UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA - CAMPUS DE RECIFE
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FRANCISCO WILSON RODRIGUES JÚNIOR

A RoboTool plug-in for RoboWorld

Recife

2023

FRANCISCO WILSON RODRIGUES JÚNIOR

A RoboTool plug-in for RoboWorld

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação

Orientador: Gustavo Henrique Porto de Carvalho

Recife

2023

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

R696r Rodrigues Júnior, Francisco Wilson
A robotool plug-in for roboworld / Francisco Wilson Rodrigues Júnior. – 2023.
90 f.: il., fig.

Orientador: Gustavo Henrique Porto de Carvalho.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2023.
Inclui referências.

1. Engenharia de software. 2. Robótica. 3. Linguagem natural controlada. I. Carvalho, Gustavo Henrique Porto de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2023-172

Francisco Wilson Rodrigues Júnior

"A RoboTool plug-in for RoboWorld"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 21 de setembro de 2023.

BANCA EXAMINADORA

ACKNOWLEDGEMENTS

I am very grateful to God for giving me inspiration and strength to build this work in the last years and to be essential in my journey. Also, I am grateful to my wife, my brother and my grandmother who helped me by providing good energies, motivation and being my basis, providing strength as well. Also, I would like to thank my parents who always helped me the whole way and I know that they will be present in all moments of my life. I am also grateful to my supervisor for all the support and patient necessary to produce this work.

ABSTRACT

Developing robotic systems is a challenging task due to the inherently underlying complexity. Additionally, the lack of customised techniques and tools means that the current practice of software engineering for robotics is outdated. Therefore, model-driven software engineering, as opposed to simulation- and code-centric approaches, has been advocated for robotics. Considering this, the RoboStar framework, via its associated tool (RoboTool), provides a collection of domain-specific languages tailored for different aspects of the development of robotic systems. One of these languages is RoboWorld: a controlled natural language (CNL) for specifying operational requirements. In this work, we develop a RoboTool plug-in dealing with RoboWorld. This plug-in has a graphical user interface that enables the customisation of project-specific dictionaries. Moreover, it provides surface and structural editors of RoboWorld documents. Integration with underlying natural-language processing techniques and tools is transparent, and, thus, hidden from the end user. From an intermediate representation of RoboWorld documents, which is derived automatically, the plug-in also has automatic support for checking well-formedness conditions, and generating a formal CyPhyCircus semantics. The RoboTool plug-in for RoboWorld was validated considering three case studies: a rescue drone, a ranger robot, and a foraging robot.

Keywords: robotics; controlled natural language; grammatical framework; robostar; cyphycircus; tool.

RESUMO

O desenvolvimento de sistemas robóticos é uma tarefa desafiadora devido à complexidade associada. Além disso, a falta de técnicas e ferramentas específicas faz com que a prática atual de Engenharia de Software para robótica esteja desatualizada. Portanto, aplicar técnicas de desenvolvimento baseado em modelos, em oposição às abordagens centradas em simulação e código, tem sido defendido pela Engenharia de Software. Considerando isto, o framework RoboStar, através de sua ferramenta (RoboTool), fornece uma coleção de linguagens específicas de domínio adaptadas para diferentes aspectos do desenvolvimento de sistemas robóticos. Uma dessas linguagens é RoboWorld: uma linguagem natural controlada (CNL) para especificar requisitos operacionais. Neste trabalho, desenvolveu-se um plug-in de RoboTool para lidar com RoboWorld. Este plug-in possui uma interface gráfica que permite a manutenção de dicionários específicos de cada projeto. Além disso, fornece editores de superfície e estruturais para documentos RoboWorld. A integração com técnicas e ferramentas de processamento de linguagem natural é transparente e, portanto, escondida do usuário final. A partir de uma representação intermediária de documentos RoboWorld, que é derivada automaticamente, o plug-in também verifica automaticamente condições de boa formação e gera uma semântica formal em CyPhyCircus. O plug-in de RoboTool para RoboWorld foi validado considerando três estudos de caso: um drone de resgate, um robô andarilho e um robô coletor de recursos.

Palavras-chaves: robótica; linguagem natural controlada; grammatical framework, robostar; cyphycircus; ferramenta.

LIST OF FIGURES

LIST OF CODE

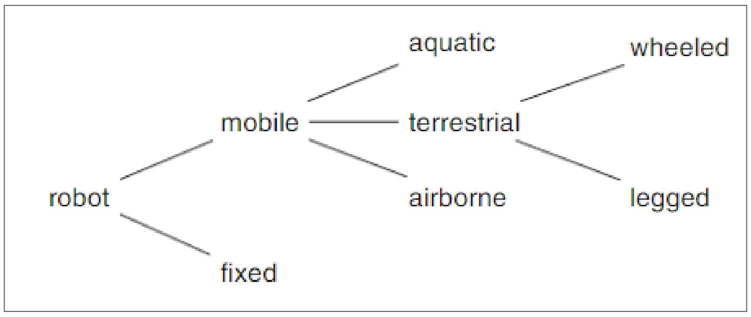
CONTENTS

1 INTRODUCTION

Robotic systems are real-time systems able to make decisions and actions based on perception and information obtained from the environment in a bounded time interval (MONTHE; NANA; KOUAMOU, 2022). This information is collected by sensors and provided to the control software, which makes the robot interact with its environment via actuators. Robotic systems are, therefore, composed by physical elements (e.g., sensors, actuators), but also by an immaterial part, which contains the knowledge (software) that controls the robot. This control enables the robot to operate in a complex, dynamic and unstructured environment.

The difference between robotic systems and simple automation is around the following question: how complex are their actions? Generally speaking, simple automation is not flexible, while robotic systems are more flexible and dynamic (BEN-ARI; MONDADA, 2018). Robotic systems can be classified in some categories according to the context they are inserted. Figure 1 presents an overview of such a classification.

Figure 1 – Robotic Systems Classification.



Source: (BEN-ARI; MONDADA, 2018)

The main distinction is concerned with the robot being mobile or fixed. Fixed robots work in a specific context, like the industry, and it performs repetitive tasks. In contrast, mobile robots are more flexible. They perform tasks in larger, ill-defined and unknown environments. Mobile robots can be further classified as aquatic, terrestrial and airbone. However, this is not a general categorisation, since it is also possible to have amphibious robots. Terrestrial robots can have legs or wheels, while aerial robots can be characterised as fixed- or rotary-wing.

Another important classification is related to how autonomous the robot is. For instance, there are mobile robots that can inspect remote places and recover objects under the control of an operator. These robots cannot be considered as autonomous. Some of these robots can be semi-autonomous, being able to realise some activities automatically (e.g., stabilise the flight assuming that the human pilot have provided the appropriate flight path) (BEN-ARI; MONDADA, 2018). Some robots can be regarded as fully autonomous. For instance, robotic vacuum cleaners are widely available and used because they are able to navigate autonomously an indoor environment that contains many obstacles.

Developing robotic systems is a challenging task due to the inherently underlying complexity of the requirements. Additionally, the lack of customised techniques and tools means that the current practice of software engineering for robotics is outdated (CAVALCANTI et al., 2021). Therefore, model-driven software engineering, as opposed to simulation- and code-centric approaches, has been advocated for robotics (CAVALCANTI et al., 2021). Moreover, as robotics systems are being increasingly applied to critical domains, such as healthcare, establishing the trustworthiness of such systems also pose a key challenge for design and assurance.

In light of the aforementioned challenges (i.e., moving to a model-driven development approach and reaching high trustworthiness levels), the RoboStar group (a centre of excellence in software engineering for robotics) has proposed a framework for rigorous modelling, verification, simulation and testing of mobile and autonomous robots (CAVALCANTI et al., 2021).

The RoboStar framework comprises a collection of domain-specific languages tailored for different aspects of the development of robotic systems. For instance, RoboChart (MIYAZAWA et al., 2019) is an event-based and diagrammatic notation similar to the Unified Modelling Language (UML) for the design of robotic systems, RoboSim (CAVALCANTI et al., 2019) is a cycle-based notation (also UML-like) for simulation, and RoboWorld (CAVALCANTI; BAXTER; CARVALHO, 2021) is a controlled natural language (CNL) for specifying operational requirements. In what follows, we focus on RoboWorld, which is the language closely related to this work.

1.1 OPERATIONAL REQUIREMENTS

Robots affect and are affected by their surrounding environment. As a consequence, it is typical to have in mind restrictions that must be satisfied for the robots to operate well; for example, one may have made assumptions about the weather, the weight of the robot, and the layout of rooms. Rarely, however, these restrictions are recorded precisely or at all. The usual code-centric approach adopted in software development for robotics often leads to tests that take these restrictions into account, but no record beyond the test base, if any, is normally produced.

In using models in testing and proof, however, we often need to have a record of assumptions about the environment. For example, tests generated from a model that does not cater for environment assumptions may characterise invalid scenarios and may be, therefore, useless. In addition, properties of the system may depend fundamentally on assumptions of the environment. For instance, a robot that starts too close to an obstacle may not be able to avoid it in time. An account of operational requirements is, therefore, an important design artefact.

RoboWorld, part of the RoboStar framework, is a controlled natural language for writing these additional documents that describe operational requirements of a robotic system for use in simulation, test generation, and proof. As a controlled natural language, RoboWorld provides an accessible notation to describe operational requirements, and to communicate them to stakeholders who must ensure they are met (such as end users). On the other hand, RoboWorld's formal semantics enables its use in rigorous verification techniques, with automatic generation insulating roboticists from the need to deal with the mathematical notations used to describe the semantics when generating tests or even carrying out proofs.

Natural language processing techniques can be statistical or symbolic (RANTA, 2011). Statistical approaches assume that a large dataset of (raw) text is available, from which techniques such as machine learning extract processing rules by creating models. Differently, symbolic approaches rely on grammars to define rules for analysing and producing valid text; these rules define a controlled natural language. While statistical approaches are more general, since they can process unrestricted text, inferring the correct interpretation of the text is a challenge due to the huge variety of possible writing styles. The control imposed by symbolic approaches can make this inference process easier, since we restrict ourselves to a controlled subset of styles. The challenge is to achieve a compromise between naturalness, expressiveness, and control.

RoboWorld is devised as a controlled natural language for the following two reasons. First, as mentioned, operational requirements of robotic systems are frequently left implicit and, thus, we do not have large datasets to develop statistical models. Second, the structure imposed by a symbolic approach enables us to provide automatically a formal semantics for such requirements. Nevertheless, RoboWorld is a natural, expressive and extensible language, even if controlled.

Providing tool support for RoboWorld is an essential task, since the ultimate goal is to provide roboticists with a natural-language notation, yet controlled, for describing operational requirements from which a formal semantics is derived automatically.

1.2 RESEARCH PROBLEM

The main research problem of this work is posed as follows: how can we provide appropriate tool support for RoboWorld, such that a formal semantics can be derived automatically from controlled natural-language specifications of operational requirements? Answering this research question unfolds into meeting the following research goals:

- Goal 1 (**G1**): provide means for editing RoboWorld documents;
- Goal 2 (**G2**): check well-formedness conditions of RoboWorld documents;
- Goal 3 (**G3**): translate RoboWorld documents into an intermediate representation;
- Goal 4 (**G4**): generate formal semantics of RoboWorld documents.

According to (LUTEBERGET, 2019), there are two predominant paradigms when writing sentences to adhere to a CNL: structural and surface editing. In structural editing, the user mostly follows a structural approach (for instance, clicking on predefined possibilities) that prevents the writing of invalid sentences according to the grammar of the CNL. In surface editing, the user inputs texts with varying degrees of guidance from the editor. In such an approach, it is possible to write sentences that are invalid. Therefore, the validity of the sentences needs to be checked afterwards. Goal G1 is associated with providing the user with a combination of these two editing styles.

In addition to checking adherence to defined CNL, it is necessary to analyse whether some well-formedness conditions are met. For example, the specification should be consistent with respect to the dimensionality of the environment; if the robot is going to operate on a 2D environment (i.e., the z-axis is not relevant for the specific application), the RoboWorld document should only refer to two dimensions (i.e., x- and y-axes). Goal G2 aims at mechanising such verifications.

The semantics of a RoboWorld document should be defined in terms of an intermediate representation (IR) of that document. With this representation, we can insulate the semantics generation from some evolutions of the RoboWorld language. Goal G3 concerns the automatic translation of RoboWorld documents into the corresponding intermediate representations.

The semantics of RoboWorld documents is provided using CyPhyCircus (FOSTER et al., 2020; MUNIVE; STRUTH; FOSTER, 2020), a state-rich and hybrid version of a process algebra for refinement cast in the Unifying Theories of Programming (UTP) of Hoare and He (HOARE; JIFENG, 1998), and formalised in the theorem prover Isabelle (FOSTER et al., 2018). Goal G4 implies in mechanising the generation of such a semantics.

1.3 OVERVIEW OF SOLUTION

The four aforementioned goals are achieved in this work by developing a RoboTool[1] plug-in for RoboWorld. In this section, we provide an overview of such a solution. To illustrate it, we rely on a running example: a rescue drone. It is a robot able to fly and search within an environment with aims to reach objects there.

In this application, the arena (that it, the area in which the robot is expected to work) must be three-dimensional, since the robot can move in three different coordinates. In the arena, there are two particular regions of interest: the origin, where the robot is positioned initially, and the target, the place the drone should reach. Additionally, the control software is designed considering that the robot is initially orientated towards the target region. These assumptions about the arena and the robot are recorded by the following RoboWorld sentences.

```
## ARENA ASSUMPTIONS ##
The arena is three-dimensional.
The arena has an origin region.
The arena has a target region.

## ROBOT ASSUMPTIONS ##
The robot is initially at the origin.
The robot is initially orientated towards the target.
```

Another important assumption is that the target region is far apart from the origin; in practical terms, it would not be reasonable to employ a rescue drone to search for something that is already in the origin region. Without this assumption, the RoboStar framework could consider a test where the drone takes off, and immediately finds the target. This scenario is valid from the control software perspective, but not realistic. Assumptions about the origin and the target regions are recorded as follows.

[1] Default tool of the RoboStar framework: <https://robostar.cs.york.ac.uk/robotool/>.

```
## ELEMENT ASSUMPTIONS ##
The origin has an x-width of 1 m and a y-width of 1 m.
The target has an x-width of 1 m and a y-width of 1 m.
The distance from the target to the origin is greater than 1 m.
```

RoboWorld documents also record information about how the environment is affected and affects the control software. For instance, the events `found` and `origin` are input events provided from the robotic platform to the control software to indicate that the robot is within the target and origin regions, respectively. These assumptions are recorded as follows.

```
## MAPPING OF INPUT EVENTS ##
The event found occurs when the robot is inside the target.
The event origin occurs when the robot is inside the origin.
```

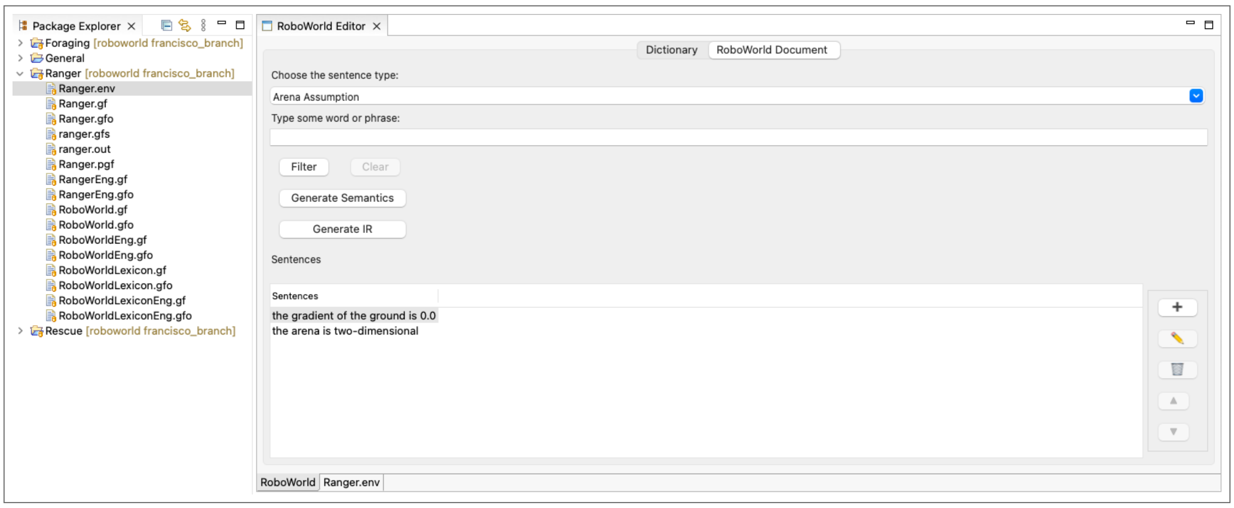
The syntax of RoboWorld is defined using the Grammatical Framework (GF) (RANTA, 2011), which provides support for inflection paradigms (for example, singular and plural forms), as well as agreement between elements of a sentence (for instance, the subject-verb number agreement), for more than 35 languages.

The RoboTool plug-in for RoboWorld developed in this work integrates with GF in order to support surface and structural editing of RoboWorld documents. Figure 2 presents a screenshot of this plug-in. It also allows the user to edit a project-specific dictionary; for instance, the nouns `target` and `origin` are specific to the rescue drone example. When the dictionary is edited, the tool dynamically rebuilds the RoboWorld grammar.

The developed plug-in also checks a number of well-formedness conditions (e.g., the dimensionality consistency, as explained before) that are required in order to generate the associated intermediate representation (IR). The IR is an intermediate representation of the RoboWorld document in terms of an object-oriented model. It is the basis for deriving the corresponding CyPhyCircus semantics. To facilitate the inspection of the semantics, the plug-in produces a LaTeX-based representation.

The derived RoboWorld semantics has many applications. By combining it with the semantics of the control software (given by its RoboChart model), the assumptions on the environment can be taken into account in simulation, formal verification, and testing. However, using the semantics of RoboWorld documents in order to exploit formal verification, simulation and testing of robotic systems is not within the scope of this work.

Figure 2 – Screenshot of the RoboTool Plug-in for RoboWorld.



Choose the sentence type: Arena Assumption

Type some word or phrase:

Sentences

the gradient of the ground is 0.0

the arena is two-dimensional

Source: Current Author

1.4 SUMMARY OF MAIN CONTRIBUTIONS

In summary, the main contributions of this work are the following ones. In (BAXTER et al., 2023), we give a concise account of the developed RoboTool plug-in for RoboWorld.

- A GUI for editing project-specific dictionaries;
- A GUI for surface and structural editing of RoboWorld documents;
- Tool support for checking well-formedness conditions of RoboWorld documents;
- Tool support for generating an intermediate representation of RoboWorld documents;
- Tool support for deriving a CyPhyCircus semantics;
- Validation of the RoboWorld plug-in in the context of a number of case studies.

1.5 STRUCTURE

This document is structured as follows. Chapter 2 discusses the necessary background information: the RoboStar framework, the Grammatical Framework, and the language CyPhyCircus. Chapter 3 covers aspects of RoboWorld documents: the underlying syntax, the intermediate representation, and its formal semantics. Chapter 4 details the RoboTool plug-in for RoboWorld, and Chapter 5 presents its use in the context of two case studies: a ranger and a foraging robot. Finally, Chapter 6 concludes this document by addressing related and future work.

2 BACKGROUND

In this chapter, we present the main concepts and theories that are necessary to better understand the present work. In Section 2.1, we discuss the RoboStar framework that enables the development of safe robotic systems. Afterwards, in Section 2.2, we present the Grammatical Framework (GF), which is used in this work to support the writing of operational requirements according to the grammar of a controlled natural language. Finally, in Section 2.3, we present the fundamentals of the language CyPhyCircus, which is used to give formal semantics to RoboWorld documents.

2.1 THE ROBOSTAR FRAMEWORK

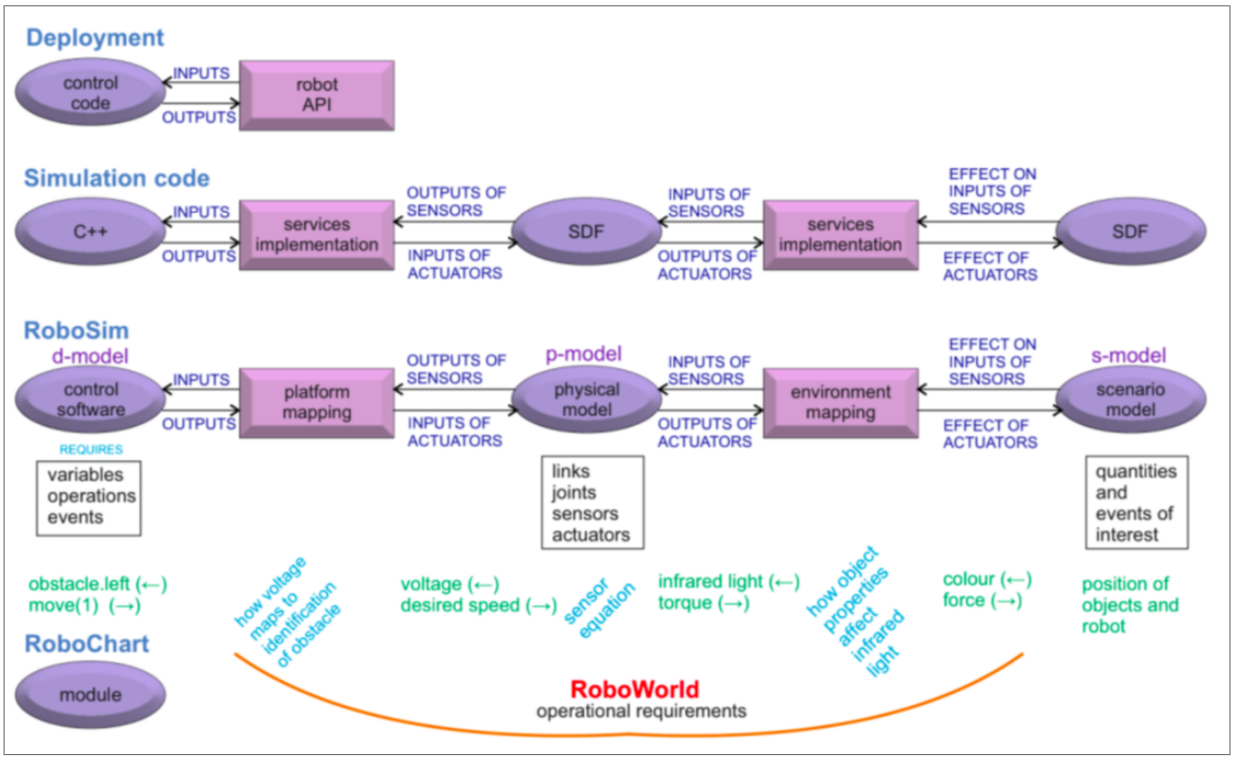
RoboStar emerged as a framework for leveraging software engineering for robotics. It has a strong focus on formal modelling and verification, aiming at situations where high trustworthiness levels are required, and therefore supports a variety of analytical technologies to model checking, theorem proving and testing (YE; FOSTER; WOODCOCK, 2022). Tool support is provided by RoboTool, an Eclipse-based application.

As mentioned before, the RoboStar framework comprises a collection of domain-specific languages. These languages provide a solid foundation for software engineering for robotics. Figure 3 shows an overview of the RoboStar framework and how it integrates into the development of robotic systems.

RoboChart is a profile of the Unified Modelling Language (UML) for the modelling of robotic applications. A RoboChart module is composed by a control software, described in terms of state machines, that interacts with the robotic platform interface. This abstract design is refined in terms of RoboSim models.

As indicated in Figure 3, RoboSim targets the simulation level. It includes a notation for describing control software (d-models), physical models (p-models), and simulation scenarios (s-models). At this level, the control software operates in a cycle fashion: during each cycle, sensors are read, input data is provided to the control software, and output data is produced to actuators. The relation between these three notations is established by means of platform and environment mappings. RoboWorld relates to RoboChart and RoboSim models by providing a controlled natural language for documenting operational requirements. These requirements

Figure 3 – RoboStar Ecosystem.



Source: (CAVALCANTI; BAXTER; CARVALHO, 2021)

refer to RoboChart definitions (further refined by d-models) that must be satisfied by RoboSim p-models and s-models (CAVALCANTI; BAXTER; CARVALHO, 2021).

RoboChart and RoboSim d-models have a discrete process algebraic semantics based on CSP (HOARE, 2022). The semantics of other notations (e.g., p-models and RoboWorld documents) need to account for both discrete and continuous behaviour. So, a state-rich and hybrid extension of CSP, called CyPhyCircus, is considered in such situations.

Another important aspect of the RoboStar framework highlighted in Figure 3 is that the modularity obtained from RoboSim d-models, p-models and s-models, which are defined separately and connected by mappings, is transported to simulation code (CAVALCANTI; BAXTER; CARVALHO, 2021). From RoboSim models, it is possible to generate C++ simulation code. In Figure 3, SDF refers to the Simulation Description Format [1]: an XML notation that describes objects and environments for robot simulators. The simulation code will ultimately guide the development of the deployed code, which interacts with the actual robot API.

In what follows, we provide more details on the RoboChart notation that are necessary to understand RoboWorld documents. In this work, it is not necessary to cover the cyclic nature of d-models, neither the details of p-models and s-models. This knowledge would

[1] SDF website: <http://sdformat.org/>.

be necessary when assessing whether the specified operational requirements are satisfied by RoboSim models. However, this is not part of the scope of this work.

RoboChart is a diagrammatic modelling language that offers elements suitable for describing robotic applications, in addition to primitives to capture timeouts and deadlines. A key element of a RoboChart model is the block that specifies the services of a robotic platform, abstracting how these services are realised by the actual robot. In other words, this block records the assumptions related to the features and functionalities required of the physical robot (MIYAZAWA et al., 2019).

A robotic platform may have variables (information made available by the platform to the control software), events (atomic interactions between the robot and the environment), and operations (functionalities provided by the physical robot); thus, abstracting sensors, actuators and embedded software. The robotic platform interacts with controllers, which describe the behaviour of the control software in terms of state machines. A module refers to the robotic platform and associated controllers in order to define the boundaries of the robotic application being modelled.

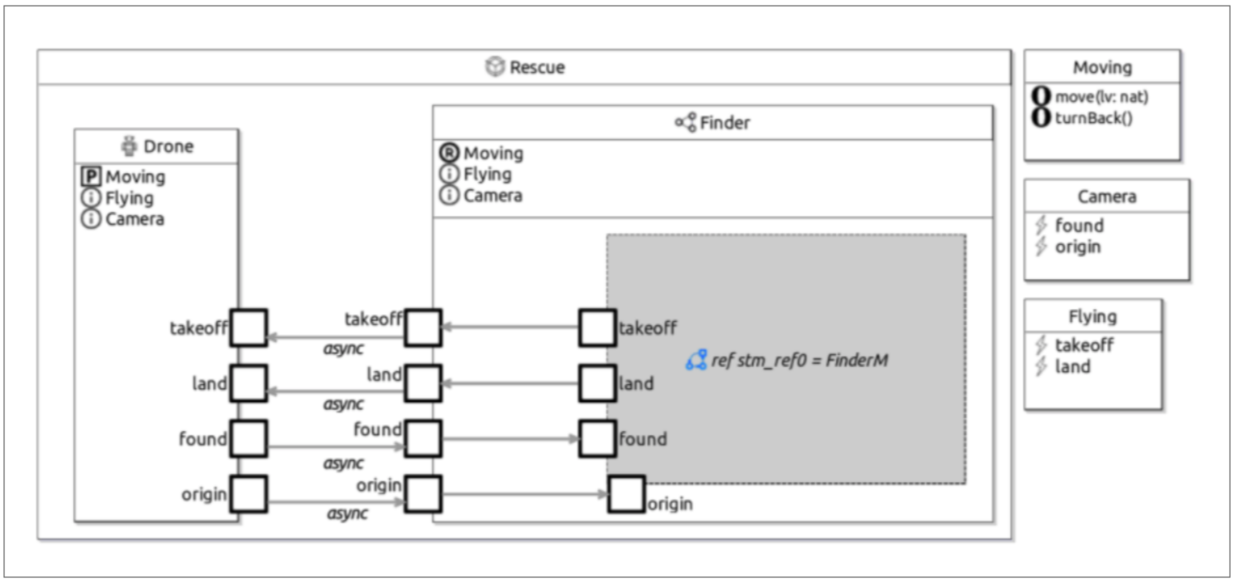
Figure 4 shows the module definition of our running example. In this module, `Drone` is the robotic platform, whereas `Finder` is a controller. The behaviour of the controller is specified in terms of the state machine `FinderM`, which is not presented in this figure. The platform provides the interface `Moving`, which is required by the controller. This interface specifies the operations `move` and `turnBack`; the former moves the robot horizontally according to the linear velocity `lv`, and the latter rotates the robot 180 degrees.

Two other interfaces are also part of this module: `Camera` and `Flying`. The `Camera` interface defines two input events (information that is sent from the platform to the controller): `found` and `origin`. The first event occurs when the robot is within the target region, whereas the second occurs when the robot is within the origin region. The `Flying` interface defines two output events (information that is sent from the controller to the platform): `takeoff` and `land`. These events signal to the platform that the robot should take off and land, respectively.

For a more comprehensive explanation of RoboChart, we refer the reader to Miyazawa et al. (2019). A complete account of RoboChart is available online in its reference manual[2].

[2] RoboChart reference manual: <https://robostar.cs.york.ac.uk/notations/>.

Figure 4 – RoboChart Module of the Rescue Drone.



Source: (CAVALCANTI; BAXTER; CARVALHO, 2021)

2.2 THE GRAMMATICAL FRAMEWORK

The Grammatical Framework (GF) is a open-source programming language for developing multilingual grammar applications. It was created in 1998 at Xerox Research Centre Europe, Grenoble, and can be used for developing applications such as translation systems, natural-language interfaces, spoken dialog systems and software localisation, for example (RANTA, 2011).

In GF, although it is possible to define a grammar using EBNF notation, grammars are normally defined using functions to cater for context-sensitive languages. Since RoboWorld is defined using functions, we focus on this presentation style. We illustrate the main features of GF using a toy version of RoboWorld (called `ToyRoboWorld`). In this toy language, we can write clauses about `robots` and `wheels`, using exclusively the verb “`to have`”. The following clauses are valid in `ToyRoboWorld`: “`the robot has a wheel`”, “`the robot has wheels`”, “`the robots have wheels`”.

In Source Code 2.1, we define the abstract grammar of `ToyRoboWorld`. The starting symbol (category) of the language is `Clause` (see Line 2). The terminals and non-terminals (called categories) are defined on Lines 4–6. The lexicon comprises determiners (in singular and plural forms), two nouns and one verb (see Lines 8–11). To finish, on Lines 13-16, we define how clauses can be created from the other categories using functions. The function `mkNounPhrase` makes a noun phrase from a determiner and a noun; `mkVerbPhrase` makes a verb phrase from

Source Code 2.1 – Abstract Grammar of ToyRoboWorld.

```
abstract ToyRoboWorld = {
  flags startcat = Clause ;
-----------------------------------------------------------------------------------------
  cat -- categories
    Determiner ; Noun ; Verb ;
    NounPhrase ; VerbPhrase ; Clause ;
-----------------------------------------------------------------------------------------
  fun -- lexicon
    a_SgDeterminer : Determiner ; a_PlDeterminer : Determiner ;
    the_SgDeterminer : Determiner ; the_PlDeterminer : Determiner ;
    robot_Noun : Noun ; wheel_Noun : Noun ; have_Verb : Verb ;
-----------------------------------------------------------------------------------------
  fun -- functions
    mkNounPhrase : Determiner -> Noun -> NounPhrase ;
    mkVerbPhrase : Verb -> NounPhrase -> VerbPhrase ;
    mkClause : NounPhrase -> VerbPhrase -> Clause ;
}
```

Source: Current Author

a verb and a noun phrase, and mkClause defines that a clause encompasses a noun phrase and a verb phrase.

The concrete grammar of ToyRoboWorld, called ToyRoboWorldEng, defines how to implement the aforementioned abstract concepts in English, covering expected spellings and grammatical rules (see Source Code 2.2). To do this, we define two parameter types (Number and VerbForm) to capture simplified notions of number and verb forms in English (Lines 3–5). These two parameter types define the concepts (constructors) Sg, Pl, and VPresent, allowing them to be used in the following definitions.

In GF, the implementations of abstract definitions are called linearisations. On Lines 7–13, we provide linearisations for the categories of ToyRoboWorld. A Determiner and a NounPhrase are implemented as records with two fields, s and n, storing the spelling (as a string, that is, a value of the GF type Str) and the number information. A Noun is a record with a single field s, defined as a table from Numbers to Strings. Similarly, Verbs are records in which the field s is a table from VerbForms to Strings. Tables are similar to functions, but their arguments must be of a parameter type (param). A VerbPhrase is also a record combining a verb (v) and a noun phrase (np).

On Lines 15–22, we define the linearisation of the lexicon of ToyRoboWorldEng. This is the place where we provide their English spelling. These definitions take into account the inflections. For instance, we provide the singular and plural forms of nouns (Lines 20 and 21) and verbs (Line 22).

Source Code 2.2 – Concrete Grammar of ToyRoboWorld.

```
 1  concrete ToyRoboWorldEng of ToyRoboWorld = {
 2  ------------------------------------------------------------------------------------------
 3    -- parameters
 4    param Number = Sg | Pl ;
 5    param VerbForm = VPresent Number ;
 6  ------------------------------------------------------------------------------------------
 7    lincat -- categories
 8      Determiner = {s : Str ; n : Number} ;
 9      Noun = {s : Number => Str} ;
10      Verb = {s : VerbForm => Str } ;
11      NounPhrase = {s : Str ; n : Number} ;
12      VerbPhrase = {v : Verb ; np : NounPhrase} ;
13      Clause = Str ;
14  ------------------------------------------------------------------------------------------
15    lin -- lexicon
16      a_SgDeterminer = {s = "a" ; n = Sg} ;
17      a_PlDeterminer = {s = "" ; n = Pl} ;
18      the_SgDeterminer = {s = "the" ; n = Sg} ;
19      the_PlDeterminer = {s = "the" ; n = Pl} ;
20      robot_Noun = {s = table {Sg => "robot" ; Pl => "robots"}} ;
21      wheel_Noun = {s = table {Sg => "wheel" ; Pl => "wheels"}} ;
22      have_Verb = {s = table {VPresent Sg => "has" ; VPresent Pl => "have"}} ;
23  ------------------------------------------------------------------------------------------
24    lin -- functions
25      mkNounPhrase det noun = {s = det.s ++ (noun.s ! det.n) ; n = det.n} ;
26      mkVerbPhrase v np = {v = v ; np = np} ;
27      mkClause np vp = np.s ++ (vp.v.s ! (VPresent np.n)) ++ vp.np.s ;
28  }
```

Source: Current Author

Lines 24–27 give the linearisations for the other functions. When creating a noun phrase (Line 25), its number information is inherited from the associated determiner (`n = det.n`). Moreover, the string representation of the noun phrase enforces agreement between the determiner and the noun. This string is created by concatenating (++) the determiner with the inflection form of the noun that shares the same number of the determiner; `noun.s ! det.n` yields a string containing the inflection form of the noun whose number information is given by `det.n`. We recall that `noun.s` is a table, a construct similar to a function; the symbol ! denotes table (function) application in GF.

In `ToyRoboWorld`, the following `NounPhrase` is not valid: "`a wheels`". In this example, the number information of the determiner "`a`" is `n = Sg` (see Line 16 in Source Code 2.2), and "`wheels`" is the inflection form associated with number `Pl` (see Line 21 in Source Code 2.2). According to the function `mkNounPhrase`, when creating noun phrases, the noun should be linearised with the inflection form that matches the number of the determiner (`noun.s ! det.n`). Therefore, in such a situation, we should read instead "`a wheel`", since `wheel` is the inflection form associated with `Sg`.

For a verb phrase, on Line 26, we just collect the verb and the noun in a record. Finally, when creating clauses, we enforce agreement between the noun phrase and the verb (Line 27). The clause is the `String` obtained from the concatenation of three strings: (1) `np.s` – the representation of the noun phrase, (2) `vp.v.s ! (VPresent np.n)` – the representation of the inflection form of the verb (`vp.v.s`) that is in the `VPresent` tense and that shares the same number of np, (3) `vp.np.s` – the representation of the noun phrase embedded in the verb phrase.

To facilitate the application of grammatical rules, such as the agreement situations presented before, as well as to deal with the particularities of different natural languages, GF offers the Resource Grammar Library (RGL); it covers a morphological and grammatical structure that is far from trivial, catering currently for more than 30 languages.

RGL defines basic categories such as adjectives (`A`), adverbs (`Adv`), determiners (`Det`), and so on. When a category has a number appended to its name (for instance, `V3`), that number denotes the amount of expected arguments (places). For example, a two-place verb (that is, a member of `V2`) expects the verb and one complement: in "the robot has an odometer", the verb "to have" is classified as a two-place verb. The verb here is "`has`" and the complement is "`an odometer`". One-place categories do not have numbers attached to their names.

The basic categories are used to create more elaborate grammatical constructions, offering support for great variety. To provide some figures, there are at least 15, 25, 20, and 30 different ways (functions) to create common nouns, noun phrases, verb phrases, and declarative clauses alone. In addition, when creating sentences, we can also consider different tenses and polarities. RoboWorld is built on RGL, inheriting its flexibility and expressiveness.

2.3 CYPHYCIRCUS

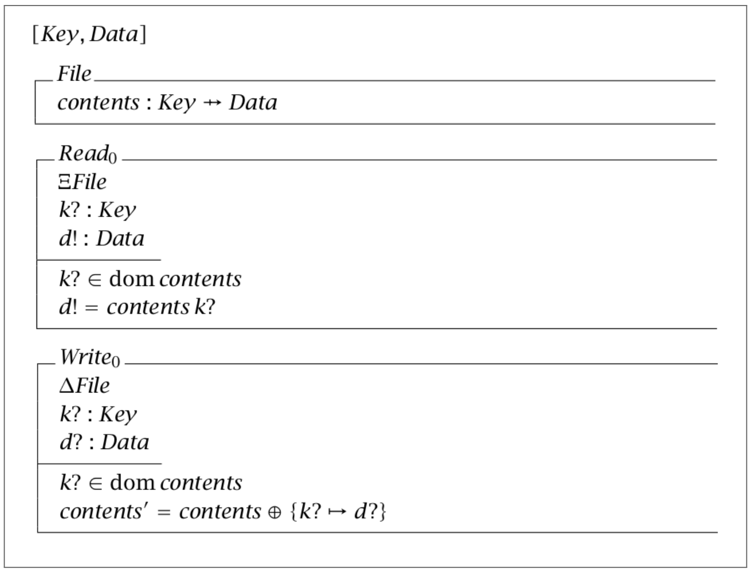
The semantics of RoboWorld documents is given in CyPhyCircus (FOSTER et al., 2020). This is a hybrid extension of Circus (WOODCOCK; CAVALCANTI, 2001) that provides facilities to specify and reason about discrete and continuous behaviour. Circus itself combines two well-established formal specification languages, Z (WOODCOCK; DAVIES, 1996) and CSP (HOARE, 2022), in order to support the specification of both data and behaviour aspects of concurrent systems. In what follows, we provide an overview of these languages.

The Z Notation

The language Z is based upon set theory and a first-order predicate calculus. The considered set theory includes standard set operators, set comprehensions, Cartesian products, and power sets (WOODCOCK; DAVIES, 1996). A characteristic feature of Z is the use of types, where every object in the language is represented by a specific and unique type.

A Z specification is structured according to paragraphs. There are several kinds of Z paragraphs, for instance: definition of basic types, axiomatic descriptions, constraints, and schemas, among others. A key element of Z is the use of schemas to describe the state of the system, and how it evolves. Figure 5 shows a Z specification for modelling a file system. Key and Data are given sets; we known that they exist (i.e., they denote valid types), but defining their internal structure is not necessary for the purpose of this modelling exercise. $File$ is a schema that specifies what a file is made of. A file has $contents$, which is modelled as a partial function ($\rightarrow\!\!\!\!\!\!+\,$) from Key to $Data$.

Figure 5 – Z Specification of a File System.



$[Key, Data]$

File
$contents : Key \rightarrow\!\!\!\!\!\!+\, Data$

Read$_0$
$\Xi File$
$k? : Key$
$d! : Data$

$k? \in \operatorname{dom} contents$
$d! = contents\, k?$

Write$_0$
$\Delta File$
$k? : Key$
$d? : Data$

$k? \in \operatorname{dom} contents$
$contents' = contents \oplus \{k? \mapsto d?\}$

Source: (WOODCOCK; DAVIES, 1996)

The schema $Read_0$ defines the read operation. Since reading a file does not change its state, $Read_0$ includes the schema $File$ such that the contents of a file are not changed during the operation; this is indicated by the symbol Ξ. This operation receives an input (a key $d?$) and yields an output (some data $d!$); the symbol $?$ denotes inputs, whereas ! denotes outputs. This operation has two constraints: (1) the key $k?$ must be valid (i.e., it should belong to the domain of $contents$, formalised as $k? \in \text{dom } contents$), and (2) the yielded output should be the data associated with the provided key (formalised as $d! = contents\ k?$).

The write operation is defined by the schema $Write_0$. Differently from $Read_0$, this operation possibly changes the state of a file. This is indicated by using the symbol Δ when including the schema $File$; $contents$ and $contents'$ denote the contents of a file before and after executing the operation, respectively. The write operation receives two inputs: a valid key $k?$ and the data $d?$ to be written to the file indexed by $k?$. The predicate $contents' = contents \oplus \{k? \mapsto d?\}$ states that $contents$ is updated ($\oplus$) such that $k?$ now maps to ($\mapsto$) the data provided ($d?$) to the write operation.

For a more comprehensive explanation of the Z notation, we refer the reader to Woodcock and Davies (1996). Z is not intended for modelling concurrent behaviour, which is the focus of the language Communicating Sequential Processes (CSP).

Communicating Sequential Processes (CSP)

CSP is a specification language for reasoning about concurrent behaviour. A central concept of CSP is the definition of processes. A process defines a behaviour by means of communication with its external environment. Communication is represented by the occurrence of events. The two most basic processes are `STOP` and `SKIP`; the former represents a deadlock situation, where no communication is possible, whereas the latter denotes successful termination by communicating the event ✓.

We consider the classical dining philosophers problem (see Source Code 2.3) to cover the basics of CSP. This code uses a machine-readable version of CSP called CSP$_\text{M}$, which combines CSP with a functional language. The dining philosophers problem is characterised by `N` philosophers around a circular table. There are also `N` forks in the table. The philosophers and forks are indexed by numbers ranging from `0` to `N-1`; the sets `PHILNAMES` and `FORKNAMES` comprise this interval.

The behaviour of the i-th philosopher is described by the process `PHIL(i)`. The communications that occur within this process are the ones defined by the `channel`s on Lines 6–7. The channels `thinks`, `sits`, `eats`, and `getsup` communicate a value in `PHILNAMES` (e.g., the event `thinks.0` denotes the 0-th philosopher thinking). The channels `picks` and `putsdown` communicate two values in `PHILNAMES` and `FORKNAMES`, respectively (e.g., the event `picks.0.1` denotes the 0-th philosopher picking the 1st fork).

Source Code 2.3 – CSP Specification of the Dining Philosophers Problem.

```
N = 5

PHILNAMES = {0..N-1}
FORKNAMES = {0..N-1}

channel thinks, sits, eats, getsup : PHILNAMES
channel picks, putsdown : PHILNAMES.FORKNAMES

PHIL(i) = thinks.i -> sits.i -> picks!i!i -> picks!i!((i+1)%N) ->
          eats.i -> putsdown!i!((i+1)%N) -> putsdown!i!i -> getsup.i -> PHIL(i)

FORK(i) = picks!i!i -> putsdown!i!i -> FORK(i)
          []  picks!((i-1)%N)!i -> putsdown!((i-1)%N)!i -> FORK(i)

PHILS = ||| i : PHILNAMES @ PHIL(i)
FORKS = ||| i : FORKNAMES @ FORK(i)

SYSTEM = PHILS [| {|picks, putsdown|} |] FORKS
```

Source: (ROSCOE, 2010)

The behaviour of a philosopher is as follows. First, the i-th philosopher thinks (thinks.i). Then, the philosopher sits (sits.i). To eat, a philosopher needs to have two forks. After eating (eats.i), the philosopher puts down the forks in reverse picking order. Finally, the philosopher gets up (getsup.i) and starts to think again (represented by the recursive call to PHIL(i)). In PHIL(i), the operator -> is used to define a sequential occurrence of events.

FORK(i) specifies the behaviour of the i-th fork. Since a fork is shared by two philosophers, the behaviour of the i-th fork is characterised by the choice ([]) of being picked by the i-th or the (i-1)%N-th philosophers. After being picked by a philosopher, the fork only returns to the table after being put down by the same philosopher. The operator [] is called external choice, where the choice is performed by an external process (e.g., the philosopher decides upon which fork is being picked). Differently, an internal choice (|~|) consists of a choice internal to the involved process.

PHILS combines all philosophers in parallel using the indexed version (i : ... @ ...) of the interleaving (|||) operator. In such a situation, there is no synchronisation between the processes describing philosophers; they operate independently. Similarly, FORKS defines the interleaving of fork processes. The SYSTEM is characterised by a synchronous parallel combination ([| ... |]) of processes. Here, the processes PHILS and FORKS must synchronise on all events associated with the channels picks and putsdown; the operator {|C|} expands C to all events that can be communicated via this channel. In other words, a philosopher can only pick a fork if, and only if, this fork is ready to be picked by this philosopher (e.g., this fork cannot be in the hands of other philosopher or far away from this philosopher).

Although CSP_M incorporates a functional language, it offers limited to support to define and manipulate complex data structures. Circus (WOODCOCK; CAVALCANTI, 2001) is a language that combines Z and CSP to support the specification of both data and behaviour aspects of concurrent systems.

Circus

Circus combines communications (events), parallelism, and choices from CSP with data types from Z. In this language, a specification is formed by a (possibly empty) sequence of `CircusParagraphs`, where each one can be a Z paragraph (`Paragraph`), a channel definition (`ChannelDefinition`), a channel set definition (`ChanSetDefinition`) or a process definition (`ProcessDefinition`). Figure 6 presents a fragment of the BNF description of the syntax of Circus. N^+ is used to denote a non-empty comma-separated list of identifiers; the same rationale applies to `Expression`$^+$. The categories `Paragraph`, `Schema-Exp`, and `Expression` are taken from the Z notation language.

Figure 6 – Circus Syntax.

Program	::=	CircusParagraph*
CircusParagraph	::=	Paragraph
	|	ChannelDefinition | ChanSetDefinition | ProcessDefinition
ChannelDefinition	::=	**channel** CDeclaration
CDeclaration	::=	SimpleCDeclaration | SimpleCDeclaration; CDeclaration
SimpleCDeclaration	::=	N^+ | N^+ : Expression | Schema-Exp
ChanSetDefinition	::=	**chanset** N == CSExpression
ProcessDefinition	::=	**process** N $\widehat{=}$ Process
Process	::=	**begin** PParagraph* • Action **end** | N
	|	Process; Process | Process □ Process | Process ⊓ Process
	|	Process |[CSExpression]| Process | Process ||| Process
	|	Process \ CSExpression
	|	Declaration ⊙ Process | Process⌊Expression$^+$⌋ | Process[N^+ := N^+]
	|	Declaration • Process | Process(Expression$^+$)
	|	[N^+]Process | Process[Expression$^+$]
PParagraph	::=	Paragraph | N $\widehat{=}$ Action
Action	::=	Schema-Exp | CSPActionExp | Command

Source: (WOODCOCK; CAVALCANTI, 2001)

The Circus support to channels is very similar to the one offered by CSP. A difference is a dedicated keyword (`chanset`) to define sets of previously defined channels. A Circus process can resemble like a pure CSP process (i.e., be defined in terms of operators such as interleaving `|||`, external choice `[]`, among others), but also consist of a sequence of process paragraphs (`PParagraph`), followed by a nameless action, delimited by `begin` and end. To give a concrete example, let us consider the process shown in Figure 7.

Figure 7 – Circus Process - Fibonacci generator.

process $Fib \mathrel{\widehat=}$ **begin**

$FibState \mathrel{\widehat=} [\, x, y : \mathbb{N} \,]$

$InitFibState \mathrel{\widehat=} [\, FibState' \mid x' = y' = 1 \,]$
$InitFib \mathrel{\widehat=} out!1 \rightarrow out!1 \rightarrow InitFibState$

$OutFibState \mathrel{\widehat=} [\, \Delta FibState;\ next! : \mathbb{N} \mid next! = y' = x + y \land x' = y \,]$
$OutFib \mathrel{\widehat=} \mu\, X \bullet \mathbf{var}\ next : \mathbb{N} \bullet OutFibState;\ out!next \rightarrow X$

$\bullet\ InitFib;\ OutFib$

end

Source: (WOODCOCK; CAVALCANTI, 2001)

The process Fib generates the Fibonacci sequence, sending its values via the channel out. We have five process paragraphs associated with Fib: $FibState$, $InitFibState$, $InitFib$, $OutFibState$, and $OutFib$. The behaviour of Fib is defined by the action $InitFib\ ;\ OutFib$, which composes sequentially (;) the actions $InitFib$ and $OutFib$. $InitFib$ produces the first two numbers of the Fibonacci sequence, and then it behaves as $InitFibState$. This action initialises the schema $FibState$, which is used to describe the internal state of the process. $FibState$ declares two variables x and y of type $\mathbb{N}$ to keep track of the last two produced numbers; initially, both variables store the value 1.

The action $OutFib$ describes a recursive $(\mu\, X)$ behaviour as follows: first, it performs `OutFibState`, which changes the state of the process by updating y to $x+y$ (i.e., $y' = x+y$) and assigning to x the old value of y (i.e., $x' = y$). Additionally, a local variable $next$ is declared and the new value of y is assigned to it (i.e., $next! = y'$). Finally, the process outputs the value of $next$ (i.e., $out!next$) and recurses. For a more detailed presentation of Circus, we refer the reader to Woodcock and Cavalcanti (2001).

CyPhyCircus

For RoboWorld, due to the continuous nature of the arena and of the associated movements, it is necessary to describe discrete and continuous behaviours (CAVALCANTI; BAXTER; CARVALHO, 2021). In this case, an extension of Circus, called CyPhyCircus (FOSTER et al., 2020), is more appropriate. The main goal of CyPhyCircus is to leverage Circus to model hybrid (a combination of discrete and continuous) systems; this is achieved by allowing the use of differential equations to specify the evolution of the system state. Concrete examples of specifications using CyPhyCircus are discussed later, when describing the semantics of RoboWorld documents.

3 ROBOWORLD

In this chapter, we present the core aspects of RoboWorld, which is a controlled natural language (part of the RoboStar framework) for documenting operational requirements of a robotic systems to be used in simulation, proof and test generation. This chapter is structured as follows. In Section 3.1, we briefly discuss the RoboWorld document of our running example (a rescue drone). In Section 3.2, we cover the syntax of RoboWorld documents, which are automatically translated into an intermediate representation (as explained in Section 3.3). A well-formed RoboWorld document should comply to with a set of well-formedness conditions, which are discussed in Section 3.4. In Section 3.5 we explain the formal CyPhyCircus semantics of RoboWorld documents.

3.1 ROBOWORLD DOCUMENT OF THE RESCUE DRONE

A RoboWorld document is structured according to seven sections. First, it describes assumptions about the arena (`ARENA ASSUMPTIONS`) and its constituent components (`ROBOT` and `ELEMENT ASSUMPTIONS`). In what follows, we present the assumptions of the rescue drone.

```
## ARENA ASSUMPTIONS ##
The arena is three-dimensional.
The arena has an origin region.
The arena has a target region.

## ROBOT ASSUMPTIONS ##
The robot is initially at the origin.
The robot is initially orientated towards the target.
The robot is a point mass.

## ELEMENT ASSUMPTIONS ##
The origin has an x-width of 1 m and a y-width of 1 m.
The gradient of the ground under the origin is the value 0.
The origin is on the ground.
The target has an x-width of 1 m and a y-width of 1 m.
The gradient of the ground under the target is the value 0.
The target is on the ground.
The distance from the target to the origin is greater than 1 m.
```

As it can be seen, the arena is three-dimensional and it has two special regions, named `origin` and `target`. These are the assumptions made about the arena. Initially, the robot is at the origin, and its orientation is towards the target region. Additionally, the shape of the robot is abstracted (i.e., it is treated as a point mass). These are the robot assumptions. Element assumptions record relevant information about the `origin` and `target` regions (i.e., their size, their location, and separation distance). There is also information about the gradient of the ground, which refers to its slope change; in this case, as it is defined to be 0, it means that the ground under the `origin` and the `target` is flat.

A RoboWorld document also provides mapping information, which describes how the control software is affected and affects the environment. This information should precisely match the (output/input) events, operations, and variables of the (possibly) associated RoboChart diagram. Mapping information of the rescue drone example is recorded as follows.

```
## MAPPING OF OUTPUT EVENTS ##
When the event takeoff occurs, the velocity of the robot is set to 1 m/s upward.
When the event land occurs, the velocity of the robot is set to 1 m/s downward.

## MAPPING OF INPUT EVENTS ##
The event found occurs when the robot is inside the target.
The event origin occurs when the robot is inside the origin.

## MAPPING OF OPERATIONS ##
When the operation move() is called, the velocity of the robot is set to 1 m/s
    towards the orientation of the robot.
The operation turnBack() is defined by a diagram where one time unit is 1 s.

## MAPPING OF VARIABLES ##
```

The output events (i.e., information sent from the control software to the robotic platform) affect the environment by changing the velocity of robot, which is part of it. The defined input events (i.e., information sent from the robotic platform to the control software) occur when the robot is inside the two aforementioned regions.

The operation move also affects the robot velocity. The mapping information for turnBack shows that, when the information becomes too complicated to be described in English, one can also rely on RoboStar diagrams to specify such information. Finally, since this example has no variables, this section is left empty.

When deriving the formal semantics of a RoboWorld document, the document as a whole is considered to generate a consistent formalisation. For instance, regarding the operation move, although it is not explicitly said in the associated sentence (see MAPPING OF OPERATIONS), a three-dimensional orientation is considered, since another sentence states that the arena is three-dimensional (see ARENA ASSUMPTIONS).

3.2 THE SYNTAX OF ROBOWORLD DOCUMENTS

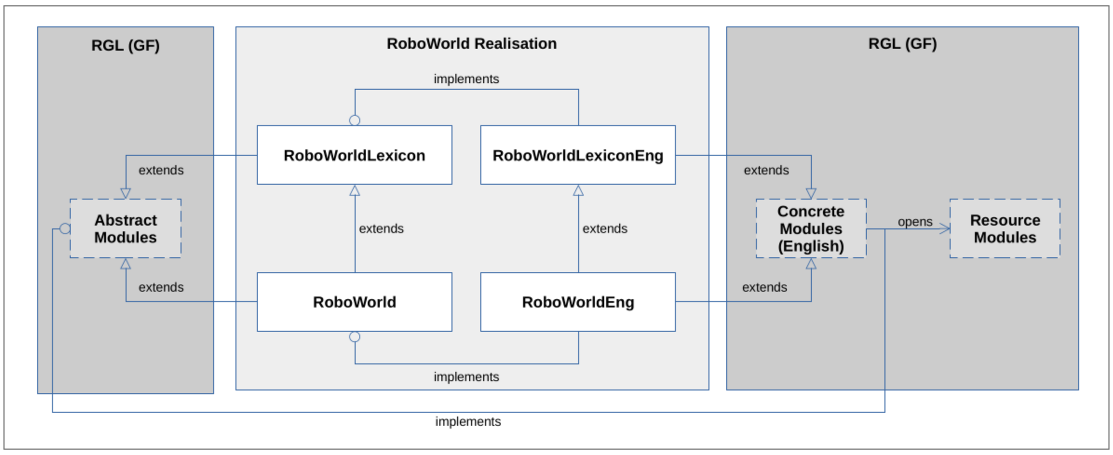
Figure 8 shows the structure of RoboWorld in GF. Our approach, which defines an abstract and a concrete syntax, along with the support provided by RGL, means that we have a general mechanisation of RoboWorld that is language independent. Using this structure, we can provide concrete implementations for RoboWorld considering other languages, such as Portuguese, French, and others. RGL takes into account more than 30 languages. Here, we restrict ourselves to English.

In Figure 8, a module is represented as a box, and a collection of RGL modules as a dashed box. The RoboWorld abstract syntax is realised by the abstract grammar `RoboWorld`. The concrete grammar `RoboWorldEng` describes how sentences in English correspond to elements of the abstract syntax.

The collections of RGL modules used in our realisation of RoboWorld are shown on the left and on the right in Figure 8. RGL is concerned with morphology and syntax rules of languages. The RGL abstract grammars that we use, shown on the left in Figure 8, cover terms such as noun phrases and clauses, for instance, which are common to many languages. On the right, Figure 8 shows RGL modules that implement the abstract modules in the English language.

In the middle box in Figure 8, we show the grammars that we have defined specifically for RoboWorld. `RoboWorldEng` implements the grammar `RoboWorld`, and they both extend a lexicon (`RoboWorldLexicon` in the case of the abstract `RoboWorld` grammar, and `RoboWorldLexiconEng` for the concrete `RoboWorldEng`). The grammars `RoboWorldLexicon`

Figure 8 – Architecture of RoboWorld syntax in GF.



Source: Current Author

and RoboWorldLexiconEng define the RoboWorld lexicon, that is, its vocabulary. All these grammars use RGL grammars to cater for general concepts.

The RoboWorld lexicon contains words that are common to the specification of robotic systems, such as arena, robot, orientation, velocity, three-dimensional, among others. Currently, the RoboWorld lexicon comprises more than 100 words. The abstract version of the lexicon (RoboWorldLexicon) defines the grammatical classes of these words (for instance, robot is a noun, one-dimensional is an adjective), but it does not give their spelling.

The concrete lexicon of RoboWorld (RoboWorldLexiconEng) implements the abstract one considering the English language, and its particularities, by extending the RGL support for English. For instance, Modern English largely does not have grammatical gender, which would require all nouns to have masculine, feminine, and neutral inflections. Therefore, when defining a noun in RoboWorldLexiconEng, it suffices to provide the spellings of the singular and plural inflections.

The RoboWorld grammar extends the RoboWorld lexicon, and defines the abstract structure of sentences (for example, sentences in the passive or active voice, or in the present or past tense, and so on) that we can write to specify assumptions and mappings. The concrete grammar RoboWorldEng implements RoboWorld observing the rules that apply to the writing of sentences in English.

Source Code 3.1 – Excerpts of the RoboWorld lexicon.

```
1  abstract RoboWorldLexicon = Cat ** {
2    ...
3    fun a_Det : Det;
4    ...
5    fun box_N : N;
6    ...
7    fun take_V2 : V2;
8    ...
9  }
10
11 concrete RoboWorldLexiconEng of RoboWorldLexicon = CatEng **
12 open MorphoEng, ResEng, ParadigmsEng, IrregEng, Prelude in {
13   ...
14   lin a_Det = mkDeterminer singular "a" | mkDeterminer singular "an";
15   ...
16   lin box_N = mkN "box" "boxes";
17   ...
18   lin take_V2 = mkV2 (mkV "take" "takes" "took" "taken" "taking");
19   ...
20 }
```

Source: Current Author

3.2.1 RoboWorld lexicon

Source Code 3.1 presents excerpts of the abstract and concrete grammars of the RoboWorld lexicon, that is, RoboWorldLexicon and RoboWorldLexiconEng. There Cat is a core abstract grammar of the RGL, declaring categories for nouns (N) and clauses (Cl), for example, among many others. CatEng is its implementation for English. On Line 1 of Source Code 3.1 we declare RoboWorldLexicon as an abstract grammar that extends Cat. On Lines 3, 5, and 7, for illustration, we show the definitions that a determiner (a_Det), a noun (box_N), and a verb (take_V2) are part of the RoboWorld lexicon.

RoboWorldLexiconEng extends CatEng and opens resource modules (e.g., MorphoEng and IrregEng) to deal with morphology rules and irregular inflections (Lines 11–12). It specifies spelling and inflection forms in English for the abstract definitions of RoboWorldLexicon. For example, a_Det is a singular determiner with two linearisation forms: "a" and "an" (Line 14). The symbol | is used to enumerate variations. Regarding box_N, RoboWorldLexiconEng defines its singular and plural forms (Line 16). Finally, for take_V2, we define the inflections for the present tense (plural and singular forms), past tense, past participle tense, and gerund (Line 18). The RGL functions mkDeterminer, mkN and mkV2 create determiners, nouns and two-place verbs.

Source Code 3.2 – Excerpts of the grammar: BasicItem and CompoundItem.

```
1  ...
2  mkBasicItem_single_noun : Cat.N -> BasicItem ;
3  ...
4  mkBasicItem_Unit : Unit -> BasicItem ;
5  ...
6  mkCompoundItem_AdverbCI : Item -> Adv -> CompoundItem ;
7  mkCompoundItem_AdverbCI_from_adjective : Item -> A -> CompoundItem ;
8  ...
```

Source: Current Author

It is possible to extend the RoboWorld lexicon to cover application-specific vocabulary. We use "dictionary" to refer to the words in the RoboWorld pre-defined and application-specific lexicons. To enrich the dictionary, we need to create new abstract and concrete grammars that extend `RoboWorld` and `RoboWorldEng`. Our tool makes this transparent: to add a word, we just need to provide it, its category, and inflections, as later explained.

3.2.2 Building blocks: ItemPhrases

Sentences in RoboWorld relate `ItemPhrases` by means of verbs. An `ItemPhrase` may refer to pronouns (`PronounIP`), determined `ItemPhrases` (`DeterminedIP`), quantified `ItemPhrases` (`QuantifiedIP`), `Items` (`BasicItem`, `CompoundItem`), and literals (e.g., `FloatLiteralIP`). In the concrete level, `BasicItems`, `CompoundItems`, and `Items` are defined as common nouns (`CatEng.CN`); `ItemPhrases` are defined as noun phrases (`CatEng.NP`). So, the functions in our grammar identify the expected forms of common nouns and noun phrases. For instance, in Source Code 3.2, we define that a `BasicItem` can be created from a noun (Line 2) or a `Unit` (Line 4), a type that we define to include the SI base units, among others.

We use RGL to make RoboWorld more flexible and expressive. For example, an `AdverbCI` is a `CompoundItem` that modifies an `Item` by an adverb. In the GF-realisation, we expect both adverbs (`Adv`) and adjectives (`A`) – see Source Code 3.2, Lines 6 and 7. In the second case, we use an RGL function to create an adverb from a given adjective (see Source Code 3.3). In the linearisation of `mkCompoundItem_AdverbCI_from_adjective`, after extracting the string embedded in the adjective (using `lin A adj`), the adverb is constructed by the RGL function `SyntaxEng.mkAdv`, turning, for example, "`initial`" into "`initially`". The function `mkCN` is also from RGL and creates a common noun given another common noun (`item`) and an adverb (`adv`). So, if we apply it to "`objects`" and "`initially`", we get the common noun "`objects`

Source Code 3.3 – Linearisation of `mkCompoundItem_AdverbCI_from_adjective`.

```
mkCompoundItem_AdverbCI_from_adjective item adj =
  let adv : CatEng.Adv = SyntaxEng.mkAdv (lin A adj)
  in mkCN item adv ;
```

Source: Current Author

`initially`" used, for instance, in "`The source contains 5 objects initially.`" – a sentence that appears in an another case study considered by this work.

The realisation of `ItemPhrases` in GF, using functions such as `mkItemPhrase_PronounIP` and `mkItemPhrase_QuantifiedIP_with_digits`, considers eight different types of quantifiers to add expressiveness. We can write, for instance, "one m", "1 m", "0.5 m", "no obstacles" and "this obstacle". For a comprehensive explanation of `ItemPhrases`, we refer to the RoboWorld reference manual[1].

3.2.3 Statements in RoboWorld

Assumptions and mapping information are defined in terms of `Statements`, which are sentences that relate `ItemPhrases` by means of verbs. In terms of RGL definitions, a statement is a sentence (`CatEng.S`). The category `Statements` comprises multiple statements linked by the conjunctions "and" or (exclusive) "or". The RoboWorld language is quite flexible in terms of writing structures with respect to `Statements`; for instance, it allows writing statements in passive and active voices, in present and past tenses, and in positive and negative polarities. Intransitive, transitive, and modal verbs are also supported. Source Code 3.4 shows some examples.

The name of the functions is quite intuitive, and describes the writing structure supported by its definition. For example, `mkStatement_PassiveVoice_IntransitiveVerb` accepts a statement in the passive voice that is written using an intransitive verb. It is worth noting that the linearisations of these functions rely on RGL built-in functions to adhere to grammatical rules. To give a concrete example, in the context of the passive voice, the function `passiveVP` yields the past participle inflection of a verb, accompanied by the verb to be, enforcing number-agreement rules. Therefore, "`the odometer of the robot is reset`" would be accepted, whereas "`the odometer of the robot are reset`" would not. Another interesting example is the function `mkS : Pol -> Cl -> S`, which creates a sentence (`S`) from a given polarity

[1] RoboWorld reference manual: <https://robostar.cs.york.ac.uk/roboworld/>

Source Code 3.4 – Excerpts of the grammar: Statement.

```
fun -- Statement
  -- the odometer of the robot is reset
  mkStatement_PassiveVoice_IntransitiveVerb :
    ItemPhrase -> V -> Statement ;
  ...
  -- the operation takeOff was called in 20 minutes before
  mkStatement_PastTense_PassiveVoice_TransitiveVerb_Preposition_ItemPhrase :
    ItemPhrase -> V -> Prep -> ItemPhrase -> Statement ;
  ...
  -- the arena is three-dimensional
  mkStatement_ActiveVoice_ToBe_Adjective :
    ItemPhrase -> A -> Statement ;
  ...
  -- the gradient of the ground is 0.0
  mkStatement_ActiveVoce_ToBe_ItemPhrase :
    ItemPhrase -> ItemPhrase -> Statement ;
  ...
  -- the robot may carry 1 object
  mkStatement_ActiveVoice_Modal_TransitiveVerb_DirectObject :
    ItemPhrase -> VV -> V2 -> ItemPhrase -> Statement ;
  ...
  -- the robot is carrying an object
  mkStatement_ActiveVoice_Positive_Progressive_TransitiveVerb_DirectObject :
    ItemPhrase -> V2 -> ItemPhrase -> Statement ;
  ...
   -- the robot is not carrying an object
  mkStatement_ActiveVoice_Negative_Progressive_TransitiveVerb_DirectObject :
    ItemPhrase -> V2 -> ItemPhrase -> Statement ;
```

Source: Current Author

(Pol) and clause (Cl). It suffices to call “mkS UncNeg cl” to change a clause cl to a negative polarity. More information about the supported writing structures is available in the RoboWorld reference manual.

3.2.4 Arena, robot, and elements assumptions

A RoboWorld document is structured into sections. The first three sections cover assumptions about the arena (ARENA ASSUMPTIONS), the robot (ROBOT ASSUMPTIONS), and other elements (ELEMENT ASSUMPTIONS) of the arena. Source Code 3.5 presents the abstract definitions of the associated categories (ArenaAssumption, RobotAssumption, and ElementAssumption, respectively), as well as the functions that create instances of these categories.

In concrete terms, these three categories are defined as sentences (i.e., CatEng.S), and an instance is built directly from the provided statement. As a consequence, the linearisation

Source Code 3.5 – Abstract definition of assumptions.

```
--------------------------------------------------------------------------------
cat -- ArenaAssumption
  ArenaAssumption ;
--------------------------------------------------------------------------------
fun -- ArenaAssumption
  -- some locations of the arena except the source and the nest contain 1 obstacle
  mkArenaAssumption_Statement : Statement -> ArenaAssumption ;
--------------------------------------------------------------------------------
cat -- RobotAssumption
  RobotAssumption ;
--------------------------------------------------------------------------------
fun -- RobotAssumption
  -- the robot is a point mass
  mkRobotAssumption_RobotStatement : Statement -> RobotAssumption ;
--------------------------------------------------------------------------------
cat -- ElementAssumption
  ElementAssumption ;
--------------------------------------------------------------------------------
fun -- ElementAssumption
  -- the source has an x-width of 0.25 m and a y-width of 0.25 m
  mkElementAssumption_Statement : Statement -> ElementAssumption ;
```

Source: Current Author

of all functions shown in Source Code 3.5 concerns an identity function. In other words, any valid statement can be used to record an assumption about the arena, the robot, and other elements of the arena.

3.2.5 Mapping information

The last four sections of a RoboWorld document provide mapping information about input and output events (`InputEventMapping` and `OutputEventMapping`, respectively), operations (`OperationMapping`), and variables (`VariableMapping`); this describes how the control software affects and is affected by the environment. Similar to the assumptions, these categories are also defined in the concrete level as sentences (`CatEng.S`). However, differently from assumptions, their construction is not defined in terms of identify functions. Source Code 3.6 shows some ways of recording mapping information. For a complete reference, we refer to the RoboWorld reference manual.

Given the name of an event (`String`) and the associated conditions (`Conditions`), the function `mkInputEventMapping_InputSometimes` yields an `InputEventMapping`. An instance of `Conditions` is created from a `Subjunction` (e.g., "*when*") and `Statements`. In the sentence "*The event obstacle occurs when the distance from the robot to an obstacle is less than 1*

Source Code 3.6 – Abstract definition of mapping information.

```
--------------------------------------------------------------------------------
fun
  -- the event obstacle occurs
  -- when the distance from the robot to an obstacle is less than 1 m
  mkInputEventMapping_InputSometimes : String -> Conditions -> InputEventMapping ;
  ...
  -- the output event spray is defined by a diagram where one time unit is 1 s
  mkOutputEventMapping_DiagramaticOutput : String -> Float -> Unit -> OutputEventMapping ;
  ...
  -- when the operation Store() is called
  -- as soon as the distance from the robot to the source is less than 1 m
  -- the robot places an object in the nest
  mkOperationMapping_OutputSometimes_WithConditions :
    String -> Conditions -> Statements -> OperationMapping ;
  ...
```

Source: Current Author

m.", the event name is "*obstacle*", and the associated condition is "*when the distance from the robot to an obstacle is less than 1 m*".¨

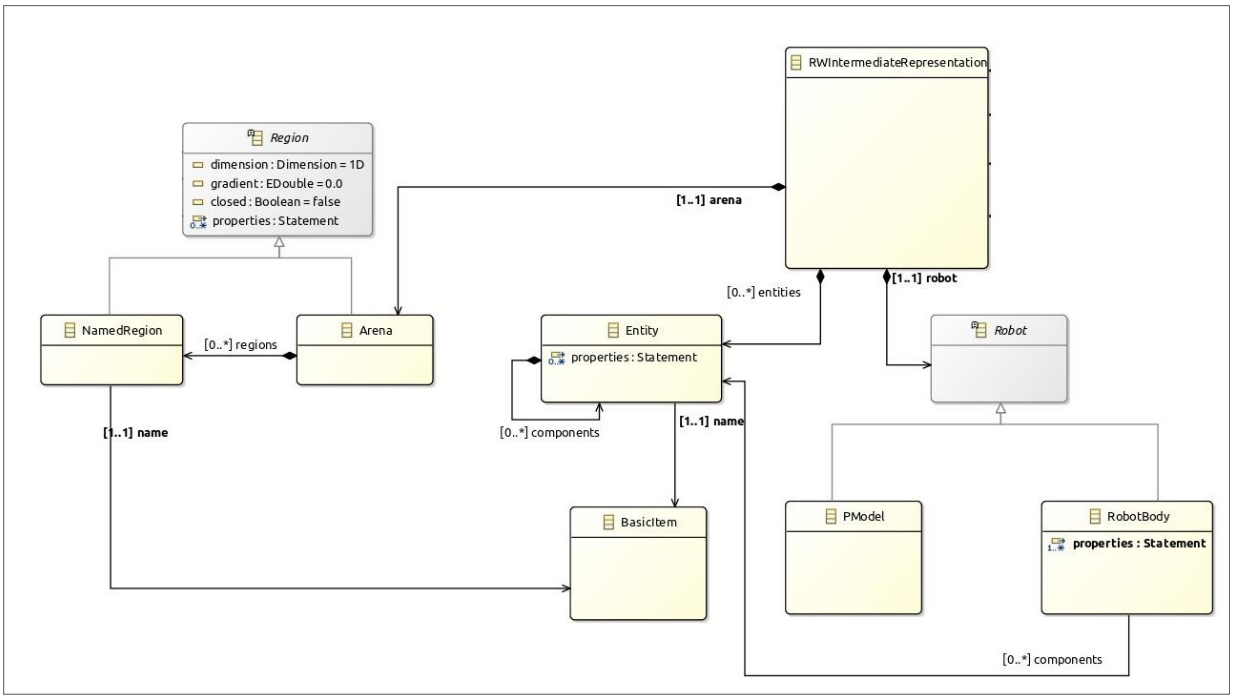
The function mkOutputEventMapping_DiagramaticOutput shows that RoboStar diagrams can be used to record mapping information, when it becomes too complicated to be described in English (i.e., when it is not straightforward to describe the mapping information with one sentence). The last function shown in Source Code 3.6 illustrates that mapping information can combine conditions and statements. In "*When the operation Store() is called, as soon as the distance from the robot to the source is less than 1 m, the robot places an object in the nest.*", the sentence documents the effect of invoking the operation "*Store*", which is "*the robot places an object in the nest*". However, this effect only occurs when the following condition holds: "*as soon as the distance from the robot to the source is less than 1 m*".

3.3 INTERMEDIATE REPRESENTATION

The semantics of a RoboWorld document is defined in terms of an intermediate representation (IR) of that document. With this representation, we insulate the semantics specification from some evolutions of RoboWorld. For example, further case studies are likely to suggest different phrasings for the same meanings, which we may be able to support by extension of the RoboWorld dictionary or of its concrete grammar. With the IR, such extensions, which are important to make the language more flexible, do not affect the semantics definition. Furthermore, in the IR, information about the arena, the robot, and the other elements is grouped, and structured, which simplifies the automatic generation of the semantics.

In this section, we provide an overview of RoboWorld's IR, whose metamodel is partially depicted in Figure 9. Given a valid RoboWorld document, its intermediate representation is characterised by an instance of the class `RWIntermediateRepresentation`. This object has one `Arena`, one `Robot`, and possibly multiple `Entities`. `Arena` is a subclass of `Region` and, thus, the object `arena` has a dimension, its floor has a gradient, it can be closed or open, and other properties must be established by means of `Statements` (see Section 3.2.3). Additionally, the `arena` may have named regions, such as the `target` and `origin` regions from our running example. Note that, although information about the arena may be defined in a RoboWorld document across multiple sentences (see Section 3.1), in terms of the metamodel, this information is grouped within the object `arena`, facilitating the generation of the semantics.

Figure 9 – Fragment of RoboWorld Intermediate Representation.



Source: (BAXTER et al., 2023)

The object `robot` groups all properties about the robot, which can be given in terms of `Statements` or by a p-model (`PModel` – see Section 2.1). Similarly, the properties of other elements of the arena are grouped in instances of `Entity`. Although not presented in Figure 9, the IR also comprises specific classes to capture mapping information in a structured way. In this document, more information about the IR is presented on demand. The complete metamodel of RoboWorld's IR is described in its reference manual.

3.4 WELL-FORMEDNESS CONDITIONS

Besides the expected restrictions of the English grammar (enforced by GF), and the structure imposed by the RoboWorld grammar, there are some general well-formedness conditions (WFCs) that need to be enforced. For example, the use of measurement units must be consistent with the relevant physical quantity. For instance, length (distance, x-width, y-width, z-width, width, depth, or height) must be measured in meters or its prefixes. Time must be measured in units derived from seconds, and so on. Another example concerns consistency with the informed arena dimension. For instance, if the arena is two-dimensional, no sentence should refer to the z-width. These general restrictions are a form of well-typedness rules, and can be naturally enforced using the intermediate representation described in the previous section.

Therefore, before providing a formal semantics to a RoboWorld document, it is necessary to check whether some well-formedness conditions are met. In this section, we concentrate on explaining some well-formedness conditions, as presented in what follows. Other WFCs are documented in the RoboWorld reference manual.

- **RW1**: The dimension of the arena must not be null.
- **RW2**: The gradient of the arena must not be a negative number.
- **RW3**: The x/y/z-width of the arena must be consistent with the arena dimension.
- **RW4**: The arena properties must refer to existing definitions.
- **RW5**: The dimension of all named regions must not be null.
- **RW6**: The gradient of all named regions must not be a negative number.
- **RW7**: The x/y/z-width of all named regions must be consistent with its dimension.
- **RW8**: The properties of all named regions must refer to existing definitions.
- **RW9**: The definite article "*the*" must always be used when referring to the robot.
- **RW10**: The output events must refer to existing robot components or entities.

RW1–RW4 are related to the arena. Its dimension must not be null (RW1); we assume that the arena is one-dimensional, if no information about its dimensionality is provided in the RoboWorld document (see Figure 9). The gradient of the floor must not be a negative number

(RW2). Additionally, whenever referring to the x/y/z-width of the arena, it is necessary to be consistent with the declared arena dimension (RW3), as explained in the beginning of this section. Finally, all arena properties must refer to existing entities (RW4); for example, we cannot say that "*the height of the arena is greater than the height of the building*" if there is no building in the arena. RW5–RW8 are related to named regions; they specify well-formedness conditions analogous to the ones just explained.

RW9 states that the definite article "*the*" must always be used when referring to the robot, since, at this moment, the RoboWorld semantics considers that there is a single robot in the arena. RW10 are analogous to RW4 and RW8, but within the context of mapping information of output events. We emphasise that the well-formedness conditions presented above is a subset of the existing ones.

3.5 ROBOWORLD SEMANTICS

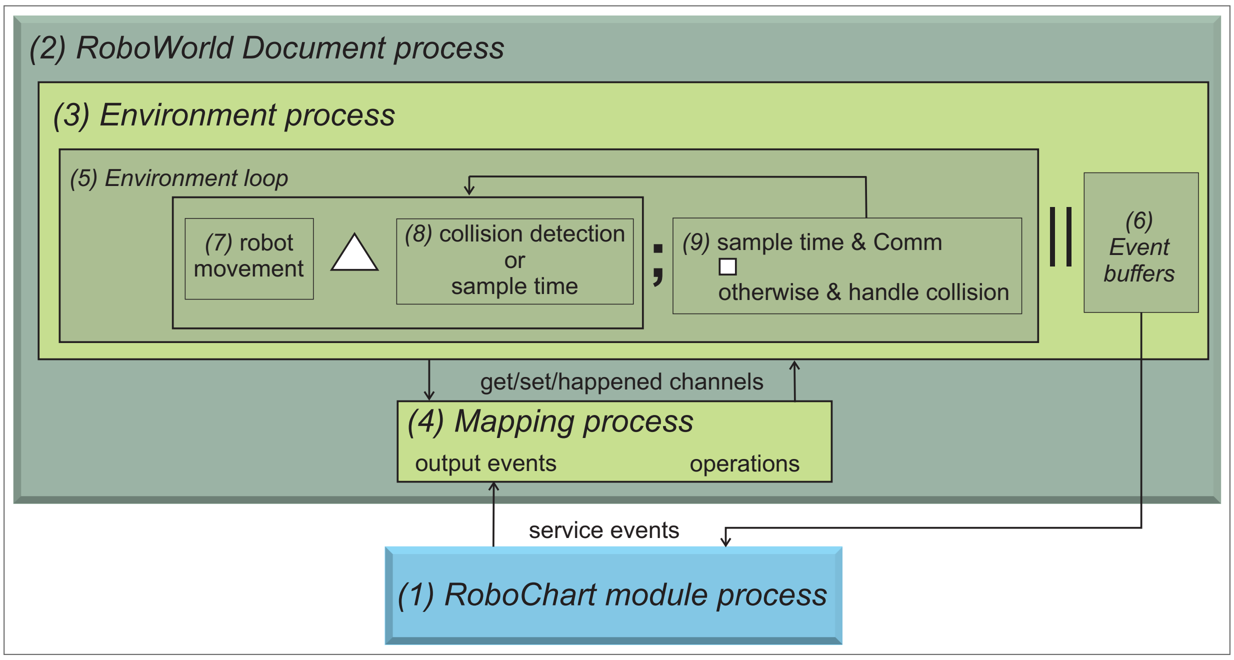
Here, we first present an overview of the formal semantics of RoboWorld documents (Section 3.5.1). Afterwards, considering the running example (a rescue drone), we discuss the CyPhyCircus characterisation of the semantics (Section 3.5.2).

3.5.1 Semantics overview

The RoboWorld semantics is a hybrid (discrete and continuous) model based on the continuous nature of the arena and of the robot's movement. The formal specification of the semantics uses CyPhyCircus, where processes share information with others by communicating events. The overall structure of the RoboWorld semantics and how it connects with the semantics of RoboChart is indicated in Figure 10. The semantics of RoboWorld documents may be possibly connected with that of RoboSim models too. However, this perspective is yet to be analysed in details. Therefore, Figure 10 refers only to the connection with RoboChart models.

The semantics of a RoboWorld document (box 2 in Figure 10) is defined by two parallel processes: the *environment process* (box 3 in Figure 10), which represents the objects inside the arena and handles the triggering of events, and a *mapping process* (box 4 in Figure 10), which contains the semantics for the mapping of output events and operations. To analyse the resulting semantics of the control software (box 1 in Figure 10) in conjunction with the

Figure 10 – RoboWorld Document Semantics.



Source: (BAXTER et al., 2023)

associated assumptions and mapping information, these two processes (represented by boxes 1 and 2) should be composed in parallel.

The environment process is defined by the parallelism of two actions (boxes 5 and 6), indicated by parallel bars in Figure 10. The first one (*environment loop*) is composed by a continuous loop related to evolving the state, communicating with the mapping process via get and set channels, and buffering information about inputs. The evolution of the robot movement can be interrupted by collision detection or by the time reaching a sample time. In Figure 10, the symbols $\triangle$, $\fatsemi$, and $\Box$ denote process interruption, sequential composition, and external choice, respectively.

When an interruption occurs due to collision, the robot is stopped (its velocity and acceleration are set to zero), and the loop action starts again. Otherwise, in case of interruption by reaching the sample time, the loop action verifies the conditions for the input events, shares this information with the parallel action (*event buffers*), and then communicates with the mapping process to get and set the values of the associated variables.

The action *event buffers* defines a set of buffers for input and output events. The input event buffers store information about detected events in the time step, and provides this to the RoboChart process. The output buffers store the time when an output/operation last happened, obtaining this information from the mapping process by the *happened* channel. The *mapping process* (box 4 in Figure 10) is defined as the interleaving of processes that accept output events and operations. These processes capture the mapping definitions provided in the RoboWorld document and share information with the environment process. In what follows, we present a more detailed account of the semantics of RoboWorld documents in CyPhyCircus.

3.5.2 Semantics of the rescue drone

Based on the IR derived from the RoboWorld document, Z schemas and global definitions are used to characterise the arena and its (static) constituent elements. For instance, regarding the rescue drone, in Figure 11, we see the schema *ArenaProperty*, which formalises properties of the arena. It has an *xwidth*, an *ywidth*, and a *zwdith*, which are real values. The locations of the arena comprise the continuous coordinates ranging from (0,0,0) to (*xwidth*,*ywdith*,*zwidth*). Properties of the ground are defined by another schema (*GroundProperty*); its *xwidth* and *ywidth* are equal to those of the arena. Properties of the origin and target regions are defined by the schemas *OriginProperty* and *TargetProperty*, respectively.

Figure 11 – ArenaProperty for the Rescue Drone.

$$
\begin{array}{|l}
\hline
\textit{ArenaProperty} \\
\quad xwidth, ywidth, zwidth : \mathbb{R} \\
\quad windSpeed : \mathbb{R} \\
\quad locations : \mathbb{P}\, Position \\
\quad ground : GroundProperty \\
\quad origin : OriginProperty \\
\quad target : TargetProperty \\
\hline
\quad locations = \{x : 0 \mathrel{..} xwidth;\ y : 0 \mathrel{..} ywidth;\ z : 0 \mathrel{..} zwidth\} \\
\quad ground.xwidth = xwidth \land ground.ywidth = ywidth \\
\hline
\end{array}
$$

Source: Current Author

A global definition $arena : ArenaProperty$ declares an instance of the schema *ArenaProperty*, which is further constrained by the assumptions made about the environment. For instance, in Figure 12, we can see that the size of the target region is constrained according to the sentence "*The target has an x-width of 1 m and a y-width of 1 m.*" (see Section 3.1). Additionally, we have two predicates to enforce that all locations of this region are on the ground and within the arena.

The semantics of the dynamic aspects of the RoboWorld document is given by the process *RoboWorldDocument*. Regarding the rescue drone, Figure 13 presents the associated CyPhyCircus semantics. The sets *getSetChannels* and *eventHappenedChannels* indicate the events

Figure 12 – Properties Constraining the Target Region.

$$
\begin{array}{|l}
\hline
\mathit{arena.target.xwidth} = 1.0 \\
\mathit{arena.target.ywidth} = 1.0 \\
\mathit{locsOnLocs}\ \mathit{arena.target.locations}\ \mathit{arena.ground.locations} = \mathbf{True} \\
\mathit{arena.target.locations} \subseteq \mathit{arena.locations}
\end{array}
$$

Source: Current Author

that will be used to establish the communication between the environment and mapping processes. The *RoboWorldDocument* is defined by the parallel composition ([...]) of processes *Environment* and *Mapping*. The union of the sets *getSetChannels*, *eventHappenedChannels* and *proceed* indicates that a synchronisation is required between *Environment* and *Mapping* on these channels; *proceed* is a channel used by *Mapping* to indicate to the *Environment* process that it can continue with the loop. After the parallel composition, these events are hidden by the operator $\setminus$ to prevent undesired synchronisation from external processes.

Figure 13 – The RoboWorld Document Semantics for Rescue Drone Example.

$$
\begin{array}{l}
\mathbf{channelset}\ \mathit{getSetChannels} == \{\!| \\
\quad \mathit{getRobotPosition}, \mathit{getRobotVelocity}, \mathit{getRobotAcceleration}, \\
\quad \mathit{getRobotOrientation}, \mathit{getRobotAngularVelocity}, \mathit{getRobotAngularAcceleration}, \\
\quad \mathit{setRobotPosition}, \mathit{setRobotVelocity}, \mathit{setRobotAcceleration}, \\
\quad \mathit{setRobotOrientation}, \mathit{setRobotAngularVelocity}, \mathit{setRobotAngularAcceleration}\ |\!\} \\
\mathbf{channelset}\ \mathit{eventHappenedChannels} == \\
\quad \{\!|\ \mathit{takeoffHappened}, \mathit{landHappened}, \mathit{moveHappened}, \mathit{turnBackHappened}\ |\!\} \\
\\
\mathbf{process}\ \mathit{RoboWorldDocument} \mathrel{\widehat=} \\
\quad (\mathit{Environment}\ [\![\ \mathit{getSetChannels} \cup \mathit{eventHappenedChannels} \cup \{\!|\ \mathit{proceed}\ |\!\}\ ]\!]\ \mathit{Mapping}) \\
\qquad \setminus\ \mathit{getSetChannels} \cup \mathit{eventHappenedChannels} \cup \{\!|\ \mathit{proceed}\ |\!\}
\end{array}
$$

Source: Current Author

The state of the *Environment* process is defined by the schema *EnvironmentState* (see Figure 14). The robot is part of the environment. Its initial position should be within the arena, and it is marked as **visible**, so that the behaviour of the environment is characterised by the evolution of the value of this component over time. *Environment* also encapsulate some time-related components: *time* represents a global clock, *stepTimer* is used to measure the duration of the present execution cycle (*timeStep*, not shown here, is a global constant that defines the size of the execution cycle), and *EventTimes* records the occurrence of events and operations.

The robot movement is defined using a special kind of schema (*RobotMovement* – see

Figure 14 – Semantics to EnvironmentState and EnvironmentStateInit.

process *Environment* $\widehat{=}$ **begin**

state *EnvironmentState*

EnvironmentState

$$\begin{array}{|l}
\textbf{visible } robot : RobotProperty \\
time : \mathbb{R} \\
stepTimer : \mathbb{R} \\
EventTimes \\
\hline
robotInit.position \in arena.locations
\end{array}$$

Source: Current Author

Figure 15), that is specifically available in CyPhyCircus (but not in Z or Circus). Such schemas are indicated by a Λ declaration of the state to specify evolution according to a set of given differential equations. The body of *RobotMovement* has, for instance, differential equations describing the movement of the robot and the evolution of timers. As shown in Figure 15, the robot's position evolves with a derivative equal to its velocity. The time-related components of the state (time and stepTimer) evolve with a derivative of 1, so that they keep track of the time in the environment. Every component in *EnvironmentState* not mentioned in the equations of *RobotMovement* remains the same throughout the evolution.

Figure 15 – Semantics of Robot Movement.

RobotMovement

$$\begin{array}{|l}
\Lambda EnvironmentState \\
\hline
\frac{\mathrm{d}robot.position}{\mathrm{d}t} = robot.velocity \\
\frac{\mathrm{d}robot.velocity}{\mathrm{d}t} = robot.acceleration \\
\frac{\mathrm{d}robot.acceleration}{\mathrm{d}t} = (0, 0) \\
\frac{\mathrm{d}robot.orientation}{\mathrm{d}t} = robot.angularVelocity \\
\frac{\mathrm{d}robot.angularVelocity}{\mathrm{d}t} = robot.angularAcceleration \\
\frac{\mathrm{d}robot.angularAcceleration}{\mathrm{d}t} = 0 \\
\frac{\mathrm{d}time}{\mathrm{d}t} = 1 \\
\frac{\mathrm{d}stepTimer}{\mathrm{d}t} = 1
\end{array}$$

Source: Current Author

The core action of *Environment* is defined as the parallel composition of the action *EnvironmentLoop* and the action *EventBuffers*, as presented in Figure 10. The CyPhyCircus definition of these two actions mimics the behaviour described in the previous section. Here, considering the input event *found* of the rescue drone, we illustrate how the mapping information on input events (i.e., information sent to the control software) is formalised.

The RoboWorld document states that "*The event found occurs when the robot is inside the target.*". Therefore, as it can be seen in Figure 16, this information is translated to a predicate that checks whether the robot position is within the locations of the target region. If this is the case, this information is sent to the respective input buffer via the communication *foundTriggered*!**True**, which performs the event *found.in* that is captured by the control software (this part is not shown here).

Figure 16 – Semantics of Input Event *found*.

```
$found\_InputEventMapping \mathrel{\widehat=}$
    $\mathbf{if}(robot.position \in arena.target.locations) \longrightarrow$
        $foundTriggered!\mathbf{True}$
         $\longrightarrow foundOccurred, foundTime := \mathbf{True}, time$
    $[\!] \neg\ (robot.position \in arena.target.locations) \longrightarrow$
        $foundTriggered!\mathbf{False} \longrightarrow \mathbf{Skip}$
    $\mathbf{fi}$
```

Source: Current Author

The mapping information on output events and operations is formalised by the definition of the *Mapping* process. For instance, the following sentence "*When the event takeoff occurs, the velocity of the robot is set to 1 m/s upward.*" is captured by the definition shown in Figure 17. After synchronisation on the event *takeoff.out*, performed by the control software, the velocity of the robot (a component of the state of the environment, as previously explained) is updated accordingly.

Figure 17 – Semantics of Output Event *takeoff*.

```
$takeoff\_Semantics \mathrel{\widehat=}$
    $(takeoff.out \longrightarrow$
        $((setRobotVelocity!(0.0, 0.0, 1.0) \longrightarrow Skip));$
        $takeoff\_Semantics)$
    $\Box$
    $proceed \longrightarrow takeoff\_Semantics$
```

Source: Current Author

We refer the reader to (BAXTER et al., 2023) for a full account of how CyPhyCircus is used to formalise the semantics of RoboWorld documents. Here, other aspects of the semantics are explained on demand in the following chapters.

4 ROBOWORLD PLUG-IN

The main goal of the RoboTool plug-in for RoboWorld is to facilitate the edition and use of RoboWorld documents. This plug-in provides means for maintaining a project-specific dictionary, authoring RoboWorld sentences, checking well-formedness conditions, and generating a formal semantics. It is implement in Java, as an Eclipse plug-in.

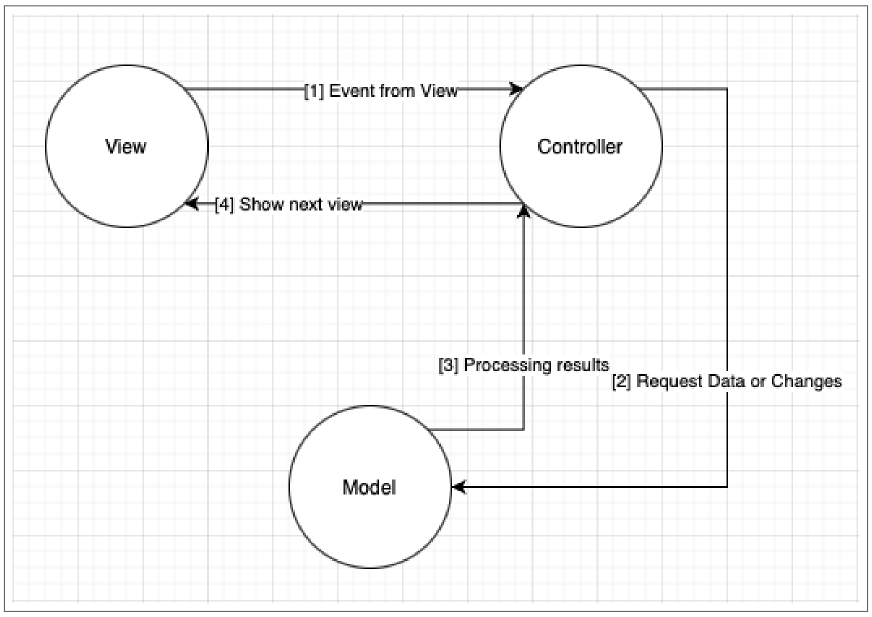
First (Section 4.1), we present the plug-in architecture. Afterwards, we show the support provided to edit RoboWorld documents (Section 4.2). Then, considering the derived intermediate representation (Section 4.3), we cover the verification of well-formedness conditions (Section 4.4), and the generation of the CyPhyCircus semantics (Section 4.5).

4.1 TOOL ARCHITECTURE

The architecture of the RoboWorld plug-in is based on the Model-View-Controller architecture (MVC), whose main purpose is to provide a bridge between how information is represented internally and how information is presented to the user. Figure 18 presents a general overview of the MVC architecture, containing its three layers: the Model, the View and the Controller. To illustrate the role of each layer, consider a scenario where the user is trying to access some data stored by the application. First, following an interaction between the user and the View, an event is triggered from the View to the Controller containing the user request. In the next step, the Controller sends the request to the Model layer, which will process it and send the associated data back to the Controller. To finish this interaction scenario, the Controller sends the received data to the View, which will present the requested information in an appropriate way. Concerning the RoboWorld plug-in, its implementation, adhering to the MVC architecture, has about twelve thousand (12,000) lines of code.

The Model layer is responsible for managing the data associated with the application in a way that is appropriate for computational means. Figure 19 shows the Model layer of the RoboWorld plug-in (tool), which comprises sixty four (64) classes. The entities are grouped into four blocks: Dictionary Model, RoboWorld Document Model, IR Model, and CyPhyCircus Model. The upper-left one contains the entities associated with the model of the dictionary (i.e., data related to adjectives, adverbs, verbs, etc.). The upper-right one contains the entities associated with the model of a RoboWorld document (i.e., data related to assumptions

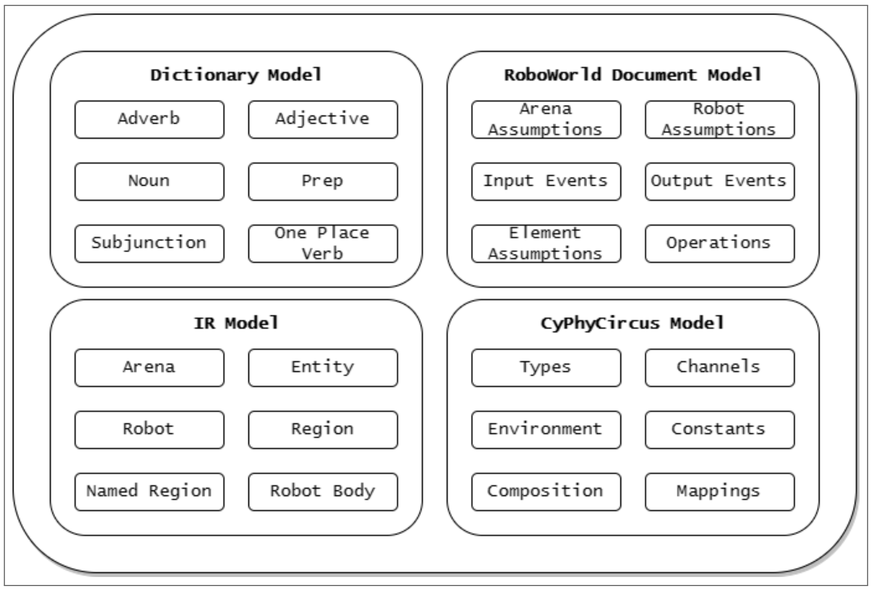
Figure 18 – The MVC Architecture.



Source: Current Author

about the arena, the robot, and other elements. Besides mapping information of output/input events, operations and variables). The lower-left one contains the entities associated with the intermediate representation of RoboWorld documents, whereas the lower-right block comprises entities associated with the CyPhyCircus semantics of RoboWorld documents.

Figure 19 – RoboWorld Tool: Model Layer.

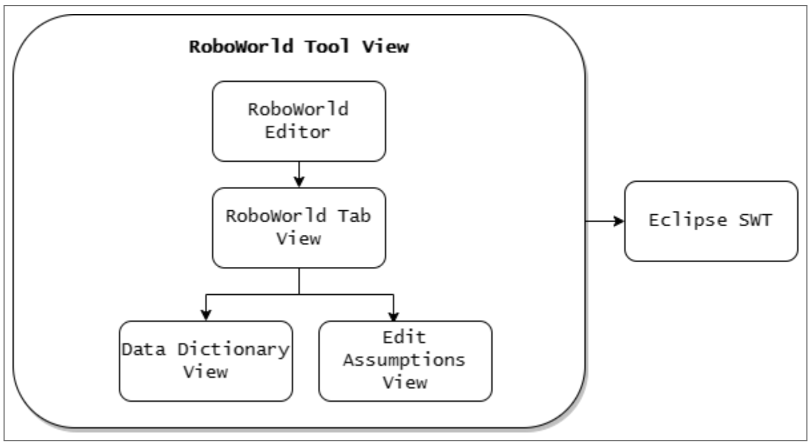


Source: Current Author

The View layer is responsible for showing the graphical user interface (i.e., forms, buttons, and other graphical elements), and controlling the way data is displayed to the user. Figure 20 shows the View layer of the RoboWorld tool, which comprises four (4) classes. The RoboWorld

Editor controls the edition of RoboWorld documents, and it contains a RoboWorld Tab View. The latter comprises two specific tabs: one for managing the project-specific dictionary (Data Dictionary View), and other for editing assumptions and mapping information (Edit Assumptions View). Each component uses resources and graphical components provided by the Eclipse Standard Widget Toolkit (Eclipse SWT) to display data and manage user interaction.

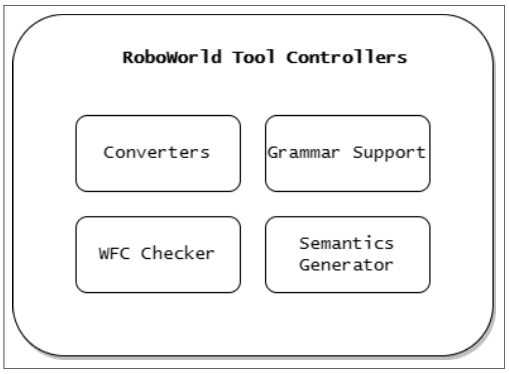
Figure 20 – RoboWorld Tool: View Layer.



Source: Current Author

The Controller layer is responsible for handling events sent by the View layer (originated from user interaction with graphical components). It manages the relationship between the View and the Model layers. Figure 21 shows an abstract representation of the entities within the Controller layer of the RoboWorld tool; there are actually five (5) classes within this layer.

Figure 21 – RoboWorld Tool: Controller Layer.



Source: Current Author

The Converters transform entities of the Dictionary model into GF definitions, while the Grammar Support is responsible for invoking GF, which compiles the underlying grammar and parses RoboWorld sentences. The other two entities (WFC Checker and Semantics Generator) are responsible for checking well-formedness conditions, and generating the CyPhyCircus semantics, respectively.

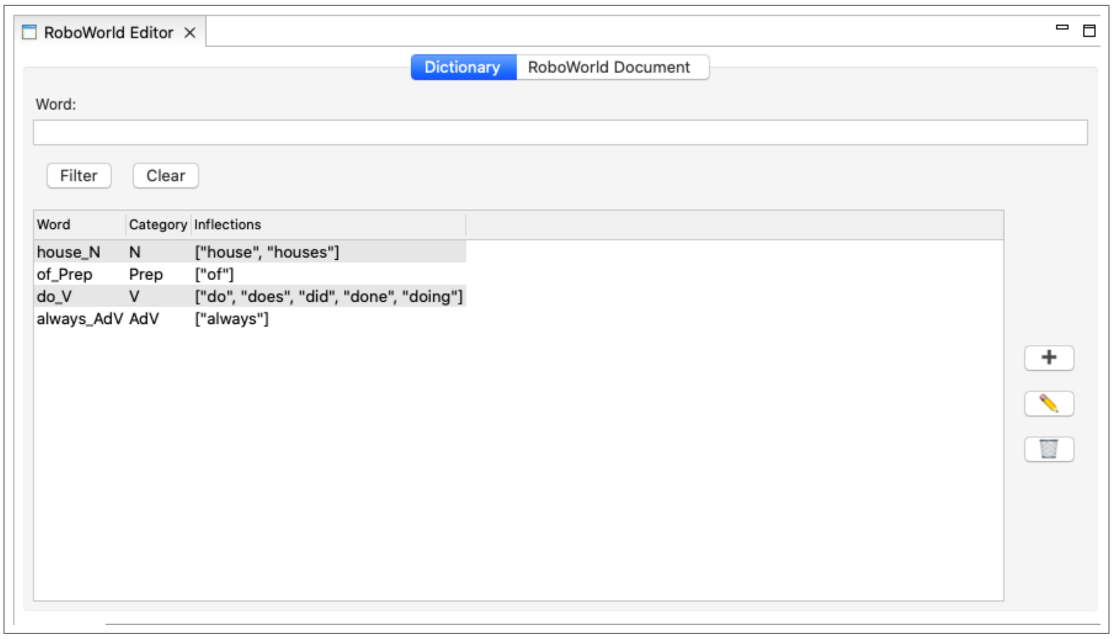
4.2 EDITING ROBOWORLD DOCUMENTS

The RoboWorld plug-in is activated when the user opens a *.env* file. As it can be seen in Figure 22, it is organized in two tabs: Dictionary and RoboWorld Document. In the first one, the user can edit the project-specific dictionary. In the second one, the user can write sentences recording assumptions about the environment and mapping information. The next sections (Section 4.2.1 and Section 4.2.2) provide more information about these two tabs.

4.2.1 Dictionary view

Figure 22 shows the Dictionary tab. It is possible to create, edit or delete word definitions. By doing this, the RoboWorld base lexicon is extended considering project-specific words. In this tab, for each word, we have three columns: Word (the word spelling), Category (the associated GF category), and Inflections (the variations of this word according to its category).

Figure 22 – RoboWorld Tool: Dictionary Tab.



Word	Category	Inflections
house_N	N	["house", "houses"]
of_Prep	Prep	["of"]
do_V	V	["do", "does", "did", "done", "doing"]
always_AdV	AdV	["always"]

Source: Current Author

In Figure 22, we have four entries in the project-specific dictionary: *house*, *of*, *do*, and *always*, whose categories are noun (N), preposition (Prep), verb (V), and adverb directly attached to a verb (AdV). In GF, a verb-phrase modifying adverb is classified differenty, as an instance of Adv (instead of AdV – note the capital "V"). For each word, the appropriate inflections are informed; for instance, regarding the noun *house*, we have its singular and plural forms.

When adding a new word, the user needs to first inform the base form of the new word and its category. After doing this, the RoboWorld plug-in searches automatically whether this word is already defined in an English dictionary provided by the RGL, which contains more than 60 thousand definitions. If this is the case, the user does not need to provide the inflection forms of this word, since this information is retrieved from the RGL dictionary. The project-specific dictionary is saved in a *.gf* file, which is stored within the open project.

As the dictionary evolves, the tool recompiles all files related to the RoboWorld grammar. This is necessary to be able to recognise sentences referring to the newly added words. When compiled, a GF grammar is translated into a portable format (PGF). In the RoboWorld plug-in, we use a PGF interpreter in Java, which is provided by the Grammatical Framework.

As mentioned before, we emphasise that the user only needs to define words that are specific to the current project, since general terms of robotics (e.g., *arena*, *robot*, *velocity*, among others) are defined as part of the RoboWorld lexicon, which itself can also evolve as required. As a consequence, the dictionary of a project tends to be quite concise.

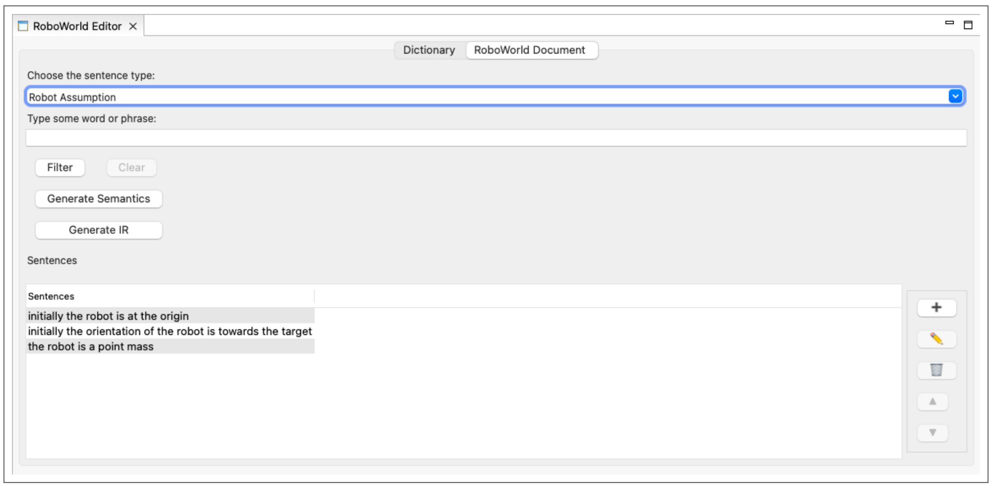
4.2.2 RoboWorld document view

Figure 23 shows the RoboWorld Document tab. It is possible to manage sentences, which shall adhere to the underlying RoboWorld grammar, describing environment assumptions (i.e., arena assumptions, robot assumptions, and element assumptions) and mapping information (input events mapping, output events mapping, operations mapping, and variables mapping).

Additionally, in this tab, by clicking on *Generate IR*, the user can derive the intermediate representation, whose textual representation is saved in the file *ir.txt* within the open project. After generating the IR, well-formedness conditions are checked. By clicking on *Generate Semantics*, the tool generates the CyPhyCircus semantics, which is saved in the file *semantics.tex* within the open project. This file describes the semantics using the LaTeX style developed for creating CyPhyCircus specifications. We provide later more information on the IR and semantics generation.

In the RoboWorld Document tab, when the user selects the desired sentence type, the GUI

Figure 23 – RoboWorld Tool: RoboWorld Document Tab.



Choose the sentence type: Robot Assumption

Type some word or phrase:

Sentences

initially the robot is at the origin
initially the orientation of the robot is towards the target
the robot is a point mass

Source: Currrent Author

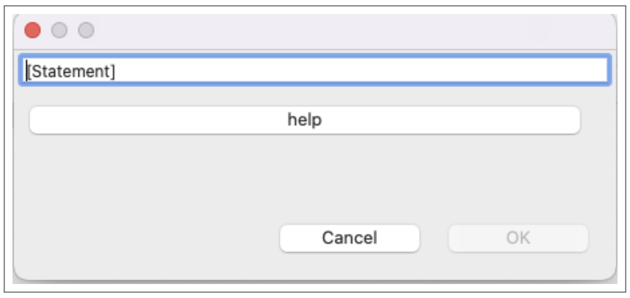
filters the RoboWorld document and shows only the sentences of the selected type. It is also possible to filter the RoboWorld document by words and phrase fragments. The button *Clear* clears the applied filter. Filtering is also available in the Dictionary tab (see Figure 22). By clicking on the up and down buttons (right-side of Figure 23), the user can reorder how the sentences are displayed. The sentences of a RoboWorld document are stored in a *.env* file.

Whenever a new sentence is added, or an old one is edited, the tool parses it, and any errors are indicated to the user. Parsing is performed by GF, and it is also supported by the provided PGF interpreter in Java. To facilitate the process of writing sentences adhering to the RoboWorld grammar, the RoboWorld plug-in provides the user with a combination of structural and surface editing. In structural editing, the user follows a structural approach by clicking on predefined writing possibilities, which prevents the writing of invalid sentences. In surface editing, the user inputs texts freely.

To illustrate the provided combination of structural and surface editing, consider the scenario described in what follows. Suppose that a user selects the sentence type Robot Assumption, and then clicks on the add button (see Figure 23). Figure 24 shows the dialog that is displayed. The text within brackets (i.e., "[Statement]") indicates that a robot assumption is created from a Statement (see Source Code 3.5 in Section 3.2.4). If the user is familiar with the RoboWorld grammar, she can delete the string that is shown in the dialog and write the desired statement freely (i.e., surface editing).

However, if the user is not familiar with the RoboWorld grammar, she can select the string "Statement" and click on the *help* button. By doing this, the tool displays the dialog that

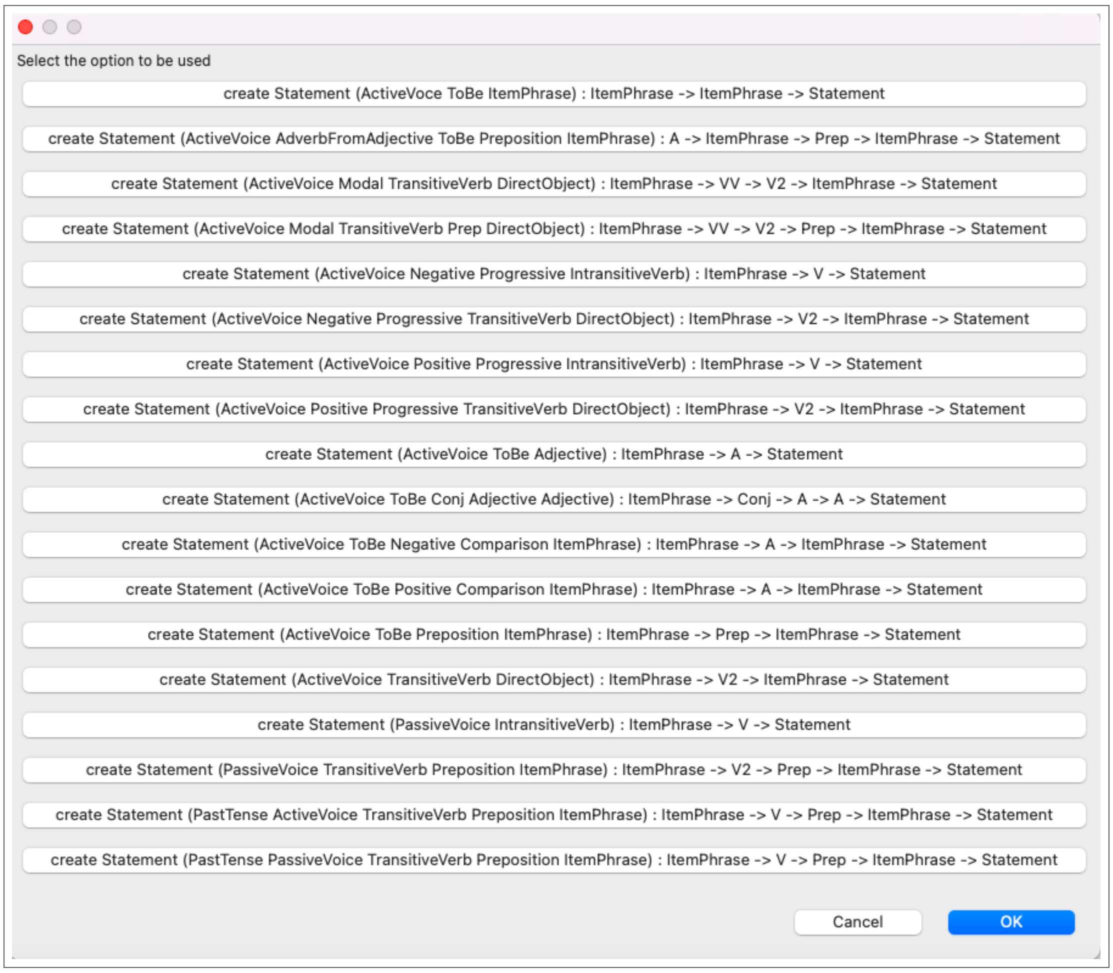
Figure 24 – Add New Sentence Dialog.



Source: Current Author

is shown in Figure 25. The tool dynamically retrieves all possibilities of writing a statement according to the RoboWorld grammar (i.e., structural editing).

Figure 25 – Possibilities of Writing a Statement.


Select the option to be used

create Statement (ActiveVoce ToBe ItemPhrase) : ItemPhrase -> ItemPhrase -> Statement
create Statement (ActiveVoice AdverbFromAdjective ToBe Preposition ItemPhrase) : A -> ItemPhrase -> Prep -> ItemPhrase -> Statement
create Statement (ActiveVoice Modal TransitiveVerb DirectObject) : ItemPhrase -> VV -> V2 -> ItemPhrase -> Statement
create Statement (ActiveVoice Modal TransitiveVerb Prep DirectObject) : ItemPhrase -> VV -> V2 -> Prep -> ItemPhrase -> Statement
create Statement (ActiveVoice Negative Progressive IntransitiveVerb) : ItemPhrase -> V -> Statement
create Statement (ActiveVoice Negative Progressive TransitiveVerb DirectObject) : ItemPhrase -> V2 -> ItemPhrase -> Statement
create Statement (ActiveVoice Positive Progressive IntransitiveVerb) : ItemPhrase -> V -> Statement
create Statement (ActiveVoice Positive Progressive TransitiveVerb DirectObject) : ItemPhrase -> V2 -> ItemPhrase -> Statement
create Statement (ActiveVoice ToBe Adjective) : ItemPhrase -> A -> Statement
create Statement (ActiveVoice ToBe Conj Adjective Adjective) : ItemPhrase -> Conj -> A -> A -> Statement
create Statement (ActiveVoice ToBe Negative Comparison ItemPhrase) : ItemPhrase -> A -> ItemPhrase -> Statement
create Statement (ActiveVoice ToBe Positive Comparison ItemPhrase) : ItemPhrase -> A -> ItemPhrase -> Statement
create Statement (ActiveVoice ToBe Preposition ItemPhrase) : ItemPhrase -> Prep -> ItemPhrase -> Statement
create Statement (ActiveVoice TransitiveVerb DirectObject) : ItemPhrase -> V2 -> ItemPhrase -> Statement
create Statement (PassiveVoice IntransitiveVerb) : ItemPhrase -> V -> Statement
create Statement (PassiveVoice TransitiveVerb Preposition ItemPhrase) : ItemPhrase -> V2 -> Prep -> ItemPhrase -> Statement
create Statement (PastTense ActiveVoice TransitiveVerb Preposition ItemPhrase) : ItemPhrase -> V -> Prep -> ItemPhrase -> Statement
create Statement (PastTense PassiveVoice TransitiveVerb Preposition ItemPhrase) : ItemPhrase -> V -> Prep -> ItemPhrase -> Statement

Source: Current Author

When the user selects one of the available possibilities, the dialog shown in Figure 24 is updated to reflect the required arguments for creating the selected type of statement. Now, the user can select other categories than Statement to get help on how to write them. This process of structural editing can continue until the level of words in the dictionary. However, the user can also write directly any text fragment whenever he fills confident enough.

4.3 DERIVING THE INTERMEDIATE REPRESENTATION

The IR generation produces an instance of the class `RwIntermediateRepresentation` shown in Figure 9. The syntax trees obtained after parsing the RoboWorld document are stored in an object called `rwDoc`, whose type is `RWDocument`. These trees are grouped according to the associated sentence type (i.e., arena assumption, robot assumption, etc.).

Source Code 4.1 presents the top-level pseudocode that produces an IR from a given RoboWorld document. After creating a default IR (line 2), its attributes are derived from the corresponding sentences; for example, information about the arena (`rwIR.arena`) is obtained from the arena assumptions (`rwDoc.arenaAssumptions`).

Source Code 4.1 – Generation of IR.

```
 1  mapRWDoc(rwDoc : RWDocument) : RWIntermediateRepresentation =
 2    rwIR = new RWIntermediateRepresentation();
 3    rwIR.arena = mapArena(rwDoc.arenaAssumptions);
 4    rwIR.robot = mapRobot(rwDoc.robotAssumptions);
 5    rwIR.entities = mapEntities(rwDoc);
 6    rwIR.inputEventMappings = mapInputEvents(rwDoc.inputEventMappings);
 7    rwIR.outputEventMappings = mapOutputEvents(rwDoc.outputEventMappings);
 8    rwIR.operationMappings = mapOperations(rwDoc.operationMappings);
 9    rwIR.variableMappings = mapVariables(rwDoc.variableMappings);
10    return rwIR;
```

Source: Current Author

To identify entities and their properties, it is necessary to traverse different sentences of a RoboWorld document (line 5), and, thus, the entire document is provided as argument. Typically, there will be an arena assumption saying that a given entity may appear in the arena, and an element assumption will define expected properties of this particular entity.

For illustration, we present here the definition of `mapArena` in Source Code 4.2. First (line 2), it creates a default arena object. Then, the code iterates over each arena assumption (line 3) updating the arena object accordingly. To do this, this code relies on auxiliary definitions that check, for instance, whether the given assumption contains information about the arena dimensionality.

The declaration of the aforementioned auxiliary definitions relies on the control imposed by the RoboWorld grammar, which enables the automatic generation of the intermediate

Source Code 4.2 – Mapping Arena Information.

```
mapArena(assumptions : List(ArenaAssumption)) : Arena =
  arena = new Arena();
  foreach (assumption : assumptions)
  {
    if (findArenaDimensionInfo(assumption))
    {
      arena.dimension = getArenaDimensionInfo(assumption);
    }
    else if (findArenaClosedInfo(assumption))
    {
      arena.closed = getArenaClosedInfo(assumption);
    }
    ...
  }
  return arena;
```

Source: Current Author

representation. To illustrate this, consider Source Code 4.3, which checks whether a given assumption has information about the arena dimensionality.

Source Code 4.3 – Finding Arena Dimensionality Information.

```
1  findArenaDimensionInfo(assumption : ArenaAssumption) : boolean =
2    if (refersToArena(assumption.statement.itemPhrase)
3        && assumption.statement instanceof mkStatement_ActiveVoice_ToBe_Adjective
4        && isPositiveSentence(assumption.statement))
5    {
6            adj = ((mkStatement_ActiveVoice_ToBe_Adjective) assumption.statement).a;
7            return isDimensionAdjective(adj);
8    }
9    ...
10   else { return false; }
```

Source: Current Author

First, note that all RoboWorld statements have a leading `ItemPhrase` (see Source Code 3.4). Therefore, in Source Code 4.3, we check whether the leading `ItemPhrase` refers to the arena (line 2). Additionally, considering the RoboWorld writing possibilities, dimensionality information can be given as a statement in the active voice using the verb to be and an adjective (see Source Code 3.4, lines 10–12). This expectation is translated to the verification performed on line 3). We also check that the statement is positive (line 4). If these conditions are met, we extract the adjective embedded in the statement (line 6), and check whether it is an adjective associated with dimensionality information (e.g., three-dimensional). If this is the case, this function yields true (line 7); otherwise, it yields false (line 8). Sentences with other structures may also say something about the arena dimensionality, and additional checks are performed, as indicated on line 10.

Regarding other parts of a RoboWorld document (i.e., assumptions about the robot and other elements, as well as mapping information), the mapping process from statements to objects of the IR follows the general ideas presented before: the provided statements are traversed and analysed considering the structure of the RoboWorld grammar.

Since RoboWorld is a flexible natural-language notation, yet controlled, it is possible that one can write sentences adhering to its grammar, but whose information cannot be automatically mapped to the derived IR. In such a situation, an incomplete IR is obtained, representing the RoboWorld document partially. It would be necessary to update the code that generates the IR to be able to extract the missing information. However, it is important to say that the present code is capable of producing successfully the IR of different types of robotic applications, as discussed in Chapter 5.

4.4 CHECKING WELL-FORMEDNESS CONDITIONS

After generating the IR, we check for violations of well-formedness conditions (WFCs). This is done by traversing the derived `rwIR` object (see Section 4.3). In this section, we illustrate the verification of some WFCs, namely: RW2, RW6, and RW10 (see Section 3.4). RW2 states that the gradient of the arena must not be a negative number. As it can be seen in Source Code 4.4, this verification is straightforward: we check the value of the attribute `gradient` of the `arena` object (line 2). If its value is not as expected, an exception is raised (line 4).

Source Code 4.4 – Checking for violations of RW2.

```
1  checkWFC_RW2(rwIR : RWIntermediateRepresentation) : void =
2    if (rwIR.arena.gradient < 0.0)
3    {
4      throw new WFCException("The gradient of the arena must not be a negative number.");
5    }
```

Source: Current Author

The verification of RW6, which states that the gradient of all named regions must not be a negative number, is also straightforward (see Source Code 4.5). It suffices to iterate over each named region (line 2), and check whether the gradient is a negative number (line 4). In such a situation, an exception is also raised (lines 6–7).

Verifying RW10 is more interesting. This WFC states that the output events must refer to

Source Code 4.5 – Checking for violations of RW6.

```
checkWFC_RW6(rwIR : RWIntermediateRepresentation) : void =
  foreach (region : rwIR.arena.regions)
  {
    if (region.gradient < 0.0)
    {
      throw new WFCException("The gradient of the " + region.name
        + " must not be a negative number.");
    }
  }
```

Source: Current Author

existing robot components or entities. In Source Code 4.6, we iterate over each output event (line 2), and get the referred entities (line 4). Afterwards, for each referred entity (line 5), we first check whether it is a robot component. If this is not the case (line 7), it should be an entity of the IR. Therefore, on line 9 we provide the referred entity as an argument to the operation `findEntity`. If no corresponding entity is found (line 10), an exception is raised (lines 12–14).

Source Code 4.6 – Checking for violations of RW10.

```
 1  checkWFC_RW10(rwIR : RWIntermediateRepresentation) : void =
 2    foreach (outputEventMapping : rwIR.outputEventMappings)
 3    {
 4      refEntities = getReferredEntities(outputEventMapping);
 5      foreach (refEntity : refEntities)
 6      {
 7        if (!rwIR.robot.isRobotComponent(refEntity))
 8        {
 9          entity = rwIR.findEntity(refEntity);
10          if (entity == null)
11          {
12            throw new WFCException("The " + outputEventMapping.name + " output event must"
13              + " refer to existing robot components or entities (reference not found: "
14              + refEntity.name + " ).");
15          }
16        }
17      }
```

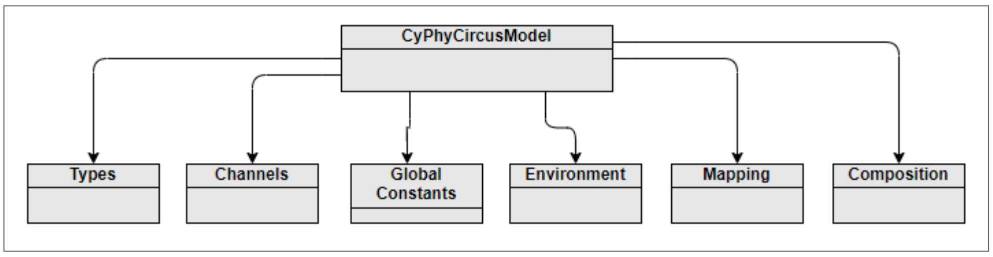
Source: Current Author

The verification of other WFCs follows the same general idea as presented above: we traverse the rwIR object and check for violations.

4.5 GENERATING THE CYPHYCIRCUS SEMANTICS

In this section, we describe how the CyPhyCircus semantics of RoboWorld documents are generated. This is done by traversing the `rwIR` object (see Section 4.3). As a result, it is created an instance of `CyPhyCircusModel`; Figure 26 shows the top-level classes of our CyPhyCircus model. Finally, the tool generates a textual representation (in LaTeX) of this object.

Figure 26 – CyPhyCircus Model.



Source: Current Author

In Source Code 4.7, we show the creation of the `semantics` object from the given intermediate representation. The class `Types` concerns the definition of general types associated to the robot (e.g., `Position`, `Velocity`, `Acceleration`), but also schemas for describing properties of the ground, the arena, the named regions, and the robot. For example, the definition of `ArenaProperty`, which is presented in Figure 11, is part of a `Types` object.

Source Code 4.7 – Generation of CyPhyCircus Semantics.

```
mapSemanticsl(rwIR : RWIntermediateRepresentation) : CyPhyCircusModel =
  semantics = new CyPhyCircusModel();
  semantics.types = mapTypes(rwIR);
  semantics.channels = mapChannels(rwIR);
  semantics.globalConstants = mapGlobalConstants(rwIR);
  semantics.environment = mapEnvironment(rwIR);
  semantics.mapping = mapMapping(rwIR);
  semantics.composition = mapComposition(rwIR);
  return semantics;
```

Source: Current Author

`Channels` concerns get/set channels, channels associated with input/output events, among others. These channels are used to send and receive information across the multiple parts of the semantics specification. In `GlobalConstants`, we have the definition of global variables, such as the arena, along with the associated constraints (e.g., the one presented in Figure 12).

In `Environment`, we have the definitions related to box 2 of Figure 10; that is, information about the environment state (see Figure 14), the robot movement (see Figure 15), the collision detection procedure, and communication with input and output buffers (see Figure 16). The semantics of mapping information on output events and operations are stored in Mapping (see Figure 17). Finally, in Composition, we have the parallel composition of the semantics of Environment and Mapping (see Figure 13). In what follows, we illustrate the generation of fragments of the CyPhyCircus semantics.

Source Code 4.8 presents the creation of the Composition object. As it can be seen in Figure 13, the process RoboWorldDocument is defined as the parallel composition of Environment and Mapping via the synchronisation on getSetChannels and eventHappenedChannels. The definition of RoboWorldDocument is standard, but the definition of the synchronisation channels depends on the specific robotic system being considered. Therefore, in Source Code 4.8, we call auxiliary functions to create the specific channels from a given IR (lines 3–4).

Source Code 4.8 – Generation of Composition.

```
1  mapComposition(rwIR : RWIntermediateRepresentation) : Composition =
2    composition = new Composition();
3    composition.getSetChannels = createCompGetSetChannels(rwIR);
4    composition.eventHappened = createCompEventHappened(rwIR);
5    return composition;
```

Source: Current Author

In Source Code 4.9, first, we consider default communication channels (line 3), such as getRobotPosition, setRobotPosition, among others. Then, if the arena may have obstacles, we define specific channels to deal with this information (lines 4–7). Similarly, if the robot is equipped with an odometer, additional channels are created (lines 8–11).

Source Code 4.9 – Generation of getSet channels of Composition.

```
1  createCompGetSetChannels(rwIR : RWIntermediateRepresentation) : List<Channel> =
2    channels = new List<Channel>();
3    channels.append(createCommonChannels());
4    if (rwIR.hasObstacle())
5    {
6            channels.append(createObstacleChannels());
7    }
8    if (rwIR.robot.hasOdometer())
9    {
10           channels.append(createOdometerChannels());
11   }
12   return channels;
```

Source: Current Author

A Happened channel is created for each output event and operation. Therefore, in Source

Code 4.10, we iterate over the list of output events and operations to define the necessary Happened channels.

Source Code 4.10 – Generation of getSet channels of `Composition`.

```
createCompEventHappened(rwIR : RWIntermediateRepresentation) : List<Channel> =
  channels = new List<Channel>();
  foreach (outputEvent : rwIR.outputEventMappings)
  {
    channels.append(createHappenedChannelName(outputEvent.name));
  }
  foreach (operation : rwIR.operationMappings)
  {
    channels.append(createHappenedChannelName(operation.name));
  }
  return channels;
```

Source: Current Author

The generation of other parts of the semantics (e.g., `Types`, `Channels`, etc.) is performed by the corresponding map operation (see Source Code 4.7). To obtain the textual representation of the semantics, one should invoke the operation `toString` on the `semantics` object. As mentioned before, the output is saved in the file *semantics.tex* within the open project. This file describes the semantics using the LaTeX style developed for creating CyPhyCircus specifications. Similarly to when generating an IR from a RoboWorld document, it is possible that a complete semantics cannot be automatically derived from a given IR. In such a situation, the semantics generation algorithms should be updated accordingly. It is important to say that the present algorithms are capable of generating successfully the semantics of different types of robotic applications, as discussed in Chapter 5.

As we comment in Section 6.2, the integration of the semantics of RoboWorld documents and RoboChart models, in order to exploit formal verification, simulation and testing of robotic systems is left as future work.

5 CASE STUDIES

The plug-in for RoboWorld was validated considering three case studies. Besides the running example (a rescue drone), we considered a ranger (Section 5.1) and a foraging (Section 5.2) robot. The plug-in, as well as all files related to the three case studies, are publicly available online[1]. We consider the validation as successful, since we managed to create the required RoboWorld documents, to check its well-formedness conditions, and to generate the expected CyPhyCircus semantics. In the following sections, we provide more details about these two additional case studies; for each one, we present the RoboChart model, the RoboWorld document, and fragments of the derived semantics.

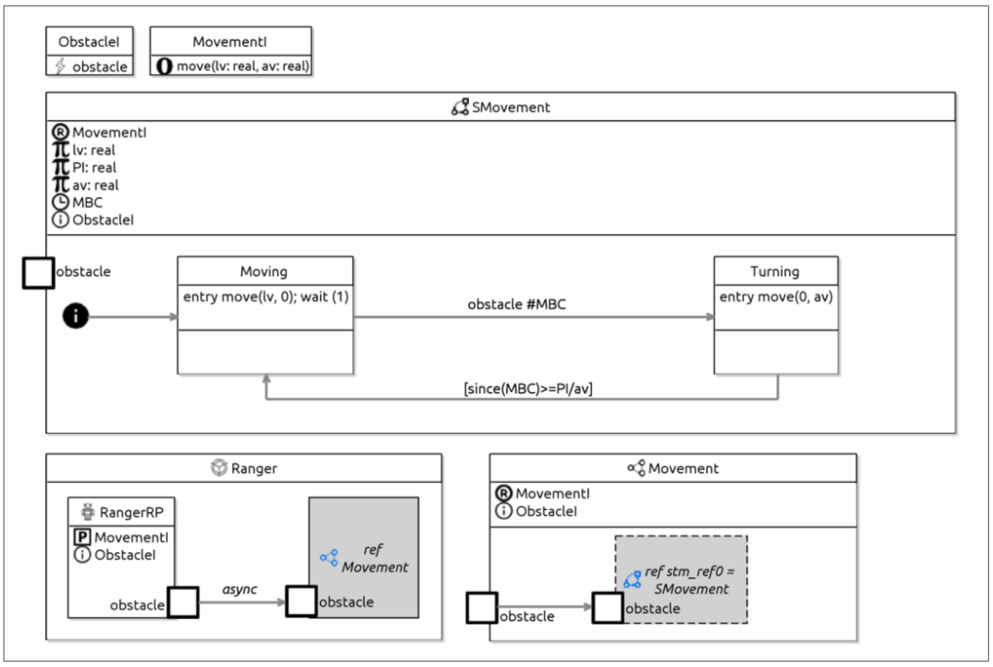
5.1 RANGER ROBOT

The ranger robot moves in the arena avoiding obstacles. Figure 27 shows the RoboChart model of the ranger. In the module `Ranger`, we have the composition of the robotic platform (`RangerRP`) with the control software, which refers to `Movement`. The robotic platforms declares two interfaces: `MovementI` and `ObstacleI`; the former is a provided interface, whereas the latter is a defined one. Operations (in this case, `move`) are declared in provided interfaces. Events (in this case, `obstacle`) are declared in defined interfaces. The operation `move` is responsible for moving the robot according to the provided linear velocity `lv` and angular velocity `av`. The input event `obstacle` signals the detection of an obstacle.

The controller `Movement` is defined in terms of an instance (`stm_ref0`) of the state machine `SMovement`. Note that the interface `MovementI` is required by the control software in order to operate correctly. The state machine `SMovement` has three constants (`lv`, `av`, and `PI`), and a clock (`MBC`), in addition to requiring the interface `MovementI` and considering the interface `ObstacleI`. The initial state is `Moving`. It invokes the operation `move` when reached. After waiting one time unit, the machine starts to monitor the occurrence of the input event `obstacle`. When it occurs, the clock `MBC` is reset, and the machine goes to the `Turning` state. As an entry action, it performs the operation `move` to turn the robot in order to avoid the obstacle. After waiting at least `PI/av` time units, the machine returns to the state `Moving`. In what follows, we present the RoboWorld document of this case study (Section 5.1.1), besides fragments of the intermediate representation (Section 5.1.2) and semantics (Section 5.1.3).

[1] Link: <https://cin.ufpe.br/~ghpc/msc_fwrj_dataset.zip>.

Figure 27 – Ranger RoboChart Model.



Source: (CAVALCANTI; BAXTER; CARVALHO, 2021)

5.1.1 RoboWorld document

In what follows, we have the RoboWorld document for the ranger robot. The document has seven sections, divided among assumptions (the first three sections) and mapping information (the last four sections). In the first section (`Arena Assumptions`), we have assumptions to state that the arena is two-dimensional (i.e., the z-dimension is not relevant for this robotic application), the gradient of the ground is 0.0 (i.e., the floor is flat), and there are obstacles in the arena. Regarding the robot (`Robot Assumptions`), its body is not relevant for modelling purposes, and, thus, it is treated as a point mass. In `Element Assumptions`, we have the information that one quarter of the arena contains obstacles.

```
## ARENA ASSUMPTIONS ##
The arena is two-dimensional.
The gradient of the ground is 0.0.
The arena has obstacles.


## ROBOT ASSUMPTIONS ##
The robot is a point mass.
```

```
## ELEMENT ASSUMPTIONS ##
One quarter of the arena contains obstacles.

## MAPPING OF INPUT EVENTS ##
The event obstacle occurs when
  the distance from the robot to an obstacle is less than 0.5 m.

## MAPPING OF OUTPUT EVENTS ##

## MAPPING OF OPERATIONS ##
When the operation move(lv,av) is called,
  the linear velocity of the robot is set to lv m/s
  towards the orientation of the robot,
  and the angular velocity of the robot is set to av rad/s.

## MAPPING OF VARIABLES ##
```

As shown in Figure 27, the robotic platform of the ranger provides one operation (`move`), and defines one input event (`obstacle`). In the RoboWorld document, we have the information that this input event occurs when the distance from the robot to an obstacle is less than 0.5 m (`Mapping of Input Events`). Concerning the `move` operation, it affects the environment, particularly, the robot, by changing its linear and angular velocity according to the received arguments `lv` and `av`, respectively. Since this robotic application has no output events or variables, we have no mapping information associated with these two sections. To recognise all sentences presented above, it was necessary to add just two words (*lv* and *av*) to the project-specific dictionary.

5.1.2 Intermediate representation

Source Code 5.1 shows a textual representation of the IR derived from the RoboWorld document presented in the previous section. `RWIntermediaryRepresentation` is the object that contains the attributes of the IR. The first attribute (`arena`; lines 2–11) records assumptions about the arena: its dimensionality, and the gradient of the floor. Since no restrictions were imposed to its size, the attributes xwidth, ywidth, and zwidth are null. Unless otherwise stated, all arenas are assumed to be open environments. As no named regions were informed in the RoboWorld document, the attribute regions is an empty list. The attribute properties is also an empty list.

The information that the arena has obstacles is used to identify such an entity (lines 14–17). The assumption that one quarter of the arena contains obstacles is recorded as a property of the entity named "obstacle" (line 16). The attribute robot has the assumption that the robot is abstracted as a point mass.

Each mapping information relates to a specific object. The mapping information about the input event obstacle relates to the object of type InputEventMappingIR. The condition necessary to trigger the event obstacle is recorded as a property of an InputSometimesIR object. Similarly, the assumption about move is kept in an instance of OperationMappingIR. Since this robotic application has no output events, nor variables, the corresponding lists are empty.

5.1.3 CyPhyCircus semantics

The CyPhyCircus semantics generated from the IR follows the structure discussed in Section 3.5. Here, we show fragments of the semantics, with a focus on the input event obstacle and on the operation move.

Figure 28 shows the formalisation of the input event obstacle. We test whether there is at least one obstacle (obstacle1), such that it has a location (loc) whose distance to the robot position is less than 0.5 m. If such a condition is met, the event obstacleTriggered!**True** is performed. Otherwise, **False** is communicated over obstacleTriggered.

In parallel, we have the buffer obstacle_Buffer, which synchronises on communications over the channel obstacleTriggered. Every time an information is sent over this channel, the buffer saves it in the variable obstacleTrig. When this variable becomes **True**, the event obstacle.in is performed, signalling to the control software that the input event obstacle has occurred.

Figure 30 shows the formalisation of the operation move. Since this robotic application has no output events, and move is the only operation available, the process Mapping is defined as the mapping of move_OperationMapping. The latter is defined as the parallel synchronisation of the move semantics (move_Semantics) and its monitor (move_Monitor).

Source Code 5.1 – Ranger intermediate representation.

```
RWIntermediaryRepresentation {
  arena = Arena {
    dimension = 2D,
    gradient = 0.0,
    xwidth = null,
    ywidth = null,
    zwidth = null,
    closed = false,
    properties = [],
    regions = []
  }

  entities = [
    Entity {
      name = "obstacle",
      properties = ["one quarter of the arena contains obstacles"]
    }
  ]

  robot = RobotBody {
    properties = ["the robot is a point mass"],
    components = []
  }

  inputEventMappings = [
    InputEventMappingIR {
      name = "obstacle",
      conditionalInput = InputSometimesIR {
        properties = ["the distance from the robot to an obstacle is less than 0.5 m"]
        communications = null
      }
    }
  ]

  outputEventMappings = []

  operationMappings = [
    OperationMappingIR {
      signature = Signature ( name = "move", parameters = ["lv", "av"] )
      output = OutputAlways {
        statements = ["the linear velocity of the robot is set to lv m/s"
           + "towards the orientation of the robot",
           "the angular velocity of the robot is set to av rad/s"]
      }
    }
  ]

  variableMappings = []
}
```

Source: Current Author

When the controller calls the move operation (`moveCall?lv?av`), the provided arguments (`lv` and `av`) are received from the channel `moveCall`. Then, the environment is affected by the

Figure 28 – Ranger Semantics: Input Event `obstacle`.

$$
\begin{array}{l}
obstacle\_InputEventMapping \mathrel{\widehat{=}} \\
\quad \mathbf{if}(\exists\, obstacle1 : \operatorname{ran} obstacles \bullet \exists\, loc : obstacle1.locations \bullet \\
\qquad (norm\,(loc - robot.position) < 0.5)) \longrightarrow \\
\qquad\quad obstacleTriggered!\mathbf{True} \\
\qquad\quad \longrightarrow obstacleOccurred, obstacleTime := \mathbf{True}, time \\
\quad [\!] \ \neg\,(\exists\, obstacle1 : \operatorname{ran} obstacles \bullet \exists\, loc : obstacle1.locations \bullet \\
\qquad (norm\,(loc - robot.position) < 0.5)) \longrightarrow \\
\qquad\quad obstacleTriggered!\mathbf{False} \longrightarrow \mathbf{Skip} \\
\quad \mathbf{fi}
\end{array}
$$

Source: Current Author

Figure 29 – Ranger Semantics: Buffers.

$$
\begin{array}{l}
obstacle\_Buffer \mathrel{\widehat{=}} \mathbf{var}\ obstacleTrig : \mathbb{B} \bullet obstacleTrig := \mathbf{False}; \\
\quad \mu X \bullet \left(\begin{array}{l} obstacleTriggered?b \longrightarrow obstacleTrig := b \\ \Box \\ (obstacleTrig = \mathbf{True})\ \&\ obstacle.in \longrightarrow \mathbf{Skip} \end{array} \right);\ X
\end{array}
$$

Source: Current Author

Figure 30 – Ranger Semantics: Mapping Process.

$$
\begin{array}{l}
\mathbf{process}\ move\_OperationMapping \mathrel{\widehat{=}} \mathbf{begin} \\
\\
move\_Semantics \mathrel{\widehat{=}} \\
\quad moveCall?lv?av \longrightarrow \\
\qquad (getRobotOrientation?robotOri \longrightarrow \\
\qquad ((setRobotVelocity!(lv * orientationToVector\ robotOri) \longrightarrow Skip); \\
\qquad (setRobotAngularVelocity!av \longrightarrow \mathbf{Skip})); \\
\qquad move\_Semantics) \\
\quad \Box \\
\quad proceed \longrightarrow move\_Semantics \\
\\
move\_Monitor \mathrel{\widehat{=}} moveCall?lv?av \longrightarrow moveHappened \longrightarrow move\_Monitor \\
\\
\bullet\ move\_Semantics\ [\![\, \varnothing \mid \{\!|\, moveCall \,|\!\} \mid \varnothing \,]\!]\ move\_Monitor \\
\\
\mathbf{end} \\
\\
\\
\mathbf{process}\ Mapping \mathrel{\widehat{=}} move\_OperationMapping
\end{array}
$$

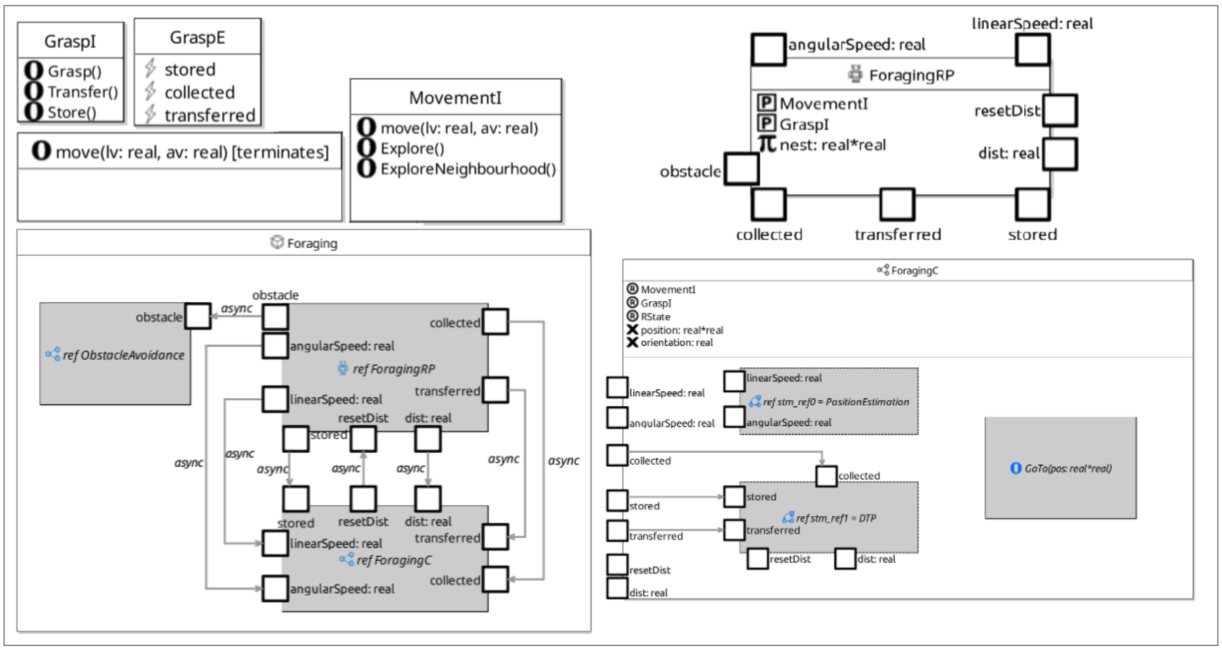
Source: Current Author

call as defined in the RoboWorld document: the linear velocity of the robot is set to `lv` towards its current orientation, and the angular velocity of the robot is set to `av`. The controller call (moveCall) is also captured by the move monitor, which performs the event moveHappened. This event synchronises with a buffer of the environment (see Figure 10), which records the time this operation was called by the controller. This part of the semantics is not shown here.

5.2 FORAGING ROBOT

The foraging robot explores the arena, collecting objects, and bringing them back to the nest. Among the considered case studies, the foraging robot is the most complex one. Figure 31 presents the top-level definitions. The module Foraging is composed by a robotic platform (ForagingRP) and two controllers (ObstacleAvoidance and ForagingC), which operate in parallel and are responsible for avoiding collision and collecting resources, respectively.

Figure 31 – Foraging RoboChart Model.



Source: <https://robostar.cs.york.ac.uk/case_studies/>

We have seven input events: obstacle indicates the presence of an obstacle, collected indicates that a resource has been collected, stored indicates that the collected resource has been stored in the nest, transferred indicates that a resource has been transferred between robots. The other three events (dist, angularSpeed, and linearSpeed) are always available and serve to communicate the value of the robot's odometer, the angular speed of the robot, and the linear speed of the robot, respectively. There is just one output event (resetDist), which is used by the control software to reset the value of the odometer.

A number of operations are provided by the platform: `Explore` makes the robot explore the arena, `ExploreNeighborhood` makes the robot explore the vicinity of a source of resources, `Transfer` makes the robot transfer resources to another robot (if more than one robot is deployed), `Store` makes the robot store a resource in the nest, `Grasp` makes the robot collect a resource, and `move` makes the robot move. The controllers are defined by referring to state machines that describe how the robot avoids obstacles, how the robot's position is estimated, and how resources are collected, transferred, and stored. Here, we do not provide further details on these state machines. The given overview is sufficient to understand the validation of RoboWorld that is described in the next sections. In what follows, we present the RoboWorld document of this case study (Section 5.2.1), besides fragments of the intermediate representation (Section 5.2.2) and semantics (Section 5.2.3).

5.2.1 RoboWorld document

In what follows, we have the RoboWorld document for the foraging robot. The first three sections describe the assumptions about the environment. The arena is two-dimensional, and it has obstacles and objects. Differently from the other case studies, the arena is closed; that is, we assume that is enclosed by colliding walls. Additionally, there are two named regions in the arena: the `nest`, and the `source`. Similarly to the other case studies, we abstract the body of the robot, and it is treated as a point mass. Here, the robot may carry one object. The other assumptions state that the `nest` and the `source` do not have obstacles inside, but both of them block movement. We also have assumptions about the size of these two regions, and they are at least 1.0 m apart. The RoboWorld document also informs that the `source` contains five objects initially, and the `nest` may contain up to five objects.

```
## ARENA ASSUMPTIONS ##
The arena is two-dimensional.
The arena has obstacles.
The arena has objects.
The arena is closed.
The arena has a nest region.
The arena has a source region.
```

```
## ROBOT ASSUMPTIONS ##
The robot is a point mass.
The robot has an odometer.
The robot may carry 1 object.

## ELEMENT ASSUMPTIONS ##
The nest has no obstacles.
The source has no obstacles.
The nest has an x-width of 0.25 m and a y-width of 0.25 m.
The source has an x-width of 0.25 m and a y-width of 0.25 m.
The nest region blocks movement.
The source region blocks movement.
The distance from the nest to the source is greater than 1.0 m.
The source contains 5 objects initially.
The nest may contain up to 5 objects.
```

Concerning the input events, the event `obstacle` occurs when the robot is close enough to an obstacle; its distance to the obstacle is less than 1.0 m. The events `collected` occurs when the robot is close enough to the `source`, and it has already collected the object. Similarly, the event `stored` occurs when the robot is close enough to the `nest`, and it has already delivered the object. Although the control software has been designed considering the presence of multiple robots, and objects being transferred between them, the RoboWorld document has an assumption stating that the event `transferred` will never happens. In other words, it considers a simpler application scenario where objects are not transferred between robots. At this moment, the RoboWorld semantics can only cater for a single robot. The last three sentences of `Mapping of Input Events` describe the value that is communicated by the events `dist`, `angularSpeed`, and `linearSpeed`, which are always available.

```
## MAPPING OF INPUT EVENTS ##
The event obstacle occurs
  when the distance from the robot to an obstacle is less than 1.0 m.
The event collected occurs
  when the distance from the robot to the source is less than 1.0 m,
  and the robot is carrying an object.
```

```
The event stored occurs
  when the distance from the robot to the nest is less than 1.0 m,
  and the robot is not carrying an object.
The event transferred never happens.
The event dist is always available,
  and it communicates the value of the odometer.
The event angularSpeed is always available,
  and it communicates the angular velocity of the robot.
The event linearSpeed is always available,
  and it communicates the magnitude of the velocity of the robot.

## MAPPING OF OUTPUT EVENTS ##
When the event resetDist occurs the odometer is reset.

## MAPPING OF OPERATIONS ##
The operation Explore()
  is defined by a diagram where one time unit is 1.0 s.
The operation ExploreNeighbourhood()
  is defined by a diagram where one time unit is 1.0 s.
When the operation Transfer() is called, nothing happens.
When the operation Store() is called,
  as soon as the distance from the robot to the source is less than 1.0 m,
  the robot places an object in the nest.
When the operation Grasp() is called,
  as soon as the distance from the robot to the nest is less than 1.0 m,
  the robot takes an object from the source.
When the operation move(lv,av) is called,
  the linear velocity of the robot is set to lv m/s
  towards the orientation of the robot,
  and the angular velocity of the robot is set to av rad/s.

## MAPPING OF VARIABLES ##
```

As mentioned before, when the output event resetDist is performed by the control software, it affects the robot by resetting its odometer. This is also recorded as part of the RoboWorld document. The effect on the environment of calling the operations Explore and ExploreNeighbourhood are described by diagrams, which are not presented here. When the operation Transfer is called, nothing happens due to the simplification explained before. The RoboWorld document also records how the operations Store and Grasp affect the environment: the former makes the robot make an object in the nest, whereas the latter makes the robot take an object from the source. When the operation move is called, the linear and angular velocities of the robot are updated accordingly. This robotic application does not have variables. To recognise all sentences presented above, it was necessary to add just five words (*nest*, *source*, *object*, *lv* and *av*) to the project-specific dictionary.

5.2.2 Intermediate representation

Source Code 5.2 shows an excerpt from the IR associated with this case study. Here, we highlight the definition of named regions, and different types of input event mappings. As defined in the RoboWorld document (see Section 5.2.1), the arena comprises two named regions: the nest (lines 5–16) and the source (lines 17–27). As it can be seen, the objects of type NamedRegion group the information related to these regions: their dimensions (lines 9–10, 21–22), and additional properties (lines 13–15, lines 25–26). Additionally, these regions are considered closed (lines 12 and 24), since they block movement. As explained before, grouping this information facilitates the generation of the semantics.

Regarding mapping information of input events, we highlight two situations. Since the event transferred never happens, in the IR (lines 33–36), its conditionalInput is of type InputNever. The event dist (lines 37–42), which is always available, illustrates the use of the attribute communications, which records the value that is communicated by the event.

5.2.3 CyPhyCircus semantics

The CyPhyCircus semantics generated from the IR follows the structure discussed in Section 3.5. Here, we show fragments of the semantics, with a focus on how assumptions about named regions and the robot are formalised. Figure 32 shows the global definitions generated for the foraging semantics. Besides the arena, we have obstacles. The assumption that the

Source Code 5.2 – Foraging intermediate representation.

```
RWIntermediaryRepresentation {
  arena = Arena {
    ...
    regions = [
      NamedRegion = {
        name = "nest region",
        dimension = 2D,
        gradient = 0.0,
        xwidth = RDimension{value = 0.25},
        ywidth = RDimension{value = 0.25},
        zwidth = RDimension{value = 0.0},
        closed = true,
        properties = ["the nest has no obstacles",
          "the distance from the nest to the source is greater than 1.0 m",
          "the nest may contain up to 5.0 objects"],
      },
      NamedRegion = {
        name = "source region",
        dimension = 2D,
        gradient = 0.0,
        xwidth = RDimension{value = 0.25},
        ywidth = RDimension{value = 0.25},
        zwidth = RDimension{value = 0.0},
        closed = true,
        properties = ["the source has no obstacles",
          "the source contains 5.0 objects initially"],
      }
    ]
  }
  ...
  inputEventMappings = [
    ...
    InputEventMappingIR {
      name = "transferred",
      conditionalInput = InputNever {}
    }
    InputEventMappingIR {
      name = "dist",
      conditionalInput = InputAlways {
        communications = ["the value of the odometer"]
      }
    }
    ...
  ]
  ...
}
```

Source: Current Author

nest and the source have no obstacles is formalised by two predicates, one for each region: for each `obstacle` in the arena, its position cannot be within the `nest` (`source`). Information about the size of the regions is also captured by the semantics. Moreover, we have a predicate to formalise the assumption about the separation of these two regions: for each location `loc1` of the `nest`, and for each location `loc2` of the `source`, the distance between `loc1` and `loc2` needs to be greater than 1.0 m.

Figure 32 – Foraging Semantics: Global Constants.

$$
\begin{array}{|l}
arena : ArenaProperty \\
obstacles : ObstacleID \rightarrow ObstacleProperty \\
\hline
\forall\, obstacle : \mathrm{ran}\ obstacles \bullet obstacle.position \notin arena.nest.locations \\
arena.nest.xwidth = 0.25 \\
arena.nest.ywidth = 0.25 \\
\forall\, loc1 : arena.nest.locations \bullet \forall\, loc2 : arena.source.locations \bullet \\
\qquad norm\,(loc2 - loc1) > 1.0 \\
arena.nest.locations \subseteq arena.locations \\
\hline
\forall\, obstacle : \mathrm{ran}\ obstacles \bullet obstacle.position \notin arena.source.locations \\
arena.source.xwidth = 0.25 \\
arena.source.ywidth = 0.25 \\
arena.source.locations \subseteq arena.locations
\end{array}
$$

Source: Current Author

Figure 33 shows the schema `EnvironmentState`, which defines the dynamic aspects of the environment; in this case study, in addition to the robot, we have objects that may be carried around the arena.

Figure 33 – Foraging Semantics: Environment State.

$$
\begin{array}{|l}
\hline
EnvironmentState \\
\mathbf{visible}\ robot : RobotProperty \\
\mathbf{visible}\ objects : ObjectID \rightarrowtail\!\!\!\!\!\!\!+ \ ObjectProperty \\
time : \mathbb{R} \\
stepTimer : \mathbb{R} \\
EventTimes \\
\hline
\#\{objectID : \mathrm{dom}\ objects \mid (objects\ objectID).carried = \mathbf{True}\} \leq 1 \\
robot.position \in arena.locations \\
\#\{objectID : \mathrm{dom}\ objects \mid (objects\ objectID).position \in arena.nest.locations\} \leq 5 \\
\forall\, object : \mathrm{ran}\ objects \mid object.carried = \mathbf{True} \bullet \\
\qquad object.position = robot.position \\
\forall\, object : \mathrm{ran}\ objects \bullet \\
\qquad object.position \in arena.locations \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
EnvironmentStateInit \\
EnvironmentState' \\
\hline
\#\{objectID : \mathrm{dom}\ objects \mid (objects\ objectID).position \in arena.source.locations\} = 5 \\
time' = 0.0 \\
stepTimer' = 0.0 \\
EventTimesInit \\
\hline
\end{array}
$$

Source: Current Author

The assumption that the robot can only carry one object at a time is formalised using set comprehension. The first predicate of the schema `EnvironmentState` creates a set with all objects that are being carried), then, it states that the size of this set is less than or equal to one. Moreover, since objects are carried by the robot, we have a universal quantification (fourth predicate of `EnvironmentState`) stating that if an object is being carried, its position is always equal to the position of the robot.

In the RoboWorld document, we have the assumption that the `nest` may contain up to five objects. This information is also formalised using set comprehension (third predicate of `EnvironmentState`): the number of objects whose position is within the `nest` is less than or equal to five. Regarding the `source`, it is assumed that it has five objects initially. This is captured by the schema `EnvironmentStateInit`, which initialises the variables of the environment state. The first predicate of this schema defines that the number of objects whose position is within the `source` is equal to five. We remember that the developed RoboWorld plug-in, as well as all files related to the three case studies, are publicly available (see the link provided at the beginning of this chapter).

6 CONCLUSION

In this work, we have developed a RoboTool plug-in for RoboWorld, expanding, as a result, the tool support of the RoboStar framework. In (BAXTER et al., 2023), we give a concise account of the developed plug-in. This plug-in has a graphical user interface that enables the customisation of project-specific dictionaries. Moreover, it provides surface and structural editors of RoboWorld documents. Integration with underlying natural-language processing techniques and tools is transparent, and, thus, hidden from the end user.

From an intermediate representation of RoboWorld documents, which is derived automatically, the plug-in also has automatic support for checking well-formedness conditions, and generating a formal CyPhyCircus semantics. The RoboTool plug-in for RoboWorld was validated considering three case studies: a rescue drone (our running example), a ranger robot, and a foraging robot. These features enable the rigorous modelling, verification, simulation and testing of mobile and autonomous robots in conjunction with their surrounding environment. Therefore, we claim that we have reached the research goals defined in Section 1.2.

6.1 RELATED WORK

Here, we first position RoboWorld with respect to the literature on approaches to consider the environment when developing software systems in general, including control software for robotic systems (Section 6.1.1). Afterwards, we discuss the adoption of controlled natural languages, especially, when used in robotics or with a formal semantics (Section 6.1.2). Finally, we cover tool support for processing controlled natural languages (Section 6.1.3).

6.1.1 Software system environments

Taking a view of environment as the context in which a software operates, we can say that a software system always affects and is affected by its surrounding environment. Taking into account the environment is, therefore, fundamental during specification, design, development, and verification tasks. For example, in (TKACHUK; DWYER; PASAREANU, 2003), the authors describe an approach for generating environments of Java program fragments from formally specified assumptions and abstractions. Here, the environment is a group of classes.

When specifying concurrent systems using a process algebra, such as CSP for example, as mentioned in the very beginning (Chapter 1) of (ROSCOE, 2010), we set up and reason about processes that interact with their environments. Those environments are mechanisms that can be (potentially) represented by other processes. Since the focus of RoboStar is on robotics, RoboWorld is tailored to address physical environments of cyber-physical systems.

In (FURIA; ROSSI; MANDRIOLI, 2007), the authors argue that the modelling activity in the development of software systems should formalise as much as possible of the environment, since analysing the correctness of the system relies on an accurate model of the environment, of the software, and of their interaction. RT-Tester is a tool for automatic test generation, execution, and real-time evaluation (PELESKA et al., 2011) that follows this point of view. The expected software behaviour is modelled as state machines, which are also used to describe the system environment. When generating test cases, for instance, the models of the software and the environment are considered together to focus on valid scenarios, where realistic operational requirements are assumed.

In (LARSEN; MIKUCIONIS; NIELSEN, 2005), a timed input/output conformance relation (s rtioco$_e$ t) is proposed to relate correct implementations s of a specification t, under the environmental constraints expressed by e. The models of s and t, and even of the environment assumptions e, are all given as timed automata.

Closer to RoboWorld is the work reported in (SANTOS; CARVALHO; SAMPAIO, 2018), where environment restrictions are specified according to a controlled natural language. That work is for cyber-physical systems in general, and the language is inspired on concepts of LTL (linear temporal logic). The semantics is given in CSP and used as part of a test-generation technique.

None of the above notations is tailored for robotics. In contrast, RoboWorld includes domain-specific concepts such as a mobile robotic platform, including its services and their definitions, and arenas. In this way, RoboWorld facilitates the specification, since concepts of the robotics domain have a pre-defined semantics, which does not need to be specified for each application.

When modelling robotic systems, some works consider the environment to avoid unrealistic designs. For instance, in (DESAI et al., 2017; QUOTTRUP; BAK; IZADI-ZAMANABADI, 2004), implicit assumptions of the environment are to some extent captured by 3D and 2D grid maps. They describe a specific scenario where the designed robots are assumed to work, as opposed to general assumptions that might identify a collections of maps.

In (ASKARPOUR et al., 2021), a UML profile is used for designing human-robot collaborative systems. This profile has specific stereotypes to model entities from a scenario that interact with the robot in class and component diagrams. The RoboWorld notion of arena corresponds to that of a layout in (ASKARPOUR et al., 2021), but layouts are discrete spaces divided in sections that can be obstructed. In a component diagram, each section is a component, with connections representing adjacency. The component diagram is, therefore, a sort of map. Mathematical models for verification automatically generated use a temporal logic with a notion of discrete time. Differently, RoboWorld has a hybrid semantics, which accounts for the continuous nature of space and movement, for example.

In (MAOZ; RINGERT, 2015), the MontiArcAutomaton language (RINGERT; RUMPE; WORTMANN, 2015) is used for modelling components of robotic systems. In this approach, environment assumptions are specified as LTL properties, using AspectLTL (MAOZ; SAAR, 2011), a language whose syntax is similar to that of the SMV model checker. In RoboWorld, at the user discretion, properties of the environment are described in a more natural way, considering a controlled natural language, or referring to diagrams. Therefore, RoboWorld distinguishes itself by its flexibility on specifying general assumptions of the environments where a robot can work.

In (LESTINGI et al., 2023), the authors acknowledge the importance of modelling the environment of robotic applications; this is done using Stochastic Hybrid Automata (SHA). Formal analyses are performed using statistical model checking (SMC) and the UPPAAL model checker (BEHRMANN et al., 2006) to assess how likely the missions of the robotic system are to end in success.

Explicitly employing semi-formal or formal notations to describe the environment may hinder practical application, since the stakeholders typically aware of environment assumptions are not familiar with such notations. So, considering the use of (controlled) natural language to hide formal models is a promising alternative. This is addressed in the next section.

6.1.2 Controlled natural languages

As discussed in Section 1.1, techniques designed to process natural language specifications are commonly based on statistical approaches (that is, model-driven artificial intelligence techniques), where it is assumed that a large dataset of raw text is available to extract processing rules. This is not the reality of the development of robotic systems, where operational requirements are frequently left implicit. Employing statistical approaches trained with text not related to the robotics field would impose challenges to the automatic generation of a formal semantics. Therefore, RoboWorld is devised as a natural language, controlled, yet flexible, whose underlying structure favours automation. In the following paragraphs, we comment on other CNLs, mostly with formal semantics.

In (NOGUEIRA; SAMPAIO; MOTA, 2014), use cases are used as source for the generation of a CSP specification. The devised CNL is tailored for mobile applications. In (SCHWITTER, 2002), PENG is proposed as a restricted computer-processable CNL for writing unambiguous and precise requirements. The specifications written in PENG can be translated into first-order predicate logic. In (ESSER; STRUSS, 2007), requirements are written in a limited standardised format according to a strict if-then sentence template. This enables the translation of requirements into the Formal Requirement Language (FRL) proposed by the authors. In (SCHNELTE, 2009), assuming that the system specification is manually represented conforming to a set of templates, developed for automotive systems, a Temporal Qualified Expression (TQE) is derived.

More recently, in (LUTEBERGET et al., 2017), the authors propose RailCNL: a CNL for the railway domain that was designed as a middle ground between informal regulations and Datalog code. As RoboWorld, this CNL is also implemented using GF. In (GIANNAKOPOULOU et al., 2021), the authors use a structured natural language (FRETISH) that incorporates previous knowledge from NASA applications and has a Real-Time Graphical Interval Logic (RTGIL) semantics. The proposed CNL was used to capture and analyse requirements for a Lockheed Martin Cyber–Physical System challenge.

CNLs are also employed by the Behaviour Driven Development (BDDl) approach that allows the tester or business analyst to create test cases in a simple text language (English). This language helps even non-technical team members to understand what is going on in the software project. For instance, Cucumber[1] is a tool for BDD whose language (Gherkin) uses a set of special keywords to give structure and meaning to executable specifications. For a comprehensive survey of English-based CNLs, we refer to (KUHN, 2014), where 100 languages, covering the literature since 1930, are described.

In the robotics domain, previous works have also investigated the use of (controlled) natural languages. In (LINCOLN; VERES, 2013), the authors present a form of natural language called system-English (sEnglish) tailored for programming complex robotic systems. In (DRAGULE et al., 2021), besides presenting a survey of domain-specific languages for robot mission specifica-

[1] https://cucumber.io/

tion, the authors propose PROMISE, a textual and graphical domain specific language (DSL) for describing complex multi-robot missions. To the best of our knowledge, there are no CNLs tailored to the description of assumptions about the environment of a robotic application. So, the effort to specify such assumptions using an existing CNL would involve defining from scratch the semantics of concepts specific to the robotics domain, which are pre-defined in RoboWorld.

RoboWorld is realised using GF, which has been previously used to define CNLs with formal semantics. In addition to (LUTEBERGET et al., 2017), a language for deontic-based specifications for normative systems is presented in (CAMILLERI; PAGANELLI; SCHNEIDER, 2014). In that work, a semi-automatic way is provided to extract a description using the CNL from free-form texts. The language has a timed-automata semantics suitable for use of UPPAAL for verification (CAMILLERI; HAGHSHENAS; SCHNEIDER, 2018). Like we do here, the semantics is defined using a translation into an intermediate XML-like language. Like in our work, syntactic queries to check simple validity constraints can use the intermediate language, while more complex semantic queries use UPPAAL.

In (CAVALCANTI; BAXTER; CARVALHO, 2021), it is provided an overview of the RoboWorld syntax, semantics, and tool support using a couple of examples. As a general discussion of RoboWorld, this previous publication does not present the metamodel of RoboWorld, neither details its realisation using GF. The intermediate representation, as well as how it is derived from RoboWorld documents, and the RoboWorld semantics generation are also not addressed. In (BAXTER et al., 2023), these topics are properly covered, providing, therefore, a comprehensive definition of RoboWorld: metamodel, grammar, well-formedness conditions, formal semantics, and the RoboTool mechanisation. The latter, is the main contribution of this work.

6.1.3 Tool support

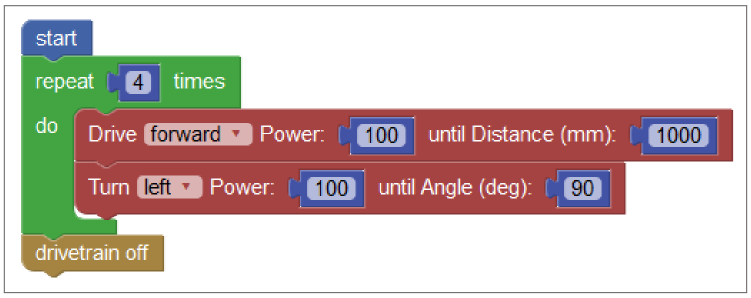
The tool support for editing text adhering to a controlled natural language ranges from basic surface (structural) editors to elaborate combinations of both editing approaches. In this section, we briefly comment on some of these tools.

The Robot Mesh Studio IDE[2] provides a graphical DSL for specifying missions of robotic systems. Although this language has fragments of a controlled natural language, the user de-

[2] Link: <http://docs.robotmesh.com/ide-project-page>

fines missions using a Blockly-based[3] interface (see Figure 34). Considering the many different writing possibilities enabled by RoboWorld, this Blockly-based approach is not suitable for editing RoboWorld documents; RoboWorld sentences do not adhere to limited writing templates where only some slots can be edited.

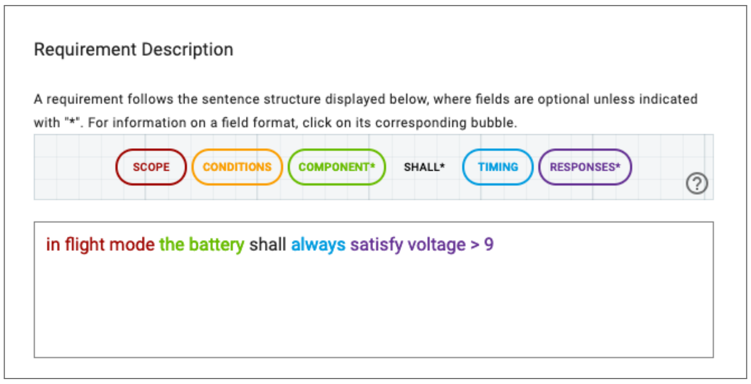
Figure 34 – Graphical DSL of the Robot Mesh Studio IDE.


start
repeat 4 times
do
Drive forward Power: 100 until Distance (mm): 1000
Turn left Power: 100 until Angle (deg): 90
drivetrain off

Source: <https://docs.robotmesh.com/v5p-1-vex-v5-education-guide-introduction>

The editing style adopted by the RoboWorld plug-in resembles the one supported by FRET: a Formal Requirements Elicitation Tool (GIANNAKOPOULOU et al., 2021). Experienced users, can write requirements directly in a surface editor, which provides syntax highlight and autocomplete (see Figure 35). In the Roboworld plug-in, this is similar to using the dialog for adding new sentences (see Figure 24).

Figure 35 – Writing Requirements in FRET.



Requirement Description

A requirement follows the sentence structure displayed below, where fields are optional unless indicated with "*". For information on a field format, click on its corresponding bubble.

in flight mode the battery shall always satisfy voltage > 9

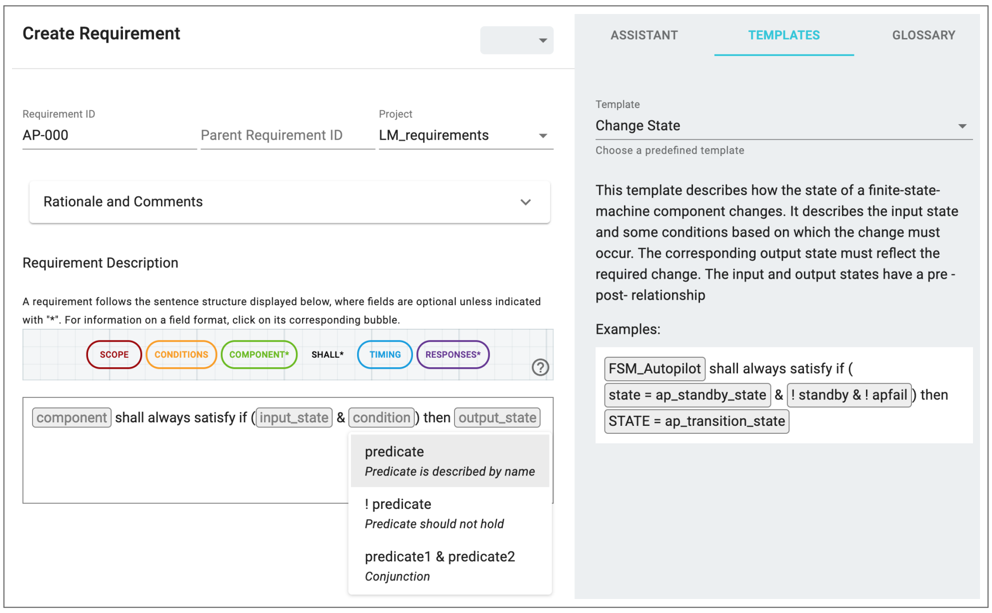
Source: <https://github.com/NASA-SW-VnV/fret/blob/master/fret-electron/docs/_media/user-interface/writingReqs.md>

As the RoboWorld plug-in (see Figure 25), FRET also combines its surface editor with a structural one, where the user can write sentences by following flexible writing templates (see Figure 36).

Some features available in FRET, such as autocomplete, can be implemented in the RoboWorld plug-in to improve its support for editing RoboWorld documents.

[3] Link: <https://developers.google.com/blockly>

Figure 36 – Using Templates in FRET.



Create Requirement

Requirement ID: AP-000
Parent Requirement ID
Project: LM_requirements

Rationale and Comments

Requirement Description

A requirement follows the sentence structure displayed below, where fields are optional unless indicated with "*". For information on a field format, click on its corresponding bubble.

component shall always satisfy if (input_state & condition) then output_state

predicate
Predicate is described by name

! predicate
Predicate should not hold

predicate1 & predicate2
Conjunction

Template: Change State
Choose a predefined template

This template describes how the state of a finite-state-machine component changes. It describes the input state and some conditions based on which the change must occur. The corresponding output state must reflect the required change. The input and output states have a pre - post- relationship

Examples:

FSM_Autopilot shall always satisfy if (state = ap_standby_state & ! standby & ! apfail) then STATE = ap_transition_state

Source: <https://github.com/NASA-SW-VnV/fret/blob/master/fret-electron/docs/_media/user-interface/writingReqs.md>

6.2 FUTURE WORK

In future work, we plan to mainly address to the following topics.

- **Improve the support for writing RoboWorld sentences**. Although surface and structural editors are available, some additional features should be developed to improve the support provided for writing RoboWorld sentences. The surface editor can be enhanced with an autocomplete feature. Regarding the structural editor, in addition to listing the writing possibilities (see Figure 25), it should show concrete examples (i.e., valid text) for each supported structure.

- **Update the RoboWorld tool support**. The present version of the RoboTool plug-in for RoboWorld is not updated with respect to the newest version of RoboWorld. As reported in (BAXTER et al., 2023), a more challenging case study (a firefighting UAV) has been recently considered. As a consequence, the RoboWorld language, along with its intermediate representation and formal semantics, have evolved to capture new types of assumptions and mapping information. Therefore, it is necessary to update the developed plug-in to support these newest extensions.

- **Exploit RoboWorld for verification, simulation and testing**. As mentioned before, RoboWorld enables the rigorous modelling, verification, simulation and testing of mobile and autonomous robots in conjunction with their surrounding environment. A preliminary application on test generation is reported in (BAXTER et al., 2023). Nevertheless, there are still many possibilities to investigate. For instance, a RoboWorld document can be used to analyse whether a robotic system can be employed in a particular scenario. In other words, checking whether this scenario meets the assumptions made.

- **Consider additional case studies**. To further validate RoboWorld and its tool support, it is necessary to consider more case studies, with varying types of assumptions and mapping information. Additionally, although RoboWorld is being developed with feedback from roboticists, it is important to carry out controlled studies with practitioners to evaluate the benefits and limitations of adopting RoboWorld.

REFERENCES

ASKARPOUR, M.; LESTINGI, L.; LONGONI, S.; IANNACCI, N.; ROSSI, M.; VICENTINI, F. Formally-based model-driven development of collaborative robotic applications. *Journal of Intelligent & Robotic Systems*, v. 102, n. 3, p. 59, 2021.

BAXTER, J.; CARVALHO, G.; CAVALCANTI, A.; JÚNIOR, F. R. RoboWorld: Verification of Robotic Systems with Environment in the Loop. *Form. Asp. Comput.*, Association for Computing Machinery, New York, NY, USA, 2023. ISSN 0934-5043.

BEHRMANN, G.; DAVID, A.; LARSEN, K. G.; HAKANSSON, J.; PETTERSON, P.; YI, W.; HENDRIKS, M. UPPAAL 4.0. In: *3rd International Conference on the Quantitative Evaluation of Systems*. [S.l.]: IEEE Computer Society, 2006. p. 125–126.

BEN-ARI, M.; MONDADA, F. Robots and their applications. In: ______. [S.l.: s.n.], 2018. p. 1–20. ISBN 978-3-319-62532-4.

CAMILLERI, J. J.; HAGHSHENAS, M. R.; SCHNEIDER, G. A Web-Based Tool for Analysing Normative Documents in English. In: *33rd Annual ACM Symposium on Applied Computing*. [S.l.]: Association for Computing Machinery, 2018. p. 1865–1872.

CAMILLERI, J. J.; PAGANELLI, G.; SCHNEIDER, G. A CNL for Contract-Oriented Diagrams. In: DAVIS, B.; KALJURAND, K.; KUHN, T. (Ed.). *Controlled Natural Language*. [S.l.]: Springer International Publishing, 2014. p. 135–146.

CAVALCANTI, A.; BARNETT, W.; BAXTER, J.; CARVALHO, G.; FILHO, M. C.; MIYAZAWA, A.; RIBEIRO, P.; SAMPAIO, A. RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing. In: ______. *Software Engineering for Robotics*. Cham: Springer International Publishing, 2021. p. 249–293. ISBN 978-3-030-66494-7.

CAVALCANTI, A.; BAXTER, J.; CARVALHO, G. RoboWorld: Where Can My Robot Work? In: CALINESCU, R.; PASAREANU, C. S. (Ed.). *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings*. Springer, 2021. (Lecture Notes in Computer Science, v. 13085), p. 3–22. ISBN 978-3-030-92124-8. Available at: <https://doi.org/10.1007/978-3-030-92124-8_1>.

CAVALCANTI, A.; DONGOL, B.; HIERONS, R.; TIMMIS, J.; WOODCOCK, J. *Software Engineering for Robotics*. Springer International Publishing, 2021. ISBN 9783030664947. Available at: <https://books.google.com.br/books?id=Two3EAAAQBAJ>.

CAVALCANTI, A.; SAMPAIO, A.; MIYAZAWA, A.; RIBEIRO, P.; Conserva Filho, M.; DIDIER, A.; LI, W.; TIMMIS, J. Verified simulation for robotics. *Science of Computer Programming*, v. 174, p. 1–37, 2019. ISSN 0167-6423.

DESAI, A.; SAHA, I.; YANG, J.; QADEER, S.; SESHIA, S. DRONA: A Framework for Safe Distributed Mobile Robotics. In: *8th International Conference on Cyber-Physical Systems*. [S.l.]: IEEE, 2017. p. 239–248.

DRAGULE, S.; GONZALO, S. G.; BERGER, T.; PELLICCIONE, P. Languages for specifying missions of robotic applications. In: ______. *Software Engineering for Robotics*. Cham: Springer International Publishing, 2021. p. 377–411. ISBN 978-3-030-66494-7.

ESSER, M.; STRUSS, P. Obtaining Models for Test Generation from Natural-Language like Functional Specifications. In: *International Workshop on Principles of Diagnosis*. [S.l.: s.n.], 2007. p. 75–82.

FOSTER, S.; BAXTER, J.; CAVALCANTI, A. L. C.; MIYAZAWA, A.; WOODCOCK, J. C. P. Automating Verification of State Machines with Reactive Designs and Isabelle/UTP. In: BAE, K.; ÖLVECZKY, P. C. (Ed.). *Formal Aspects of Component Software*. Cham: Springer, 2018. p. 137–155. Available at: <papers/FBCMW18.pdf>.

FOSTER, S.; CAVALCANTI, A. L. C.; CANHAM, S.; WOODCOCK, J. C. P.; ZEYDA, F. Unifying theories of reactive design contracts. *Theoretical Computer Science*, v. 802, p. 105 – 140, 2020. Available at: <papers/FCCWZ20.pdf>.

FURIA, C. A.; ROSSI, M.; MANDRIOLI, D. Modeling the environment in software-intensive systems. In: *International Workshop on Modeling in Software Engineering (MISE'07: ICSE Workshop 2007)*. [S.l.: s.n.], 2007. p. 11–11.

GIANNAKOPOULOU, D.; PRESSBURGER, T.; MAVRIDOU, A.; SCHUMANN, J. Automated formalization of structured natural language requirements. *Information and Software Technology*, v. 137, p. 106590, 2021. ISSN 0950-5849. Available at: <https://www.sciencedirect.com/science/article/pii/S0950584921000707>.

HOARE, C. *Communicating Sequencial Processes*. 2022. Available at: <http://www.usingcsp.com/cspbook.pdf>. Accessed on: 05 feb. 2023.

HOARE, C. A. R.; JIFENG, H. *Unifying Theories of Programming*. [S.l.]: Prentice-Hall, 1998.

KUHN, T. A survey and classification of controlled natural languages. *Comput. Linguist.*, MIT Press, Cambridge, MA, USA, v. 40, n. 1, p. 121?170, mar 2014. ISSN 0891-2017. Available at: <https://doi.org/10.1162/COLI_a_00168>.

LARSEN, K.; MIKUCIONIS, M.; NIELSEN, B. Online Testing of Real-time Systems Using UPPAAL. In: GRABOWSKI, J.; NIELSEN, B. (Ed.). *Formal Approaches to Software Testing*. [S.l.]: Springer Berlin Heidelberg, 2005. p. 79–94.

LESTINGI, L.; ZERLA, D.; BERSANI, M. M.; ROSSI, M. Specification, stochastic modeling and analysis of interactive service robotic applications. *Robotics and Autonomous Systems*, v. 163, p. 104387, 2023. ISSN 0921-8890.

LINCOLN, N.; VERES, S. M. Natural Language Programming of Complex Robotic BDI Agents. *Journal Intelligent Robotics Systems*, Kluwer Academic Publishers, v. 71, n. 2, p. 211–230, 2013.

LUTEBERGET, B. *Automated Reasoning for Planning Railway Infrastructure*. Phd Thesis (PhD Thesis), 2019.

LUTEBERGET, B.; CAMILLERI, J. J.; JOHANSEN, C.; SCHNEIDER, G. Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL. In: CIMATTI, A.; SIRJANI, M. (Ed.). *Software Engineering and Formal Methods*. [S.l.]: Springer International Publishing, 2017. p. 87–103.

MAOZ, S.; RINGERT, J. Synthesizing a Lego Forklift Controller in GR(1): A Case Study. In: CERNÝ, P.; KUNCAK, V.; MADHUSUDAN, P. (Ed.). *4th Workshop on Synthesis*. [S.l.: s.n.], 2015. (EPTCS, v. 202), p. 58–72.

MAOZ, S.; SAAR, Y. AspectLTL: An Aspect Language for LTL Specifications. In: *10th International Conference on Aspect-Oriented Software Development*. [S.l.]: Association for Computing Machinery, 2011. p. 19–30.

MIYAZAWA, A.; RIBEIRO, P.; LI, W.; CAVALCANTI, A.; TIMMIS, J.; WOODCOCK, J. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, v. 18, p. 1–53, 10 2019.

MONTHE, V.; NANA, L.; KOUAMOU, G. A model-based approach for common representation and description of robotics software architectures. *Applied Sciences*, v. 12, p. 2982, 03 2022.

MUNIVE, J. H. Y.; STRUTH, G.; FOSTER, S. Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In: *18th International Conference on Relational and Algebraic Methods in Computer Science*. [S.l.]: Springer, 2020. (Lecture Notes in Computer Science, v. 12062), p. 169–186.

NOGUEIRA, S.; SAMPAIO, A. C. A.; MOTA, A. C. Test generation from state based use case models. *Formal Aspects of Computing*, v. 26, n. 3, p. 441–490, 2014.

PELESKA, J.; VOROBEV, E.; LAPSCHIES, F.; ZAHLTEN, C. *Automated Model-Based Testing with RT-Tester*. [S.l.], 2011.

QUOTTRUP, M.; BAK, T.; IZADI-ZAMANABADI, R. Multi-robot planning: a timed automata approach. In: *IEEE International Conference on Robotics and Automation*. [S.l.: s.n.], 2004. v. 5, p. 4417–4422.

RANTA, A. *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications, 2011. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

RINGERT, J.; RUMPE, B.; WORTMANN, A. Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. *CoRR*, abs/1509.04505, 2015. Available at: <http://arxiv.org/abs/1509.04505>.

ROSCOE, A. W. *Understanding Concurrent Systems*. Springer, 2010. (Texts in Computer Science). ISBN 978-1-84882-257-3. Available at: <https://doi.org/10.1007/978-1-84882-258-0>.

SANTOS, T.; CARVALHO, G.; SAMPAIO, A. Formal modelling of environment restrictions from natural-language requirements. In: MASSONI, T.; MOUSAVI, M. (Ed.). *Formal Methods: Foundations and Applications*. [S.l.]: Springer International Publishing, 2018. p. 252–270.

SCHNELTE, M. Generating Test Cases for Timed Systems from Controlled Natural Language Specifications. In: *Proceedings of International Conference on System Integration and Reliability Improvements*. USA: [s.n.], 2009. p. 348–353.

SCHWITTER, R. English as a Formal Specification Language. In: *Proceedings of The International Workshop on Database and Expert Systems Applications*. France: [s.n.], 2002.

TKACHUK, O.; DWYER, M.; PASAREANU, C. Automated environment generation for software model checking. In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* [S.l.: s.n.], 2003. p. 116–127.

WOODCOCK, J.; CAVALCANTI, A. A concurrent language for refinement. In: . [S.l.: s.n.], 2001.

WOODCOCK, J.; DAVIES, J. *Using Z*. 1996. Available at: <https://www.cs.cmu.edu/~15819/zedbook.pdf>. Accessed on: 03 feb. 2023.

YE, K.; FOSTER, S.; WOODCOCK, J. Formally Verified Animation for RoboChart using Interaction Trees. *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022*, v. 69, 10 2022.