2, RoboArch specifically targets architectural patterns used in robotics software and has specialised components with semantics given by model transformations to RoboChart. The transformations to a formalised language is a novel feature of our work supporting precision and rigour through the use of mathematical techniques for verification and proof.

RoboArch is not related to the homonym in [106], which is a tool to support the development of mobile robots. The focus in [106] is on implementation, not modelling, of hardware-software co-designs based on hardware and software components, and code generation for FPGA, not software architectures. Moreover, there is no semantics or support for verification beyond simulation for the notation adopted by RoboArch to define the compositions.

The closest relevant work is the WRIGHT architecture description language [84] where CSP is used to give a formal semantics to its component and connectors for analysing architectures by verification of properties. While the use of CSP is common with RoboArch, WRIGHT does not define any patterns for robotics.

More recent work RsaML [83] is an architectural modelling language to describe robotic systems. RsaML has a similar scope to RobotML in that its metamodel captures a complete system but with additional support for some of the patterns used in the robotics domain, for example, layers and parts of subsumption: suppressors and inhibitors. RsaML is similar to RoboArch in its support for architectural patterns, but differs in not having a formal semantics.

Finally, the third contribution of the RoboArch tool and framework means that there is automated support to describe robotic system architectures for the patterns presented. The facilities provided by the tool, for example, unit tests for testing subsets of transformation rules, can be used by others to validate new patterns as extensions to RoboArch. Therefore, allowing RoboArch to act as a library of precisely specified patterns that roboticists can draw upon when designing systems.

6.3 Future Work

To maximise the value RoboArch can offer to the robotics community additional patterns need to be characterised. In particular, patterns for the executive and planning layers so that systems designed to perform diverse tasks with increasing autonomy can be supported. A pattern that is popular for the executive layer and has been used in the planning layer is the Hierarchical Task Network (HTN) [10, p. 295]. The pattern has tasks, actions, and monitors that can be visualised as nodes of a tree diagram. They can be mapped to machines in a RoboChart model.

Further case studies should be conducted to investigate the expected properties of patterns and verify them. Another avenue for further work is comparative studies that review previous RoboChart case studies [20] and recreate them in RoboArch using patterns to study the effect of architecture on the system. This could give insights that enable detailed guidelines for patterns use to be established.

Another area of required work is the interface between layers and their pattern. The input and output events of layers and the connections defined among them specifies the layers' interfaces. The pattern used by a layer can have different interfaces, therefore, adaptors between the pattern and a layer's interface will be required. Currently the user must define their own adaptor by customising the generated RoboChart model. Being able to support the automatic generation of these adaptors is the expectation, however, this is not trivial because all combinations of patterns and layers have to be considered. Future standardisation may play a part in easing this situation.

Development of new verification techniques can also contribute to improved system verification. For instance, reverse transformations would allow static verification to ensure changes to the generated RoboChart model do not violate the specified system architecture.

Identifying properties of patterns and recording them in transformation rules to RoboChart assertions would mean that checks for a pattern's common properties can be automatically generated. Developers can verify these properties, assisting them in evaluating their system and the architecture they select.

Software architecture is often documented diagrammatically and there is general consensus among robotics DSLs to have a graphical notation. Therefore, giving RoboArch its own graphical notation and visual shorthands for connections would be worthwhile to complement the textual notation.

Appendices

A Lawn-Mowing System

A.1 Assertions

```
1  assertion MM_1: mower::MowManager is deterministic
2
3  assertion MM_2: mower::MowManager is divergence-free
4
5  assertion MM_3: mower::MowManager is deadlock-free
6
7  assertion MM_4: mower::MowManager does not terminate
8
9  assertion MM_5:
10     mower::MowManager::Charging is reachable in mower::
       MowManager
11
12 assertion MM_6:
13     mower::MowManager::Mowing is reachable in mower::MowManager
14
15 assertion MM_7:
16     mower::MowManager::AvoidingObstacle is reachable in mower::
       MowManager
17
18 assertion MM_8:
19     mower::MowManager::Turning is reachable in mower::MowManager
20
21 assertion MM_9:
22     mower::MowManager refines PAO  in the failures model
23
24 assertion MM_10:
25     mower::MowManager refines PTB in the failures model
26
27 assertion M_1: mower::Mower is deterministic
28 assertion M_2: mower::Mower is divergence-free
29 assertion M_3: mower::Mower is deadlock-free
30 assertion M_4: mower::Mower does not terminate
31
32 assertion LMS_1: lawnmower_system::Lawnmower is deterministic
33 assertion LMS_2: lawnmower_system::Lawnmower is divergence-free
34 assertion LMS_3: lawnmower_system::Lawnmower is deadlock-free
35
36
```

```
37 // MM_9 - Property Avoid Obstacle (PAO)
38 csp PAO csp-begin
39
40   PAO =
41     let
42       waitevents = {| mower_MowManager::boundary.in,
     mower_MowManager::fullPower.in, mower_MowManager::
     lowPower.in, mower_MowManager::enableCutterCall,
     mower_MowManager::moveForwardsCall, mower_MowManager::
     avoidCall, mower_MowManager::turnCall |}
43
44       PAO = [] ev: waitevents @ ( ev -> PAO )
45               []
46               mower::MowManager::obstacle.in ->
47                mower_MowManager::avoidCall -> PAO
48     within
49       PAO
50
51 csp-end
52
53
54 //MM_10 - Property Turn at Bundary (PTB)
55 csp PTB csp-begin
56
57   PTB =
58     let
59       waitevents = {| mower_MowManager::obstacle.in,
     mower_MowManager::fullPower.in, mower_MowManager::
     lowPower.in, mower_MowManager::enableCutterCall,
     mower_MowManager::moveForwardsCall, mower_MowManager::
     avoidCall, mower_MowManager::turnCall |}
60
61       PTB = [] ev: waitevents @ ( ev -> PTB )
62               []
63               mower::MowManager::boundary.in ->
64                mower_MowManager::turnCall -> PTB
65     within
66       PTB
67
68 csp-end
69
```

A.2 Results

Table A.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 PAO (MM_9) [traces model]	14	17	true
mower_MowManager is refined by PTB (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

B RoboArch Rules

This thesis presents the key rules; the complete rule reference can be found on the following website:
https://robostar.cs.york.ac.uk/publications/reports/roboarch_rules.pdf

B.1 Top-Level

RULE B.1.1:

ⓐ	**Name**	LayerToTypes
ⓑ	**Parameter** *name:type*	amlyr: Layer
ⓒ	**Result** *name:type*	rctyp: Set(TypeDecl)
ⓔ	**Definition**	amlyr.pattern ∈ ReactiveSkills ⇒ rctyp = ReactiveSkillsLayerToTypes(amlyr) amlyr.pattern ∈ Subsumption ⇒ rctyp = SubsumptionLayerToTypes(amlyr) amlyr.pattern ∈ Htn ∧ amlyr ∈ ExecutiveLayer ⇒ rctyp = HtnExecutiveLayerToTypes(amlyr) amlyr.pattern ∈ Htn ∧ amlyr ∈ PlanningLayer ⇒ rctyp = HtnPlanningLayerToTypes(amlyr) amlyr.pattern ∈ PlannerScheduler ⇒ rctyp = PlannerSchedulerLayerToTypes(amlyr)

RULE B.1.2:

ⓐ	**Name**	LayerToFunctions
ⓑ	**Parameter** *name:type*	amlyr: Layer
ⓒ	**Result** *name:type*	rcfuns: Set(Function)
ⓔ	**Definition**	amlyr.pattern ∈ ReactiveSkills ⇒ rctyp = ReactiveSkillsLayerToFunctions(amlyr) amlyr.pattern ∈ Subsumption ⇒ rctyp = SubsumptionLayerToFunctions(amlyr) amlyr.pattern ∈ Htn ∧ amlyr ∈ ExecutiveLayer ⇒ rctyp = HtnExecutiveLayerToFunctions(amlyr) amlyr.pattern ∈ Htn ∧ amlyr ∈ PlanningLayer ⇒ rctyp = HtnPlanningLayerToFunctions(amlyr) amlyr.pattern ∈ PlannerScheduler ⇒ rctyp = PlannerSchedulerLayerToFunctions(amlyr)

RULE B.1.3:

ⓐ	**Name**	LayerToInterface
ⓑ	**Parameter** *name:type*	amlyr: Layer
ⓒ	**Result** *name:type*	rclyri: Set(Interface)
ⓔ	**Definition**	amlyr.pattern ∈ ReactiveSkills ⇒ rclyri = ReactiveSkillsLayerToInterface(amlyr) amlyr.pattern ∈ Subsumption ⇒ rclyri = SubsumptionLayerToInterface(amlyr) amlyr.pattern ∈ Htn ∧ amlyr ∈ ExecutiveLayer ⇒ rclyri = HtnExecutiveLayerToInterface(amlyr) amlyr.pattern ∈ Htn ∧ amlyr ∈ PlanningLayer ⇒ rclyri = HtnPlanningLayerToInterface(amlyr) amlyr.pattern ∈ PlannerScheduler ⇒ rclyri = PlannerSchedulerLayerToInterface(amlyr)

RULE B.1.4:

ⓐ	**Name**	ControlLayerToRoboticPlatform
ⓑ	**Parameter** *name:type*	amcl: Set(ControlLayer)
ⓒ	**Result** *name:type*	rcrpr: Set(RoboticPlatform)

Continued on next page

RULE B.1.4 – continued from previous page

ⓔ	**Definition**	#amcl = 1 ∧ (μ l : amcl • l.pattern) ∈ ReactiveSkills ⇒
		rcrpr = {ReactiveSkillsLayerToRoboticPlatform(
		(μ l : amcl • l)
		)}
		#amcl = 1 ∧ (μ l : amcl • l.pattern) ∈ Subsumption ⇒
		rcrpr = {SubsumptionLayerToRoboticPlatform(
		(μ l : amcl • l)
		)}
		#amcl ≠ 1 ⇒
		rcrpr = ∅

B.2 Reactive Skills Pattern

RULE B.2.1:

ⓐ	**Name**	ReactiveSkillsLayerToTypes
ⓑ	**Parameter** *name:type*	amrsl: Layer
ⓒ	**Result** *name:type*	rctyp: Set(TypeDecl)
ⓓ	**Pre** *condition*	amrsl.pattern ∈ ReactiveSkills
ⓔ	**Definition**	rctyp = { SkillsToSkillEnumeration(amrsl.pattern.skills), MonitorsToEventEnumeration(amrsl.pattern.skills) }

RULE B.2.2:

ⓐ	**Name**	ReactiveSkillsLayerToFunctions
ⓑ	**Parameter** *name:type*	amrsl: Layer
ⓒ	**Result** *name:type*	rcfun: Set(Function)
ⓓ	**Pre** *condition*	amrsl.pattern ∈ ReactiveSkills
ⓔ	**Definition**	rcfun = { ⟪ name= SkillsManagerFunctionUpdateValueName(), type= ref("boolean", PrimitiveType), parameters= ⟨ ⟪ name= "value", type= anyNatProduct ⟫$_{Parameter}$, ⟪ name= "newValue", type= anyNatProduct ⟫$_{Parameter}$ ⟩ ⟫$_{Function}$ } --- ***where*** anyNatProduct = ⟪ types = ⟨ ⟪ identifier= "X" ⟫$_{AyType}$, ref("nat", PrimitiveType) ⟩ ⟫$_{ProductType}$

RULE B.2.3:

ⓐ	**Name**	ReactiveSkillsLayerToInterface
ⓑ	**Parameter** *name:type*	amrsl: Layer
ⓒ	**Result** *name:type*	rcrsi: Set(Interface)
ⓓ	**Pre** *condition*	amrsl.pattern ∈ ReactiveSkills
ⓔ	**Definition**	rcrsi = {⦇ name = SkillsManagerInterfaceName() events = SkillsManagerInterfaceInputs(amrsl) ∪ SkillsManagerInterfaceOutputs(amrsl) ⦈$_{Interface}$} ∪ { skl : amrsl.pattern.skills • SkillToInterface(skl) }

RULE B.2.4:

ⓐ	**Name**	SkillsManagerInterfaceInputs
ⓑ	**Parameter** *name:type*	amrsl: Layer
ⓒ	**Result** *name:type*	ips: Set(Events)
ⓓ	**Pre** *condition*	amrsl.pattern ∈ ReactiveSkills

Continued on next page

RULE B.2.4 – continued from previous page

ⓔ	**Definition**	ips = {⟨ name = ActivationRequestName(), type = ref(SkillsEnumName(), Enumeration) ⟩$_{Event}$, ⟨ name = DeactivationRequestName(), type = ref(SkillsEnumName(), Enumeration) ⟩$_{Event}$, ⟨ name = ActiveSkillsRequestName() ⟩$_{Event}$, ⟨ name = "initiateEventMonitor", type = ref(EventsEnumName(), Enumeration) ⟩$_{Event}$, ⟨ name = "stopEventMonitor", type = ref(EventsEnumName(), Enumeration) ⟩$_{Event}$ } ∪ ⋃ { skl:amrsl.pattern.skills • { par : skl.parameters • ⟨ name = SkillParameterSetRequestName(skl.name, par.name) type = par.type ⟩$_{Event}$, ⟨ name = SkillParameterGetRequestName(skl.name, par.name) ⟩$_{Event}$ } } ∪ ⋃ { skl : amrsl.pattern.skill • { out : skl.outputs • ⟨ name= SkillValueRequestName(skl.name, out.name) ⟩$_{Event}$ } }

RULE B.2.5:

ⓐ	**Name**	SkillsManagerInterfaceOutputs
ⓑ	**Parameter** *name:type*	amrsl: Layer
ⓒ	**Result** *name:type*	ops: Set(Events)
ⓓ	**Pre** *condition*	amrsl.pattern ∈ ReactiveSkills
ⓔ	**Definition**	ops = {⟪ name = ActiveSkillsReturnName(), type = ⟪ domain = ref(SkillsEnumName(), Enumeration) ⟫$_{SetType}$ ⟫$_{Event}$, ⟪ name = MonitoredEventReturnName(), type = ref(EventsEnumName(), Enumeration) ⟫$_{Event}$ } ∪ ⋃ { skl : amrsl.pattern.skills • { par : skl.parameters • ⟪ name= SkillParameterReturnName(skl.name, par.name), type = par.type ⟫$_{Event}$ } } ∪ ⋃ { skl : amrsl.pattern.skills • { out : skl.outputs • ⟪ name= SkillValueReturnName(skl.name, out.name), type= out.type ⟫$_{Event}$ } }

RULE B.2.6:

ⓐ	**Name**	SkillToInterface
ⓑ	**Parameter** *name:type*	amskl: Skill
ⓒ	**Result** *name:type*	rcskl: Interface
ⓔ	**Definition**	rcskl rcskl.name=amskl.name rcskl.events= {⟨ name= "activate" + amskl.name ⟩$_{Event}$, ⟨ name= "deactivate" + amskl.name ⟩$_{Event}$, ⟨ name= "execute" + amskl.name ⟩$_{Event}$, ⟨ name= SkillCompleteEventName(amskl.name) ⟩$_{Event}$ } ∪ { par: amskl.parameters • ⟨ name= amskl.name + par.name + "Param", type= par.type ⟩$_{Event}$ } ∪ comSkillEvents --- ***where*** comSkillEvents = { inp: amskl.inputs • ⟨ name = SkillEventName(amskl.name, inp.name), type = inp.type ⟩$_{Event}$ } ∪ { out: amskl.outputs • ⟨ name = SkillEventName(amskl.name, out.name), type = ⟨ types = { out.type, ref("nat", PrimitiveType) ⟩$_{ProductType}$ ⟩$_{Event}$}

B.3 Functions

FUNCTION B.3.1

ⓐ	**Name**	GetConnectionsForCommonInterface
ⓑ	**Parameter** *name:type*	lctla: Controller, lctlb: Controller, intnm: Name, intin: Set(Event), intout: Set(Event)
ⓒ	**Result** *name:type*	rccon: Set(Connection)
ⓓ	**Pre** *condition*	$(\exists_1$ i : lctla.interfaces • i.name = intnm ∧ (i.events = intin ∪ intout)) ∧ $(\exists_1$ i : lctlb.interfaces • i.name = intnm)
ⓔ	**Definition**	rccon = { ev: commonInterface | ev ∈ intin • ⟪ from = ref(lctlb.name, Controller), efrom = ev, to = ref(lctla.name, Controller), eto = ev, ⟫$_{Connection}$ } ∪ { ev: commonInterface | ev ∈ intout • ⟪ from = ref(lctla, Controller), efrom = ev, to = ref(lctlb, Controller), eto = ev, ⟫$_{Connection}$ } ***where*** commonInterface = (μ i : lctla.interfaces | i.name = intnm • l.events)

FUNCTION B.3.2

ⓐ	**Name**	SkillsManagerBaseVariables
ⓑ	**Parameter** *name:type*	-
ⓒ	**Result** *name:type*	smbvs: Set(Variable)
ⓔ	**Definition**	smbvs = { ⟨ name = SkillsManagerVarCurrentActiveSkills-Name(), type = ⟨ domain = ref(SkillsEnumName(), Enumeration) ⟩$_{SetType}$ ⟩$_{Variable}$, ⟨ name = SkillsManagerVarActivationRequestName(), type = ref(SkillsEnumName(), Enumeration) ⟩$_{Variable}$, ⟨ name = SkillsManagerVarExecutingSkillName(), type = ref(SkillsEnumName(), Enumeration) ⟩$_{Variable}$, ⟨ name = SkillsManagerVarCurrentActiveMonitors-Name(), type = ⟨ domain= ref(SkillsEnumName(), Enumeration) ⟩$_{SetType}$ ⟩$_{Variable}$, ⟨ name = SkillsManagerVarMonitorRequestName(), type = ref(EventsEnumName(), Enumeration) ⟩$_{Variable}$, ⟨ name = SkillsManagerVarCycleSkillsName(), type = ⟨ domain = ref(SkillsEnumName(), Enumeration) ⟩$_{SeqType}$ ⟩$_{Variable}$

FUNCTION B.3.3

ⓐ	**Name**	SkillsManagerNewSkillInputFlagVariables
ⓑ	**Parameter** *name:type*	skill: Name
ⓒ	**Result** *name:type*	smifvs: Set(Variable)
ⓔ	**Definition**	smifvs = ⋃ { sk: skls ∣ sk ∈ CSkill • { inp: sk.inputs • ⟨ name = SkillsManagerNewInputFlagName(sk.name, inp.name), type = ref("boolean", PrimitiveType) ⟩$_{Variable}$ } ∪ { otp: sk.outputs • ⟨ name = SkillsManagerNewInputFlagName(sk.name, ot.name), type = ref("boolean", PrimitiveType) ⟩$_{Variable}$ } } ∪ ⋃ { sk: skls ∣ sk ∈ DSkill • { act: sk.inputs • ⟨ name = SkillsManagerNewInputFlagName(sk.name, act.name), type = ref("boolean", PrimitiveType) ⟩$_{Variable}$ } }

FUNCTION B.3.4

ⓐ	**Name**	SkillsManagerParameterVariables
ⓑ	**Parameter** *name:type*	skill: Name
ⓒ	**Result** *name:type*	smpvs: Set(Variable)
ⓔ	**Definition**	smpvs = ⋃ { sk: skls • { pm: sk.parameters • ⦇ name = SkillsManagerVarParameterName(sk.name, pm.name) type = pm.type ⦈ *Variable* } }

FUNCTION B.3.5

ⓐ	**Name**	SkillsManagerInputOutputVariables
ⓑ	**Parameter** *name:type*	skill: Name
ⓒ	**Result** *name:type*	smiovs: Set(Variable)
ⓔ	**Definition**	smiovs = ⋃ { sk: skls ∣ sk ∈ CSkill • { inp: sk.inputs • ⦇ name = SkillsManagerInputOutputVariable-Name(sk.name, inp.name),

Continued on next page

FUNCTION B.3.5 – continued from previous page

```
                type =
                ⟪ types = { inp.type, ref("nat", PrimitiveType) }
                  ⟫ProductType
    ⟫Variable }
    ∪
    { otp: sk.outputs •
          ⟪ name = SkillsManagerInputOutputVariable-
Name(
                                          sk.name, otp.name ),
          type =
            ⟪ types = { otp.type, ref("nat", PrimitiveType) }
              ⟫ProductType
    ⟫Variable }
}
∪
⋃ { sk: skls | sk ∈ DSkill •
    { sdt: sk.outputs •
          ⟪ name = SkillsManagerInputOutputVariable-
Name(
                                          sk.name, sdt.name ),
          type =
            ⟪ types = { sdt.type, ref("nat", PrimitiveType) }
              ⟫ProductType
    ⟫Variable }
    ∪
    { act: sk.inputs •
          ⟪ name = SkillsManagerInputOutputVariable-
Name(
                                          sk.name, act.name ),
          type =
            ⟪ types = { act.type, ref("nat", PrimitiveType) }
              ⟫ProductType
    ⟫Variable }
}
```

FUNCTION B.3.6

ⓐ	**Name**	SkillsManagerConstants
ⓑ	**Parameter** *name:type*	-
ⓒ	**Result** *name:type*	smcs: Set(Variable)
ⓔ	**Definition**	smcs = { ⟪ name = SkillsManagerConstRequestHandlingDelay-Name(), type = ref("nat", PrimitiveType) ⟫$_{Variable}$, ⟪ name= SkillsManagerConstAsyncSkillsName(), type = ⟪ domain = ref("Skills", Enumeration) ⟫$_{SeqType}$ ⟫$_{Variable}$ }

FUNCTION B.3.7

ⓐ	**Name**	SkillsManagerPatternStates
ⓑ	**Parameter** *name:type*	skls: Set(Skill), scons: Set(SkillConnection), smons: Set(Monitor)
ⓒ	**Result** *name:type*	smps: Set(State)

Continued on next page

FUNCTION B.3.7 – continued from previous page

ⓔ	**Definition**	smps = { ⟪ name = "Initial" ⟫$_{Initial}$, ⟪ name = SkillsManagerInitialiseName(), actions = { ⟪ action= SkillsManagerInitialiseStatements(skls) ⟫$_{EntryAction}$ } ⟫$_{State}$, ⟪ name = SkillsManagerHandleRequestsName()⟫$_{State}$, SkillsManagerDoNextSkillState(), SkillsManagerUpdateRecordState(skls, scons), SkillsManagerCheckMonitorsState(smons) } ∪ SkillsManagerExecutingStates(skls)

FUNCTION B.3.8

ⓐ	**Name**	SkillsManagerJunctionEmptyCycleSkillsName
ⓑ	**Parameter** *name:type*	-
ⓒ	**Result** *name:type*	jname: Name
ⓔ	**Definition**	jname = "JunctionEmptyCycleSkills"

FUNCTION B.3.9

ⓐ	**Name**	SkillsManagerPatternTransitions
ⓑ	**Parameter** *name:type*	skls: Set(Skill)
ⓒ	**Result** *name:type*	smpt: Set(Transitions)
ⓔ	**Definition**	{ ⟨ name = SkillsManagerInitialName() + "To" + SkillsManagerInitialiseName(), source = ref(SkillsManagerInitialName(), State), target = ref(SkillsManagerInitialiseName(), State) ⟩*Transition* , ⟨ name = SkillsManagerInitialiseName() + "To" + SkillsManagerHandleRequestsName() , source = ref(SkillsManagerInitialiseName(), State), target = ref(SkillsManagerHandleRequestsName(), State) ⟩*Transition* , ⟨ name = SkillsManagerHandleRequestsName() + "To" + SkillsManagerDoNextSkillName(), source = ref(SkillsManagerHandleSetParameterName(), State), target = ref(SkillsManagerDoNextSkillName(), State), condition= SkillsManagerHandleTimeGuardExpression(), action= SkillsManagerResetRequestClock() ⟩*Transition* ,

Continued on next page

FUNCTION B.3.9 – continued from previous page

```
    ⟪ name = SkillsManagerUpdateRecordName() + "To"
+
                SkillsManagerCheckMonitorsName() ,
       source = ref(SkillsManagerUpdateRecordName(),
State),
       target = ref(SkillsManagerCheckMonitorsName(),
State),
      condition =
       ref(SkillsManagerVarUpdateRecordCompleteName(),
           Variable)
     ⟫Transition ,
    ⟪ name = SkillsManagerCheckMonitorsName() + "To"
+
                SkillsManagerCleanupName() ,
      source = ref(SkillsManagerCheckMonitorsName(),
                   State),
      target = ref(SkillsManagerCleanupName(), State)
      condition =
        ⟪ left = ref(SkillsManagerVarCycleSkillsName(),
           Variable),
          right = ⟪ values = ⟨⟩ ⟫SeqExp
        ⟫Equals
     ⟫Transition ,
    ⟪ name = SkillsManagerCheckMonitorsName() + "To"
+
                SkillsManagerDoNextSkillName() ,
      source = ref(SkillsManagerCheckMonitorsName(),
                   State),
      target = ref(SkillsManagerDoNextSkillName(), State)
      condition =
        ⟪ function= ref("isNonEmpty", Function) ,
          args= ref(SkillsManagerVarCycleSkillsName(),
                    Variable) ⟫CallExp
```

Continued on next page

FUNCTION B.3.9 – continued from previous page

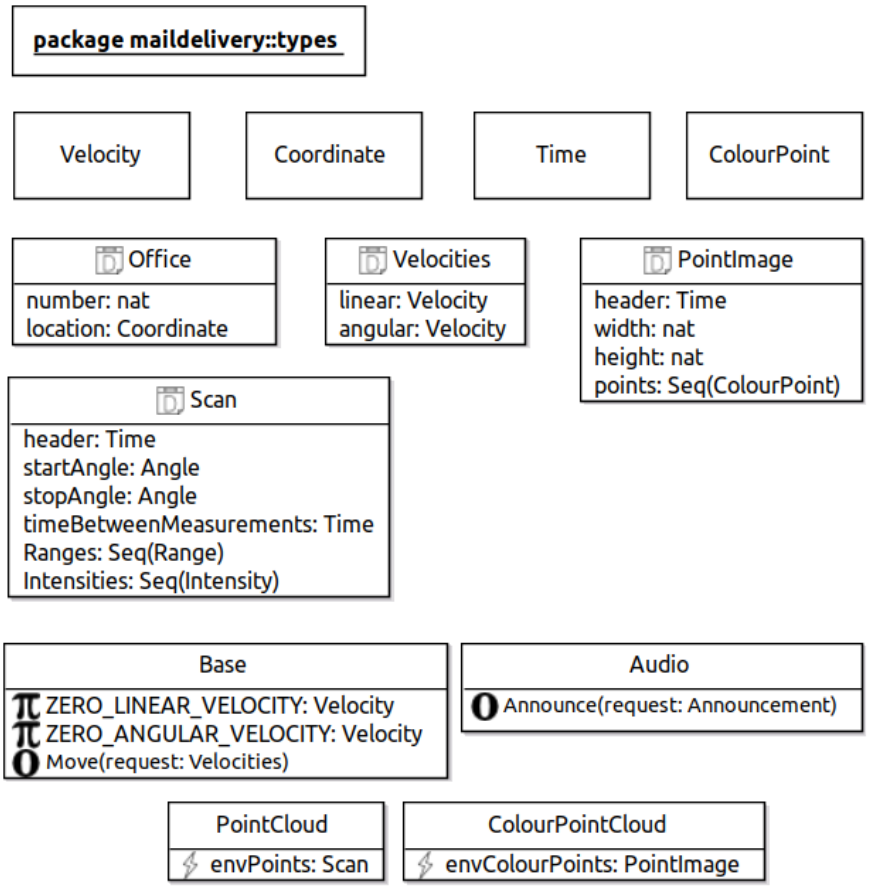
	action = ⟨| left= ref(SkillsManagerVarCycleSkillsName(), Variable) right= ⟨| function= ref("tail", Function) , args= ref(SkillsManagerVarCycleSkillsName(), Variable) |⟩$_{CallExp}$ |⟩$_{Assignment}$ |⟩$_{Transition}$, ⟨| name = SkillsManagerCleanupName() + "To" + SkillsManagerHandleValueQueriesName() , source = ref(SkillsManagerCleanupName(), State), target = ref(SkillsManagerHandleValueQueriesName(), State), action = SkillsManagerResetRequestClock() |⟩$_{Transition}$ } ∪ SkillsManagerExecutingSkillsTransitions(skls) ∪ SkillsManagerHandlerTransitions(skls)

FUNCTION B.3.10

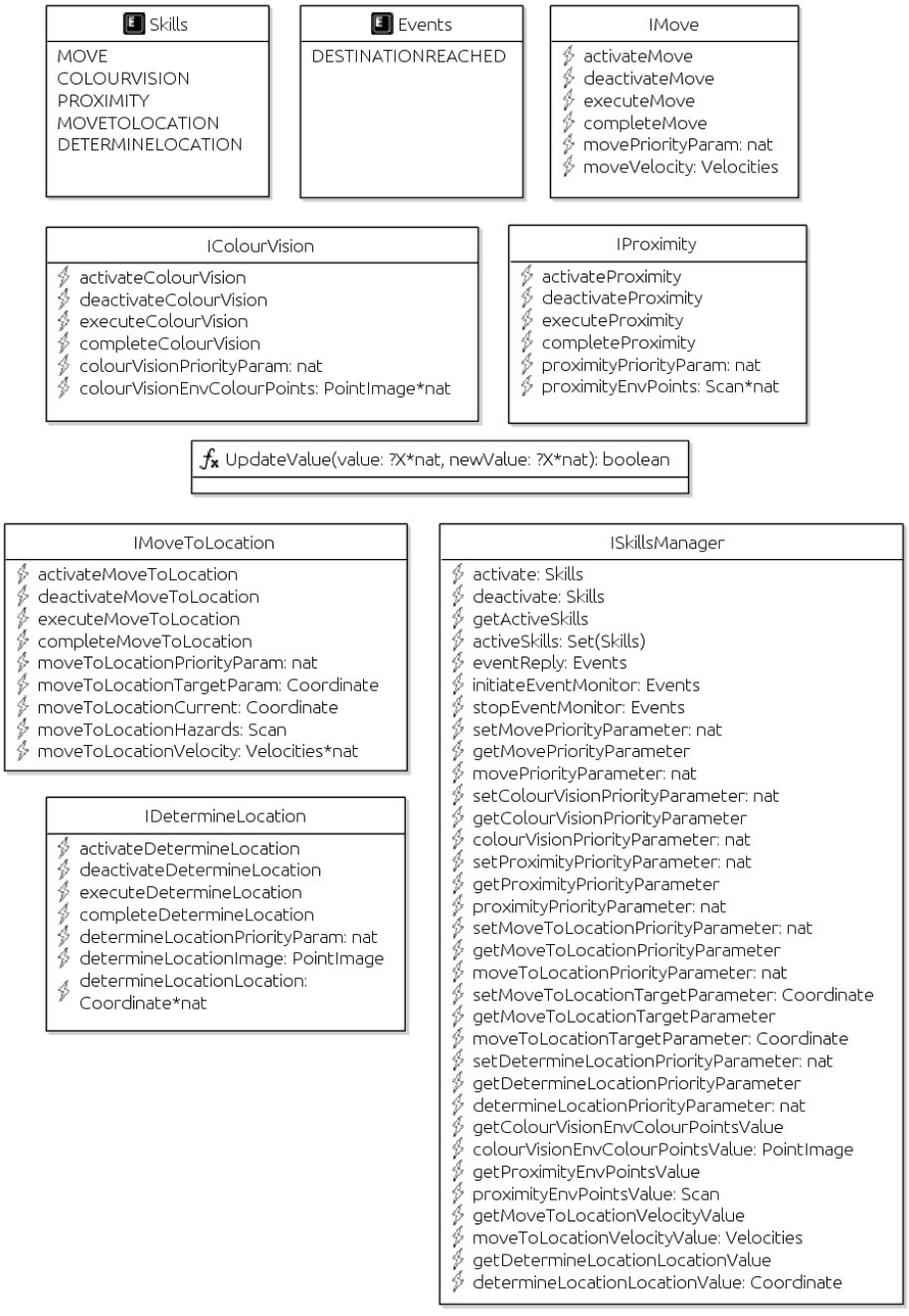
ⓐ	**Name**	SkillsManagerClockRequestHandlingName
ⓑ	**Parameter** *name:type*	-
ⓒ	**Result** *name:type*	vname: Name
ⓔ	**Definition**	sname = "requestHandling"

C Mail Delivery Case Study

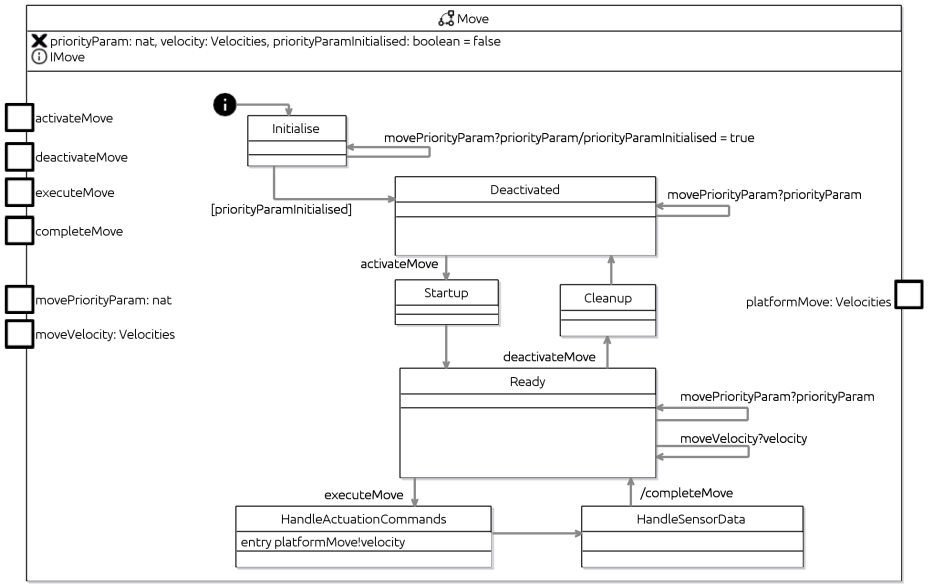
C.1 Types



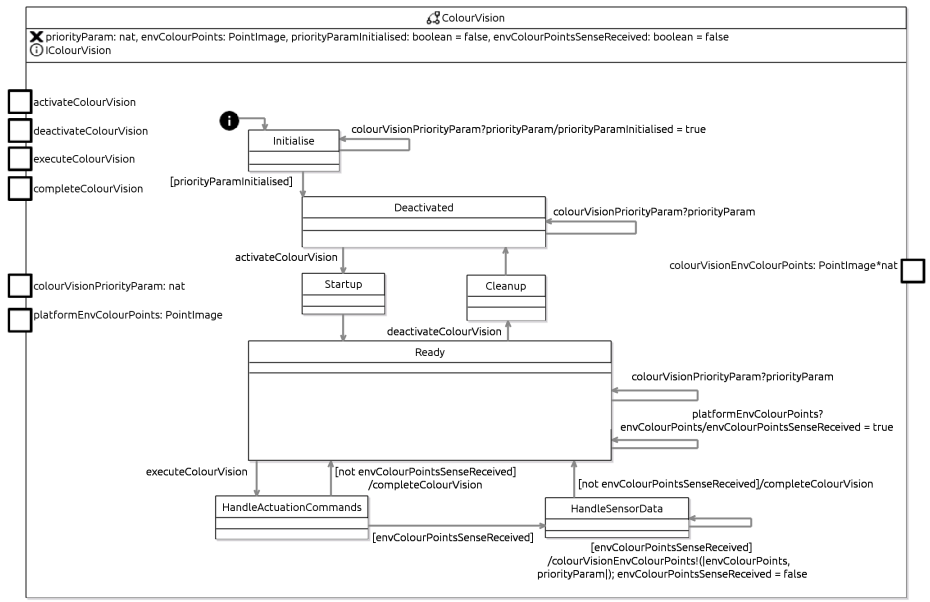
C.2 Reactive Skills: Types



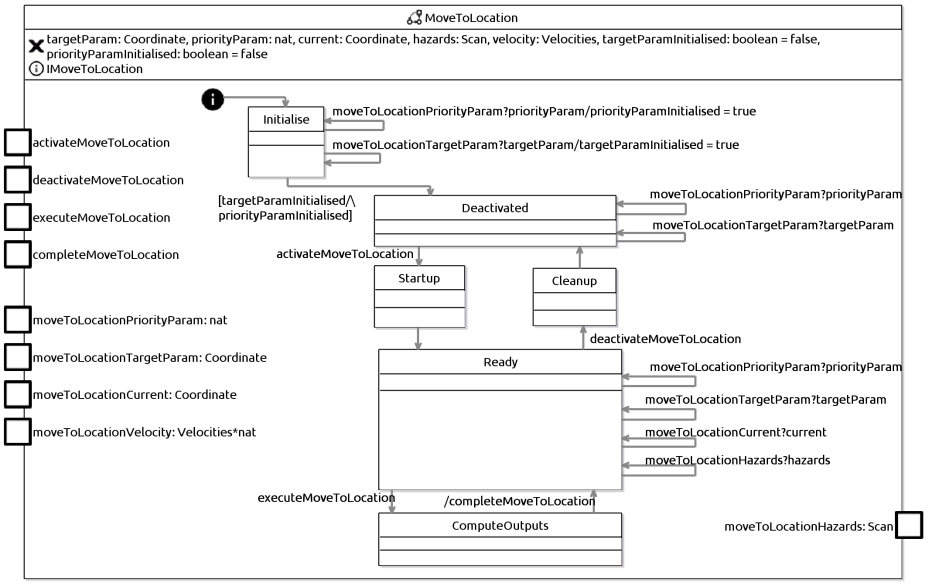
C.3 Reactive Skills: Move D-Skill Machine



C.4 Reactive Skills: ColourVision D-Skill Machine



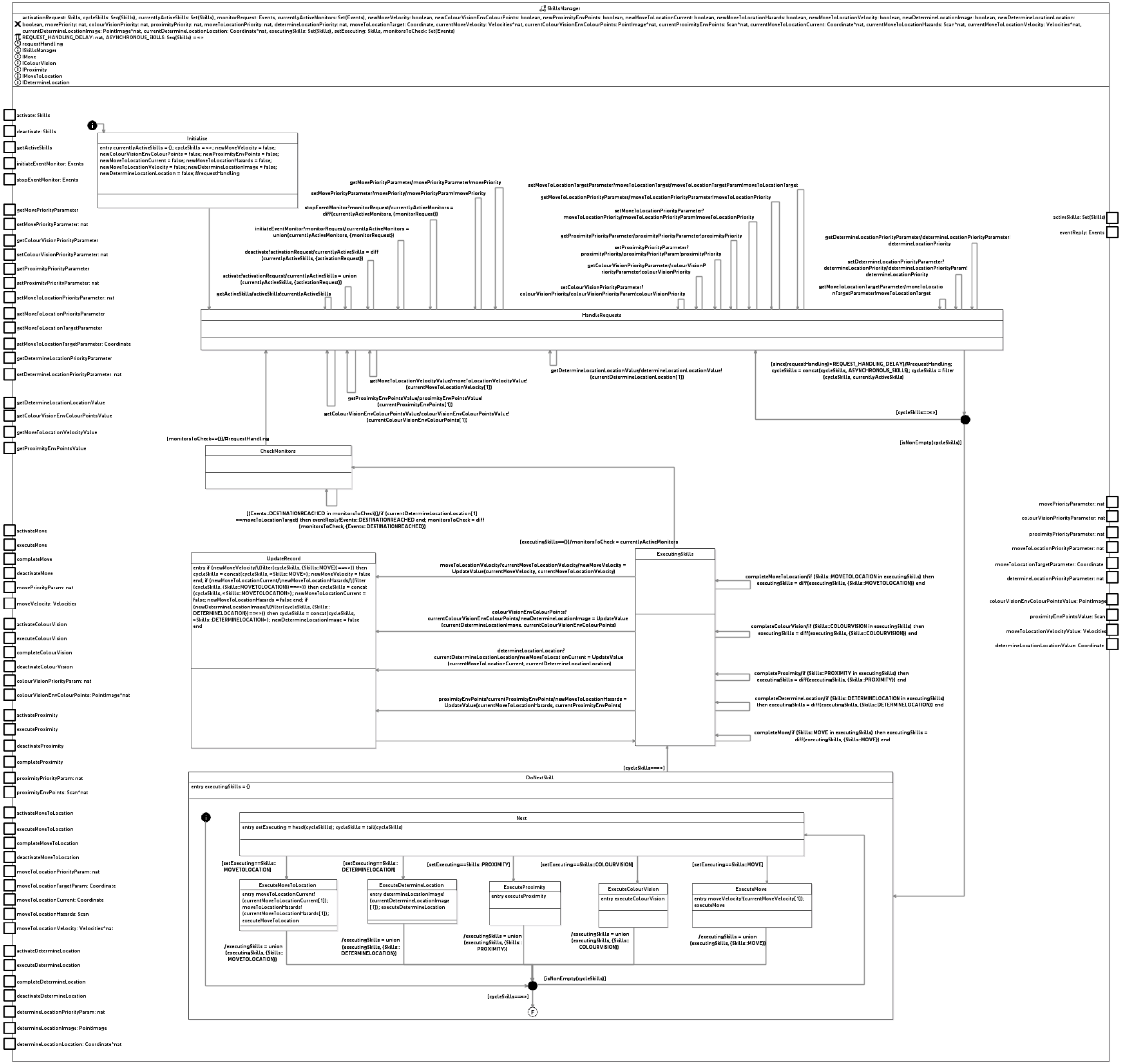
C.5 Reactive Skills: MoveToLocation C-Skill Machine



C.6 Reactive Skills: SkillsManager Machine

Fold the following page and adhere here as a foldout.

Folded size: (12cm, 8cm)


SkillsManager
activationRequest: Skills, cycleSkills: Seq(Skills), currentlyActiveSkills: Set(Skills), monitorRequest: Events, currentlyActiveMonitors: Set(Events), newMoveVelocity: boolean, newColourVisionEnvColourPoints: boolean, newProximityEnvPoints: boolean, newMoveToLocationCurrent: boolean, newMoveToLocationHazards: boolean, newMoveToLocationVelocity: boolean, newDetermineLocationImage: boolean, newDetermineLocationLocation: boolean, movePriority: nat, colourVisionPriority: nat, proximityPriority: nat, moveToLocationPriority: nat, determineLocationPriority: nat, moveToLocationTarget: Coordinate, currentMoveVelocity: Velocities*nat, currentColourVisionEnvColourPoints: PointImage*nat, currentProximityEnvPoints: Scan*nat, currentMoveToLocationCurrent: Coordinate*nat, currentMoveToLocationHazards: Scan*nat, currentMoveToLocationVelocity: Velocities*nat, currentDetermineLocationImage: PointImage*nat, currentDetermineLocationLocation: Coordinate*nat, executingSkills: Set(Skills), setExecuting: Skills, monitorsToCheck: Set(Events)
REQUEST_HANDLING_DELAY: nat, ASYNCHRONOUS_SKILLS: Seq(Skills) = <>
requestHandling
ISkillsManager
IMove
IColourVision
IProximity
IMoveToLocation
IDetermineLocation
activate: Skills
deactivate: Skills
getActiveSkills
initiateEventMonitor: Events
stopEventMonitor: Events
getMovePriorityParameter
setMovePriorityParameter: nat
getColourVisionPriorityParameter
setColourVisionPriorityParameter: nat
getProximityPriorityParameter
setProximityPriorityParameter: nat
setMoveToLocationPriorityParameter: nat
getMoveToLocationPriorityParameter
getMoveToLocationTargetParameter
setMoveToLocationTargetParameter: Coordinate
getDetermineLocationPriorityParameter
setDetermineLocationPriorityParameter: nat
getDetermineLocationLocationValue
getColourVisionEnvColourPointsValue
getMoveToLocationVelocityValue
getProximityEnvPointsValue
activateMove
executeMove
completeMove
deactivateMove
movePriorityParam: nat
moveVelocity: Velocities
activateColourVision
executeColourVision
completeColourVision
deactivateColourVision
colourVisionPriorityParam: nat
colourVisionEnvColourPoints: PointImage*nat
activateProximity
executeProximity
deactivateProximity
completeProximity
proximityPriorityParam: nat
proximityEnvPoints: Scan*nat
activateMoveToLocation
executeMoveToLocation
completeMoveToLocation
deactivateMoveToLocation
moveToLocationPriorityParam: nat
moveToLocationTargetParam: Coordinate
moveToLocationCurrent: Coordinate
moveToLocationHazards: Scan
moveToLocationVelocity: Velocities*nat
activateDetermineLocation
executeDetermineLocation
completeDetermineLocation
deactivateDetermineLocation
determineLocationPriorityParam: nat
determineLocationImage: PointImage
determineLocationLocation: Coordinate*nat
activeSkills: Set(Skills)
eventReply: Events
movePriorityParameter: nat
colourVisionPriorityParameter: nat
proximityPriorityParameter: nat
moveToLocationPriorityParameter: nat
moveToLocationTargetParameter: Coordinate
determineLocationPriorityParameter: nat
colourVisionEnvColourPointsValue: PointImage
proximityEnvPointsValue: Scan
moveToLocationVelocityValue: Velocities
determineLocationLocationValue: Coordinate
Initialise
entry currentlyActiveSkills = {}; cycleSkills = <>; newMoveVelocity = false; newColourVisionEnvColourPoints = false; newProximityEnvPoints = false; newMoveToLocationCurrent = false; newMoveToLocationHazards = false; newMoveToLocationVelocity = false; newDetermineLocationImage = false; newDetermineLocationLocation = false; #requestHandling
getMovePriorityParameter/movePriorityParameter!movePriority
setMovePriorityParameter?movePriority/movePriorityParam!movePriority
stopEventMonitor?monitorRequest/currentlyActiveMonitors = diff(currentlyActiveMonitors, {monitorRequest})
initiateEventMonitor?monitorRequest/currentlyActiveMonitors = union(currentlyActiveMonitors, {monitorRequest})
deactivate?activationRequest/currentlyActiveSkills = diff (currentlyActiveSkills, {activationRequest})
activate?activationRequest/currentlyActiveSkills = union (currentlyActiveSkills, {activationRequest})
getActiveSkills/activeSkills!currentlyActiveSkills
setMoveToLocationTargetParameter?moveToLocationTarget/moveToLocationTargetParam!moveToLocationTarget
getMoveToLocationPriorityParameter/moveToLocationPriorityParameter!moveToLocationPriority
setMoveToLocationPriorityParameter?moveToLocationPriority/moveToLocationPriorityParam!moveToLocationPriority
getProximityPriorityParameter/proximityPriorityParameter!proximityPriority
setProximityPriorityParameter?proximityPriority/proximityPriorityParam!proximityPriority
getColourVisionPriorityParameter/colourVisionPriorityParameter!colourVisionPriority
setColourVisionPriorityParameter?colourVisionPriority/colourVisionPriorityParam!colourVisionPriority
getDetermineLocationPriorityParameter/determineLocationPriorityParameter!determineLocationPriority
setDetermineLocationPriorityParameter?determineLocationPriority/determineLocationPriorityParam!determineLocationPriority
getMoveToLocationTargetParameter/moveToLocationTargetParameter!moveToLocationTarget
HandleRequests
getMoveToLocationVelocityValue/moveToLocationVelocityValue!(currentMoveToLocationVelocity[1])
getProximityEnvPointsValue/proximityEnvPointsValue!(currentProximityEnvPoints[1])
getColourVisionEnvColourPointsValue/colourVisionEnvColourPointsValue!(currentColourVisionEnvColourPoints[1])
getDetermineLocationLocationValue/determineLocationLocationValue!(currentDetermineLocationLocation[1])
[since(requestHandling)>REQUEST_HANDLING_DELAY]/#requestHandling; cycleSkills = concat(cycleSkills, ASYNCHRONOUS_SKILLS); cycleSkills = filter (cycleSkills, currentlyActiveSkills)
[cycleSkills==<>]
[isNonEmpty(cycleSkills)]
[monitorsToCheck=={}]/#requestHandling
CheckMonitors
[(Events::DESTINATIONREACHED in monitorsToCheck)]/if (currentDetermineLocationLocation[1] ==moveToLocationTarget) then eventReply!Events::DESTINATIONREACHED end; monitorsToCheck = diff (monitorsToCheck, {Events::DESTINATIONREACHED})
[executingSkills=={}]/monitorsToCheck = currentlyActiveMonitors
UpdateRecord
entry if (newMoveVelocity/\(filter(cycleSkills, {Skills::MOVE})==<>)) then cycleSkills = concat(cycleSkills, <Skills::MOVE>); newMoveVelocity = false end; if (newMoveToLocationCurrent/\newMoveToLocationHazards/\(filter (cycleSkills, {Skills::MOVETOLOCATION})==<>)) then cycleSkills = concat (cycleSkills, <Skills::MOVETOLOCATION>); newMoveToLocationCurrent = false; newMoveToLocationHazards = false end; if (newDetermineLocationImage/\(filter(cycleSkills, {Skills::DETERMINELOCATION})==<>)) then cycleSkills = concat(cycleSkills, <Skills::DETERMINELOCATION>); newDetermineLocationImage = false end
ExecutingSkills
moveToLocationVelocity?currentMoveToLocationVelocity/newMoveVelocity = UpdateValue(currentMoveVelocity, currentMoveToLocationVelocity)
colourVisionEnvColourPoints?currentColourVisionEnvColourPoints/newDetermineLocationImage = UpdateValue (currentDetermineLocationImage, currentColourVisionEnvColourPoints)
determineLocationLocation?currentDetermineLocationLocation/newMoveToLocationCurrent = UpdateValue (currentMoveToLocationCurrent, currentDetermineLocationLocation)
proximityEnvPoints?currentProximityEnvPoints/newMoveToLocationHazards = UpdateValue(currentMoveToLocationHazards, currentProximityEnvPoints)
completeMoveToLocation/if (Skills::MOVETOLOCATION in executingSkills) then executingSkills = diff(executingSkills, {Skills::MOVETOLOCATION}) end
completeColourVision/if (Skills::COLOURVISION in executingSkills) then executingSkills = diff(executingSkills, {Skills::COLOURVISION}) end
completeProximity/if (Skills::PROXIMITY in executingSkills) then executingSkills = diff(executingSkills, {Skills::PROXIMITY}) end
completeDetermineLocation/if (Skills::DETERMINELOCATION in executingSkills) then executingSkills = diff(executingSkills, {Skills::DETERMINELOCATION}) end
completeMove/if (Skills::MOVE in executingSkills) then executingSkills = diff(executingSkills, {Skills::MOVE}) end
[cycleSkills==<>]
DoNextSkill
entry executingSkills = {}
Next
entry setExecuting = head(cycleSkills); cycleSkills = tail(cycleSkills)
[setExecuting==Skills::MOVETOLOCATION]
[setExecuting==Skills::DETERMINELOCATION]
[setExecuting==Skills::PROXIMITY]
[setExecuting==Skills::COLOURVISION]
[setExecuting==Skills::MOVE]
ExecuteMoveToLocation
entry moveToLocationCurrent!(currentMoveToLocationCurrent[1]); moveToLocationHazards!(currentMoveToLocationHazards[1]); executeMoveToLocation
ExecuteDetermineLocation
entry determineLocationImage!(currentDetermineLocationImage[1]); executeDetermineLocation
ExecuteProximity
entry executeProximity
ExecuteColourVision
entry executeColourVision
ExecuteMove
entry moveVelocity!(currentMoveVelocity[1]); executeMove
/executingSkills = union (executingSkills, {Skills::MOVETOLOCATION})
/executingSkills = union (executingSkills, {Skills::DETERMINELOCATION})
/executingSkills = union (executingSkills, {Skills::PROXIMITY})
/executingSkills = union (executingSkills, {Skills::COLOURVISION})
/executingSkills = union (executingSkills, {Skills::MOVE})
[isNonEmpty(cycleSkills)]
[cycleSkills==<>]
F

D Obstacle Avoidance Case Study

The RoboArch description and the RoboChart and CSP models used for property verification of the assertions in Sections D.2 to D.5 can be found at the following website https://github.com/UoY-RoboStar/roboarch-examples.

D.1 Module

Fold the following page and adhere here as a foldout.

Folded size: (12cm, 15cm)

ObstacleAvoidance

explorePriorityParameter: nat
getExplorePriorityParameter
setExplorePriorityParameter: nat
proximityPriorityParameter: nat
getProximityPriorityParameter
setProximityPriorityParameter: nat
exploreMaxSpeedParameter: real
getExploreMaxSpeedParameter
setExploreMaxSpeedParameter: real
exploreSafetyDistanceParameter: int
getExploreSafetyDistanceParameter
setExploreSafetyDistanceParameter: int
movePriorityParameter: nat
getMovePriorityParameter
setMovePriorityParameter: nat

ref Executive::Executive

exploreVOutValue: Velocities
getExploreVOutValue
proximityGapValue: int
getProximityGapValue

activate: Skills
deactivate: Skills
getActiveSkills
activeSkills: Set(Skills)
eventReply: Events
initiateEventMonitor: Events
stopEventMonitor: Events

async

activate: Skills
deactivate: Skills
getActiveSkills
activeSkills: Set(Skills)
eventReply: Events
initiateEventMonitor: Events
stopEventMonitor: Events

setMovePriorityParameter: nat
getMovePriorityParameter
movePriorityParameter: nat
setExploreSafetyDistanceParameter: int
getExploreSafetyDistanceParameter
exploreSafetyDistanceParameter: int
setExploreMaxSpeedParameter: real
getExploreMaxSpeedParameter
exploreMaxSpeedParameter: real
setProximityPriorityParameter: nat
getProximityPriorityParameter
proximityPriorityParameter: nat
setExplorePriorityParameter: nat
getExplorePriorityParameter
explorePriorityParameter: nat

getProximityGapValue
proximityGapValue: int
getExploreVOutValue
exploreVOutValue: Velocities

ref Control::Control

proximity: int
move: Velocities

proximity: int
move: Velocities

ref Platform::PuckRobot

D.2 Skill Explore Assertions

```
csp ExploreSkillEvents csp-begin
  ... See 5.2.3 for skill events
csp-end

csp EBS1  csp-begin
  ... See E.1 for EBS1 process
csp-end

csp EBS2 csp-begin
  ... See E.1 for EBS2 process
csp-end

csp EBS3 csp-begin
  ... See E.1 for EBS3 process
csp-end

csp EBS4 csp-begin
  ... See E.1 for EBS4 process
csp-end

csp EBS5 csp-begin
  ... See E.1 for EBS5 process
csp-end

csp EBS6 csp-begin
  ... See E.1 for EBS6 process
csp-end

csp EBCS1 csp-begin
  ... See E.2 for EBCS1 process
csp-end


assertion EBS1:
  Control::CSkill::Explore::Explore refines EBS1 in the failures
    model

assertion EBS2:
  Control::CSkill::Explore::Explore refines EBS2 in the failures
    model

assertion EBS3:
  Control::CSkill::Explore::Explore refines EBS3 in the failures
    model

assertion EBS4:
```

```
44   Control::CSkill::Explore::Explore refines EBS4 in the failures
       model
45
46 assertion EBS5:
47   Control::CSkill::Explore::Explore refines EBS5 in the failures
       model
48
49 assertion EBS6:
50   Control::CSkill::Explore::Explore refines EBS6 in the failures
       model
51
52 assertion EBCS1:
53   Control::CSkill::Explore::Explore refines EBCS1 in the
       failures model
```

D.3 Skill Proximity Assertions

```
1  csp ProximitySkillEvents csp-begin
2    ... See 5.2.3 for skill events
3  csp-end
4
5  csp EBDS1 csp-begin
6    ... See E.3 for EBDS1 process
7  csp-end
8
9  csp EBDS3 csp-begin
10   ... See E.3 for EBDS3 process
11 csp-end
12
13 assertion EBDS1:
14   Control::DSkill::Proximity::Proximity refines EBDS1 in the
       failures model
15
16 // EBDS2 - not applicable to Proximity as it receives no
       actuation command values from the skills manager.
17
18 assertion EBDS3:
19   Control::DSkill::Proximity::Proximity refines EBDS3 in the
       failures model
```

D.4 Skill Move Assertions

```
1  csp MoveSkillEvents csp-begin
2    ... See 5.2.3 for skill events
```

```
csp-end

csp EBDS1 csp-begin
  ... See E.3 for EBDS1 process
csp-end

csp EBDS2 csp-begin
  ... See E.3 for EBDS2 process
csp-end


assertion EBDS1:
  Control::DSkill::Move::Move refines EBDS1 in the failures
    model

assertion EBDS2:
  Control::DSkill::Move::Move refines EBDS2 in the failures
    model

// EBDS3 - not applicable to Move as it receives no sensor
    values from the platform.


```

D.5 Skills Manager Assertions

```
csp SkillsManagerEvents csp-begin
  ... See 5.2.3 for skills manager events
csp-end


csp EBSM1 csp-begin
  ... See E.4 for EBSM1 process
csp-end

csp EBSM2_ExplorePriority csp-begin
  ... See E.4 for EBSM2 process
csp-end

csp EBSM3_ExplorePriority csp-begin
  ... See E.4 for EBSM3 process
csp-end

csp EBSM4 csp-begin
  ... See E.4 for EBDS4 process
csp-end

```

```
22 assertion EBSM1:
23   Control::SkillsManager::SkillsManager refines EBSM1 in the
       failures model
24
25 assertion EBSM2_ExplorePriority:
26   Control::SkillsManager::SkillsManager refines
       EBSM2_ExplorePriority in the failures model
27
28 assertion EBSM3_ExplorePriority:
29   Control::SkillsManager::SkillsManager refines
       EBSM3_ExplorePriority in the failures model
30
31 assertion EBSM4:
32   Control::SkillsManager::SkillsManager refines EBSM4 in the
       failures model
```

E Reactive Skills Properties

This appendix presents the CSP processes described in Section 5.2.3 that were used for verifying the properties of the generated reactive skills RoboChart models. Section E.1 contains the process definitions for properties common to all skills. Section E.2 contains the process definitions for properties of C-Skills. Section E.3 contains the process definitions for properties of D-Skills. Finally, Section E.4 contains the process definitions for properties of the Skills Manager.

The process definitions of the properties, unless straightforward, have a three-part structure: *PRE*, *POS*, and *CON*. The *PRE* portion specifies the events that can occur before the property of interest. The *CON* portion specifies the events that can occur at the time the property of interest occurs. The *POS* portion specifies the events that can occur after the property of interest has concluded. This means that the process defining a property will initially behave as *PRE*. When the condition of interest occurs, the process then behaves as *CON*. Once the condition of interest's behavior has been fulfilled, the process behaves as *POS*, where any event is permissible.

E.1 Skills

EB-S1

A skill initially accepts a value for any of the parameters.

$$EBS1 = \Box\, ev : skillParameterEvents \bullet ev \longrightarrow CHAOS(allSkillEvents)$$

EBS1 $\sqsubseteq_F$ Skill

EB-S2

After accepting one of each parameter value the skill accepts an activate event and any of its parameter values.

$$
\begin{aligned}
&EBS2 = \text{let}\\
&\quad PRE(\{\}) = \Box\, ev : skillParameterEvents \bullet ev \longrightarrow PRE(\{ev\})\\
&\quad PRE(evs) = \text{if}(\ (evs \cap \{|sp_1|\} \neq \varnothing) \wedge (evs \cap \{|sp_2|\} \neq \varnothing) \wedge (evs \cap \{|sp_3|\} \neq \varnothing)) \text{ then}\\
&\qquad\qquad CON\\
&\qquad\quad \text{else}\\
&\qquad\qquad \Box\, ev : skillParameterEvents \bullet ev \longrightarrow PRE(\{ev\} \cup evs)\\
&\qquad CON = \Box\, ev : skillParameterEvents \bullet ev \longrightarrow CON\\
&\qquad\quad \Box\\
&\qquad\quad activate \longrightarrow POS\\
&\qquad POS = CHAOS(allSkillEvents)\\
&\text{within } PRE(\{\})
\end{aligned}
$$

EBS2 $\sqsubseteq_F$ Skill

EB-S3

After accepting an activate event a skill accepts any of its parameter values, input values, an execute event or deactivate event.

$$
\begin{aligned}
&EBS3 = \text{let}\\
&\qquad PRE = \sqcap\, ev : (allSkillEvents \backslash activate) \bullet ev \longrightarrow PRE\\
&\qquad\quad \sqcap\, activate \longrightarrow CON\\
&\qquad CON = \Box\, ev : (skillParameterEvents \cup skillInputEvents) \bullet ev \longrightarrow CON\\
&\qquad\quad \Box
\end{aligned}
$$

$$
\begin{array}{l}
\quad\quad\quad\quad execute \longrightarrow POS \\
\quad\quad\quad\quad \Box \\
\quad\quad\quad\quad deactivate \longrightarrow POS \\
\quad\quad POS = CHAOS(allSkillEvents) \\
\text{within } PRE
\end{array}
$$

$EBS_3 \sqsubseteq_F$ Skill

EB-S4

After accepting an execute event a skill must output a complete event before any of the other skill control events.

$$
\begin{array}{l}
EBS4 = \text{let} \\
\quad\quad PRE = \sqcap ev : (allSkillEvents \setminus \{execute\}) \bullet ev \longrightarrow PRE \\
\quad\quad\quad\quad \sqcap \\
\quad\quad\quad\quad execute \longrightarrow CON \\
\\
\quad\quad CON = \sqcap ev : (allSkillEvents \setminus skillControlEvents) \bullet ev \longrightarrow CON \\
\quad\quad\quad\quad \sqcap \\
\quad\quad\quad\quad complete \longrightarrow POS \\
\quad\quad POS = CHAOS(allSkillEvents) \\
\text{within } PRE
\end{array}
$$

$EBS_4 \sqsubseteq_F$ Skill

EB-S5

After accepting a deactivate event a skill can only accept its parameter values or an activate event.

$$
\begin{aligned}
EBS5 = \text{let}\\
\quad PRE &= \sqcap ev : (allSkillEvents \setminus \{deactivate\}) \bullet ev \longrightarrow PRE\\
&\quad \sqcap\ deactivate \longrightarrow CON\\
\\
CON &= \Box ev : (skillParameterEvents) \bullet ev \longrightarrow CON\\
&\quad \Box\\
&\quad activate \longrightarrow POS\\
POS &= CHAOS(allSkillEvents)\\
\text{within } PRE
\end{aligned}
$$

$EBS5 \sqsubseteq_F$ Skill

EB-S6

No skill outputs are communicated until the skill's priority parameter event is accepted. After accepting a priority parameter event, the skill's output value always includes the most recently accepted priority value.

$sp_{priority} \in skillParameterEvents$

$$
\begin{aligned}
EBS6 = \text{let}\\
\quad PRE &= \sqcap ev : (allSkillEvents \setminus (skillOutputEvents \cup \{|sp_{priority}|\})\ ) \bullet\\
&\quad ev \longrightarrow PRE\\
&\quad \sqcap\\
&\quad sp_{priority}?p \longrightarrow CON(p)
\end{aligned}
$$

$$
\begin{aligned}
CON(priority) = {} & \sqcap ev : (allSkillEvents \setminus (skillOutputEvents \cup \{|sp_{priority}|\})) \bullet \\
& \quad ev \longrightarrow CON(priority) \\
& \sqcap \\
& \sqcap ev : \{v : T_{v_{op_1}} \bullet op_1.(v, priority)\} \bullet ev \longrightarrow CON(priority) \\
& \sqcap \\
& \sqcap ev : \{v : T_{v_{op_2}} \bullet op_2.(v, priority)\} \bullet ev \longrightarrow CON(priority) \\
& \sqcap \\
& \sqcap ev : \{v : T_{v_{op_3}} \bullet op_3.(v, priority)\} \bullet ev \longrightarrow CON(priority) \\
& \quad \vdots \\
& \sqcap \\
& sp_{priority}?p \longrightarrow CON(p) \\
\text{within } & PRE
\end{aligned}
$$

EBS6 $\sqsubseteq_F$ Skill

E.2 C-Skills

EB-CS1

After accepting an execute event the skill communicates zero or more output value(s) before communicating a complete event.

$$
\begin{aligned}
EBCS1 = \text{let}\ & \\
& PRE = \sqcap ev : (allSkillEvents \setminus \{execute\}) \bullet ev \longrightarrow PRE \\
& \qquad \sqcap\ execute \longrightarrow CON \\
& \\
& CON = \sqcap ev : (skillOutputEvents) \bullet ev \longrightarrow CON \\
& \qquad \sqcap \\
& \qquad complete \longrightarrow POS \\
& POS = CHAOS(allSkillEvents) \\
\text{within}\ & PRE
\end{aligned}
$$

EBCS1 $\sqsubseteq_F$ Skill

E.3 D-Skills

EB-DS1

After accepting an execute event and before communicating a complete event, a D-Skill can communicate zero or more platform actuation value(s) and zero or more output value(s).

$$
\begin{aligned}
&EBDS1 = \text{let} \\
&\qquad PRE = \sqcap ev : (allSkillEvents \backslash execute) \bullet ev \longrightarrow PRE \\
&\qquad\qquad \sqcap\ execute \longrightarrow CON \\
&\qquad CON = \sqcap ev : (skillPlatformActuationEvents \cup skillOutputEvents) \bullet ev \longrightarrow CON \\
&\qquad\qquad \sqcap \\
&\qquad\qquad complete \longrightarrow POS \\
&\qquad POS = CHAOS(allSkillEvents) \\
&\text{within } PRE
\end{aligned}
$$

$\text{EBDS1} \sqsubseteq_{\text{F}} \text{Skill}$

EB-DS2

After accepting an activate or complete event followed by an execute event, the latest input event(s) values that were accepted since the last activate or complete event, if any, are communicated to the skill's corresponding platform actuation output.

$$
\begin{aligned}
&EBDS2 = \text{let} \\
&\qquad PRE = \\
&\qquad\qquad \sqcap ev : (allSkillEvents \backslash \{activate, complete\}) \bullet ev \longrightarrow PRE \\
&\qquad\qquad\qquad \sqcap \\
&\qquad\qquad\qquad activate \longrightarrow PRE'(\langle\rangle, \langle\rangle, \langle\rangle)
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad \sqcap \\
\qquad\qquad complete \longrightarrow PRE'(\langle\rangle, \langle\rangle, \langle\rangle)
\end{array}
$$

$$
\begin{array}{l}
PRE'(ac1, ac2, ac3) = \\
\qquad \sqcap ev : (skillParameterEvents \cup skillPlatformSensingEvents) \bullet \\
\qquad\qquad ev \longrightarrow PRE'(ac1, ac2, ac3) \\
\qquad \sqcap \\
\qquad in_1?ac \longrightarrow PRE'(\mathrm{seq}\, ac, ac2, ac3) \\
\qquad \sqcap \\
\qquad in_2?ac \longrightarrow PRE'(ac1, \mathrm{seq}\, ac, ac3) \\
\qquad \sqcap \\
\qquad in_3?ac \longrightarrow PRE'(ac1, ac2, \mathrm{seq}\, ac) \\
\qquad\qquad \vdots \\
\qquad \sqcap \\
\qquad execute \longrightarrow CON(ac1, ac2, ac3) \\
\qquad \sqcap \\
\qquad deactivate \longrightarrow POS
\end{array}
$$

$$
\begin{array}{l}
CON(ac1, ac2, ac3) = \\
\qquad (OUT1(ac1) \;|||\; OUT2(ac2) \;|||\; OUT3(ac3)); \; POS
\end{array}
$$

$$
\begin{array}{l}
OUT1(ac1) = \mathrm{if}(ac1 \neq \langle\rangle) \text{ then} \\
\qquad\quad spa_1!head(ac1) \longrightarrow SKIP \\
\qquad \text{else } SKIP
\end{array}
$$

$$
\begin{aligned}
&OUT2(ac2) = \text{if}(ac2 \neq \langle\rangle) \text{ then} \\
&\qquad spa_2!head(ac2) \longrightarrow SKIP \\
&\quad \text{else } SKIP \\
&OUT3(ac3) = \text{if}(ac3 \neq \langle\rangle) \text{ then} \\
&\qquad spa_3!head(ac3) \longrightarrow SKIP \\
&\quad \text{else } SKIP \\
&POS = CHAOS(allSkillEvents)
\end{aligned}
$$

within PRE

EBDS2 $\sqsubseteq_F$ Skill

EB-DS3

After accepting an activate or complete event followed by an execute event, the latest platform sensing event(s) values that were accepted since the last activate or complete event, if any, are communicated to the corresponding output(s).

$$
\begin{aligned}
&EBDS3 = \text{let} \\
&\quad PRE = \\
&\qquad \sqcap ev : (allSkillEvents \setminus \{activate, complete\}) \bullet ev \longrightarrow PRE \\
&\qquad\qquad \sqcap \\
&\qquad\qquad activate \longrightarrow PRE'(\langle\rangle, \langle\rangle, \langle\rangle) \\
&\qquad\qquad \sqcap \\
&\qquad\qquad complete \longrightarrow PRE'(\langle\rangle, \langle\rangle, \langle\rangle) \\
&\quad PRE'(sd1, sd2, sd3) = \\
&\qquad \sqcap ev : (skillParameterEvents \cup skillInputEvents) \bullet
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad ev \longrightarrow PRE'(sd1, sd2, sd3) \\
\sqcap \\
sps_1?sd \longrightarrow PRE'(\mathrm{seq}\, sd, sd2, sd3) \\
\sqcap \\
sps_2?sd \longrightarrow PRE'(sd1, \mathrm{seq}\, sd, sd3) \\
\sqcap \\
sps_3?sd \longrightarrow PRE'(sd1, sd2, \mathrm{seq}\, sd) \\
\qquad\qquad \vdots \\
\sqcap \\
execute \longrightarrow CON(sd1, sd2, sd3) \qquad\qquad \sqcap \\
deactivate \longrightarrow POS
\end{array}
$$

$$
\begin{array}{l}
CON(sd1, sd2, sd3) = \\
\qquad (OUT1(sd1) \;|||\; OUT2(sd2) \;|||\; OUT3(sd3)); \; POS
\end{array}
$$

$$
\begin{array}{l}
OUT1(sd1) = \mathrm{if}(sd1 \neq \langle\rangle) \text{ then} \\
\qquad\qquad \sqcap ev : \{p : core\_nat \bullet so_1.(head(sd1), p\} \bullet ev \longrightarrow SKIP \\
\qquad\qquad \text{else } SKIP
\end{array}
$$

$$
\begin{array}{l}
OUT2(sd2) = \mathrm{if}(sd12 \neq \langle\rangle) \text{ then} \\
\qquad\qquad \sqcap ev : \{p : core\_nat \bullet so_2.(head(sd2), p\} \bullet ev \longrightarrow SKIP \\
\qquad\qquad \text{else } SKIP
\end{array}
$$

$$
OUT3(sd3) = \mathrm{if}(sd2 \neq \langle\rangle) \text{ then}
$$

$$\sqcap ev : \{p : core\_nat \bullet so_3.(head(sd3), p\} \bullet ev \longrightarrow SKIP$$

else *SKIP*

$$POS = CHAOS(allSkillEvents)$$

within *PRE*

EBDS$_3 \sqsubseteq_F$ Skill

E.4 Skills Manager

```
handlingEvents = Union(\{ \{| activate, deactivate, getActiveSkills,
                           initiateEventmonitor, stopEventMonitor |\},
                        skillParameterRequestEvents,
                        skillValueRequestEvents
                      \})

skillParameterRequestEvents = \{| getS1P1parameter,
                                  getS1P2parameter,
                                  getS2P1parameter,
                                  ...
                                  setS1P1parameter,
                                  setS1P2parameter,
                                  setS2P1parameter,
                                  ...
                                |\}

skillValueRequestEvents = \{| getS1V1value,
                              getS1V2value,
                              getS2V1value,
                              ...
                            |\}

responseEvents = \{| activeSkills,
                     eventReply,
```

```
                    S1P1parameter,
                    S1P2parameter,
                    S1V1value,
                    ...

                    S2P1parameter,
                    S2V1value,
                    ...

                    S3P1parameter,
                    ...
                   |\}

handlingResponseEvents = diff( responseEvents, \{|eventReply|\} )

handlingOutputEvents = Union( cSkillS1Parameters,
                              cSkillS2Parameters )

cskillInputs = \{| S1in1, S1in2,
                  ...
                  S2in1,
                  ...
                |\}

cskillOutputs = \{| S1ou1, S2ou1,
                   ...
                 |\}

dskillInputs = \{| S3ac1,
                             ...
                           |\}

dskillOutputs = \{| S3\_sd1,
                      ... |\}

skillControlEvents = \{ skillActivateEvents, skillDeactivateEvents,
                        skillExecuteEvents, skillCompleteEvents \}
skillActivateEvents = \{ activate\_s1, activate\_s2, activate\_s3 ... \}
skillDeactivateEvents = \{ deactivate\_s1, deactivate\_s2,
                           deactivate\_s3 ... \}
```

```
skillExecuteEvents = \{ execute\_s1, execute\_s2, execute\_s3 ... \}
skillCompleteEvents \{ complete\_s1, complete\_s2, complete\_s3 ... \}

allSkillsManagerEvents = Union(\{ handlingEvents, responseEvents,
                                  cskillInputs, cskillOutputs,
                                  dskillInputs,
                                  dskillOutputs,
                                  skillControlEvents
                                \})
```

EB-SM1

After accepting a getActiveSkills handling request the response is communicated via an activeSkills event.

```
EBSM1 = let
   PRE =  getActiveSkills -> CON
          |~|
          |~| ev: diff( allSkillsManagerEvents,
                        \{|getActiveSkills|\} ) @ ev -> PRE

   CON = |~| ev: \{|activeSkills|\} @ ev -> POS

   POS = CHAOS(allSkillsManagerEvents)

within PRE
-----------------------
EBSM1 [F= SkillsManager
```

EB-SM2

After accepting a get(skill.name)(param.name)Parameter handling request the response is communicated via a (skill.name)(param.name)Parameter event.

```
EBSM2_s1p1 = let
   PRE =  getS1P1Parameter -> CON
          |~|
          |~| ev: diff( allSkillsManagerEvents,
                        {|getS1P1Parameter|} ) @ ev -> PRE

   CON =  |~| ev: {|S1P1Parameter|} @ ev -> POS

   POS = CHAOS(allSkillsManagerEvents)

within PRE
 .
 .
 .

** Additional rules for each skill and parameter.

----------------------------
EBSM2_s1p1 [F= Skill
```

EB-SM3

After accepting a set(skill.name)(param.name)Parameter handling request the new parameter value is communicated via a (skill.name)(param.name)Param event.

```
EBSM3_s1p1 = let
  PRE =  setS1P1Parameter?v -> CON(v)
         |~|
         |~| ev: diff( allSkillsManagerEvents,
                       {|setS1P1Parameter|} ) @ ev -> PRE

   CON(v) = S1P1Param!v -> POS

   POS = CHAOS(allSkillsManagerEvents)

within PRE
  .
  .
  .

 ** Additional rules for each skill and parameter. \\

------------------------------
EBSM3_s1p1 [F= SkillsManager
```

EB-SM4

No handling response can happen until a handling request has been accepted.

```
EBSM4 = let
   CON = ( [] ev : handlingEvents @  ev -> CON' )
           |~|
         ( |~| ev: diff( allSkillsManagerEvents,
                         Union({handlingEvents,
                                handlingResponseEvents,
                                handlingOutputEvents})
                        ) @  ev -> CON )

   CON' = ( |~| ev: union(handlingResponseEvents,
                          handlingOutputEvents) @
                       ev -> CON |~| CON )

within CON
------------------------------
EBSM4 [T= SkillsManager
```

Acronyms

API Application Programming Interface. 42, 44, 52

CPU Central Processing Unit. 152

CSP Communicating Sequential Processes. 32, 33, 38, 40, 48, 49, 147, 148, 151, 158

DSL Domain-Specific Languages. 3, 31, 32, 39, 40, 49, 123, 124, 157–159

EBNF Extended Backus-Naur Form. 124

EGL Epsilon Generation Language. 124

EMF Eclipse Modeling Framework. 37, 124, 157

ETL Epsilon Transformation Language. 124, 125, 127–129, 157

HTN Hierarchical Task Network. 159

IDE Integrated Development Environment. 33, 35, 123

MDD Model Driven Development. 124

MDE Model Driven Engineering. 3, 40

MPS Meta Programming System. 123

OCL Object Constraint Language. 37

SPA Sense Plan Act. 9

VLF Very Low Frequency. 42

References

[1] W. Barnett, A. Cavalcanti and A. Miyazawa, "Architectural modelling for robotics: RoboArch and the CorteX example," *Frontiers in Robotics and AI*, vol. 9, 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/frobt.2022.991637.

[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] Guidehouse Inc., *Guidehouse insights leaderboard: Automated driving vehicles*, Accessed: 2020-9-26, 2020. [Online]. Available: https://guidehouseinsights.com/reports/guidehouse-insights-leaderboard-automated-driving-systems.

[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] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, en. Chichester: Wiley, 1996, vol. 1.

[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] 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.

[15] 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.

[16] 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., InTech, Feb. 2012.

[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] A. Cavalcanti, W. Barnett, J. Baxter *et al.*, "Robostar technology: A roboticist's toolbox for combined proof, simulation, and testing," in *Software Engineering for Robotics*, A. Cavalcanti, B. Dongol, R. Hierons, J. Timmis and J. Woodcock, Eds. Cham: Springer International Publishing, 2021, pp. 249–293, ISBN: 978-3-030-66494-7. DOI: 10.1007/978-3-030-66494-7_9. [Online]. Available: https://doi.org/10.1007/978-3-030-66494-7_9.

[21] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation* (Domain-Specific Modeling: Enabling Full Code Generation). Hoboken, New Jersey: John Wiley and Sons, 2008.

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

[23] "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, Dec. 2011. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2011.6129467.

[24] J. McGovern, S. W. Ambler, M. E. Stevens, J. Linn, E. K. Jo and V. Sharan, *A Practical Guide to Enterprise Architecture* (Professional Technical Reference), en. Upper Saddle River, NJ: Prentice Hall, 2004.

[25] P. Clements, F. Bachmann, L. Bass *et al.*, *Documenting Software Architectures: Views and Beyond*, second. Addison-Wesley Professional, 2011.

[26] E. Gamma, R. Helm, R. Johnson, R. E. . Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (professional computing series), en. London: Addison-Wesley, 1995.

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

[28] 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)*, May 2018, pp. 93–100.

[29] P. Backes, K. Edelberg, P. Vieira *et al.*, Eds., *The intelligent robotics system architecture applied to robotics testbeds and research platforms*, vol. 2018-March, IEEE Computer Society, 2018.

[30] 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)*, Apr. 2018, pp. 75–7509.

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

[32] 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, Dec. 2017.

[33] 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, Jul. 2006.

[34] 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.

[35] 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, Dec. 2004.

[36] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras and H. Das, Eds., *The CLARAty architecture for robotic autonomy*, vol. 1, Big Sky, MT, 2001.

[37] 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.

[38] 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.

[39] 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.

[40] J.-J. Borrelly, E. Coste-Manière, B. Espiau *et al.*, "The ORCCAD architecture," *Int. J. Rob. Res.*, vol. 17, no. 4, pp. 338–359, Apr. 1998.

[41] 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, Jun. 1995.

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

[43] 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, Nov. 1993.

[44] 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.

[45] 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., Aug. 1991, pp. 1225–1231.

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

[47] 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*, Oct. 1989.

[48] R. C. Arkin, "Towards cosmopolitan robots: Intelligent navigation in extended Man-Made environments," Ph.D. dissertation, University of Massachusetts, 1987.

[49] 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.

[50] 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, Mar. 1998.

[51] S. Bensalem, L. de Silva, F. Ingrand and R. Yan, *A verifiable and Correct-by-Construction controller for robot functional levels*, 2013. [Online]. Available: http://arxiv.org/abs/1309.0442.

[52] 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, Sep. 2003.

[53] I. A. D. Nesnas, R. Simmons, D. Gaines *et al.*, "CLARAty: Challenges and steps toward reusable robotic software," *Int. J. Adv. Rob. Syst.*, vol. 3, no. 1, pp. 023–030, 2006.

[54] T. Huntsberger, P. Pirjanian, A. Trebi-Ollennu *et al.*, "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.

[55] 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, Apr. 2000, pp. 300–307.

[56] 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, Dec. 2016.

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

[58] 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.

[59] C. Wong, D. Kortenkamp and M. Speich, "A mobile robot that recognizes people," in *Proceedings of 7th IEEE International Conference on Tools with Artificial Intelligence*, Nov. 1995, pp. 346–353. [Online]. Available: http://dx.doi.org/10.1109/TAI.1995.479652.

[60] R. J. Firby, R. E. Kahn, P. N. Prokopowicz and M. J. Swain, "An architecture for vision and action," in *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1*, Morgan Kaufmann Publishers Inc., Aug. 1995, pp. 72–79. [Online]. Available: http://dl.acm.org/citation.cfm?id=1625855.1625865.

[61] R. J. Firby, M. G. Slack and C. Drive, "Task execution: Interfacing to reactive skill networks," in *Lessons Learned from Implemented Software Architectures for Physical Agents: Papers from the 1995 Spring Symposium*, K. D. Hexmoor Henry, Ed., ser. Technical Report SS-95-02, Menlo Park, CA, 1995, pp. 97–111.

[62] R. P. Bonasso, R. Kerri, K. Jenks and G. Johnson, "Using the 3T architecture for tracking shuttle RMS procedures," in *Proceedings. IEEE International Joint Symposia on Intelligence and Systems (Cat. No.98EX174)*, May 1998, pp. 180–187. [Online]. Available: http://dx.doi.org/10.1109/IJSIS.1998.685440.

[63] R. A. Brooks, "A robot that walks; emergent behaviors from a carefully evolved network," in *Proceedings, 1989 International Conference on Robotics and Automation*, May 1989, 692–4+2 vol.2. [Online]. Available: http://dx.doi.org/10.1109/ROBOT.1989.100065.

[64] J. H. Connell, "A colony architecture for an artificial creature," Ph.D. dissertation, Massachusetts Institute of Technology, 1989.

[65] J. C. Posso, A. T. Sampson, J. Simpson and J. Timmis, "Process-Oriented subsumption architectures in swarm robotic systems," in *33th Communicating Process Architectures Conference, CPA 2011, organised under the auspices of WoTUG, Limerick, Ireland, June 19th, 2011*, 2011, pp. 303–316. [Online]. Available: https://doi.org/10.3233/978-1-60750-774-1-303.

[66] R. A. Brooks, "The behavior language; user's guide," Massachusetts Institute Of Technology Artificial Intelligence Lab, Tech. Rep., Apr. 1990. [Online]. Available: http://people.csail.mit.edu/brooks/papers/AIM-1227.pdf.

[67] "IEEE standard ontologies for robotics and automation," *IEEE Std 1872-2015*, pp. 1–60, Apr. 2015. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2015.7084073.

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

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

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

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

[72] 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.

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

[74] 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.

[75] 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.

[76] S. Dhouib, N. Du Lac, J.-L. Farges *et al.*, "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.

[77] 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.

[78] 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.

[79] 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, Apr. 2014.

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

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

[82] 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, Jan. 2010.

[83] V. M. Monthe, L. Nana and G. E. Kouamou, "A Model-Based approach for common representation and description of robotics software architectures," *Applied Sciences (Switzerland)*, vol. 12, no. 6, 2022. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85126939928&doi=10.3390%2fapp12062982&partnerID=40&md5=c5421118f364765efcf362c840f13b5c.

[84] R. J. Allen, "A formal approach to software architecture," Ph.D. dissertation, Carnegie Mellon University, May 1997. [Online]. Available: http://reports-archive.adm.cs.cmu.edu/anon/1997/CMU-CS-97-144.pdf.

[85] 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.

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

[87] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*, en. Prentice Hall, 1996. [Online]. Available: https://play.google.com/store/books/details?id=ua1QAAAAMAAJ.

[88] R. J. Firby, "Adaptive execution in complex dynamic worlds," Ph.D. dissertation, New Haven, CT, USA, 1989. [Online]. Available: https://dl.acm.org/citation.cfm?id=916113.

[89] F. F. Ingrand, M. P. Georgeff and A. S. Rao, "An architecture for real-time reasoning and system control," *IEEE Expert*, vol. 7, no. 6, pp. 34–44, Dec. 1992. [Online]. Available: http://dx.doi.org/10.1109/64.180407.

[90] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No.98CH36190)*, vol. 3, Oct. 1998, 1931–1937 vol.3. [Online]. Available: http://dx.doi.org/10.1109/IROS.1998.724883.

[91] V. Verma, T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons and K. Tso, "Plan execution interchange language (plexil) for executable plans and command sequences," 2005, pp. 81–88. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-28744443249&partnerID=40&md5=a059d1fe9b60c539dbf5a8671c36acc1.

[92] K. Currie and A. Tate, "O-Plan: The open planning architecture," *Artif. Intell.*, vol. 52, no. 1, pp. 49–86, 1991. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0026255231&doi=10.1016%2f0004-3702%2891%2990024-E&partnerID=40&md5=750964de11b121495bc1c68be8cc470c.

[93] R. Lallement, L. De Silva and R. Alami, "Hatp: Hierarchical agent-based task planner," cited By 0, vol. 3, 2018, pp. 1823–1825. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85054746485&partnerID=40&md5=abfbe1ecad8797e6d3fd9b1f93c20b57.

[94] N. Muscettola, "HSTS: Integrating planning and scheduling," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-93-05, Mar. 1993. [Online]. Available: https://www.ri.cmu.edu/pub_files/pub3/muscettola_nicola_1993_1/muscettola_nicola_1993_1.pdf.

[95] S. Efftinge and M. Spöenemann, *Xtext - language engineering made easy!* https://www.eclipse.org/Xtext/, Accessed: 2022-9-NA. [Online]. Available: https://www.eclipse.org/Xtext/.

[96] JetBrains, *MPS: The Domain-Specific language creator by JetBrains*, https://www.jetbrains.com/mps/, Accessed: 2022-9-NA. [Online]. Available: https://www.jetbrains.com/mps/.

[97] Eclipse Foundation, "Eclipse project briefing materials," Apr. 2003. [Online]. Available: https://www.eclipse.org/eclipse/presentation/eclipse-slides.pdf.

[98] D. Steinburg, F. Budinsky, M. Paternostro and E. Merks, *EMF: Eclipse Modeling Framework* (Eclipse), en, second, E. Gamma, L. Nackman and J. Wiegand, Eds. Addison-Wesley, 2008.

[99] *Information technology — Syntactic metalanguage — Extended BNF*, ISO/IEC 14977:1996. Geneva: International Organization for Standardization, 1996.

[100] D. S. Kolovos, R. F. Paige and F. A. C. Polack, "The epsilon transformation language," in *Theory and Practice of Model Transformations*, Springer Berlin Heidelberg, 2008, pp. 46–60. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69927-9_4.

[101] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev and P. Valduriez, "ATL: A QVT-like transformation language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '06, Portland, Oregon, USA: Association for Computing Machinery, Oct. 2006, pp. 719–720. [Online]. Available: https://doi.org/10.1145/1176617.1176691.

[102] Object Management Group (OMG), *Meta object facility (MOF) 2.0 Query/View/Transformation specification*, Jun. 2016. [Online]. Available: https://www.omg.org/spec/QVT/1.3/PDF.

[103] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige and I. Medina-Bulo, "EUnit: A unit testing framework for model management tasks," in *Model Driven Engineering Languages and Systems*, Springer Berlin Heidelberg, 2011, pp. 395–409. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24485-8_29.

[104] The Apache Software Foundation., *Apache ant - welcome*, https://ant.apache.org/, Accessed: 2023-02-NA. [Online]. Available: https://ant.apache.org/.

[105] *Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model*, ISO/IEC0 7498-1:1994. Geneva: International Organization for Standardization, Nov. 1994.

[106] V. Bonato and E. Marques, "RoboArch: A component-based tool proposal for developing hardware architecture for mobile robots," in *2009 IEEE International Symposium on Industrial Embedded Systems*, Jul. 2009, pp. 249–252. [Online]. Available: http://dx.doi.org/10.1109/SIES.2009.5196221.