



Submitted in part fulfilment for the degree of MEng

Software Engineering for Robotics: an Autonomous Robotic Vacuum Cleaner for Solar Panels

Bianca Darolți

25 April 2019

Supervisors: Prof Ana Cavalcanti and Dr Alvaro Miyazawa

To my parents, Silviu Remus Darolți and Elisabeta Florentina Darolți, whose unwavering love and support made me who I am today

ACKNOWLEDGEMENTS

Thank you to my supervisors, Prof Ana Cavalcanti and Dr Alvaro Miyazawa for their guidance, knowledge, and patience. Thanks to Gautham Vasan for answering my questions about the original vacuum cleaning robot. Finally, I would like to express my gratitude to Dominic Gill for believing in me no matter how far I reached, and to Jess Netherway for supporting me every step of the way.

STATEMENT OF ETHICS

Any work in the field of robotics requires the author to address its potential ethical implications, particularly about the safety of those interacting with the robot. We would like to argue that the usual considerations concerning to this field, or to that of AI since the robot is autonomous, need not apply in this situation—the former because we are not actually constructing a physical robot, but only a model of its software, nor is it intended to interact with a human being. The latter because the application's autonomy consists of merely following a pre-determined path, and the output of our work is a model of how these decisions would be taken if the robot were actually constructed. There is also the question of whether this work could be acquired and used for malicious purposes. We believe the results of our work cannot be used to cause any damage, as the modelled application itself is harmless.

TABLE OF CONTENTS

Executive summary	10
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Structure of document	3
2 Related Work	4
2.1 Domain-specific Languages for Robotics	4
2.1.1 RobotML	4
2.1.2 G ^{en} oM	4
2.1.3 ORCCAD	5
2.1.4 RoboFlow	6
2.1.5 RoboChart and RoboSim	6
2.1.6 Evaluation	7
2.2 RoboChart and RoboTool	9
2.3 Final Considerations	11
3 Problem analysis: autonomous vacuum robot for solar panels	12
3.1 Existing application and its documentation	12
3.2 Documentation issues	14
3.2.1 Software Diagrams	14
3.2.2 Software descriptions	16
3.3 Model requirements	17
4 Model and verification	19
4.1 Robot model	19
4.1.1 Overview	19
4.1.2 Planning layer	20
4.1.3 Executive layer	22
4.1.4 Functional layer	25
4.2 Model verification	27

Executive summary

5	Evaluation	31
6	Conclusion.....	35
	Appendix A.....	36
	Appendix B.....	46
	Bibliography.....	52

TABLE OF FIGURES

Figure 1: Vending robot module	9
Figure 2: Vending robot interfaces, datatypes, and platform	10
Figure 3: Vending robot state machine	11
Figure 4: Vacuum route [7]	12
Figure 5: Path planning algorithm [7]	13
Figure 6: Battery preservation algorithm [7]	14
Figure 7: PathPlanningController	20
Figure 8: Path planning state machine	21
Figure 9: Middle level state machine, with state actions and sinceEntry loops removed (full diagram available in appendix A)	24
Figure 10: PID State machine	26
Figure 11: Definition of NoFall	28
Figure 12: Datatypes and enumerations	36
Figure 13: Operation definitions for cleaning (top) and moving (bottom)	36
Figure 14: Mathematical functions	36
Figure 15: Linear speed function used in LinearSpeedSM	36
Figure 16: Turn functions used in MidLevelSM	37
Figure 17: Interfaces declaring variables and constants	37
Figure 18: Interfaces defining events	37
Figure 19: Interfaces defining operations	38
Figure 20: Robotic platform	38
Figure 21: Overview of module	39
Figure 22: LinearSpeedSM definition	40
Figure 23: AngularSpeedSM definition	40
Figure 24: SpeedSM definition	41
Figure 25: DisplacementSM definition	41
Figure 26: CleanSM definition	41
Figure 27: InputDuplicationSM definition	42
Figure 28: State definitions of MidLevelSM with actions	42
Figure 29: Looping sinceEntry transitions of MidLevelSM	43
Figure 30: MidLevelSM definition hiding turn transitions	44
Figure 31: MidLevelSM definition hiding move_forward and stop transitions	45
Figure 32: Recurse(S, P), NRecurse(S, P) and NoFall definitions	46
Figure 33: ReturnToCharge definition	47
Figure 34: ReturnAfterCharging: performing normal cleaning task and returning to the docking station	47
Figure 35: ReturnAfterCharging: resuming cleaning after charging	48

Executive summary

Figure 36: MovementShape: definition of what a cleaning cycle should look like in terms of movement commands.....	48
Figure 37: PathPlanningSMMovements: PathPlanningSM definition abstracting away all but the movement commands	48
Figure 38: ReachEnd definition	49
Figure 39: PathPlanningSMEndState definition: PathPlanningSM abstracting all but the ultrasonic event, and the events signaling the entering of states Go_Right_Again and Return	49
Figure 40: CleanAllPanels: possible behaviours while moving up, left, or down	50
Figure 41 CleanAllPanels: possible behaviours while moving right, checking whether a cliff has been reached	50
Figure 42: Assertion definitions	51

TABLE OF TABLES

Table 1 Properties of robotic DSLs.....	8
Table 2: Documentation issues	17
Table 3: Core assertion results for state machines.....	32
Table 4: Summary of requirement outcomes.....	34

Executive summary

This project aims to investigate the improvements modelling can bring to an existing real-world application—an autonomous vacuum cleaning robot for solar panels. By grounding our work in a pre-constructed, documented robot, we build a case study of the benefits modelling can bring to this both in terms of documentation and testing.

Modelling has been used extensively in software development to make designs more robust to change, and eliminate ambiguity from discussions about the software’s functionality. In this work we argue that the same principles could be used to obtain these results in the field of robotics. The robotic vacuum cleaner in particular exhibits many documentation issues and ambiguities that could be eliminated by using a domain-specific language (DSL) for robotics—a modelling language that abstracts away common constructs specific to robotics. Furthermore, model verification may allow developers to reason about properties of their robots and correctness of their designs prior to investing the time and resources of building or programming them.

In our work we discuss the shortcomings of the vacuum cleaning robot’s documentation, and derive from these a set of requirements for a useful model. We highlight the incompleteness and ambiguity of the specification, its lack of consistent syntax in its figures, and the absence of formal proof of the robot’s claimed properties—charging itself and resuming cleaning afterwards, achieving full coverage of the solar panels. Our objectives are to compare the available DSLs, select the most suitable language for the application, and construct a readable model that accurately captures the robot’s functionality, and includes enough detail to reason about the above properties. We set as additional objectives to test whether RoboChart, our selected language, can model a robot with such an expressive platform, whether it can capture multiple levels of abstraction in an intuitive manner, and to create the first RoboChart model of a PID controller, a common robotics component which may prove useful in future projects in the language. Our requirements for the model are to be accurate, unambiguous, clear, and to capture all assumptions, definitions, components and their interactions. Furthermore, we aim to use verification tools on our model in order to prove the claimed properties.

We create a model in RoboTool, RoboChart’s associated tool, by following the design described in the original paper. Where the documentation is ambiguous or incomplete, we rely on communication with one of the paper’s authors to clarify the intended design. When

Executive summary

this is not possible, we derive information from the specifications of the robot's hardware, and make our own design decisions where no data can be found. We document all of our assumptions and possible deviations from the original specification. We use CSP – the notation behind RoboChart – to mathematically define the robotic properties we aim to prove. Finally, we verify whether our model exhibits these properties using RoboTool and FDR, a CSP refinement checker.

Our results are a model of the original robot and a set of mathematical proofs of its properties. The model is as faithful to the original specification as is possible to achieve with the current amount of information, fulfilling all of our requirements about accuracy, completeness and lack of ambiguity. Its clarity is a subjective matter. We argue that while some components are verbose, it is overall easy to understand and reason about. We capture most hardware assumptions as required, with the exception of a few properties that we document in writing, but should incorporate into the model in future work. We fail to capture assertions on the environment, as this is not possible in RoboChart, but describe this in writing instead.

We prove that the components of our model are non-terminating, deterministic, and free of deadlock and divergence. We are unable to prove this is true for the entire model as it is too complex for FDR to feasibly reason about. We provide evidence that the modelled robot does not fall off the solar panels, charges when it is low on battery, resumes cleaning afterwards, and achieves full coverage of the panels. Due to the fact that we cannot reason about the entire robot, these assertions are done on the path planning component of the robot instead, and should be supplemented by further proof that other components are correctly integrated with this in future work.

Our model successfully builds on the existing application by capturing all of its functionality in a way that can be clearly discussed and reasoned about. As a result of this, we are able to prove claims from the original paper that were not formally verified or tested. We find RoboChart to be highly expressive and suitable for modelling real-world, complex applications. We also contribute bug reports of wrongly generated semantics we encounter in our development process, and suggest future improvements for RoboTool to make models more readable. Finally, our work highlights issues with RoboChart's optimisations—specifically, that the way it compresses the generated semantics cannot feasibly be applied to a model as complex as ours, and thus need to be improved in future versions of the language.

Since the product of our work is a model rather than a physical application, we also argue the absence of any legal, social, ethical, professional, or commercial issues that can arise from our results.

1 Introduction

1.1 Motivation

As of 2015, renewable sources accounted for 19.3% of the world's energy production [1], and continue to grow in popularity. Solar energy is at the forefront of this trend, due to a significant decrease in cost: 80% from 2008 to 2015 [2]. Photovoltaic (PV) panels have seen the greatest cut in pricing and as such are attracting more usage [3].

One main driver for the reduction of cost of PV panels has been innovation in cleaning methods. Solar energy is best harnessed in areas with hot, dry climates, where accumulation of dust poses a serious problem—it has been found to lower their efficiency by up to 50% [4] and lead to overheating and permanent damage [5].

PV panel cleaning has been achieved with great efficiency, removing over 90% of dust particles [6]. Still, complete coverage is yet to be reached, and so there is still ongoing research in alternative solutions, such as that proposed by Aravind et al.: an autonomous vacuum cleaning robot powered by the panels [7]. Their approach is a light-weight robot designed to traverse the panels cleaning them, and a docking station for whenever it needs to charge.

Avarind et al. implemented a working robot. Their outcomes show complete coverage of the panel and satisfactory cleaning.

However, the work does not include any formal or even rigorous specification of the robot's software. This is instead detailed in English descriptions and state diagrams with no formal semantics. These are open to interpretation and cause difficulty in reproducing the experiment. Additionally, the software is only tested using Code Composer Studio [8], and no systematic account of any claimed properties is performed. It would be helpful to have a more abstract description of the design, particularly such that people using different platforms could recreate the application. This abstraction, in turn, may help us reason about the robot's properties and the software's correctness before expending resources on building it.

One way to provide this abstraction is to build a model of the robot. Software models depict *what* a system does, whereas the code describes *how* it does it [9]. Modelling increases the productivity of developers, by reducing the product's sensibility to change. Expressing the design formally allows for the development process not to be affected by a change in personnel, and for the application itself to require minimal alterations when requirements or platforms are changed [10]. It also facilitates better communication and discussion

of the design prior to implementation, and helps developers manage the increasing complexity of their codebase [11]. The outcome is reduced cost and time-to-market, as well as solving recurring architectural problems.

Model-driven engineering further adds to the benefits of modelling by combining two technologies: transformations and domain-specific languages (DSLs) [12]. Two types of transformation exist: model-to-model, converting from one model to another, within the same language or to a different one, and model-to-text. Model-to-text transformations enable the generation of text files such as deployment scripts or even code directly from a model. This allows developers to translate an abstraction into platform-specific code, and bypass writing code for architectural problems they have already solved by modelling.

This is even more beneficial when paired with DSLs: modelling languages defined for an application domain. These enrich the expression power of modelling by abstracting recurring features of a domain, and providing constraints that reflect the reality of the area.

By nature, DSLs are numerous and varied [13], and particularly in robotics, there is no industry standard.

1.2 Objectives

To build upon Avarind et al.'s work, we set the following objectives. Firstly, we aim to explore the available domain specific languages for robotics and evaluate what they can bring to this application. With our selected language, we will build a software model of the robot that:

1. Specifies the robot's functionality as defined in the paper
2. Is easily readable and understandable
3. Provides enough level of detail to reason about the properties that were experimentally proven in the paper:
 - a. The robot's charging function operates correctly and it does not run out of battery.
 - b. After charging, the robot resumes cleaning from the position it paused at.
 - c. The robot achieves full coverage of the solar panels.

In carrying out this work we have chosen RoboChart [14], due to its strong mathematical foundation, provisions for reasoning about robotic properties, and intuitive graphical tool. In the use of this language we have a number of additional objectives:

1. Test whether RoboChart can model such an expressive platform as that provided by the application.
2. Test RoboChart's ability to model multiple layers of abstraction in a readable and intuitive manner.

3. Create the first RoboChart model of a PID controller, a very commonly used component in robotics, which may prove useful for future applications.

1.3 Structure of document

Chapter 2 aims to provide an overview of the available DSLs and a basic understanding of the selected modelling language. It investigates available languages that may be suitable to our undertaking, justifies our choice in more detail, and provides a short explanation of the selected language's features.

Chapter 3 focuses on the robot we intend to model, detailing the existing work, the questions its documentation does not answer, and deriving a set of model requirements from these such that our results may contribute to said documentation.

Chapter 4 presents the actual model we created and the assertions we use to verify it.

Finally, chapter 5 presents an evaluation of the model against our requirements, and chapter 6 briefly draws some conclusions from our work.

2 Related Work

In this section we cover the background material for our work. In section 2.1, we identify the DSLs for robotics that have formal semantics and are thus suitable for our task, and describe them. We provide a more detailed analysis of our language of choice in section 2.2, and draw our final considerations in section 2.3.

2.1 Domain-specific Languages for Robotics

For the purpose of accomplishing our task we are interested in languages which meet the following criteria:

- They are domain-specific languages for robotics and provide abstractions for concepts in this area.
- They hide low-level programming details, such that their usage does not require expertise in the domain of robotics.
- They have formal semantics, and thus have meaning beyond being a representation of the system’s architecture.

2.1.1 RobotML

RobotML [15] was born out of a desire to solve variability issues in robotics and abstract away from low-level coding details. It provides tools for modelling, simulation and **deployment**, focused on component-based architectures. A model is built by declaring a set of components, their behaviours, via either algorithms or finite state machines, and their communications. The user specifies the system’s environment for simulation, and the target platform, for code generation. RobotML has a well-defined and straightforward workflow.

The language uses Ontology for its **semantics**. An Ontology is a definition of a set of concepts and the relationships between them, but not their behaviour. RobotML exploits an already existing domain-specific ontology, such that concepts do not require re-definition. The result of this choice is a collection of classes that provide a suitable level of abstraction, making the language easy to use.

One can construct RobotML models via a simple **graphical tool** that is an extension of Papyrus [16], part of the Eclipse Modelling Project. The tool supports a variety of target platforms and simulation environments and is therefore fitting for a number of different robotic applications. However, it provides **no form of verification** of the model’s correctness, allowing only for static model **validation**.

2.1.2 G^{en}M

G^{en}M [17] is a language designed for code generation and testing. Like RobotML, it focuses on a component-based architecture. These components, called modules, are responsible for a physical or logical resource, performing the processing, failure detection and recovery.

Modules operate in a client/server architecture of execution requests and control requests (reading and writing parameters or interrupting execution). The language allows for the definition of a **hierarchy of processes** using these modules, all communicating via posters, areas of shared memory. A user can describe modules in a **declarative language** similar to C, specifying their inputs, outputs, execution steps, posters and timing properties. A $G^{\text{en}}\text{M}$ **parser tool** is provided.

$G^{\text{en}}\text{M}$'s strength lies in the facilities it provides for deployment, **generating code** and interactive tests. Its successor, $G^{\text{en}}\text{M3}$ [18], allows specification of optional temporal properties for execution tasks. It also supports the use of verification tools such as D-Finder.

Recent developments in $G^{\text{en}}\text{M3}$ have led to the possibility of creating Fiacre [19] and RT-BIP [20] templates that synthesise models [21]. These templates are relative to the targeted middleware to ensure that they are semantically equivalent to the generated component. Both templates allow $G^{\text{en}}\text{M3}$ users to check models for client requests and monitor components, as well as their timing properties.

While $G^{\text{en}}\text{M3}$ brings great improvements in terms of model **verification**, it is still restrictive in terms of the architecture imposed. We also argue that writing the specification in a C-like language leads to a limited level of abstraction, and reduced readability of the model.

2.1.3 ORCCAD

ORCCAD [22] provides a greater emphasis on the real-time aspects of robotic applications. Its specifications consist of Robot-Tasks and Robot-Procedures. Robot-Tasks represent elementary robotic actions, and can be defined using a **graphical user interface** in the provided toolset. Robot-Procedures **range in complexity** from simple actions to full mission specifications, and can be described using a dedicated language called **MAESTRO**. Both constructs include attributes such as pre- and post-conditions, and the user can specify temporal properties such as the period of a cyclic execution. From the completed model, the toolset can **generate** ESTEREL or C++ code.

While this is restrictive in terms of target platforms, ORCCAD excels instead at **formal verification**. One can check the synchronisation of every Robot-Task, finding any structural deadlocks or temporal inconsistencies. The tool also provides logical analysis using theorem-proving techniques tailored for MAESTRO. Additionally, application-dependent requirements can be verified interactively as relationships between events, or between events and actions. ORCCAD performs this check by constructing and analysing an automaton.

2.1.4 RoboFlow

Unlike previously discussed languages, RoboFlow [23] stands out by having a different target user. Its aim is to enable people to describe the intended behaviour of robots without any prior knowledge of programming. To this end, RoboFlow is a simple graphical flow-based programming language, which uses programming by demonstration.

RoboFlow allows for the definition of three types of procedures: manipulation, navigation, and active perception. Instantiating these procedures is similar to filling out a template. They have preconditions and postconditions, which are determined by the type of procedure, and their behaviour can be specified in terms of a set of parameters.

RoboFlow programs are **graphs**, and their **operational semantics** is provided by logical inference rules of state transitions. The language also allows for procedural abstraction: users can define nesting programs as procedures, allowing for a **layered system architecture**.

RoboFlow is implemented as a **Java applet** for the PR2 research robot. While the language's constructs are generalizable, its tool has a restricted scope in that it enforces a specific target platform, and constrains the achievable types of tasks. However, it is very expressive within its limited scope, and accessible for non-experts.

2.1.5 RoboChart and RoboSim

RoboChart [14] was born out of an initiative to build a more rigorous connection between the modelling and simulation stages of robotic development. RoboChart is the language developed for the former, while RoboSim was created for the latter.

In RoboChart, robots are specified in terms of modules, which **encapsulate** and connect platforms and controllers. Platforms represent in-built facilities of the hardware, while controllers offer implementation details of the functionality executed within the platform. Controllers describe behaviour using finite state automata based on UML [24] SMs (State Machines) stripped of features deemed non-essential for robotics. SMs are augmented to allow for the specification of timing properties, which are often crucial to robotic behaviour, and probabilistic properties, implemented using P-nodes.

The language also includes an API comprising of a set of datatypes and functions. It does not provide code for these, nor does it have any facilities for code generation. However, it is strongly tied to RoboSim [25], a state-machine based language for the definition of simulations, supporting automatic generation of simulation code.

The **semantics** of RoboChart is specified in CSP [26], a mathematical notation for communicating sequential processes. The user constructs a model via a **graphical tool** called RoboTool, a set

of Eclipse plug-ins, and this is translated into CSP. This allows for **verification** using FDR [27], a CSP refinement checker. Specifically, FDR allows one to assert the determinism of a specification, as well as absence of divergences and deadlocks. It can also be used to verify reachability of states in the SMs, and assumptions about the hardware, as long as it is not abstracted away in the model. If one of these assertions fails, FDR provides a counterexample scenario for debugging. RoboTool also provides a **validator** that implements well-formedness conditions of the language.

While it is lacking in terms of facilities for code generation and deployment, RoboChart is highly expressive, and allows for thorough verification based on a mathematical model.

2.1.6 Evaluation

Table 1 provides a summary of the above discussion. Often in robotics, developers split software into multiple layers depending on the level of abstraction involved, from planning to controller-level code. None of the discussed languages provide facilities for defining these layers, but some allow the user to create them using other constructs.

Validation, in this context, encompasses a tool’s ability to verify that a model conforms to the language’s syntax along with a set of well-formedness rules. Verification involves checking the specification against a set of requirements based on the semantics rather than the syntax. Formally, it is defined as “confirmation by examination and provision of objective evidence that specified requirements related to a product or process have been met” [28].

RoboFlow is not fit for our purpose, due to its tool targeting specific hardware different from that used in our application. Additionally, while our application could feasibly be modelled in terms of RoboFlow’s three task templates, the current tool requires us to physically manipulate the hardware, which we do not have access to.

Of the remaining languages, we consider only those with verification tools. We deem the C-like language of G^{en}oM to fall short of achieving the level of abstraction and readability we desire, particularly in comparison with ORCCAD and RoboChart.

We believe ORCCAD and RoboChart would be most suitable for this application. They both allow the user to specify software at different layers of abstraction, and have strong provisions for formal verification.

We could use ORCCAD, as it has outstanding features for specifying real-time properties of a system, which may be useful particularly in asserting that the robot returns to its docking station in time to charge. This is not to say that RoboChart does not have any timing primitives. RoboChart allows for the specification of time budgets and deadlines.

Related Work

We deem these to be sufficient, as the period and WCET properties provided by ORCCAD belong to a more concrete design level.

Language	Semantics	Tool support	Current support	Verification	Validation	Code generation	Layers
RobotML	Ontology	Graphic	✓	X	✓	✓	X
G ^{en} oM	Declarative language	Parser	✓	✓	✓	✓	✓
ORCCAD	MAESTRO and ESTEREL	Graphic, textual	✓	✓	✓	✓	✓
RoboFlow	Logic inference rules	Graphic	X	X	✓	✓	✓
RoboChart	CSP	Graphic	✓	✓	✓	X	✓

Table 1 Properties of robotic DSLs

Unlike ORCCAD, RoboChart does not currently facilitate code generation, but as we are only interested in constructing a model of the robot, this shortcoming is not relevant to us. Additionally, its API library may be useful to us, and while its real-time features are not as expressive as ORCCAD's, its conversion to CSP allows for checking

properties of the hardware that are included in the model using FDR, a provision that is unusual yet very useful to the field of robotics. Because of these reasons, we choose RoboChart as our language.

2.2 RoboChart and RoboTool

In this section we provide a description of RoboChart, and RoboTool, the tool used to build RoboChart models. We use an example of a mobile vending-machine robot. This has the task of moving to a specified location, and delivering tea or coffee according to a user's request. It navigates the building by using a stored map.

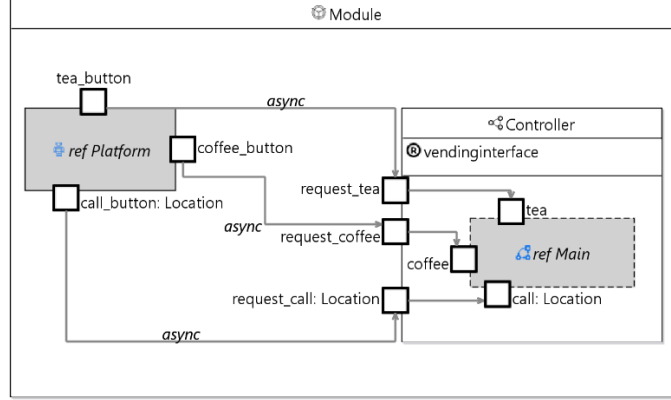


Figure 1: Vending robot module

The top-level construct of a RoboChart model is a **Module**, a structure that records assumptions made about the robot's hardware, describes its controlling software, and links these together. It either contains or references a **Platform** and any number of **Controllers**, and defines the connections between them. For example, our vending machine robot contains one module (figure 1), which references one platform, contains one controller, and connects their events.

Platforms represent in-built facilities of the hardware. These include typed variables and constants, operations the robot can perform and events. The platform of our robot specifies the operations *dispense_tea()* and *dispense_coffee()*, representing the actuators that produce a cup of tea or coffee, and the operation *move(location:Location)*, which models the robot navigating to a given position, specified by a parameter of type *Location*. The type, shown in figure 2, is composed of integers *X* and *Y* representing coordinates.

In this example, we have modelled the operation *move* as the robot planning a route to a given location and travelling to it, without describing any of the details of this process. When writing a model for the vacuum robot, we will specify the path planning algorithm in detail.

All of the above operations are contained in a **Provided Interface** called *vendinginterface*. Interfaces serve to encapsulate events,

Related Work

variables and operations. They can be of three types: provided, defined and required. The first describe variables and operations a robotic platform provides. **Defined interfaces** declare events and variables used in an element. **Required interfaces** describe operations and variables a controller or state machine assumes are provided by the platform and other controllers, allowing for behaviour to be defined independently of specific platforms.

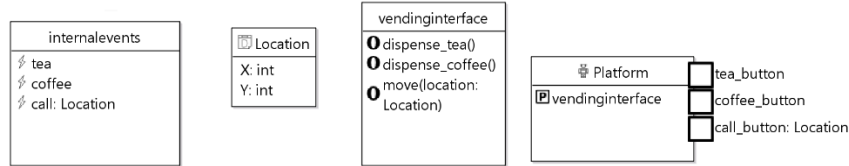


Figure 2: Vending robot interfaces, datatypes, and platform

Events represent an atomic communication. In our example, the robot's platform has the events *tea_button* and *coffee_button*, symbolising the user pressing a button to request tea or coffee respectively, as well as *call_button*, an event of type *Location*. Values of type *Location* can atomically be communicated over *call_button*.

Events exist at the platform level, as well as at Controller and State Machine level. For example, our robot's **controller** has a required interface of operations and variables it expects from the platform, which links to the interface provided by our platform. *Controller* also defines 3 events: *request_tea*, *request_coffee*, and *call_to : Location*. The *Module* contains both a reference to the platform and the definition of the controller, and links their events together: *tea_button* is linked to *request_tea* such that whenever the former occurs so does the latter. Effectively, whenever a user presses the tea button, a request for tea is passed on to the controller. The same is true for *coffee_button* and *request_coffee*, as well as *call_button : Location* and *call_to : Location*.

It is worth noting that this architecture of one-to-one event mappings at every level is part of our example, and not enforced by RoboChart. Event connections can also exist between state machines, or between controllers. Connections can be unidirectional or bidirectional, and synchronous or asynchronous, although connections involving a platform must always be asynchronous.

Having defined the platform and the controller and linked them together in the module, we define the behaviour within the controller. This is done with a **state machine**, *Main* (figure 3). This has the same required interface as the controller, a variable *location* of type *Location* and the events *tea*, *coffee*, *call : Location*, specified in a defined interface called *internalevents* and linked to the controller events as expected.

Every state machine is composed of states, junctions, and transitions. States may have actions to be performed on entry to the

state, during active stage of the state, or on exit. Junctions may not have these operations, and must have at least one outgoing transition. Their transitions may not have triggers, and their guards must form a cover. They act as unstable ‘temporary states’ that the robot must pass through and exit immediately. Every SM must have one initial junction. Ours immediately transitions into a state called *AwaitCall*, where the robot waits to be summoned to a *Location* via the *call* event.

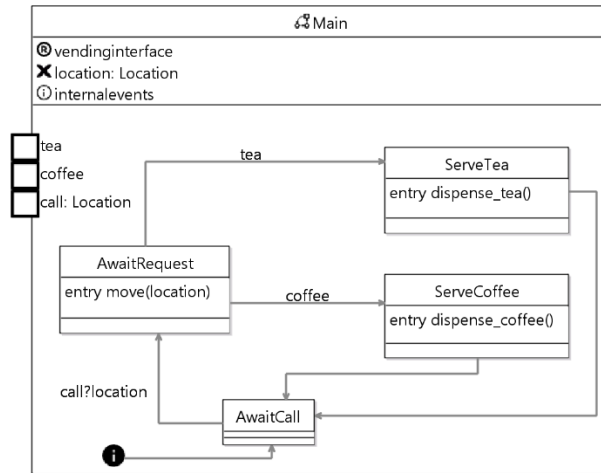


Figure 3: Vending robot state machine

Once a *Location* is received via *call*, it is stored in the variable *location* and the transition to the state *AwaitRequest* is triggered. On entry, the operation *move(location)* is performed, the robot travels to the place specified by the location, and waits for the user to request either tea or coffee. This then triggers a transition to either the state *ServeTea* or *ServeCoffee*, which, on entry, triggers one of the corresponding operations *dispense_tea()* or *dispense_coffee()*. The SM then transitions back to *AwaitCall* to await the next request.

To summarise, our robot uses data types we have defined, and its operations and external events are contained in a platform. Its behaviour is specified by a state machine, which is a component of a controller. The top level of our specification is a module, which links together the platform and controller. There is, of course, more to RoboChart than the basic building blocks described here, and we will explain any other necessary concepts as they are used.

2.3 Final Considerations

In this chapter we have discussed and compared a selection of 5 robotic modelling DSLs. We have justified our choice of using RoboChart as our implementation language and provided a brief account of its features. In the next chapter we will present the application we intend to model in more detail, review its documentation, and derive from it the requirements of our model.

3 Problem analysis: autonomous vacuum robot for solar panels

In this chapter we provide a detailed report of the existing application and our goals. We give a comprehensive description of the robot in [7] in section 3.1, an analysis of issues with the documentation in section 3.2, and a list of requirements for our model in section 3.3.

3.1 Existing application and its documentation

Aravind et al.'s goal was to create an energy efficient and therefore cost-effective vacuum cleaning solution for solar panels. They addressed four key challenges. These included cliff detection, efficient navigation on the inclined surface of a solar panel, traversal from one panel to another, and use of a generic algorithm, such that it could be deployed on different sets of panels without modification.

The robot performs a two-stage cleaning process—because the dust sticks to the panels, it uses a rolling brush to agitate it and push it towards a vacuum. The controller sends a common signal to both the brush and the vacuum motor to continuously operate. It also verifies the battery level regularly. Whenever the voltage is beneath a pre-set threshold, the robot finishes its current cleaning cycle and returns to its docking station, which is connected to the panels. The robot charges itself using two safety mechanisms. If the voltage exceeds 12.6V, it lowers the charging current to prevent battery damage. Once the battery is charged, which is also verified via comparison with a reference value, the robot disconnects itself from the charging circuit to avoid damage, and resumes cleaning from its last position.

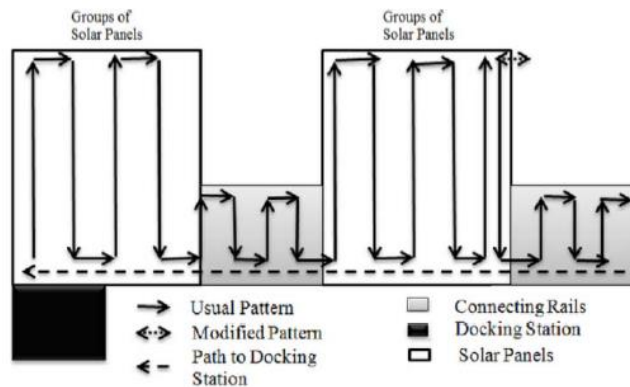


Figure 4: Vacuum route [7]

Navigation follows a pre-defined cleaning path, traversing the entirety of each solar panel. Panels are usually linked together in arrays, which Avarind et al. connect with rails to facilitate the robot's traversal. The robot uses two motors and differential drive for steering. It continuously monitors the output of two accelerometers on either side and a front mounted ultrasonic sensor in order to ensure it is

following the correct path and avoiding cliffs. It uses a Proportional Integral Derivative (PID) technique to stay on its defined path.

The pattern the robot navigates in can be seen in figure 4, and the algorithm can be visualised in figure 5, which we will revisit in detail in the next section. The robot begins by climbing up the first panel, while using the ultrasonic sensor to continuously measure the distance to the surface beneath it. Eventually, the ultrasonic's returned value exceeds a threshold, indicating that the robot has reached the edge of the panel. It performs a 90° turn to the right, and moves a pre-set distance, equal to the vacuum nozzle's length, before turning 90° to the right again. This is done in order to cover the entire panel's surface without overlapping. The robot then travels down the solar panel until it reaches its end, turns left twice in the same fashion and repeats the entire pattern until it has finished cleaning or must return to charge.

The robot records its position by counting the number of such cleaning cycles it has completed. Whenever it has finished a cycle and needs to return to the docking station, either because it has finished or because it needs recharging, it rotates 180° and follows the path forward.

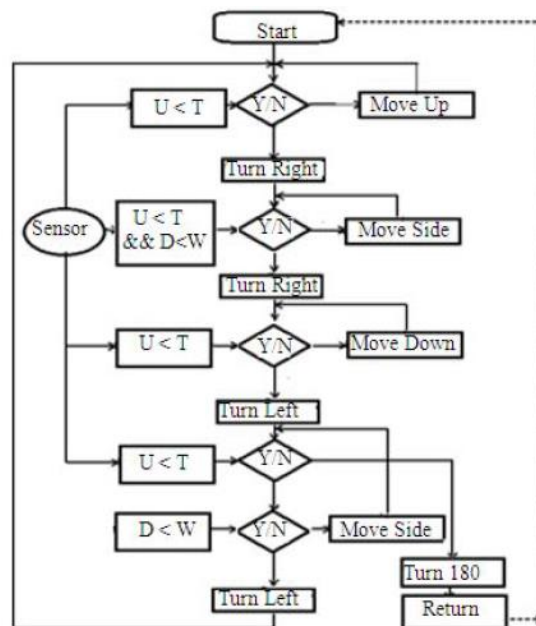


Figure 5: Path planning algorithm [7]

The diagram in figure 5 is a visualisation of this algorithm. Note that the meaning of the variables is not obvious, nor is it defined in the paper. The next section further elaborates on this aspect and the figure as a whole, as a result of discussion with the authors.

Avarind et al. created a robot that can traverse an entire array of solar panels at a speed of 2-6 cm/s, cleaning with satisfactory results. They have provided experimental evidence that it traverses inclined

surfaces with no difficulty, and that it recharges itself successfully and resumes cleaning. Their solution achieved all of its requirements while remaining simple and cost-effective.

3.2 Documentation issues

Here we discuss our identified issues in the robot's documentation.

3.2.1 Software Diagrams

While the paper gives an understandable high-level explanation of how the robot works, many questions are left unanswered. Aside from the textual explanations of its algorithms, the only available documentation is in the form of two diagrams, in figure 5, the path planning algorithm, and in figure 6, the battery preservation algorithm. These diagrams are not consistent with each other in terms of syntax. The *Return* statement is represented differently in both, and the conditionals are depicted as diamond shapes containing predicates in figure 6, but as separate conditions in rectangles and Yes/No decisions in diamonds in figure 5.

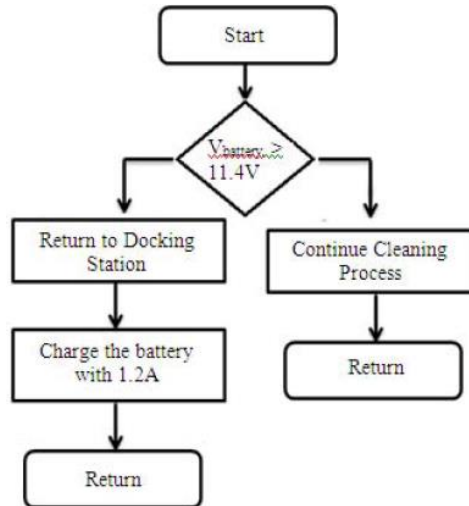


Figure 6: Battery preservation algorithm [7]

The diagrams are not only inconsistent with each other in terms of syntax, but within figure 5 the syntax also changes. The last conditional lacks input from the sensor, and it is unclear whether this is an accidental omission, or perhaps the *Sensor*, which is never defined, does not provide the values for *D* and *W*.

The meaning of the *Return* statement is also unclear. If it has the same semantics as a return statement in programming, then the loop back to *Start* is broken. It is possible that the use of a dashed line instead of a solid line like the rest of the diagram is meant to convey this, but the significance of the line style choices is never clarified.

Moreover, both diagrams are difficult to read. It is unclear which branch of the conditional is to be followed if a predicate is true, or false. While we can easily deduce this from textual explanation in figure 6, this is much more difficult in figure 5, since the variables used are not defined.

As a result of communication with the paper's authors [29], we understand that the meaning of these variables is as follows: the ultrasonic reading is labelled U , and T is the threshold for its value, beyond which the robot is considered to have encountered a cliff. D is the horizontal displacement of the robot measured using the accelerometers, and W is the length of the vacuum nozzle.

With this clarification in mind, we can deduce that the right branch of every conditional is the true/Yes branch, and the lower one is the false/No branch. Read in this manner, the diagram specifies that the robot moves up until the ultrasonic reading exceeds the cliff threshold (the first conditional statement, $U < T$), then turns right and proceeds until it reaches another cliff or has moved the distance of the vacuum nozzle (the second conditional statement, $U < T \ \&\& \ D < W$). It then moves down the panel and eventually to the left, in a similar pattern.

Note the description of the robot's movement—upon encountering the first cliff, it turns right, then performs the operation “move side”. The common approach when specifying such an algorithm is to describe its movement relative to its own orientation. Instead, here it is described relative to the top-down view of the panels, as seen in figure 4. This convention is not declared in the paper, and is all the more confusing as the turn operations are performed relative to the robot's orientation, but the move operations are not.

Finally, the second to last decision in the diagram discerns whether the robot has encountered the end of a group of panels. Following the syntax that has been used thus far, the false branch, where the edge has not been encountered, leads to the robot performing a 180° turn, and the true branch results in the robot moving forward over the cliff. This clearly cannot be the intended behaviour, but rather the lack of explicit convention leading to an error in specification.

The diagrams are also incomplete in terms of operations described. While the meaning of the movement and turning operations is intuitive, the way these highly abstracted operations are communicated and performed at a more concrete level is not described. Similarly in figure 6, the robot's return to cleaning is a single high-level operation, but this is abstracting away a number of smaller steps that are not modelled.

The same issues are encountered with the diagrams' variables. Beyond them not being declared, no details are provided on how these variables are obtained. For instance, how is the distance D derived?

The authors have confirmed to us that it is obtained from the accelerometers' output, but not how it is calculated.

The software description is meant to be supported by these diagrams. But due to their inconsistent and unclear syntax, lack of definition of variables used, unclear use of operations and incompleteness they do not help convey a clear understanding of the algorithms, on the contrary they introduce errors and confusion.

3.2.2 Software descriptions

Another issue with the documentation is that there is no centralised description of the software components. While their interactions are mentioned, neither of the diagrams model this interaction.

Some aspects of the software are not modelled at all. It is claimed that the robot returns to the place it was cleaning prior to charging, but not how it does this, apart from the mention that it keeps track of its location by number of completed cleaning cycles, presumably using a counter, which is not present in the path planning algorithm.

The constants used are also unclear. In the error calculation formula provided for PID, as seen below, the error values e_p , e_i and e_d are proportional, integral and derivative errors found via comparison to preset values, but no indication of what these predefined values represent is given.

$$e = e_p \times k_p + e_i \times k_i + e_d \times k_d [7]$$

Similarly, it is claimed the robot always returns to the charging station successfully due to the voltage monitoring algorithm, but no clarification is given as to how the reference value used is calculated. Is it a constant derived experimentally, or based on a formula to ensure the robot does not run out of battery on its way back to the docking station?

Constants, variables and their datatypes are not declared in a unified manner. It is unclear which datatypes are used, and whether this imposes any restrictions on the implementation language.

Additionally, while the paper details the specific hardware used for the implementation, the requirements expected of the hardware platform are not provided. If one would attempt to implement these algorithms on different hardware, in order to further improve on weight and cost as suggested by the authors, hidden or implied requirements may cause issues.

Much like the platform requirements, the assumptions made on the environment are also incomplete. The paper lists 5 assumptions made—limiting panel inclination to 30 degrees, the panels only being covered in dust and small impurities, weekly dust collection via the

robot, provision of rails and a docking station, and differential turns being zero radius. However, at least one other assumption is made implicitly. Namely, that the furthest point a robot must clean is close enough for it to be possible to reach it and travel back to the docking station without exhausting the battery on the way.

Moreover, important aspects of the software's interaction with the hardware are not discussed. For example, how often is the ultrasonic sensor sampled? Considering the robot's speed travelling both up and down the panels, what sampling rate is sufficient for a cliff to be detected in time to stop moving?

To add to the issues with the documentation, the software was only tested with Code Composer Studio [8], and the experimental evidence given that it meets its requirement may not be sufficient.

3.3 Model requirements

Revisiting all of the above issues, we can summarise them in the following table, such that we can refer to them easily in future sections:

Number	Issue
1	Diagrams are inconsistent with each other in terms of syntax
2	Meaning of different diagram elements (<i>Sensor</i> , <i>Return</i> , different line styles, etc) unclear
3	Unresolved ambiguity in diagrams
4	Description of movement and turn operations based on uncommunicated convention
5	Error in path planning diagram
6	Meaning of operations in diagrams unclear
7	No declaration of variables in diagrams
8	No indication of how variables and constants are obtained or what they represent
9	Lack of centralised description of software components
10	Interaction between software components not modelled
11	Software model is incomplete
12	No unified declaration of datatypes used
13	No statement of platform requirements
14	Incomplete environment requirements
15	Aspects of software interaction with hardware are not modelled
16	Only experimental evidence that the robot charges successfully
17	Only experimental evidence that the robot returns to its previous cleaning location after charging
18	Only experimental evidence that the robot achieves full coverage of solar panels

Table 2: Documentation issues

To further build upon the existing application as described in the previous section, we identify the following requirements for our model, referring to the issues we wish to address from table 2:

1. Reflect the functionality of the robot as specified in the paper and present it in a consistent manner (1, 5, 11)
2. Model the software components and their interactions (10)
3. Depict information clearly, unambiguously, and readably (2, 3, 4, 9)
4. Use a clearly defined set of variables and constants (7, 8)
5. Define the robot's operations and their parameters consistently (6)
6. Declare all its non-primitive datatypes in a unified manner (12)
7. State all assumptions on the robotic platform (13)
8. State all assumptions on the robot's environment (14)
9. Model software interaction with hardware (15)
10. Provide enough detail such that the following assertions can be tested using FDR:
 - a. The robot returns to the docking station successfully (i.e. it does not run out of battery) (16)
 - b. If cleaning is not finished after charging, the robot returns to the location it was previously cleaning and resumes the process (17)
 - c. The robot completely covers all of the solar panels (18)

We have restricted the scope of the properties we intend to prove using our model. This is because other assertions made in the paper, such as those about the robot's speed and its cleaning efficiency, require us to reason about physical properties that we are unable to include in our specification, but may be better suited for simulations.

In the following chapter we will describe our RoboChart model of the vacuum cleaning robot.

4 Model and verification

In this chapter we describe our RoboChart model of the robot and its properties, as verified in RoboTool. In the first section we discuss the model, while in the second we detail the assertions we will use to verify our required properties in the next chapter.

4.1 Robot model

Although this cannot be made explicit through RoboChart, we structure our model in the following layers: a planning, a functional, and an executive layer. We use this architecture because the original paper clearly characterises the former two layers in the form of its path planning algorithm and PID controllers respectively, and we aim to stay true to the original implementation. The paper does not mention how the two layers interact, and we therefore use an executive layer to bridge this gap. We give an overview of the model, followed by a detailed description of the components operating on each layer.

4.1.1 Overview

Our model consists of **Module**, which connects **Platform** to the only controller, **PathPlanningController**. The platform simply provides all of the operations required by the controller, and all the events that must be connected to the controller's defined events.

We define **PathPlanningController**, a model for the original robot's only controller, containing all three layers of our application. This requires the interfaces *CleaningOperations* and *MovementOperations*. The former contains the operations *vacuum(x:int)* and *brush(x:int)*, which allow the controller to output to the motors controlling the vacuum and brush. The latter contains *output_left_motor(x:int)* and *output_right_motor(x:int)*, used to controller the movement of the robot. The controller defines the interface *SensorEvents* containing the events it needs to receive information about its environment: *battery_level:int*, *ultrasonic:int*, and *charging*, which is received whenever the robot begins charging. Additionally, the controller defines the events *acc_l:TripleAxis* and *acc_r:TripleAxis* to receive input from the left and right accelerometers. The datatype *TripleAxis* models the data returned by the triple axis accelerometers used in [7]. These return three values, namely the acceleration on three different axis. These are the values X, Y, and Z of *TripleAxis* in our model.

Within the controller (figure 7), we define **PathPlanningSM** to make navigational decisions based on sensor input. These decisions are communicated to **MidLevelSM**, which translates them to target values to be sent to the two **PID** SMs regulating linear and angular speed. These speeds are provided by **LinearSpeedSM** and **AngularSpeedSM**. The accelerometer outputs from the left and right

of the robot are converted into two speed values by two **SpeedSMs**. The operation of the brush and vacuum is handled by **CleanSM**, based on **PathPlanningSM**'s commands. The displacement variable required by the latter is calculated by **DisplacementSM**.

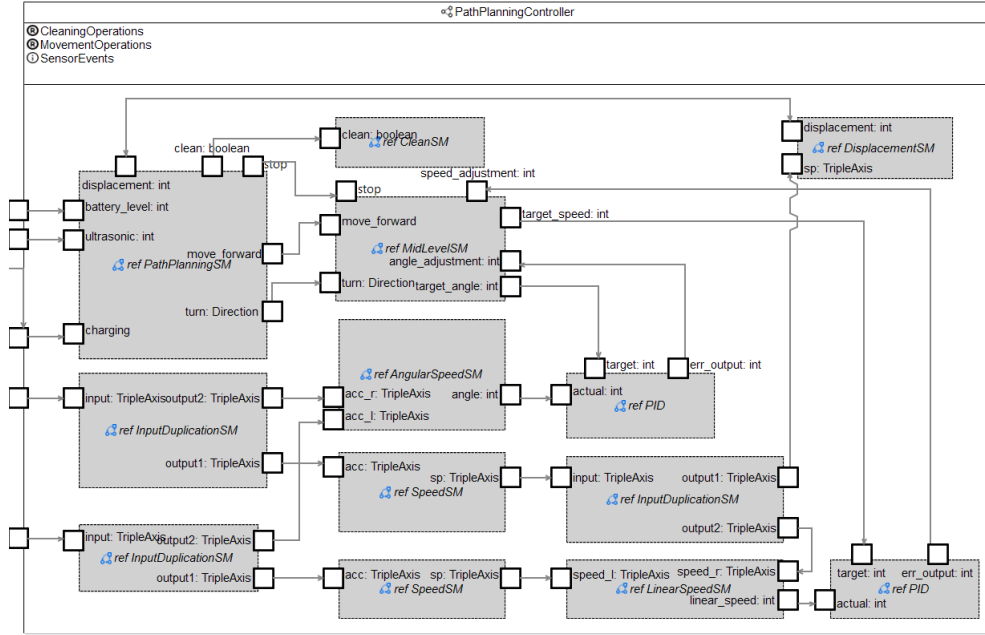


Figure 7: PathPlanningController

4.1.2 Planning layer

We begin by describing **PathPlanningSM**, depicted in figure 8, the SM operating on the planning layer, as it is a more detailed representation of the diagram depicted in figure 5, and therefore the easiest part of our model to directly relate to the source material.

This state machine is tasked with deciding which direction the robot should move in. It uses the events *move_forward*, *stop*, and *turn:Direction* (where *Direction* is an enumeration with elements *left* and *right*), defined in the interface *MovementCommands* to communicate its decisions to the lower layer state machines that enact them. It receives information about its environment via the events *ultrasonic:int*, *displacement:int*, *battery_level:int*, and *charging*, and uses the event *clean:boolean* to command the robot to activate or deactivate its vacuum and brush motors. Whenever the robot turns right it sends out the number zero on the *displacement* channel to communicate that it needs to begin measuring displacement anew. The state machine has an integer variable called *cycles*, which keeps track of the number of cleaning cycles completed, as defined in chapter 3, in order to be able to resume cleaning after charging. The constants used in figure 5 are defined in our model in the interface *SensorConstants*—*cliff* is the equivalent of *T*, the cliff threshold, and *W* is mapped onto *nozzle*, the length of the vacuum nozzle.

Model and verification

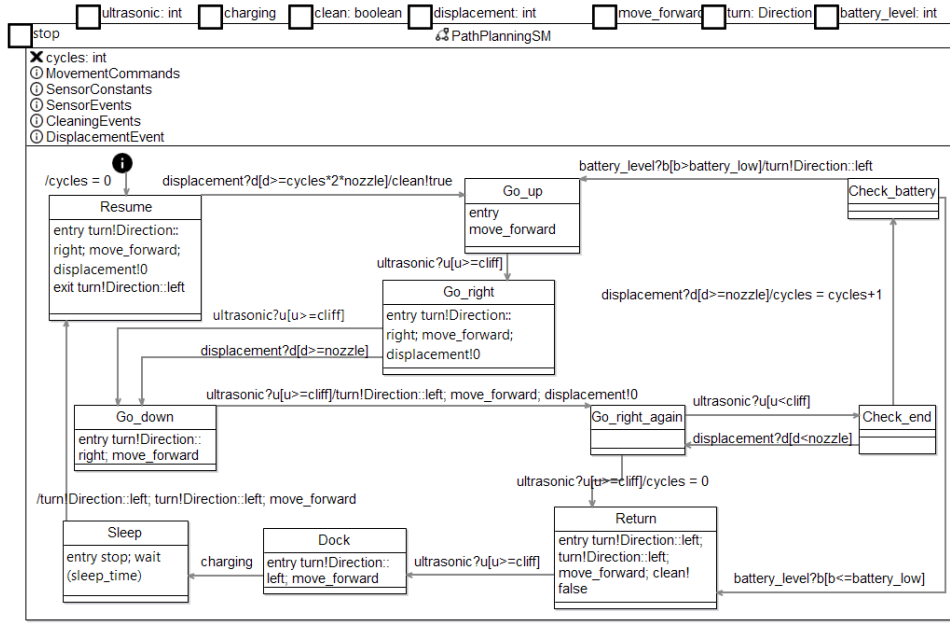


Figure 8: Path planning state machine

We assume the robot begins operating from its docking station, located at the bottom left corner of the panels, facing towards its exit. It first enters the state *Resume*, which is meant to guide it to its last position before it returned to the docking station—in this case, the start. This destination is defined by using the completed cycles. Although the details are not discussed in the original paper, Figure 4 suggests that the total horizontal length of a cleaning cycle is two times the length of the robot’s vacuum nozzle. We therefore deduce that in order to reach its last cleaning position the robot must simply travel to the right until its horizontal displacement reaches double the nozzle length multiplied by the number of completed cleaning cycles. This is the transition condition to move to the next state, *Go_up*. Although this is not discussed in the paper, as it never relates figures 5 and 6 to each other, we make the assumption that the robot checks its battery level only at the end of a cleaning cycle.

Where possible, states are named after the stage of a cleaning cycle the robot is in. The command to turn left is sent on transition to *Go_up*, followed by the command to move forward on entry. Once it encounters the end of the panel, a condition defined as *ultrasonic?u[u]>=cliff]*, it transitions to the state *Go_right*. The SM instructs the robot to move right, and at this point it may either finish moving the length of its vacuum nozzle, or reach the end of the current panel array, in both cases transitioning to *Go_down*. In the former case this is simply following the pattern of a cycle. In the latter case, this is because there may be more arrays to clean and therefore a connecting

rail at the bottom of the panels, and the same action is taken as in figure 5.

The transition to *Go_down* begins the second half of the cleaning cycle: moving down, then right again via *Go_right_again*. This state and *Check_end* capture the last two conditional nodes in figure 5. Either the robot encounters a cliff, in which case it has finished cleaning and can transition to *Return*, or its horizontal displacement reaches the length of the vacuum nozzle, in which case a cleaning cycle has been completed, and *cycles* is incremented accordingly. The SM then checks the robot's battery, as seen in figure 6, and either transitions to *Return* if it is found to be below the threshold, or transitions to *Go_up* and begins a new cleaning cycle.

When it eventually transitions to *Return*, this instructs it to move to the left until it encounters a cliff, meaning it has reached the beginning of the very first panel, and can transition to the state *Dock*, moving into the docking station. Here it waits to recharge in the state *Sleep*, and then enters *Resume* and begins its operation once more as described.

4.1.3 Executive layer

The state machine **MidLevelSM** is responsible for converting the movement commands received from **PathPlanningSM** into voltages to send to the left and right motors of the robot. It receives the events defined in *MovementCommand* and uses the channels *target_speed:int*, *target_angle:int*, *speed_adjustment:int*, and *angle_adjustment:int* to send out the desired speed and angles to the **PID SMs** and receive the PID error output. These values are necessary for determining the correct voltages the SM should output to the motors by using the operations *output_left_motor(x:int)* and *output_right_motor(x:int)*. This output is done at this level rather than the functional layer because it requires additional state information PID does not have. Figure 9 presents the state machine with all state actions hidden in order to make it easier to read. Due to how verbose the diagram is we have colour coded it to make it easier to distinguish transitions: looping transitions are grey, others are blue if they are triggered by a turn and black if they are triggered by moving forward.

The SM uses the variable *orientation* to keep track of which way it is facing. It begins under the same assumption as **PathPlanningSM**, that the robot is facing upwards. Therefore it moves into the state *Go_up*. The SM has a set of states for moving (*Go_up*, *Go_side*, *Go_down*), one for stopping (*Stop*) and two for turning (*Turn_right*, *Turn_left*). Every state has the same set of entry actions. It sends out a constant target speed it aims to achieve to the **Linear Speed PID SM**, and receives a speed adjustment in response, which it needs to incorporate in order to near its target. It then follows the same pattern with the **Angular Speed PID**. It updates its left and right motor voltage

variables, *lspeed* and *rspeed* according to the adjustments retrieved from the **PID** SMs, and outputs these values to the motors.

The states are connected by transitions that verify the robot's orientation to determine the correct state to move to, and adjust the orientation accordingly, using the functions *turn_left* and *turn_right*. Apart from the PID adjustments to angular and vertical speed, we assume the robot must move forward at a constant speed. This is represented by the constants *movesp* and *stopsp*, which *lspeed* and *rspeed* are set to in transitions when the robot begins or stops moving.

The commands issued by **PathPlanningSM** must be performed continuously until the next command arrives—for instance, if **MidLevelSM** receives a command to move forward, it must send the appropriate voltages for moving forward to its motors repeatedly until the next command causes it to change these values and so forth. This is because PID needs multiple iterations to guide **MidLevelSM** to the correct motor outputs. To accomplish this, all states have a loop transition with a *sinceEntry(state)>=step* condition—every *step* units of time, all the actions of the state are repeated, until a new command arrives, and the SM transitions to a different state. These transitions are hidden in diagram 9 to make the image less verbose.

The state machine maps every movement command to a set of target speed and target angle constants to send to PID— *climb_speed*, *descent_speed*, *side_spped*, *left_angle*, *right_angle*, and *forward_angle*. The constants used for target values act as the “predefined set of values” used in accelerometer output comparisons in Avarind et al.'s work. The actual values depend on the hardware available and should be estimated via simulation, but we have established the following conventions: the target speed should be positive for moving up, negative for down, and zero for stopping or turning; the target angle should be positive for turning right, negative for turning left, and zero for any other action.

These rules allow us to use the same PID feedback on both motors, since they often require opposite adjustments—a -90° differential drive turn, for example, is performed by reversing the left motor, and outputting positively to the right motor. Therefore, if the **Angular PID** output determines the robot is not turning to the left enough, it will return a positive value – see **PID** implementation in the next sub-section – we can add to the right motor and subtract from the left motor. In the reverse of this condition, the value returned by the **PID** would be negative, and so adding and subtracting in the same way would still work. Similarly, the **Speed PID** will return a positive value if the robot is moving too fast, and this value will be subtracted from both motors, reducing the overall speed. These conventions are not made explicit in the model, but this could be done by declaring the constants as being of restricted positive or negative integer types.

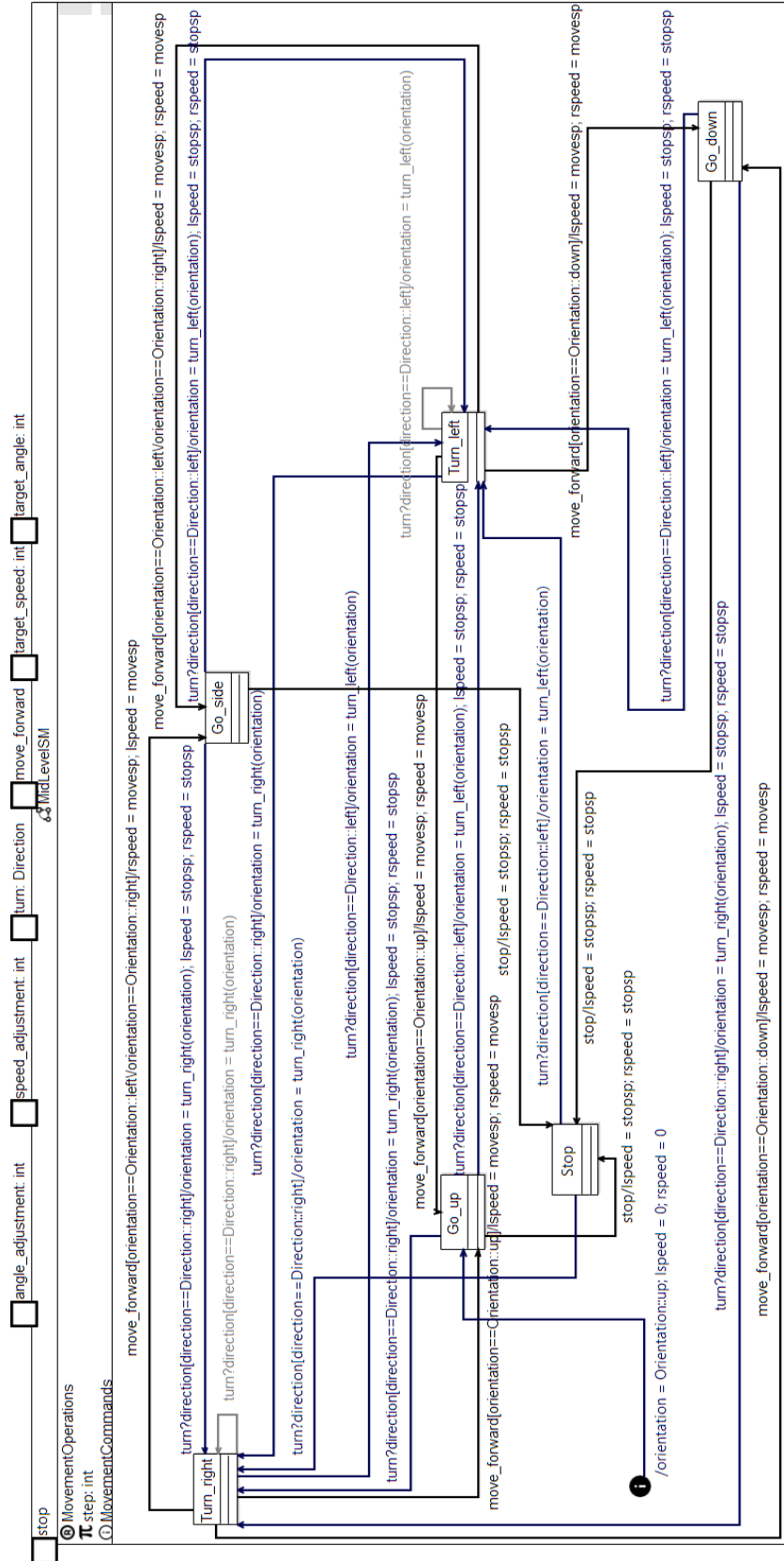


Figure 9: Middle level state machine, with state actions and sinceEntry loops removed (full diagram available in appendix A)

4.1.4 Functional layer

At the lowest layer of our model, a number of SMs process sensor data into variables that can be used by the higher layer SMs. The most critical component at this part of the hierarchy is the **PID SM**.

Avarind et al. include in the paper the use of PID “to control the navigation of the robot” [7]. They make no mention of what role exactly the PID plays in this task—it could be used to prevent the robot from steering too far to the left or right, regulating its speed, or to address the concern they highlighted earlier of the robot accelerating appropriately on inclined surfaces.

Due to the lack of information available on this aspect of the original robot, we have chosen to implement two PID controllers based on the data in the paper and our own assumptions. Avarind et al.’s experiment description indicates that PID is used to correct the robot’s orientation when it crosses a bump between panels, and as such we implement a **PID SM** to control the robot’s movement angle. This leaves us with the question of how the other issues, regulating speed and tackling incline, are handled. We assume the robot should move at approximately the same speed regardless of movement direction, and as such a constant can be assigned to the movement speed—*movesp* and *stopsp* in **MidLevelSM**. We could simply assign different speed constants for ascending and descending the panels to solve the incline problem in a similar manner, but considering the criticality of the issue – if the robot accelerates too much or too little on an incline it may fall and break – we decide on the implementation of another PID SM to continuously regulate the robot’s climb or descent speed, which we call the **linear speed PID SM**.

Both **PID SMs** work exactly in the same way. As such, our model contains only one **PID SM** and two references to it within the controller, handling linear and angular speed respectively. PID uses a feedback system to calculate a more appropriate output value based on the result of a previous iteration [30]. An error value is obtained by subtracting the target value we are trying to reach (as received from **MidLevelSM**) from the actual value observed (the linear/angular speed calculated from the accelerometers). This is the proportional error. Additionally to this, we calculate the integral of errors observed thus far in order to eliminate offsets accumulated over time, and the derivative error in order to adjust the rate of change of the output. The final error value is calculated by adding these three errors together, each multiplied by its own constant, as seen in the equation in [7].

Our implementation of PID (figure 10) stays true to this algorithm, receiving its target and actual values via the events *target:int* and *actual:int*, and calculating a discretised version of the errors to output over *err_output:int* on every iteration. The only modification we

introduce is resetting the prior error and integral to zero if the target value has changed. This is to address a problem raised by Balaji et al. [31], who highlighted PID's poor handling of sharp turns, and implemented an open loop controller to handle them separately. Rather than doing this, we erase the prior error history, as the robot's errors when turning left, for example, are irrelevant when it is trying to move forward.

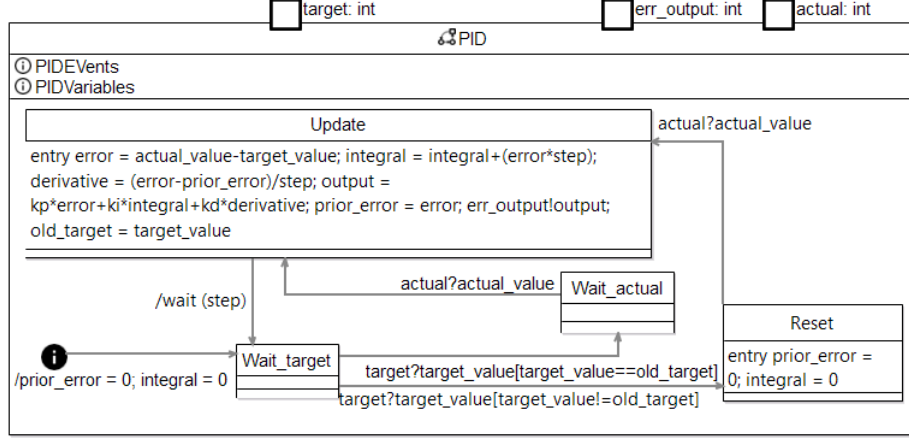


Figure 10: PID State machine

The actual value inputs are provided by **LinearSpeedSM** and **AngularSpeedSM**. **LinearSpeedSM** obtains an overall ascent or descent speed from the speed returned on the robot's left and right accelerometers' Z axis by averaging the two values. Similarly, **AngularSpeedSM** reads in both the left and right accelerometers and produces an angular speed using the formula depicted in the equation below [32], where D is the distance between the accelerometers. We retrieve the sign of one of the accelerometer values and apply this sign to the output to respect our imposed conventions for **MiddleLevelSM**. This is under the assumption that the accelerometers return negative values for acceleration to the left, and positive to the right. This assumption is rooted in the ability of the MMA7361 – the accelerometer used by Avarind et al.'s robot – to measure both positive and negative acceleration [33]. Both of these SMs, along with the other building blocks of the model, are available in Appendix A.

$$\omega = \sqrt{\frac{|a_{y2} - a_{y1}|}{D}}$$

The speeds used in **LinearSpeedSM** are retrieved from two references to **SpeedSM**, one for the left accelerometer and one for the right. This calculates the speed the robot is travelling based on its acceleration and time elapsed. Similarly, **DisplacementSM** calculates the distance it has travelled based on its speed and elapsed time, and outputs it to **PathPlanningSM**.

A state machine called **CleanSM** receives the Boolean values transmitted over the *clean* event by **PathPlanningSM** and turns the motor controlling both the brush and vacuum on or off accordingly.

It is worth noting that some of the inputs of these state machines overlap—for example, the input from the accelerometers is used both by the **SpeedSMs** and the **AngularSpeedSM**. Multisynchronisation is not possible in RoboChart. In order to work around the absence of this feature, we have created **InputDuplicationSM**, which takes in a value and outputs it on two separate events. This state machine is referenced wherever multiple copies of the same value are needed by different SMs. This solves the problem of allowing multiple SMs to use these values, but introduces dependency in the order of communication—**InputDuplicationSM** still outputs sequentially, and so the SM receiving its second output cannot receive it until the first output has reached a different SM. This could potentially lead to deadlock, but in the next chapter we verify the absence of this problem.

4.2 Model verification

In order to reason about our model we must define its constants and functions. Due to the complexity of our model, we aim to minimise the state space FDR must explore by restricting the values variables can take. Therefore, we define *int* and *real* as being the set $\{-1, 0, 1\}$. We set all constants to be within this set, and define all functions' output for every possible input. By inspection, this is sufficient to represent the different states of our SMs and overall model. This is because the decisions made in every SM are rooted in comparisons to constants and conditions based on the positive or negative nature of variables.

We define our own assertions to prove the properties we set out in requirement 10. We include a description of the assertions here, the full definitions in Appendix B, and their results in the next chapter. Ideally we would reason about the robot as a whole and verify against **Module**, but it is not feasible with the current version of RoboTool to check any assertions at this level of our model, as we will discuss in the next chapter. Instead we reason about all properties on the planning layer, and therefore verify them against **PathPlanningSM**.

We define a set of CSP processes capturing these properties. All of them are put in parallel with **RUN({tock})**, a process which accepts the event *tock*. This is done in order to be able to test these assertions on the timed semantics. Firstly, we set out to prove that the robot does not fall off the panels. For this purpose we define **NoFall** (figure 11), a CSP process that accepts and ignores any event until it receives a value via the ultrasonic channel. If the read value is below the cliff constant defined in the model, then **NoFall** proceeds as before. Otherwise, it turns left, right, or back, and resumes moving. By asserting that **PathPlanningSM** trace refines **NoFall** – that all of its

possible execution traces are a subset of **NoFall**'s traces – we assert that on the planning layer the robot is directed to always move away from the edge of the panels. There are a set of implicit assumptions here—that the cliff constant is adequate, that the software and hardware's response times are small enough to transmit and enforce the movement commands before it is too late, and that sensor noise is ignorable. We also rerun this assertion, swapping the two transitions from *Go_right_again* in **PathPlanningSM** to capture the second to last conditional as described in figure 5. This is to confirm our assumption in section 3 that this conditional does not respect the syntax and its branches should be swapped – item 5 in table 2.

```
Wait = Recurse([PathPlanningSM_move_forward.out, PathPlanningSM_turn.out.Direction_left,
                PathPlanningSM_turn.out.Direction_right, PathPlanningSM_clean.out.true, PathPlanningSM_clean.out.false,
                PathPlanningSM_charging.in, PathPlanningSM_stop.out, PathPlanningSM_displacement.in,
                PathPlanningSM_displacement.out, PathPlanningSM_battery_level.in], Wait)

[]
PathPlanningSM_ultrasonic.in?u -> CheckCliff(u)
CheckCliff(u) = let PathPlanningSM_cliff = 1 within
  if u < PathPlanningSM_cliff then Wait
  else
    PathPlanningSM_turn.out!Direction_left -> TurnedLeft
    []
    PathPlanningSM_turn.out!Direction_right -> TurnedRight
TurnedLeft = PathPlanningSM_move_forward.out -> Wait
[]
PathPlanningSM_turn.out.Direction_left -> TurnedBack
TurnedRight = PathPlanningSM_move_forward.out -> Wait
[]
PathPlanningSM_turn.out.Direction_right -> TurnedBack
TurnedBack = PathPlanningSM_move_forward.out -> Wait
[]
PathPlanningSM_turn.out.Direction_left -> Wait
NoFall = Wait ||| RUN({tock})
```

Figure 11: Definition of NoFall

Next, we assert that the robot returns to charge whenever it is running out of battery (requirement 10.a). We define the process **ReturnToCharge** similarly to **NoFall**, except that it accepts and ignores all but the *battery_level:int* event. Whenever the battery level *b* is too low, verified via **CheckBattery(b)**, the robot returns to the docking station and begins charging, via the processes **Return** and **Dock**, before starting over. This is under the assumption that the selected battery constant is large enough for the robot to be able to return to the docking station before running out of battery. This is reasonable as the constant can be calculated based on experimental results. If we deduce an average constant battery level required for the robot to move sideways on a panel a certain distance, and we know the length of a cleaning cycle, then we can combine this information into a function that can be maximised under constraints to determine an appropriate constant.

Having verified that the robot returns to charge when low on battery, we also check that it returns to its previous cleaning position afterwards (requirement 10.b). This is by far the most complex assertion, as it requires us to model cleaning cycles and keep track of their number. We define **CycleStart(n, MAX)** and **CycleEnd(n, MAX)** for this purpose, splitting the cycle in halves. The first moves up, then right, and the second moves down, then right. This split is necessary

because the end of a panel may be reached halfway through a cycle. The first argument is the current number of completed cycles. The second is the maximum allowed number of cycles—this is a restriction we impose on the process to prevent it from having infinite states, in which case FDR would not be able to reason about it. Before entering the start of a new cleaning cycle we verify via **CheckEnd(n , MAX)** that n is smaller than MAX . Depending on the ultrasonic and displacement input, these processes either recurse, or return to the docking station via **ReturnToDock(n , MAX)**. Within these processes, we abstract away from specific conditions for returning or moving forward, such as checking against the cliff constant. We are not interested in the specific behaviour by which the robot performs the cleaning cycle, only that it accurately keeps track of the cycle number.

The processes **ExitDock(n , MAX)** and **Resume(n , MAX)** capture the behaviour we are actually interested in here—leaving the docking station and returning to our previous position, as deduced in our equation in section 4.1. This is verified via synchronisation with **WaitForCycles** and **Measure(x)**, two processes that continuously capture the displacement and, when trying to return to cleaning, verify the value against our equation. **WaitForCycles** continuously receives and ignores the robot's displacement, until it receives a value via *cyclecount* from **Resume(n , MAX)**, and progresses to **Measure(x)**, which reads the displacement value and compares it to the robot's previous cleaning position. It is only when this value is reached that **Measure(x)** sends **Resume(n , MAX)** the *signal* event such that it can begin a new cleaning cycle. These parallel processes communicate over *cyclecount* and *signal*. Finally, **ReturnAfterCharging** is defined as this parallel composition, hiding *cyclecount* and *signal*, and we check that **PathPlanningSM** trace refines this process.

We argue that the CSP processes described thus far capture the intended properties by abstracting away all but the relevant behaviour. **NoFall** and **ReturnToCharge** are much simpler than the robot itself in definition, but allow a wider range of behaviour. For example, the former allows any sequence of events, except for moving forward after encountering a cliff. Similarly, the latter allows anything to occur when the battery is high enough, but enforces a sequence of events equivalent to the robot returning to the docking station whenever the battery is low. Trace refinement of these processes therefore reveals the presence of the properties they represent. While **ReturnAfterCharging** is more complicated, it still abstracts away the actual structure of cleaning cycles and returning to dock, being specific only about the counting of cycles and resuming cleaning. All of these processes are also examined using FDR's probe feature, which allows us to explore their possible execution traces, to further verify they accurately depict the properties required, and discard any information that is not relevant to these properties.

We divide requirement 10.c – that the robot completely covers the panels – into two assertions: the robot must move in the shape depicted in diagram 4, and it must reach the end of the panels.

The first is simple to prove. We reason about a version of **PathPlanningSM** that hides all but the events of *MovementCommands*. We then use only these events to define **Cycle**, the first half of a cycle, **Continue**, the second half, **GoBack**, and **ResumeCycling**, processes which capture only the movements the robot performs. We then assert that the version of **PathPlanningSM** that abstracts everything but movement trace refines **MovementShape**, defined as **ResumeCycling**, representing the first state of **PathPlanningSM**, in parallel with **RUN({tock})**.

The second assertion we can prove in two ways. We can expose the internal events signifying entry to the states *Go_right_again* and *Return* in **PathPlanningSM**, and write a process **ReachEnd** which transitions from the former to the latter when encountering an ultrasonic value above the cliff threshold. Here we have to begin using failure refinement rather than trace refinement—not only must the possible execution traces of **PathPlanningSM** be a subset of those belonging to this process, but the possible failures must be contained as well. Within the context of CSP, a failure is a pair containing a trace and a set of events which may be refused after the execution of that trace. Failure refinement enforces that both the ways a process can and cannot behave are contained in the ways the other process can and cannot behave. In our context this is important to ensure that not only does **PathPlanningSM** behave as we expect it to when encountering a cliff while going right, but it is also possible for it to encounter the cliff. This assertion is considerably simpler than our previous ones, but requires knowledge of how **PathPlanningSM** is modelled. An option which avoids this problem is to instead create a set of processes as described in **CleanAllPanels**, which keep track of the robot’s orientation, and always accept any value of ultrasonic. By using failures refinement again we ensure that the robot is able to accept an ultrasonic value above the threshold when moving right, so it is possible for it to reach the end of the panels. This coupled with our previous proof that the robot moves in the expected pattern is sufficient to demonstrate that it covers all of its assigned panels.

As seen before, **MovementShape** and **CleanAllPanels** are valid because they abstract all but the behaviour required to have a particular property. We check this by probing the processes. **ReachEnd** is simpler to validate, as we know from the definition of the model that the end of all the solar panels has been reached if and only if a transition is triggered from *Go_right_again* to *Return*.

Having described both the model and the processes used to verify it, we proceed to evaluate it against our requirements in the next chapter.

5 Evaluation

In this section we evaluate the model and assertion results against our requirements, and discuss the contributions of our work.

Our model depicts the functionality of the robot, as described by the paper, using consistent syntax throughout (**requirement 1**). Due to the ambiguity of some of the original descriptions and the omission of certain details we cannot be sure of its accuracy, but it conforms to all specifications provided, and we clearly specify additional assumptions.

We model both the components and their interactions (**requirement 2**) within the controller, otherwise the model would be invalid.

Whether the model is unambiguous, clear, and readable (**requirement 3**) is a subjective matter. The syntax of RoboChart does not permit us to be ambiguous in our model. However, the latter two properties are up to the reader. The SM descriptions are more verbose than [7]’s diagrams, because they specify the behaviour in a more complete manner. We argue that this does not detract from the model’s readability, with the exception of **MiddleLevelSM**, which is complex enough for its transition labels to be difficult to distinguish. We note that in RoboTool this is not a problem, as selecting a transition highlights its label, but in the exported diagrams this is not possible, and the purpose of each transition is not immediately clear. Currently, RoboTool allows us to mitigate this by changing the formatting of the figure, which we have used to colour code the diagram. Additionally, we could select shorter names for our variables, functions and constants, at the cost of these being less intuitive. We can also present the textual version of the diagram to clarify, but this means losing the benefits of a graphical representation. This issue would be mitigated if RoboTool allowed the user to change the orientation of transition labels—the use of vertical labels attached to the vertical lines in figure 9 would make much clearer what each label corresponds to. Additionally, automatically generating shorter labels is in discussion for future version of RoboTool in order to improve on this issue.

RoboChart requires us to define our variables, constants, operations, parameters and non-primitive datatypes as part of our model, fulfilling **requirements 4, 5, and 6**. The constants and variables are defined within interfaces, while all used operations and datatypes are defined at the model-level. All of these can be found in Appendix A.

Requirement 7 is for the model to state all assumptions about its platform. The controller explicitly declares all of its assumptions about the hardware’s provided operations within its required interfaces. These operations are outputting to the motors, the vacuum and the brush. Moreover, assumptions about the sensors provided by the hardware and the types of data they return are stated within the

controller’s events—*ultrasonic:int*, *battery_level:int*, *charging*, *acc_r:int*, and *acc_l:int*. We note that some assumptions are not captured within the model, but rather in our description. For example, our implementation of **AngularSpeedSM** relies on a property of the hardware used in [7]—the sensors return negative values for certain accelerations. Additionally, some assumptions arise from abstracting away part of the hardware’s functionality. Take for example Avarind et al.’s two safety measures for charging the robot, which we have not modelled, but assumed to be provided as part of the docking station.

Requirement 8 is for the model to state all assumptions made on the environment. Some of these assumptions, such as the presence of connecting rails, are provided by Avarind et al. and reinstated within our description of the model. However, it is not possible to state them within RoboChart. RoboTool allows for the annotation of a model, thus we could attach these assumptions as notes to the SM they regard, but they would have no semantic meaning.

We model all of the robot’s interaction with hardware as either calls to operations, such as *output_left_motor(x)*, or events provided by the platform, such as *ultrasonic:int*, thereby fulfilling **requirement 9**.

Our final requirement is to prove certain properties for the robot. First we must reason about the correctness of our model in terms of some basic properties. RoboTool defines a set of core assertions—it generates the assertions necessary to verify essential properties of all SMs, controllers, and the entire module, in both timed and untimed (ignoring timed constructs) semantics. The first assertion is a lack of deadlocks. This includes termination—if the element does not deadlock, but terminates, this assertion still fails. In this case, a second assertion verifies that the only deadlock is successful termination. Finally, two assertions verify that the element is deterministic and divergence-free. In table 3 we compile the results for every SM’s lack of termination (Term.), deadlock (Dead.), determinism (Det.) and lack of divergence (Div.).

SM	Term.	Dead.	Det.	Div.
InputDuplicationSM	✓	✓	✓	✓
SpeedSM	✓	✓	✓	✓
AngularSpeedSM	✓	✓	✓	✓
LinearSpeedSM	✓	✓	✓	✓
DisplacementSM	✓	✓	✓	✓
CleanSM	✓	✓	✓	✓
PID	✓	✓	✓	✓
MidLevelSM	✓	✓	X	✓
PathPlanningSM	✓	✓	✓	✓

Table 3: Core assertion results for state machines

The only failing assertion is the determinism of **MidLevelSM**. Debugging this in FDR we find that this is, as expected, the looping transitions. Not only does the counterexample provided by FDR confirm this, but removing the transitions and rerunning this assertion leads to a deterministic version of **MidLevelSM**. This nondeterminism is part of the behaviour we expected and therefore acceptable.

Some of these assertions did not pass at first, highlighting issues both in our model and in the language itself. The controller assertions initially failed due to RoboChart generating incorrect semantics for the multiple references to the state machine definitions. The three **InputDuplicationSM** references acted as one object, introducing unintended synchronisation between other state machines. We have raised this issue and it has been resolved. Debugging these assertion failures was particularly difficult as the generated semantics use internal events that are not documented and, in the case of multiple references to the same definition, share the same name despite being different entities. We argue that it may be useful for a future version of RoboChart to change its naming convention in these specific cases, or to make possible the translation from CSP trace counterexamples back to a format more clearly tied to the state machines defined. Currently, while the use of RoboChart does not require domain knowledge of robotics or CSP, it is not possible to debug any failed assertions without a deeper understanding of the generated CSP semantics. A translation back to RoboChart would erase this barrier.

While RoboTool defines these same assertions for the controller and module as a whole, these require reasoning about all the above SMs synchronised in parallel. This CSP process is so complex that we are unable to verify the assertions, even with the resources of the University's high performance computing cluster. Specifically, RoboTool performs a compression operation to optimise these assertions. This requires the enumeration of the SMs in parallel. It is during this enumeration that the executing cluster runs out of memory. Therefore with the current version of RoboTool it is not possible to verify that the robot's components, assembled together, are deadlock-free. While this is concerning and a great impediment towards using RoboTool to model and verify applications in industry, research is currently in progress on how to address these problems and improve the tool's optimisation strategy to handle larger examples.

The CSP processes we defined in 4.2 allow us to mathematically model the robot's properties set out in requirement 10. Our assertions about the refinement of these processes all pass, proving that, on the planning layer, the robot's decisions respect the imposed constraints. This is not complete proof that the robot as a whole has these properties, but as previously discussed it is currently impossible to reason about the entire module due to memory constraints. We

believe the planning layer proof is sufficient for the required properties, since they all relate to the robot’s movement decisions, and these are only calculated at this layer. This could, in future, be supplemented by additional proofs on the middle and executive layers that the respective SMs execute movement commands as expected. Our modified model (swapping the *Go_right_again* transitions) described in section 4.2 fails **NoFall**, proving our claim about the incorrectness of figure 5 in the original paper—the branches of the second to last conditional must be swapped, else the robot may fall of the panels.

Requirement	Outcome
1	Model conforms to provided specification
2	All components and interactions modelled
3	Unambiguous, other properties subjective
4	All variables and constants defined
5	Operations and parameters defined
6	Datatypes declared in unified manner
7	Assumptions captured in model where possible
8	Not possible in RoboChart
9	All interactions modelled
10.a	Proved using <i>returnToCharge</i>
10.b	Proved using <i>returnAfterCharging</i>
10.c	Proved using <i>movementShape</i> and <i>cleanAllPanels</i>

Table 4: Summary of requirement outcomes

Additionally to these requirements, we aimed to test RoboChart’s ability to model this expressive platform, and to represent multiple layers of abstraction, as well as provide the first model of a PID controller in the language. We have found RoboChart to be perfectly suitable for these three tasks, and in the process have contributed a number of bug reports and improvement suggestions for the language:

1. We have raised the issue of multi-synchronisation and whether this is a useful feature for the language.
2. In using multiple references to the same state SM definition in a controller we have discovered a bug in the generated semantics for SM references that has since been fixed.
3. As part of discovering the above bug we highlighted the difficulties of reading the FDR traces generated from the model’s semantics, leading to a discussion on the potential of translating these traces back to RoboChart.
4. Our model has proved that RoboChart’s current optimisations technique are not suitable for all of its applications

Finally, our model has provided further evidence that RoboChart is suitable for modelling real-world applications. It has proved the correctness of the algorithms described in Avarind et al.’s paper and formalised the requirements of a hardware platform for the robot.

6 Conclusion

In conclusion, this project provides a case study of using RoboChart to model a pre-existing, real-world application. Upon comparison of the available robotic domain-specific languages for modelling, RoboChart has been selected as the most appropriate for the task. We highlight the issues we found in Avarind et al.'s documentation and argue the benefits of constructing a model of the robot to describe its functionality more clearly and reason about its properties.

We have built our model relying on all available information about the application and its hardware as well as our documented assumptions. We have used CSP to define a set of processes capturing the desired properties of the robot, and RoboTool along with FDR to prove that these properties are satisfied. Our model respects the original specification, and defines all the robot's components and their interactions unambiguously. We have defined a set of requirements for the quality, accuracy, and readability of our model, as well as for using the model to demonstrate that the robot conforms to a set of properties defined in the original paper. Our work fully satisfies nine of the twelve requirements, and provides suggestions for future improvements on the three requirements that were not fully met.

Our result is a model that fulfils our requirements in terms of accuracy and completeness of its definition, with the exception of some hardware and environment assumptions that we document but cannot capture in RoboChart. The model is unambiguous by definition, but more verbose and thus potentially more difficult to understand in some places. However, it allows us to reason about its mathematical properties. We prove the non-termination, determinism, as well as absence of deadlocks and divergences of all components of the robot, but not the components linked together, as this is impossible to do for a model of our complexity in the current version of RoboTool. We also demonstrate that the planning layer fulfils all the required properties.

Our work contributes bug reports and future enhancement requests to RoboChart and RoboTool. It demonstrates the ability of the former to capture a very expressive platform, and includes the first definition of PID in the language, a common control scheme in robotics, particularly valuable in the application itself, where it performs the critical task of preventing the robot from falling and damaging itself.

Future work should focus on simplifying the existing component definitions in order to make them easier to read, and providing further proof at lower layers of the model that the required properties are met. Our project provides further evidence of RoboChart's advantages as a robotic DSL. Currently, code generation is not a feature of RoboTool. However, the ability to formally verify models implies that in the future, we could generate controllers that are correct by construction.

Appendix A

In this appendix we include diagrams of all components of the model mentioned in section 4 that were not depicted within the chapter.

Figure 12 depicts the datatype and enumeration definitions. Figure 13 defines the operations used throughout the state machines. The mathematical functions defined for the model are presented in figure 14. The function used to calculate linear speed in **LinearSpeedSM** is defined in figure 15. The functions used by **MidLevelSM** to adjust the *orientation* variable can be seen in figure 16.

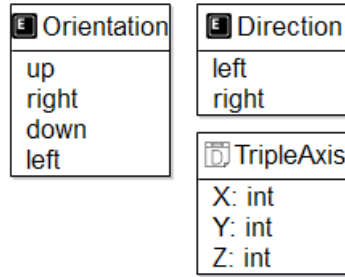


Figure 12: Datatypes and enumerations

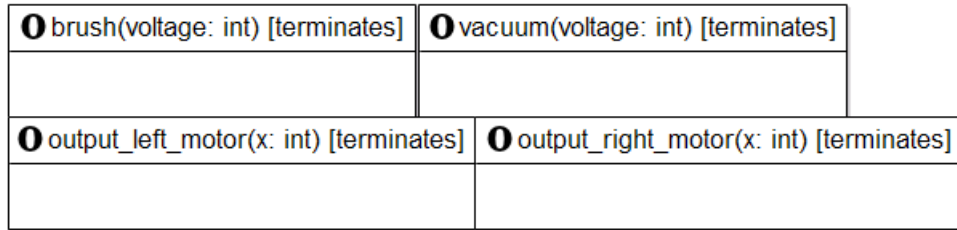


Figure 13: Operation definitions for cleaning (top) and moving (bottom)

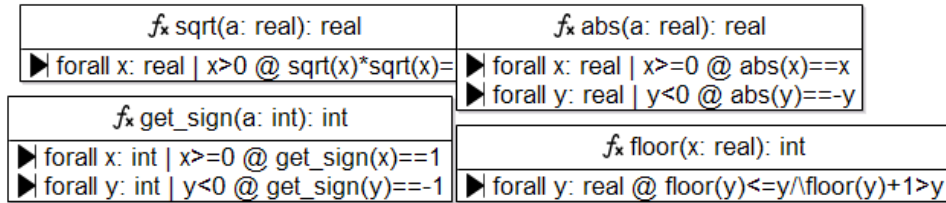


Figure 14: Mathematical functions

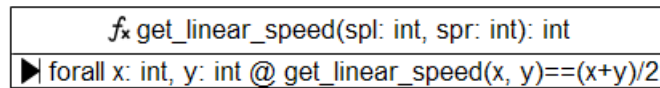


Figure 15: Linear speed function used in LinearSpeedSM

Appendix A

f_x turn_right(x: Orientation): Orientation	
▶ forall a: Orientation	a==Orientation::up @ turn_right(a)==Orientation::right
▶ forall b: Orientation	b==Orientation::right @ turn_right(b)==Orientation::down
▶ forall c: Orientation	c==Orientation::down @ turn_right(c)==Orientation::left
▶ forall d: Orientation	d==Orientation::left @ turn_right(d)==Orientation::up

f_x turn_left(x: Orientation): Orientation	
▶ forall a: Orientation	a==Orientation::up @ turn_left(a)==Orientation::left
▶ forall b: Orientation	b==Direction::left @ turn_left(b)==Orientation::down
▶ forall c: Orientation	c==Orientation::down @ turn_left(c)==Orientation::right
▶ forall d: Orientation	d==Orientation::right @ turn_left(d)==Orientation::up

Figure 16: Turn functions used in MidLevelSM

Figures 17, 18, and 19 depict the interfaces defined, required and provided within the module.

SensorConstants	MidVars	PIDVariables
π cliff: int	π climb_speed: int	\times prior_error: int
π nozzle: int	π descent_speed: int	\times integral: int
\times u: int	π side_speed: int	π kp: int
\times d: int	π forward_angle: int	π ki: int
\times b: int	π left_angle: int	π kd: int
π battery_low: int	π right_angle: int	\times error: int
π sleep_time: int	\times lspeed: int	\times derivative: int
π acc_distance: int	\times rspeed: int	\times output: int
	\times angle: int	π step: int
	\times speed_adj: int	\times target_value: int
	\times angle_adj: int	\times actual_value: int
	π turn_speed: int	\times old_target: int
	\times direction: Direction	
	\times orientation: Orientation	
	π movesp: int	
	π stopsp: int	

Figure 17: Interfaces declaring variables and constants

SensorEvents ⚡ ultrasonic: int ⚡ battery_level: int ⚡ charging	MovementCommands ⚡ move_forward ⚡ turn: Direction ⚡ stop	MovementEvents ⚡ acc_l: TripleAxis ⚡ acc_r: TripleAxis ⚡ speed_l: TripleAxis ⚡ speed_r: TripleAxis ⚡ angle: real
CleaningEvents ⚡ clean: boolean	SpeedEvents ⚡ acc: TripleAxis ⚡ sp: TripleAxis	
PIDEvents ⚡ actual: int ⚡ target: int	DisplacementEvent ⚡ displacement: int	

Figure 18: Interfaces defining events

Appendix A

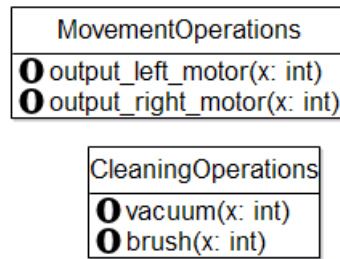


Figure 19: Interfaces defining operations

Figure 20 shows the robotic platform that provides the interfaces and events required by the controller. The full module is depicted in figure 21.

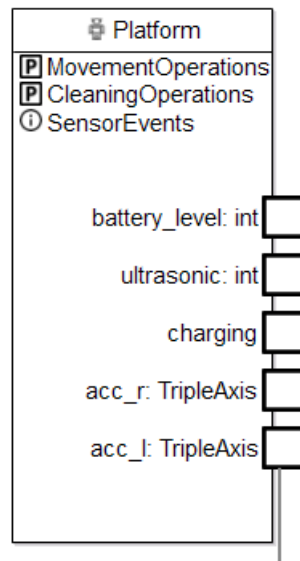


Figure 20: Robotic platform

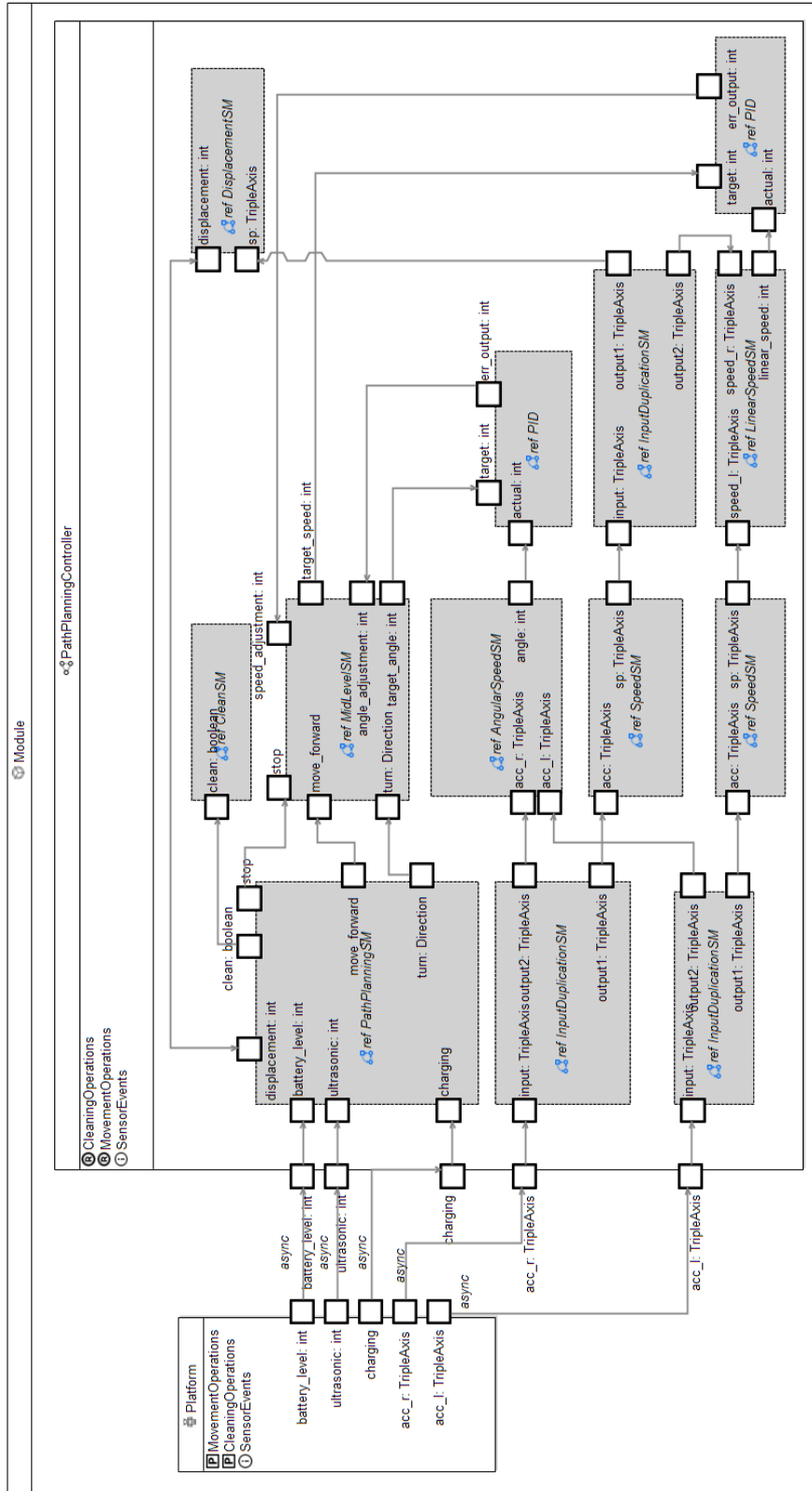


Figure 21: Overview of module

Appendix A

The linear and angular speeds required by **PID** are provided by **LinearSpeedSM** and **AngularSpeedSM**, shown in figures 22 and 23.

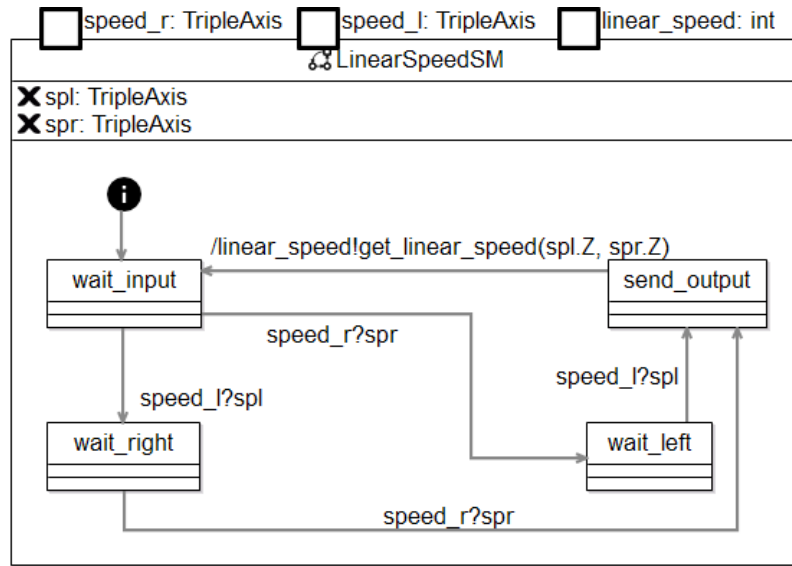


Figure 22: LinearSpeedSM definition

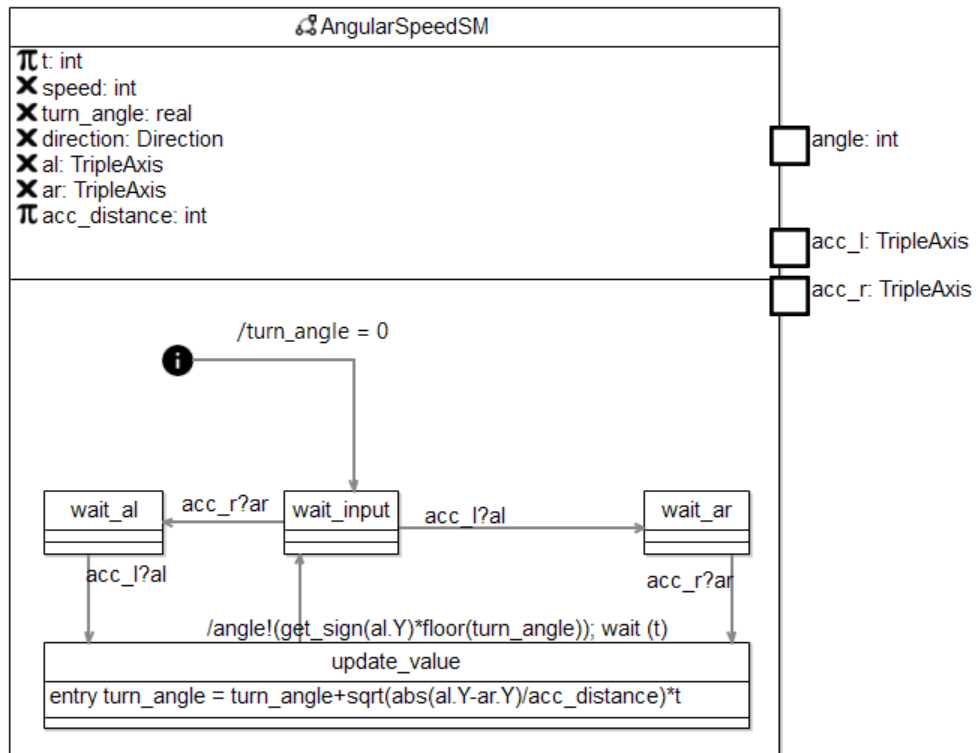


Figure 23: AngularSpeedSM definition

The speed required by these state machines is provided by two different **SpeedSMs**—one for the left speed and one for the right, both references to the definition in figure 24.

Appendix A

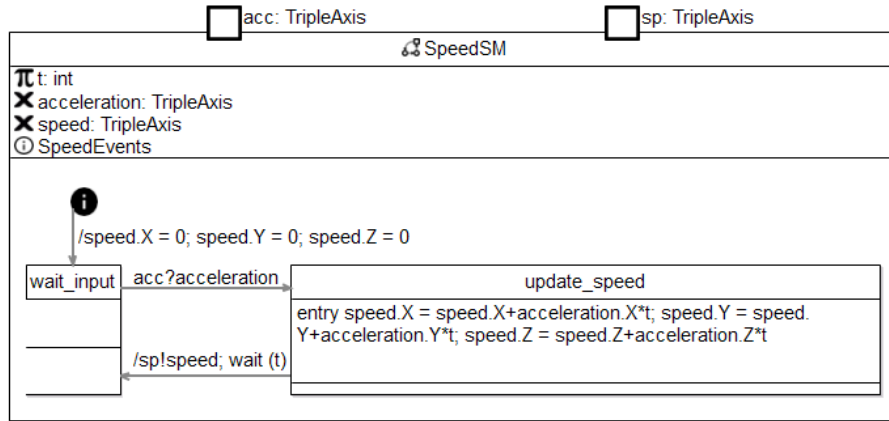


Figure 24: SpeedSM definition

Using a similar formula, the displacement required by **PathPlanningSM** is calculated in **DisplacementSM**, as seen in figure 25.

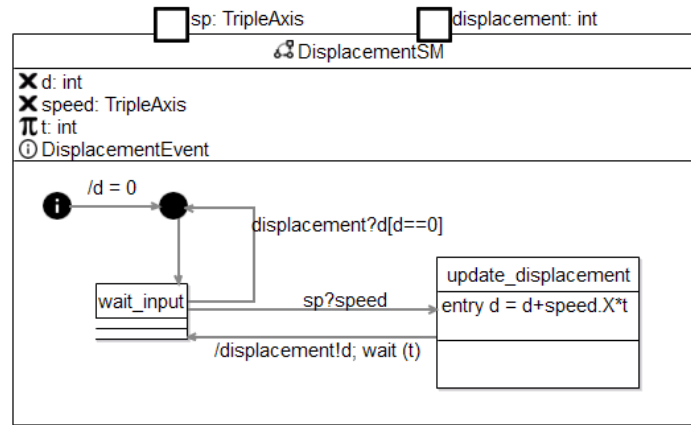


Figure 25: DisplacementSM definition

The cleaning commands sent out by **PathPlanningSM** are executed by **CleanSM** – figure 26 – using the cleaning operations defined in figure 19.

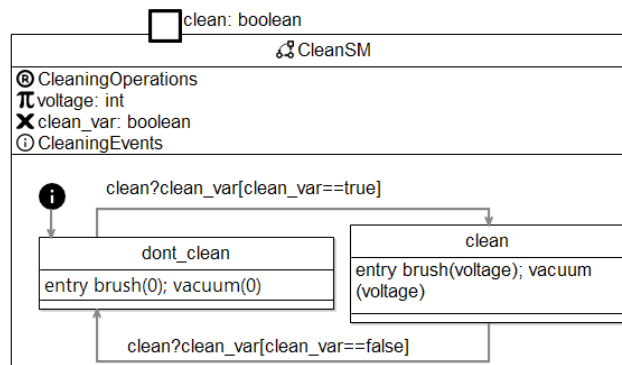


Figure 26: CleanSM definition

Appendix A

Wherever multisynchronisation of events is needed, **InputDuplicationSM** is used to duplicate an event. This simple state machine can be seen in figure 27.

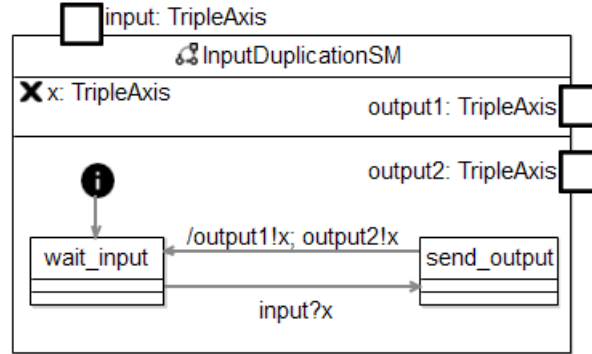


Figure 27: InputDuplicationSM definition

Due to the complexity of **MidLevelSM**, we include the definitions that were omitted in figure 9 using RoboChart's textual language. These are automatically generated by RoboTool from the diagram we produced using the graphical editor. Figure 28 contains the full state definitions, including the entry actions omitted in figure 9. The *sinceEntry* conditional loops that were also removed are presented in figure 29.

```

state Go_up {
  entry target_speed ! climb_speed ; speed_adjustment ? speed_adj ; target_angle !
  forward_angle ; angle_adjustment ? angle_adj ; lspeed = lspeed - speed_adj -
  angle_adj ; rspeed = rspeed - speed_adj + angle_adj ; output_left_motor(
  lspeed) ; output_right_motor( rspeed)
}
state Turn_right {
  entry target_speed ! turn_speed ; speed_adjustment ? speed_adj ; target_angle !
  right_angle ; angle_adjustment ? angle_adj ; lspeed = lspeed - speed_adj -
  angle_adj ; rspeed = rspeed - speed_adj + angle_adj ; output_left_motor(
  lspeed) ; output_right_motor( rspeed)
}
state Go_down {
  entry target_speed ! descent_speed ; speed_adjustment ? speed_adj ;
  target_angle ! forward_angle ; angle_adjustment ? angle_adj ; lspeed =
  lspeed - speed_adj - angle_adj ; rspeed = rspeed - speed_adj + angle_adj ;
  output_left_motor( lspeed) ; output_right_motor( rspeed)
}
state Turn_left {
  entry target_speed ! turn_speed ; speed_adjustment ? speed_adj ; target_angle !
  left_angle ; angle_adjustment ? angle_adj ; lspeed = lspeed - speed_adj -
  angle_adj ; rspeed = rspeed - speed_adj + angle_adj ; output_left_motor(
  lspeed) ; output_right_motor( rspeed)
}
state Go_side {
  entry target_speed ! side_speed ; speed_adjustment ? speed_adj ; target_angle !
  forward_angle ; angle_adjustment ? angle_adj ; lspeed = lspeed - speed_adj -
  angle_adj ; rspeed = rspeed - speed_adj + angle_adj ; output_left_motor(
  lspeed) ; output_right_motor( rspeed)
}
state Stop {
  entry target_speed ! turn_speed ; speed_adjustment ? speed_adj ; target_angle !
  forward_angle ; angle_adjustment ? angle_adj ; lspeed = lspeed - speed_adj -
  angle_adj ; rspeed = rspeed - speed_adj + angle_adj ; output_left_motor(
  lspeed) ; output_right_motor( rspeed)
}
  
```

Figure 28: State definitions of MidLevelSM with actions

Appendix A

```
transition t22 {
  from Turn_right
  to Turn_right
  trigger
  condition sinceEntry ( Turn_right ) >= step
}
transition t23 {
  from Go_side
  to Go_side
  trigger
  condition sinceEntry ( Turn_left ) >= step
}
transition t24 {
  from Go_up
  to Go_up
  trigger
  condition sinceEntry ( Go_up ) >= step
}
transition t25 {
  from Turn_left
  to Turn_left
  trigger
  condition sinceEntry ( Turn_left ) >= step
}
transition t26 {
  from Stop
  to Stop
  trigger
  condition sinceEntry ( Stop ) >= step
}
transition t27 {
  from Go_down
  to Go_down
  trigger
  condition sinceEntry ( Go_down ) >= step
}
```

Figure 29: Looping *sinceEntry* transitions of *MidLevelSM*

Finally, we present **MidLevelSM** as two separate diagrams: figure 30 hiding all *turn* transitions, and figure 31 hiding all *move_forward* and *stop* transitions. Hiding these transitions serves to present less information at once such that it is easier to follow along with the descriptions provided in 4.1.2.



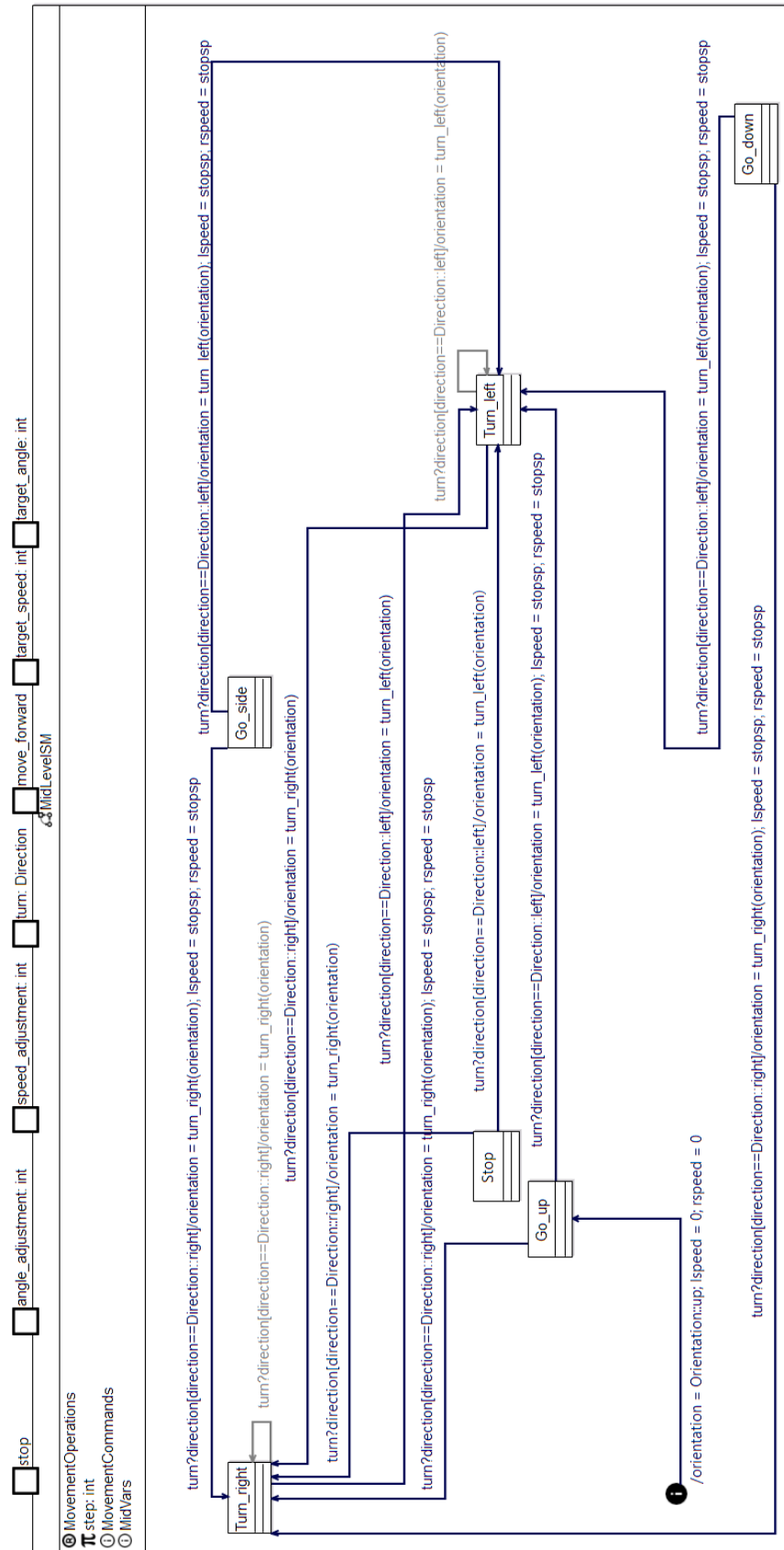


Figure 31: MidLevelSM definition hiding move_forward and stop transitions

Appendix B

In this section we include the full definitions of all CSP processes used for assertions.

We define the processes **Recurse(S, P)** and **NRecurse(S, P)**, where **S** is a set of events and **P** is a process. **Recurse(S, P)** accepts any event in **S** and proceeds as **P**. We use this in the following definitions in order to accept events but effectively discard them—for example, the way **Wait** from **NoFall** recurses when offered any event other than an ultrasonic input. **NRecurse(S, P)** provides the same behaviour, except it uses internal choice of the events in **S**, rather than external, like **Recurse(S, P)**. In other words, **NRecurse(S, P)** is nondeterministic, while **Recurse(S, P)** is deterministic.

The full explanations of how each process works are provided in chapter 4. In this appendix we merely provide the CSP specifications along with a very brief summary of the points in 4.2.

The definitions of **Recurse(S, P)**, **NRecurse(S, P)**, and **NoFall** can be seen in figure 32. The process **Wait** recurses until it receives an ultrasonic input, at which point **CheckCliff(u)** verifies whether a cliff has been encountered. **TurnedLeft**, **TurnedRight**, and **TurnedBack** enforce the behaviour of turning before continuing to move, if a cliff lies forward.

```

csp NoFall csp-begin
  Recurse(S,P) = [] ev : S @ ev -> P
  NRecurse(S, P) = [~] ev : S @ ev -> P

  Wait = Recurse([PathPlanningSM_move_forward.out, PathPlanningSM_turn.out.Direction_left,
    PathPlanningSM_turn.out.Direction_right, PathPlanningSM_clean.out.true, PathPlanningSM_clean.out.false,
    PathPlanningSM_charging.in, PathPlanningSM_stop.out, PathPlanningSM_displacement.in,
    PathPlanningSM_displacement.out, PathPlanningSM_battery_level.in], Wait)

  []
  PathPlanningSM_ultrasonic.in?u -> CheckCliff(u)
  CheckCliff(u) = let PathPlanningSM_cliff = 1 within
    if u < PathPlanningSM_cliff then Wait
    else
      PathPlanningSM_turn.out!Direction_left -> TurnedLeft
      []
      PathPlanningSM_turn.out!Direction_right -> TurnedRight
  TurnedLeft = PathPlanningSM_move_forward.out -> Wait
  []
  PathPlanningSM_turn.out.Direction_left -> TurnedBack
  TurnedRight = PathPlanningSM_move_forward.out -> Wait
  []
  PathPlanningSM_turn.out.Direction_right -> TurnedBack
  TurnedBack = PathPlanningSM_move_forward.out -> Wait
  []
  PathPlanningSM_turn.out.Direction_left -> Wait
  NoFall = Wait ||| RUN({tock})
csp-end

```

Figure 32: *Recurse(S, P), NRecurse(S, P) and NoFall definitions*

Figure 33 depicts the definition of **ReturnToCharge**, the CSP specification of the property of returning to the docking station when the battery runs low. **WaitBattery** waits for input of the battery level, which is checked using **CheckBattery(b)**. If the level is low, **Return** and **Dock** move back to the docking station.

Appendix B

```

csp ReturnToCharge csp-begin
  WaitBattery = Recurse({|PathPlanningSM_move_forward.out, PathPlanningSM_turn.out.Direction_left,
    PathPlanningSM_turn.out.Direction_right, PathPlanningSM_clean.out.true,
    PathPlanningSM_clean.out.false, PathPlanningSM_charging.in,
    PathPlanningSM_stop.out, PathPlanningSM_displacement.out,
    PathPlanningSM_displacement.in, PathPlanningSM_ultrasonic.in|},
    WaitBattery)

  []
  PathPlanningSM_battery_level.in?b -> CheckBattery(b)

  CheckBattery(b) = let PathPlanningSM_battery_low = 0 within
    PathPlanningSM_turn.out!Direction_left ->
    if b > PathPlanningSM_battery_low then Wait
    else Return

  Return = PathPlanningSM_turn.out!Direction_left -> PathPlanningSM_move_forward.out ->
    PathPlanningSM_clean.out!false -> Dock

  Dock = PathPlanningSM_ultrasonic.in?u ->
    if u < PathPlanningSM_cliff then Dock
    else PathPlanningSM_turn.out!Direction_left -> PathPlanningSM_move_forward.out ->
    PathPlanningSM_charging.in -> WaitBattery

  ReturnToCharge = WaitBattery ||| RUN({tock})
csp-end

```

Figure 33: ReturnToCharge definition

The more complicated **ReturnAfterCharging** captures the property of resuming cleaning from the spot the robot reached before running low on battery. The processes **CycleStart(n, MAX)**, **CycleEnd(n, MAX)**, **ReturnToDock(n, MAX)** and **CheckEnd(n, MAX)** model cleaning and returning to the docking station, while storing the number of completed cleaning cycles, as seen in figure 34.

```

csp ReturnAfterCharging csp-begin
  CycleStart(n, MAX) = Recurse({|PathPlanningSM_move_forward.out, PathPlanningSM_turn.out.Direction_left,
    PathPlanningSM_turn.out.Direction_right, PathPlanningSM_clean.out.true,
    PathPlanningSM_clean.out.false, PathPlanningSM_battery_level.in|},
    CycleStart(n, MAX))
  []
  PathPlanningSM_ultrasonic.in?u -> CycleStart(n, MAX) [] CycleEnd(n, MAX)
  []
  PathPlanningSM_displacement.out!0 -> CycleStart(n, MAX) [] CycleEnd(n, MAX)
  []
  PathPlanningSM_displacement.in?d -> CycleStart(n, MAX)

  CycleEnd(n, MAX) = Recurse({|PathPlanningSM_move_forward.out, PathPlanningSM_turn.out.Direction_left,
    PathPlanningSM_turn.out.Direction_right, PathPlanningSM_clean.out.true,
    PathPlanningSM_clean.out.false, PathPlanningSM_displacement.out|},
    CycleEnd(n, MAX))
  []
  PathPlanningSM_ultrasonic.in?u -> CycleEnd(n, MAX) [] ReturnToDock(0, MAX)
  []
  PathPlanningSM_battery_level.in?b -> CycleEnd(n, MAX) [] ReturnToDock(n, MAX)
  []
  PathPlanningSM_displacement.in?d -> CycleEnd(n, MAX) [] CheckEnd(n, MAX)

  ReturnToDock(n, MAX) = Recurse({|PathPlanningSM_move_forward.out, PathPlanningSM_turn.out.Direction_left,
    PathPlanningSM_turn.out.Direction_right, PathPlanningSM_clean.out.true,
    PathPlanningSM_clean.out.false, PathPlanningSM_ultrasonic.in,
    PathPlanningSM_battery_level.in, PathPlanningSM_displacement.in,
    PathPlanningSM_displacement.out|},
    ReturnToDock(n, MAX))
  []
  PathPlanningSM_charging.in -> ExitDock(n, MAX)

  CheckEnd(n, MAX) = if n < MAX then CycleStart(n+1, MAX) else ReturnToDock(0, MAX)

```

Figure 34: ReturnAfterCharging: performing normal cleaning task and returning to the docking station

The processes **ExitDock(n, MAX)** and **Resume(n, MAX)** model the robot's return to its position prior to docking. This position is measured

Appendix B

by the processes **WaitForCycles** and **Measure(x)**, depicted in figure 35.

```

channel signal
channel cyclecount: int

ExitDock(n, MAX) = Recurse([|PathPlanningSM_stop.out, PathPlanningSM_turn.out|], ExitDock(n, MAX))
[]
PathPlanningSM_move_forward.out -> Resume(n, MAX)
Resume(n, MAX) = PathPlanningSM_turn.out!Direction_right -> PathPlanningSM_move_forward.out ->
PathPlanningSM_displacement.out!0 -> cyclecount!n -> signal -> CycleStart(n, MAX)

WaitForCycles = PathPlanningSM_displacement.in?d -> WaitForCycles
[]
cyclecount?x -> Measure(x)
Measure(x) = PathPlanningSM_displacement.in?d ->
if d < x * PathPlanningSM_nozzle * 2 then Measure(x)
else signal -> WaitForCycles
ReturnAfterCharging = (Resume(0, 1) [|{|cyclecount, signal|}|] WaitForCycles)
\ {|cyclecount, signal|} ||| RUN({tock})

csp-end

```

Figure 35: ReturnAfterCharging: resuming cleaning after charging

Figure 36 contains the definition of **MovementShape**—the sequence of movement commands performed in a cleaning cycle and in returning to dock and resuming cleaning. **PathPlanningSMMovements**, illustrated in figure 37, abstracts away all but the movement commands of **PathPlanningSM**, such that we can assert that it refines **MovementShape**.

```

csp MovementShape csp-begin
ResumeCycling = PathPlanningSM_turn.out!Direction_right -> PathPlanningSM_move_forward.out -> Cycle

Cycle = PathPlanningSM_turn.out!Direction_left -> PathPlanningSM_move_forward.out ->
PathPlanningSM_turn.out!Direction_right -> PathPlanningSM_move_forward.out ->
Continue

Continue = PathPlanningSM_turn.out!Direction_right -> PathPlanningSM_move_forward.out ->
PathPlanningSM_turn.out!Direction_left -> PathPlanningSM_move_forward.out ->
(Cycle [] GoBack)

GoBack = PathPlanningSM_turn.out!Direction_left -> PathPlanningSM_turn.out!Direction_left ->
PathPlanningSM_move_forward.out -> PathPlanningSM_turn.out!Direction_left ->
PathPlanningSM_move_forward.out -> PathPlanningSM_stop.out ->
PathPlanningSM_turn.out!Direction_left -> PathPlanningSM_turn.out!Direction_left ->
PathPlanningSM_move_forward.out -> ResumeCycling

MovementShape = ResumeCycling ||| RUN({tock})
csp-end

```

Figure 36: MovementShape: definition of what a cleaning cycle should look like in terms of movement commands

```

csp PathPlanningSMMovements csp-begin
P_PathPlanningSM = let
PathPlanningSM_cliff = 1
PathPlanningSM_nozzle = 1
PathPlanningSM_battery_low = 0
PathPlanningSM_sleep_time = 1
PathPlanningSM_acc_distance = 1
within PathPlanningSM_O(0, PathPlanningSM_cliff, PathPlanningSM_nozzle,
PathPlanningSM_battery_low, PathPlanningSM_sleep_time,
PathPlanningSM_acc_distance)

PathPlanningSMMovements = P_PathPlanningSM \ {|PathPlanningSM_displacement.in, PathPlanningSM_displacement.out,
PathPlanningSM_ultrasonic.in, PathPlanningSM_battery_level.in,
PathPlanningSM_clean.out, PathPlanningSM_charging.in|}

csp-end

```

Figure 37: PathPlanningSMMovements: PathPlanningSM definition abstracting away all but the movement commands

Appendix B

Similarly, **ReachEnd**, seen in figure 38, depicts the property of the robot reaching the end of all its panels, by exposing the state machine's state information and modelling a cliff being reached while moving right at the end of a cleaning cycle. **PathPlanningSMEndState** is a definition of **PathPlanningSM** that shows only those events relevant to **ReachEnd**. This can be seen in figure 39.

```
csp ReachEnd csp-begin
  WaitGoRightAgain = PathPlanningSM_ultrasonic.in$u -> WaitGoRightAgain
  |~|
  PathPlanningSM_enteredV."PathPlanningSM_Go_right_again" -> CheckPanelEdge
  |~|
  PathPlanningSM_enteredV."PathPlanningSM_Return" -> WaitGoRightAgain
  CheckPanelEdge = PathPlanningSM_ultrasonic.in$u ->
    if u < PathPlanningSM_cliff then WaitGoRightAgain
    else PathPlanningSM_enteredV."PathPlanningSM_return" -> WaitGoRightAgain
  ReachEnd = WaitGoRightAgain
csp-end
```

Figure 38: ReachEnd definition

```
csp PathPlanningSMEndState csp-begin
  PathPlanningSMEndState = let
    PathPlanningSM_cliff = 1
    PathPlanningSM_nozzle = 1
    PathPlanningSM_battery_low = 0
    PathPlanningSM_sleep_time = 1
    PathPlanningSM_acc_distance = 1
  within
    PathPlanningSM_VS_O(0, PathPlanningSM_cliff,
      PathPlanningSM_nozzle, PathPlanningSM_battery_low,
      PathPlanningSM_sleep_time, PathPlanningSM_acc_distance)
  | \ { | PathPlanningSM_enteredV."PathPlanningSM_Go_right_again",
    PathPlanningSM_enteredV."PathPlanningSM_Return",
    PathPlanningSM_ultrasonic.in | }
csp-end
```

Figure 39: PathPlanningSMEndState definition: PathPlanningSM abstracting all but the ultrasonic event, and the events signaling the entering of states Go_Right_Again and Return

Appendix B

```

csp CleanAllPanels csp-begin
  FacingUp = NRecurse({|PathPlanningSM_stop.out, PathPlanningSM_move_forward.out,
    PathPlanningSM_ultrasonic.in, PathPlanningSM_displacement.in,
    PathPlanningSM_displacement.out, PathPlanningSM_battery_level.in,
    PathPlanningSM_charging.in, PathPlanningSM_clean.out|},
    FacingUp)

  |~|
  PathPlanningSM_turn.out.Direction_left -> FacingLeft
  |~|
  PathPlanningSM_turn.out.Direction_right -> FacingRight

  FacingLeft = NRecurse({|PathPlanningSM_stop.out, PathPlanningSM_move_forward.out,
    PathPlanningSM_ultrasonic.in, PathPlanningSM_displacement.in,
    PathPlanningSM_displacement.out, PathPlanningSM_battery_level.in,
    PathPlanningSM_charging.in, PathPlanningSM_clean.out|},
    FacingLeft)

  |~|
  PathPlanningSM_turn.out.Direction_left -> FacingDown
  |~|
  PathPlanningSM_turn.out.Direction_right -> FacingUp

  FacingDown = NRecurse({|PathPlanningSM_stop.out, PathPlanningSM_move_forward.out,
    PathPlanningSM_ultrasonic.in, PathPlanningSM_displacement.in,
    PathPlanningSM_displacement.out, PathPlanningSM_battery_level.in,
    PathPlanningSM_charging.in, PathPlanningSM_clean.out|},
    FacingDown)

  |~|
  PathPlanningSM_turn.out.Direction_left -> FacingRight
  |~|
  PathPlanningSM_turn.out.Direction_right -> FacingLeft

```

Figure 40: *CleanAllPanels: possible behaviours while moving up, left, or down*

The definition of **CleanAllPanels** is captured in figures 40 and 41. This describes the expected behaviours of the robot when facing up, down, left, and right. It enforces the possibility of encountering a cliff when moving to the right. Failure refinement allows us to use this process to ensure that the robot can in fact reach the end of all panels by encountering a cliff at the end of a cleaning cycle. If this were not the case, the failure refinement assertion would fail.

```

FacingRight = NRecurse({|PathPlanningSM_stop.out, PathPlanningSM_move_forward.out,
  PathPlanningSM_displacement.in, PathPlanningSM_displacement.out,
  PathPlanningSM_battery_level.in, PathPlanningSM_charging.in,
  PathPlanningSM_clean.out|},
  FacingRight)

  |~|
  PathPlanningSM_turn.out.Direction_left -> FacingUp
  |~|
  PathPlanningSM_turn.out.Direction_right -> FacingDown
  |~|
  PathPlanningSM_ultrasonic.in?u -> CheckUltrasonic(u)

  CheckUltrasonic(u) = if u < PathPlanningSM_cliff then FacingRight
    else (
      PathPlanningSM_turn.out.Direction_left -> FacingUp
      |~|
      PathPlanningSM_turn.out.Direction_right -> FacingDown
    )

  CleanAllPanels = FacingUp ||| CHAOS({tock})
csp-end

```

Figure 41 *CleanAllPanels: possible behaviours while moving right, checking whether a cliff has been reached*

Appendix B

Finally, figure 42 contains the definitions of all the assertions using these processes. All assertions are executed in both the timed and untimed semantics, and use the same constant definitions. All assertions pass.

```
assertion noFall : PathPlanningSM refines NoFall in the traces model with constants
  cliff set to 1,
  nozzle set to 1,
  battery_low set to 0,
  sleep_time set to 1,
  acc_distance set to 1

assertion returnToCharge : PathPlanningSM refines ReturnToCharge in the traces model with constants
  cliff set to 1,
  nozzle set to 1,
  battery_low set to 0,
  sleep_time set to 1,
  acc_distance set to 1

assertion returnAfterCharging : PathPlanningSM refines ReturnAfterCharging in the traces model with constants
  cliff set to 1,
  nozzle set to 1,
  battery_low set to 0,
  sleep_time set to 1,
  acc_distance set to 1

assertion movementShape : PathPlanningSMMovements refines MovementShape in the traces model

assertion reachEnd : PathPlanningSMEndState refines ReachEnd in the failures model

assertion cleanAllPanels : PathPlanningSM refines CleanAllPanels in the failures model
```

Figure 42: Assertion definitions

Bibliography

- [1] REN21, "Renewables 2017 Global Status Report," [Online]. Available: http://www.ren21.net/gsr-2017/chapters/chapter_01/chapter_01/. [Accessed 20 October 2018].
- [2] International Energy Agency, Next Generation Wind and Solar Power, Paris: OECD/IEA, 2016.
- [3] A. Singh, Talking renewables : a renewable energy primer for everyone, San Rafael, California: Morgan & Claypool Publishers, 2018.
- [4] S. A. Sulaiman, H. H. Hussain, N. S. H. N. Leh and M. S. I. Razali, "Effects of Dust on the Performance of PV Panels," *International Journal of Mechanical and Mechatronics Engineering*, vol. 5, no. 10, pp. 2028-2033, 2011.
- [5] R. Kohli and K. L. Mittal, Developments in Surface Contamination and Cleaning, Volume 3: Methods for Removal of Particle Contaminants, vol. 3, Oxford: William Andrew, 2011, p. 154.
- [6] H. Kawamoto and T. Shibata, "Electrostatic cleaning system for removal of sand from solar panels," *Journal of Electrostatics*, vol. 73, pp. 65-70, 2015.
- [7] A. G. G. Vasan, G. K. T.S.B and N. B. R, "A Control Strategy for an Autonomous Robotic Vacuum Cleaner for Solar," in *2014 Texas Instruments India Educators' Conference (TIIEC)*, Bangalore, India, 2014.
- [8] Texas Instruments, "Code Composer Studio (CCS) Integrated Development Environment (IDE)," [Online]. Available: <http://www.ti.com/tool/CCSTUDIO>. [Accessed 10 December 2018].
- [9] OMG Architecture Board ORMSC, "Model driven architecture (MDA)," 9 July 2001. [Online]. Available: <https://www.omg.org/cgi-bin/doc?ormsc/2001-07-01>. [Accessed 26 October 2018].
- [10] C. Atkinson and T. Kühne, "Model-Driven Development: A Metamodeling Foundation," *IEEE Software*, vol. 20, no. 5, pp. 36-41, 2003.

Bibliography

- [11] Object Management Group, "OMG Unified Modeling Language," March 2003. [Online]. Available: <https://www.omg.org/cgi-bin/doc?formal/03-03-01>. [Accessed 26 October 2018].
- [12] D. C. Schmidt, "Model-Driven Engineering," *Computer*, vol. 39, no. 2, pp. 25-31, 2006.
- [13] A. Nordmann, N. Hochgeschwender, D. Wigand and S. Wrede, "A survey on domain-specific modeling and languages in robotics," *Journal of Software Engineering in Robotics*, vol. 7, no. 1, pp. 75-99, 2016.
- [14] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis and J. Woodcock, "RoboChart: a state-machine notation for modelling and verification of mobile and autonomous robots.," York, United Kingdom, 2016.
- [15] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi and M. Ziane, "RobotML, a Domain-Specific Language to Design,," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Berlin, 2012.
- [16] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, R. S. Patrick Tessier, H. Dubois and F. Terrier, "Papyrus UML: an open source toolset for MDA,," in *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, Enschede, The Netherlands, 2009.
- [17] S. Fleury, M. Herrb and R. j. Chatila, "GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture," in *International Conference on Intelligent Robots and Systems*, 1997, 1997.
- [18] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan and F. Ingrand, "GenoM3: Building middleware-independent robotic components," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2010.
- [19] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gauffillet, F. Lang and F. Vernadat, "Fiacre: an Intermediate Language for Model Verification in the Topcased Environment," in *ERTS 2008*, Toulouse, France, 2008.
- [20] D. Socci, P. Poplavko, S. Bensalem and M. Bozga, *Modeling Mixed-critical Systems in Real-time BIP*, Taipei, Taiwan: 1st workshop on Real-Time Mixed Criticality Systems, 2013.

Bibliography

- [21] M. Foughali, F. Ingrand and A. Mallet, "GenoM3 Templates: from Middleware Independence to Formal Models Synthesis," arXiv, 2018.
- [22] J.-J. Borrelly, E. Coste-Maniere, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon and N. Turro, "The ORCCAD Architecture," *The International Journal of Robotics Research*, vol. 17, no. 4, pp. 338-359, 1998.
- [23] S. Alexandrova, Z. Tatlock and M. Cakmak, "RoboFlow: A Flow-based Visual Programming Language for Mobile Manipulation Tasks," in *IEEE International Conference on Robotics and Automation (ICRA)*, Seattle, Washington, 2015.
- [24] J. Rumbaugh, I. Jacobson and G. Booch, *Unified Modeling Language Reference Manual*, Pearson Higher Education, 2004.
- [25] A. Cavalcanti, P. Ribeiro, A. Miyazawa, A. Sampaio, M. C. Filho and A. Didier, "RoboSim Reference Manual," 2019.
- [26] S. D. Brookes, C. A. R. Hoare and A. W. Roscoe, "A theory of communicating sequential processes," *Journal of the ACM (JACM)*, vol. 31, no. 3, pp. 560-599, 1984.
- [27] University of Oxford, "FDR4 - The CSP Refinement Checker," [Online]. Available: <https://www.cs.ox.ac.uk/projects/fdr/>. [Accessed 18 April 2019].
- [28] P. Pierce, "Software verification and Validation," in *IEEE Technical Applications Conference. Northcon/96*, Seattle, 1996.
- [29] G. Vasan, *RE: Autonomous Robotic Vacuum Cleaner for Solar Panels*, 2018.
- [30] D. Sellers, "An Overview of Proportional plus Integral plus Derivative Control and Suggestions for Its Successful Application and Implementation," in *Proceedings of the First International Conference for Enhanced Building Operations*, Austin, Texas, 2001.
- [31] V. Balaji, M. Balaji, M. Chandrasekaran, M. Khan and I. Elamvazuthi, "Optimization of PID Control for High Speed Line Tracking Robots," *Procedia Computer Science*, vol. 76, pp. 147-154, 2015.
- [32] Kionix, "Using Two Tri-Axis Accelerometers for Rotational Measurements," 9 July 2015. [Online]. Available:

Bibliography

<http://kionixfs.kionix.com/en/document/AN019%20Using%20Two%20Tri-Axis%20Accelerometers%20for%20Rotational%20Measurements.pdf>. [Accessed 2 April 2019].

- [33] NXP, $\pm 1.5g$, $\pm 6g$ *Three Axis Low-g Micromachined Accelerometer*, 2008.