

RoboWorld Reference Manual

James Baxter Gustavo Carvalho
Ana Cavalcanti
Francisco Rodrigues Júnior

WWW.CS.YORK.AC.UK/ROBOSTAR

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Contents

1	Introduction	9
2	RoboStar framework	13
3	RoboWorld: overview and metamodel	17
3.1	Document structure: overview	17
3.1.1	Assumptions	17
3.1.2	Mappings	20
3.2	Metamodel	24
3.3	Well-formedness conditions	28
4	RoboWorld: realisation in the Grammatical Framework	31
4.1	RoboWorld in GF: overview	31
4.2	Background on the Grammatical Framework	33
4.3	RoboWorld lexicon	36
4.4	Building blocks: ItemPhrases	37
4.5	Writing structures: RWClauses	38
4.6	Tenses and polarities: RWSentences	39
4.7	Writing assumptions and mapping definitions	40
4.7.1	Assumptions	40
4.7.2	Mapping definitions	40
5	Intermediate representation	43
5.1	Metamodel and well-formedness conditions	43

5.2	Generation from RoboWorld documents	49
5.2.1	Generating the intermediate representation	49
5.2.2	Annotating the intermediate representation	53
6	RoboWorld: semantics	55
6.1	Formal semantics: overview	55
6.2	Semantics generation: transformation rules	65
7	RoboWorld in RoboTool: authoring RoboWorld documents	79
8	Conclusions	83
A	Complete RoboWorld Grammar	85
A.1	RoboWorld.gf	85
A.2	RoboWorldEng.gf	93
A.3	RoboWorldLexicon.gf	113
A.4	RoboWorldLexiconEng.gf	117
B	Complete RoboWorld MetaModel	123
C	Complete RoboWorld IR Metamodel	129
C.1	RoboWorldIR.ecore	129
C.2	ExpressionIR.ecore	133
D	Firefighter <i>CyPhyCircus</i> Semantics	139
D.1	Types	139
D.2	Channels	140
D.2.1	Software channels	140
D.2.2	Input event triggered channels	141
D.2.3	Output clock reset channels	141
D.2.4	Variable get/set channels	141
D.3	Global Constants	142
D.4	Environment	143
D.4.1	Environment State	143
D.4.2	Robot Movement	145
D.4.3	Collision Detection	146
D.4.4	Communication Actions that occur on the time step	146
D.4.5	Input Event Buffers	147
D.4.6	Output Event Buffers	148
D.4.7	Environment main action	149
D.5	Mapping	149
D.5.1	spray Output Event Mapping	150
D.5.2	takeoff Operation Mapping	152
D.5.3	goToBuilding Operation Mapping	153

D.5.4	goHome Operation Mapping	153
D.5.5	searchFire Operation Mapping	153
D.6	Composition	156
	Credits	157
	Bibliography	159
	Index of Semantic Rules	163

List of rules

1	Map RWDocument	50
2	Map ArenaAssumptions	50
3	Update Arena	51
4	Find dimensionality information	51
5	Map InputEventMappings	52
6	Map InputEventMapping	52
7	Map InputSometimes	53
8	Annotate Constraint	53
9	Annotate Statement	54
10	Semantics of RoboWorld Documents	66
11	Type Definitions	66
12	Type Definitions for the Arena	67
13	Type Definitions Needed for Attributes	68
14	Type Definitions for the Robot	68
15	Type Definitions for an ElementDefinition Robot	69
16	Type Definitions for an Element	69
17	Type Definitions for an ElementDefinition Element	70
18	Fields for Size Parameters of an Element	70
19	Definition of <i>locations</i> Set for an Element	71
20	Input event trigger channel definitions	71
21	Output event happened channel definitions	71
22	Variable get/set channel definitions	72
23	Robot variable get/set channel definitions	72
24	ElementDefinition variable get/set channel definitions	72
25	Component variable get/set channel definitions	73
26	Global assumptions about elements	73

27	Definition of global elements	74
28	Global Assumptions on the Arena	74
29	Mapping Process	74
30	Composition of Output Event Mappings	74
31	Composition of Operation Mappings	75
32	Environment Process	75
33	Environment Process State	76
34	Input Trigger semantics	77
35	Operation Mapping Semantics	78

1. Introduction

Recent advances in Engineering and Artificial Intelligence promise to have a transformative impact on society, as robots become ubiquitous in homes, offices, and public spaces, providing services to facilitate and enrich our lives. Development of software for robots operating in these complex environments, however, is a challenge. Roboticists often have in mind restrictions on the environment that must be satisfied for their robots to operate well: they make assumptions about temperature, wind, layout of rooms, weight of the robot, and so on, for example. Rarely, however, these operational 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.

Model-driven, as opposed to code-centric, software engineering has been widely advocated for robotics [6]. Many domain-specific languages support modelling and automated generation of code for simulation and deployment. A few have a formal semantics. The RoboStar framework¹ [5] is distinctive in that its design and simulation notations have semantics that can be automatically generated. It is provided using a state-rich hybrid version of a process algebra for refinement [22] cast in Hoare and He's Unifying Theories of Programming (UTP) [13] and formalised in Isabelle [21].

In using models for generation of tests and for verification by proof, we 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 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

¹robostar.cs.york.ac.uk

account of operational requirements is, therefore, an important design artefact.

In this paper, we present and formalise RoboWorld, a controlled natural language for documenting operational requirements of a robotic system for use in simulation, test generation, and proof. RoboWorld documents complement platform-independent design models by describing operational requirements as assumptions about the environment. The RoboWorld requirements cover aspects of the arena (that is, area) in which the robot is expected to work and of the robotic platform.

Modelling the environment of a service robot is not feasible due to its highly complex, often dynamic and unpredictable, nature. On the other hand, it is feasible to record assumptions about the environment [1], including the robotic platform. RoboWorld supports this practice by providing an accessible and extensible English-based notation for roboticists.

In current practice, the starting point to identify operational requirements is the development and use of simulation scenarios, if not of the actual program and platform. By recording requirements in a RoboWorld document, however, we can then verify whether these assumptions are satisfied by a simulation (model or code). In addition, we can generate or select tests that are guaranteed to be meaningful. Finally, we can use the assumptions to prove properties of the system.

Generally speaking, natural language processing techniques can be statistical or symbolic [10]. Statistical approaches assume that a large dataset of (raw) text is available, from which techniques such as machine learning extract processing rules by creating statistical models. Differently, symbolic approaches typically rely on grammars to define rules for analysing and producing valid text; these rules define a Controlled Natural Language (CNL).

While statistical approaches are more general, since they can process unrestricted text, inferring the correct interpretation of the text is a challenge due to huge variety of writing styles. The control imposed by symbolic approaches can make this inference process easier, since we restrict ourselves to a controlled subset of styles. However, a challenge when defining a CNL 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, yet controlled.

Tool support for RoboWorld is provided by RoboTool². It includes facilities for (graphical) modelling, validation, and automatic generation of mathematical models for existing RoboStar notations and now also RoboWorld. It also automates test and simulation generation. Proof automation relies on integration with model checkers [19, 23] and Isabelle/UTP [21].

²robostar.cs.york.ac.uk/robotool/

In [3], we have provided an overview of the RoboWorld syntax, semantics, and tool support using a couple of examples. Here, we provide a comprehensive definition of the language: metamodel, grammar, well-formedness conditions, and formal semantics, and the RoboTool mechanisation.

In terms of the semantics, we define an intermediate representation that ensures changes to the concrete syntax do not affect directly the definition of the semantics. The intermediate representation provides a syntax-independent basis to define the semantics and implement tools for RoboWorld. A set of rules defines how an intermediate representation is generated for a RoboWorld document. A second set of rules defines a mathematical semantics for RoboWorld documents by specifying functions that map the intermediate representation to *CyPhyCircus* processes [18, 22]. *CyPhyCircus* is the hybrid state-rich process algebra used in the RoboStar framework. Mechanisation of the two sets of rules allows automatic generation of *CyPhyCircus* models using RoboTool.

In the next section, we give an overview of RoboChart [7], the RoboStar notation for software modelling, to illustrate how RoboWorld can complement design models, and influence simulation, testing, and proof. RoboWorld, however, is not tied to RoboStar notations, and can be used to record and formalise operational requirements whether a RoboStar model is available or not.

Section 3 specifies the structure of RoboWorld documents: their abstract syntax via a metamodel, with associated well-formedness conditions. As an example, we present a RoboWorld document that captures the operational requirement of a firefighting UAV. The concrete grammar is defined in Section 4 using the facilities of the Grammatical Framework, a special-purpose functional programming language for developing and implementing controlled natural languages [10]. In Section 5 we describe our intermediate representation for RoboWorld documents. Section 6 formalises the semantics. RoboTool support for RoboWorld is the object of Section 7. We conclude and discuss future work in Section 8.

2. RoboStar framework

At the design level, a RoboWorld document complements (platform-independent) models of control software. In the RoboStar framework, these are written using RoboChart, a timed state-machine based notation with a specialised component model. Platform independence is achieved in this context by writing models in terms of the services of the robotic platform required by the control software. Services are described by events, operations, and variables provided by the platform; these are abstractions for sensors and actuators, and associated embedded software.

RoboWorld documents can enrich a platform-independent software design by capturing how features of the environment affect and are affected by the behaviour described by that design. This is achieved by defining how elements of the environment affect or are affected by the values of the variables, occurrences of events, and calls to operations used in the software.

How the software or simulation is described in terms of its required services is irrelevant to the reader or writer of a RoboWorld document. To illustrate our ideas, however, we give a brief overview of the RoboChart notation. For that we use a simplified model of a firefighting UAV inspired on a challenge for an international robotics competition¹. Figure 2.1 shows the drone.

RoboChart is a diagrammatic modelling language based on UML state machines, but embedding a component model suitable for robotics and time primitives to capture budgets, timeouts, and deadlines. In defining a RoboChart model, a key element is the block that specifies the services of a robotic platform that are used by the control software. In Figure 2.2, the block named UAV inside the block SimpleFireFighter is the robotic-platform block for our example.

¹www.mbzirc.com/ - Challenge 3 in 2020.



- 1 RealSense D435i depth camera
& MLX90640 thermal camera
- 2 Nozzle attached to a two-axis gimbal
- 3 Arduino Nano for pump and gimbal control
- 4 1m Carbon fibre arm
- 5 3S LiPo Battery
- 6 10bar water pump.
- 7 Onboard computer, a Raspberry Pi 4
- 8 4L Water bag
- 9 DJI M600 UAV

Figure 2.1: Firefighting UAV

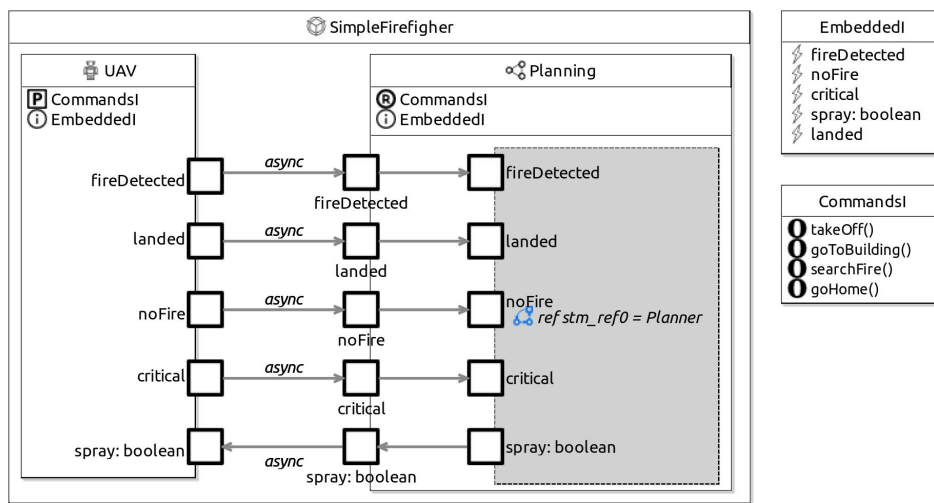


Figure 2.2: RoboChart module for a simplified firefighting UAV application

The model for the real firefighter drone defines 21 robotic-platform services. In our simpler version, we have just five events and four operations. They are declared in interface blocks called `EmbeddedI` and `CommandsI` on the right in Figure 2.2. The UAV block declares these interfaces, making their events and operations available for use by the software. Here, the software behaviour is defined by a single controller block called `Planning`. The block `SimpleFirefigher` is an example of a RoboChart module, which can be used to define a platform-independent model for the control software of a robotic system, using a robotic-platform block, and one or more controller blocks.

The services of UAV include abstractions for a camera and associated image analysis software in the form of events `fireDetected` and `noFire`. The event `critical` is an abstraction for a sensor that indicates that the level of the battery is too low. The event `spray` abstracts an actuator that turns on and off the water pump. The event `landed` represents flight-control sensors: IMU and GPS, for example. Finally, the operations of our platform abstract navigation facilities of the flight controller, which is able to follow trajectories to `takeOff()`, `goToBuilding()`, `searchFire()`, and `goHome()`.

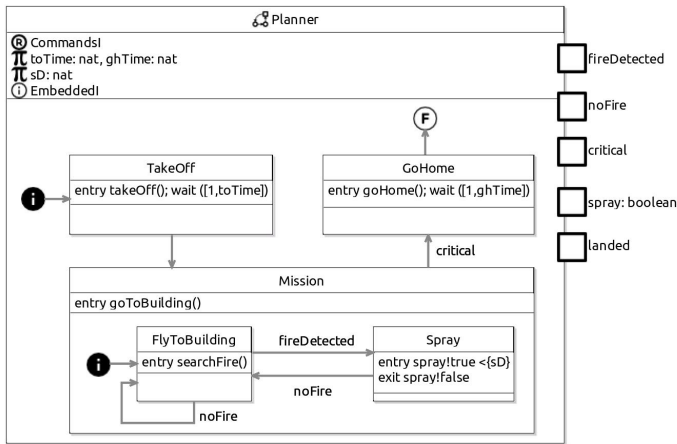


Figure 2.3: RoboChart state machine for our simplified firefighting UAV

The RoboWorld document that we present in the next section explains how all these services declared in UAV are related to elements of the drone environment. So, that RoboWorld document is associated with the RoboChart module SimpleFirefighter. These definitions are irrespective of how the services of UAV are used in the controller Planning. For completeness, however, we present in Figure 2.3 the RoboChart state machine Planner, which defines the behaviour of Planning. In general, the behaviour of a controller can be specified by a collection of parallel state machines. In the complete model of the firefighter, we have two controllers and nine machines.

In Planner, we have a state machine that is, by far and large, much like a UML machine. We note a few points, though. First, there is a context at the top that declares the required or local variables, events, and operations used in the definition of the state machines. In our example, the interface `CommandsI` is declared as required (R). This means that a controller that uses this machine needs to define these operations, or require them from the platform as it is the case here. The interface `EmbeddedI` is defined (i), so the machine uses its events to input or output from or to the controller and platform. Additionally, `Planner` declares three constants (π). They are used only in `Planner`, but are parameters of the module as a whole, since their values are left undefined. Here, these constants record time budgets and deadlines for the operations.

Second, in the actual machine defined in `Planner`, the initial junction (black circle marked with an i) leads to the state `TakeOff`, whose entry action (executed when the state is entered) calls the platform operation `takeoff()` and then pauses for between 1 and `toTime` time units. The pause is defined using a RoboChart time primitive `wait`. It is used to specify time budgets: here, an interval defining a range for the amount of time that might actually be needed for the drone to take off.

Finally, in the entry action of the state `Spray` (inside the composite state `Mission`), we have a deadline `sD` on the entry action `spray!true`. This ensures that, once the fire is detected, the robot starts the attempt to extinguish it no later than `sD` time units afterwards.

Like RoboWorld, RoboChart has a process-algebraic semantics based on CSP [11]. It uses

the discrete-time variant of CSP called *tock*-CSP, whose denotational semantics is given in [16]. The RoboChart semantics is compatible with the semantics we provide here for RoboWorld, using *CyPhyCircus* [18]. This is a hybrid process algebra that extends *Circus* [2], which itself combines CSP with Z [17] for modelling abstract data types and operations. With a RoboWorld document and its associated RoboChart model, we can reason about the robotic system as a whole.

We next describe the details of the RoboWorld language.

3. RoboWorld: overview and metamodel

In this section, we first give an overview of the structure of RoboWorld documents using the example of the firefighting drone (Section 3.1). Next, in Section 3.2, we present a metamodel for RoboWorld. Finally, Section 3.3 lists well-formedness conditions that must be satisfied by a valid RoboWorld document, and that provide guidance to designers.

3.1 Document structure: overview

In this section, we give an overview of the RoboWorld syntax using the RoboWorld document for the firefighting UAV, presented in Figures 3.1 and 3.2. As illustrated, a RoboWorld document includes assumptions and mappings. Assumptions declare and restrict elements of the environment: they are described in Section 3.1.1. The mappings define the services of an associated (RoboChart) design model using the elements defined in the assumptions. We give more details in Section 3.1.2.

3.1.1 Assumptions

The assumptions are divided into sections to distinguish assumptions about the arena, about the robot, and about (other) elements introduced in the assumptions about the arena. The first section, labelled ARENA ASSUMPTIONS, captures assumptions over the arena as a whole: its dimension, properties of the ground, if any, and, most importantly, presence of elements (obstacles, objects that may be carried, a home or target region, and so on) besides the robot. The elements may be entities that the robot may interact with or regions of the arena.

```

## ARENA ASSUMPTIONS ##
The arena is three-dimensional.

The width of the arena is 50.0 m.

The depth of the arena is 60.0 m.

The arena has a floor.
The gradient of the ground is 0.0.

The arena has one building.

The height of the arena is the height of the building plus at least 1.0 m.

The arena has fires.

The arena has a home region.

The speed of the wind is less than 8.0 m/s.

It is not raining.

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

Initially the robot is in the home region.

The robot has a tank of water.
The tank of water is either full or empty.

The robot has a searchPattern.
The searchPattern is a sequence of positions.

## ELEMENT ASSUMPTIONS ##
The building is a box.

The height of the building is not less than 6.0 m.
The height of the building is not greater than 20.0 m.

The width of the building is not less than 10.0 m.
The width of the building is not greater than 30.0 m.

The depth of the building is not less than 10.0 m.
The depth of the building is not greater than 40.0 m.

A fire can occur on the floor.
A fire can occur on the building from a height of 5.0 m to 18.0 m.

The width of the fires is 36.0 mm.
The height of the fires is 60.0 mm.
The depth of the fires is 0.0 mm.

The fires have a status.
The statuses of the fires are either burning or extinguished.

The home has an x-width of 1.0 m and a y-width of 1.0 m.
The home is on the ground.

```

Figure 3.1: Firefighter UAV RoboWorld assumptions

The assumptions in Figure 3.1 state that the arena is three-dimensional with a flat ground (gradient 0.0). The arena is not assumed to have a floor; for instance, for a drone, the existence of a floor may not be relevant. The arena has a floor if, and only if, it is explicitly said, as in Figure 3.1, or if the gradient of the ground is defined. So, in Figure 3.1, the declaration of the `floor` can be removed.

Two types of entities are declared in Figure 3.1: building and fire. The sentences that declare these entities indicate that there is a single building, but there may be none, one, or many fires.

There is also a region called home. The regions share the same dimensionality of the arena, unless we say otherwise. In addition, the arena and its regions are open, unless explicitly indicated to be closed. So, regions do not block movement, unless otherwise stated.

Another entity often declared is obstacle. For instance, the arena assumptions for a foraging

robot may declare obstacles as shown below. Entities are assumed to block movement.

Example 1

The arena has obstacles. □

In our example, we provide in separate sentences exact measurements for the width and depth of the arena, as described for the competition. These measurements can, however, be left unspecified, in which case the arena is finite, but the actual values of its dimensions are unbounded. For instance, in the example, the exact height of the arena is not specified. Another sentence provides a lower bound, based on the height of the building, which is an element previously declared.

Finally, in Figure 3.1 two sentences give properties related to the wind and rain. These are primitive concepts of RoboWorld. By default, the environment does not have any wind or rain.

Arena assumptions are optional. If not included, the implicit assumption is a three-dimensional arena, of finite, but unbounded size, without floor, and that contains just the robot.

ROBOT ASSUMPTIONS are compulsory. We need to define the assumptions about the shape of the robot. It can, however, be defined to be a point mass if the shape of the robot is not important as far as the assumptions we make about its interactions with the world are concerned. We can also define initial location, elements, and capabilities of the robotic platform. The ability to move is a feature of every robot; they all have a pose (position and orientation), velocity, and acceleration.

If the initial pose of the robot is not defined, the robot can start in any pose in the arena.

For the firefighting drone, we declare a tank of water as a robot element. After the introduction of such an element, we can also indicate relevant information that can be recorded about it; here, a separate sentence indicates that the tank of water can be full or empty. Another element of the robot is the searchPattern. This is information held by the robot, rather than a physical element. The declaration gives it type, namely, a sequence of positions.

Several other examples are available¹, and some take advantage of this facility to declare relevant elements of the robot. For instance, requirements for the foraging robot include the following.

Example 2

The robot may carry one object.

The robot has an odometer. □

In this case, elements called objects need to have been declared in the arena assumptions. Odometer is part of the RoboWorld vocabulary, and captures information related to the robot movement.

It is possible to write a detailed description of the robot shape entirely in English. This involves

¹robostar.cs.york.ac.uk

defining components of the robot, their shapes (boxes, spheres, cylinders, and so on), and their poses. If such a description becomes unwieldy, however, it may be better to use a (block) diagram.

In RoboStar, physical models for use in simulation can be specified using RoboSim [4, 9]. These models describe specific robotic platforms and scenarios for a simulation using specialised block diagrams and differential equations. In contrast, RoboWorld documents specify properties that must be satisfied by RoboSim models, called p-models, in the case of platform models, and s-models, in the case of scenario models. If, however, a detailed physical model for the robot or any other element of the arena is useful, a p-model component can be included.

In this paper, however, we focus on the facilities for descriptions in English. The use of diagrams in RoboWorld is not required, but is provided as an extra resource.

The `ELEMENT ASSUMPTIONS` describe properties of elements declared in the `ARENA ASSUMPTIONS`. We can constrain their shapes, dimensions, and locations, for example. These can be specific or underspecified. In our example, for instance, we define a range for the dimensions of the building, we define specific values for the dimensions of a fire, and we define that the home region is on the ground, but do not say specifically where on the ground.

In the competition set up, a fire was simulated by a heat plate with a hole for the water. We do not capture here some information that makes sense only for the environment especially set up for physical testing, such as the hole in the middle of the fire. We, however, provide size information. Here, we use millimetres, rather than metres. RoboWorld accepts all SI units and their prefixes.

3.1.2 Mappings

Up to four sections of a document contain mapping definitions: for `INPUT EVENTS`, `OUTPUT EVENTS`, `OPERATIONS`, and `VARIABLES`. These describe how the robotic-platform services of an associated (RoboChart) design model affect and are affected by the environment.

In Figure 3.2 we have mappings for four `INPUT EVENTS`: `fireDetected`, `noFire`, `critical`, and `landed`. The mappings determine conditions that characterise the scenarios in which the input events occur. In the conditions, we can refer to properties of the arena, of the robot, and of elements of the arena. In our example, we refer to a property `distance` related to the robot and fires in defining `fireDetected` and `noFire`, for instance. To define `landed` we refer to the position of the robot.

The event `critical` is characterised by time conditions, in relation to occurrences of an output event, namely, `spray`, and calls to the operation `takeOff`.

The mappings for `OUTPUT EVENTS` describe their effect on the environment when they occur. Similarly, the mappings for `OPERATIONS` describe their effect when they are called. For the foraging robot, if the `ROBOT ASSUMPTIONS` declare that the robot has an odometer, and we have an output event `resetDist` to abstract functionality related to the odometer, then the mapping for this event

```

## MAPPING OF INPUT EVENTS ##
When the distance from the robot to a fire is not greater than 0.5 m, the event fireDetected occurs.

When the distance from the robot to a fire is greater than 0.5 m, the event noFire occurs.

When the occurrence of the event spray was 3 minutes before or the occurrence of the operation takeOff was 20
minutes before, the event critical occurs.

When the z-position of the robot is 0.0, the event landed occurs.

## MAPPING OF OUTPUT EVENTS ##
When the event spray occurs, if the tank of water is full, then the effect is defined by a diagram where one
time unit is 1.0 s.

## MAPPING OF OPERATIONS ##
When the operation takeOff is called, the velocity of the robot is set to 1.0 m/s upwards.

When the operation goToBuilding is called, the velocity of the robot is set to 1.0 m/s towards the building.

When the operation goHome is called, the velocity of the robot is set to 1.0 m/s towards the home region.

The operation searchFire() is defined by a diagram where one time unit is 1.0 s.

```

Figure 3.2: Firefighter UAV RoboWorld mappings

can be as follows. As said, odometer is one of the sensors regarded as a primitive concept in RoboWorld.

Example 3

When the event resetDist occurs, the odometer is reset. □

For a drone, we may have an event land to abstract functionality of the autopilot. The mapping in this case can be as shown below, where we refer to the velocity of the robot.

Example 4

When the event land occurs, the velocity of the robot is set to 1.0 m/s downward. □

The effect of an output event or operation may be conditional. In the firefighter example, the effect of the output event `spray` is conditioned to the status of the tank of water being full. It changes the environment by extinguishing fires and changing the status of the tank of the robot to empty. This is defined by a state machine, shown in Figure 3.3.

As for the p-model block diagrams, state machines are provided as a resource to define mappings if their English description might be too complex. Typically, if the effect of an output or operation involves loops over a set of elements or takes time, using a state machine to define it may be simpler than giving an English description.

The notation to describe state machines is similar to that of RoboChart. In a RoboWorld machine, however, we can use events to `set` and `get` the position, orientation, velocity, and acceleration of the robot, and other declared properties of elements of the arena and robot. This is in addition to the event defined by the mapping. We can also require variables (and constants).

The state machine for an output event is named after that event. In our example, the machine is `spraymapping()`. Figure 3.3 shows on the right the declaration of `spray`. On the left, we declare

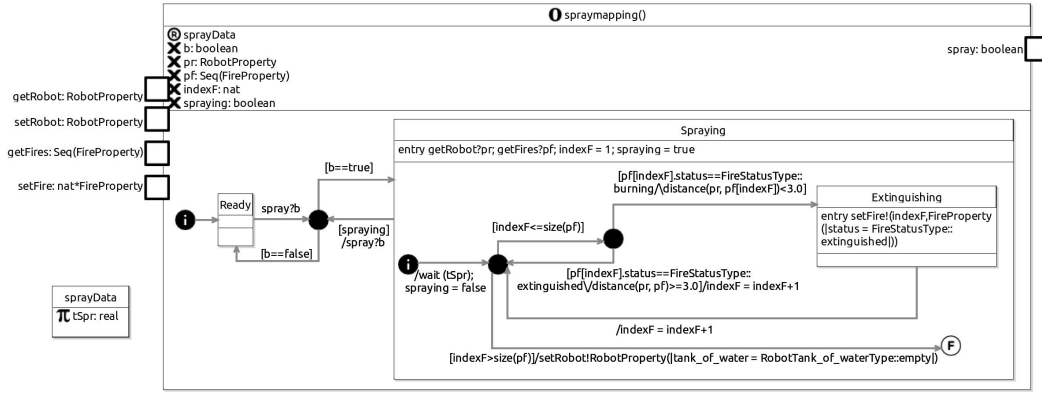


Figure 3.3: Firefighter UAV RoboWorld - mapping for spray

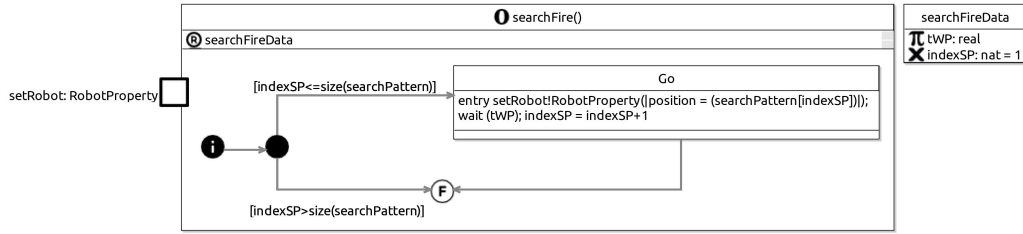


Figure 3.4: Firefighter UAV RoboWorld - mapping for searchFire()

the events to `set` and `get` properties of the robot and of the fires. Their type declarations uses record types `RobotProperty` and `FireProperty` that reflect implicit attributes related to pose, for example, and the declarations of components of the robot and of a fire. For instance, for the robot, a field `tank_of_water` has an enumeration type `RobotTank_of_waterType` containing values `empty` and `full`. Similarly, `FireProperty` has a field `status` whose type has values `burning` and `extinguished`.

If the arena may have several instances of an element, the corresponding `set` and `get` channels communicate sequences of the record type that characterises the element. For instance, in Figure 3.3, the type of `setFires` is a sequence of `FireProperty`. In addition, we have channels to `get` and `set` a particular element in such a sequence. The type of `setFire` in Figure 3.3 is a pair (constructor `*`), whose first element is an index in the sequence of fires, and whose second element is a `FireProperty`.

As mentioned, the machine defines the behaviour following the occurrence of the output event. In the example in Figure 3.3, the machine is at first in a state `Ready`, waiting for the occurrence of a `spray` event. In accordance with its declaration, the event `spray` takes a `boolean` `b` as input (`spray?b`). This is an output produced by the software (see Figures 2.2 and 2.3), and so an input of the mapping that defines its effect on the environment. The variable `b` is declared locally. If `b` is `true`, then the machine moves to the composite state `Spraying`. Otherwise, it stays in `Ready`.

In the entry action of `Spraying`, the events `getRobot` and `getFires` are used to record the properties of the robot and a sequence of properties of fires in local variables `pr` and `pf`. Finally, a local index `indexF` is initialised to 1, and a local boolean variable `spraying`, which records whether the fires close and in front of the robot still require spraying, is set to true.

Whether more spraying is required is defined by the time that the robot has been spraying. A required interface `sprayData` declares a constant `tSpr` defining the amount of time to spray. In `Spraying`, in the transition from its initial junction, after `tSpr` time units have passed (`wait(tSpr)`), `spraying` is set to false. At this point, the behaviour of the machine of `Spraying` cannot be interrupted, as it records the effect of the spraying. So, the transition out of `Spraying` that occurs if the event `spray` occurs is disabled by the guard `spraying == false`.

The state machine in `Spraying` defines a loop, where the status of each fire identified by `indexF`, that is `pf(indexF)`, is checked. If it is burning and its distance to the robot is less than 3.0 m, then the machine moves to a state `Extinguishing`. The function `distance`, whose definition we omit, uses the pose of the robot recorded in `pr`, and of the fire, in `pf(indexF)` to determine the distance between them. (A fire that is not in front of the robot is considered very far by this function.)

The entry action of `Extinguishing` uses `setFire` to update the status of the fire identified by `indexF` to extinguished. A transition out of `Extinguishing` increments `indexF` and leads to the decision junction for the loop. If `pf(indexF)` is extinguished or too far from the robot, then the only action is the `indexF` increment. When all fires have been considered, the tank of water is updated to empty.

The mappings can also use intrinsic properties of the robot, such as its velocity and acceleration. In our example, the operations `takeOff`, `goToBuilding`, and `goHome` all affect the robot's velocity.

The mapping for the operation `searchFire()` is specified by the diagram shown in Figure 3.4.

The state machine for `searchFire()` indicates that the robot moves to each waypoint recorded in `searchPattern` in sequence. This is achieved by setting the robot's position, using the event `setPos`, to the next waypoint in `searchPattern` (`setPos!(searchPattern(indexSP))`). The value of `indexSP` is initialised to 1 upon its declaration in the interface `searchData`. A less strong abstraction would set the robot's velocity and acceleration. Since, however, in this example the focus is on the mission of the drone, namely, fighting fires, rather than on its mobility, this abstraction is useful.

In any case, the amount of time units defined by the constant `tWP` is required to pass before `indexSP` is incremented and the robot moves to the next waypoint. The guard of the self-transition of the state `Go`, that is, `sinceEntry(Go) > tWP`, holds after `tWP` time units since the state `Go` is entered. At that moment, the transition is enabled and immediately taken. Once all waypoints

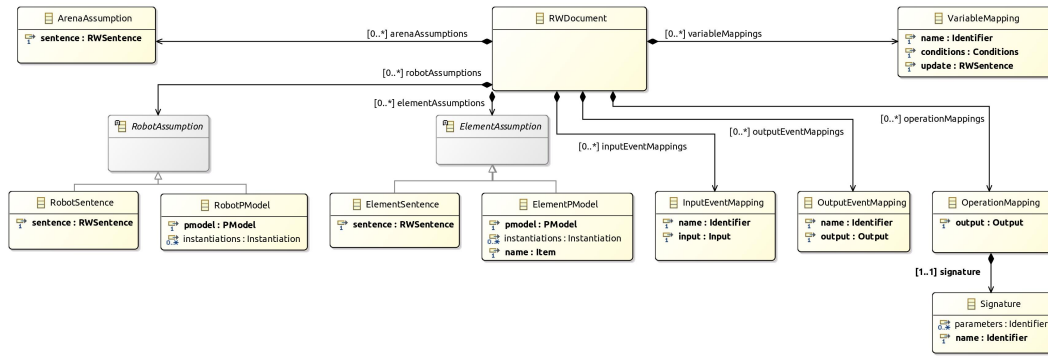


Figure 3.5: RoboWorld metamodel: top classes

have been visited ($\text{indexSP} > \text{size}(\text{searchPattern}(\text{indexSP}))$) then the operation `searchFire()` finishes.

Before finishing, `searchFire()` may be interrupted (see Figure 2.3), in which case the robot starts spraying until the fire is no longer in sight. When the operation `searchFire()` is called again, the drone continues to the next waypoint. (In the real drone, an extra operation stops the drone before spraying. We omit it here as it does not add to the objective of illustrating the use of RoboWorld.)

The final section contains the MAPPING OF VARIABLES of the robotic platform. It is empty for the firefighting UAV, since there are no robotic platform variables in its model. Variables can be used as inputs to the software, and so their definitions are similar to those for input events.

We now specify the metamodel and well-formedness conditions for RoboWorld documents.

3.2 Metamodel

Figure 3.5 presents a diagram including the top-level classes of the RoboWorld metamodel. A RoboWorld document is an element of the class `RWDocument`. It is formed by a sequence of zero or more objects of the classes for each of the assumption and mapping groups.

The assumptions and mappings are defined in terms of sentences, defined by the class `RWSentence` representing the forms of sentences allowed in RoboWorld, and `Conditions`, which are `RWSentences` prefixed by a subjunction. `RWSentences` are specified in terms of categories the English language: Noun, Adjective, Adverb, and so on.

An `ArenaAssumption` is defined by a sentence. As said, a `RobotAssumption` can be defined by a sentence, as represented by the subclass `RobotSentence`, or by a p-model, represented by the class `RobotPModel`. The attribute `pmodel` of `RobotPModel` has type `PModel`. This is a class in the RoboSim metamodel² that represents a specialised form of block

²robostar.cs.york.ac.uk/publications/techreports/reports/physmod-reference.pdf

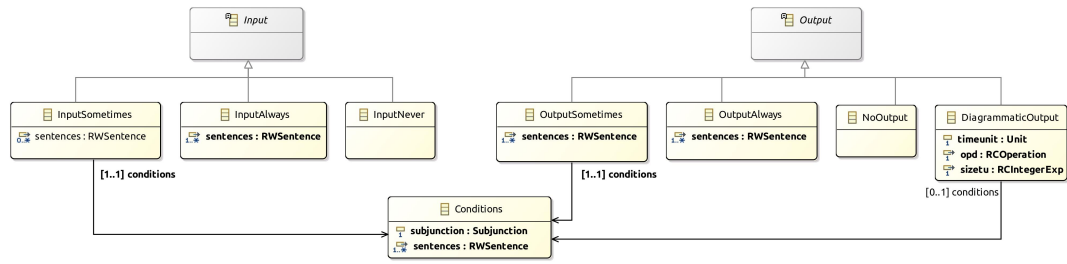


Figure 3.6: RoboWorld metamodel: inputs and outputs

diagrams that can be used to describe the links, joints, sensors, and actuators of a robot.

Here, we do not discuss block diagrams any further, but note that a PModel may have some parameters (representing sizes of rigid bodies, for example) which may be instantiated when used in a RoboWorld document. The class Instantiation, used to give type to the attribute instantiations of RobotPModel, is also in the RoboSim metamodel. Like the semantics of RoboWorld presented here, the semantics of a RoboSim PModel is also given in *CyPhyCircus*, so it integrates well.

Like a RobotAssumption, an ElementAssumption can be a sentence (ElementSentence) or a p-model (ElementPModel). In this case, however, our metamodel indicates that the name of the element is an Item as defined in Figure 3.7: the block diagram is for the element declared in the arena assumptions whose name is that Item. In the case of a p-model for the robot, the name is just robot.

The mappings all have a name, except for an OperationMapping, which has a signature, including a name and a list of parameters. The types of the parameters do not need to be defined, since they are already declared in the associated RoboChart model.

The name of an InputEventMapping identifies the input event being defined. In addition, it has information given by an input that characterises when that event can take place and, if relevant, that defines the values input. In Figure 3.6 we define the class Input as an abstract class with three concrete subclasses: InputSometimes, InputAlways, and InputNever.

In Figure 3.2, the input mappings for fireDetected, noFire, and landed all represent an element of InputSometimes, with an attribute conditions. In each case, the subjunction in conditions is “when”, and sentences, such as “the distance from the robot to a fire is not greater than 0.5 m”, define when the event occurs. In these examples, however, the InputSometimes instance itself has no sentences. We provide below more examples, where we distinguish in bold face the keywords of RoboWorld. In *italic*, we distinguish the names of the events being defined.

In the example below, in an input mapping for an event with name *angularSpeed*, we use an element of InputAlways indicated by “**is always available**”. We can also write “is always enabled”, “can always happen”, and so on. The concrete syntax identifies the possibilities (see

Section 4).

Example 5 The event *angularSpeed* is always available and it communicates the angular velocity of the robot. □

In this example, the value of sentences in `InputAlways` is the `RWSentence` “it communicates the angular velocity of the robot” introduced by the “and”. We assume that *angularSpeed* is declared in the RoboChart robotic platform to have type `real`, so we use an `RWSentence` to define the value communicated by the input: the angular velocity of the robot, which is a pre-defined property.

The keyword “and” is a separator used when we have a definition for sentences to follow. Use of an `RWSentence` is valid only when the event has a type, and so communicates values. If an event has a type, but no `RWSentence` is used to define the input value, that value is unconstrained.

In the next example of an `InputMapping` for an event *transferred*, the `Input` is an instance of `InputNever` as indicated by “never happens”. In this case, the input event never takes place, and so we do not need to include an `RWSentence` to characterise input values.

Example 6 The event *transferred* never happens. □

The `InputNever` instances are useful for abstraction. An example of where the mapping in Example 6 is useful is provided by one of our case studies³: a robot from a swarm that can transfer objects to another robot. A sensor tells when the transfer has taken place. In the initial simulation we have targetted, there is a single robot, so this part of the functionality is left out.

The output of an `OutputEventMapping` or of an `OperationMapping` can be defined in one of two ways: in English or diagrammatically (see Figure 3.6). It can be described in English using, optionally, Conditions, and some `RWSentences`. The concrete subclasses of `Output` called `OutputSometimes`, `OutputAlways`, and `NoOutput` are similar to `InputSometimes`, `InputAlways`, and `InputNever`, but define `Outputs`. For instance, in Example 3, the `OutputMapping` is for an event *resetDist*, whose output is an instance of `OutputAlways`. There is no condition, but just an `RWSentence`.

An output, however, may be defined to have no effect, for the sake of abstraction. In this case, we use an instance of `NoOutput` as illustrated below.

Example 7 When the operation *Transfer()* is called, nothing happens. □

The use case here is the same as that for the Example 6. We use this mapping to block the operation *Transfer()* when simulating a single robot from a swarm.

An output defined in a mapping by a diagram for a state machine is an instance of `Diagram-`

³robostar.cs.york.ac.uk/

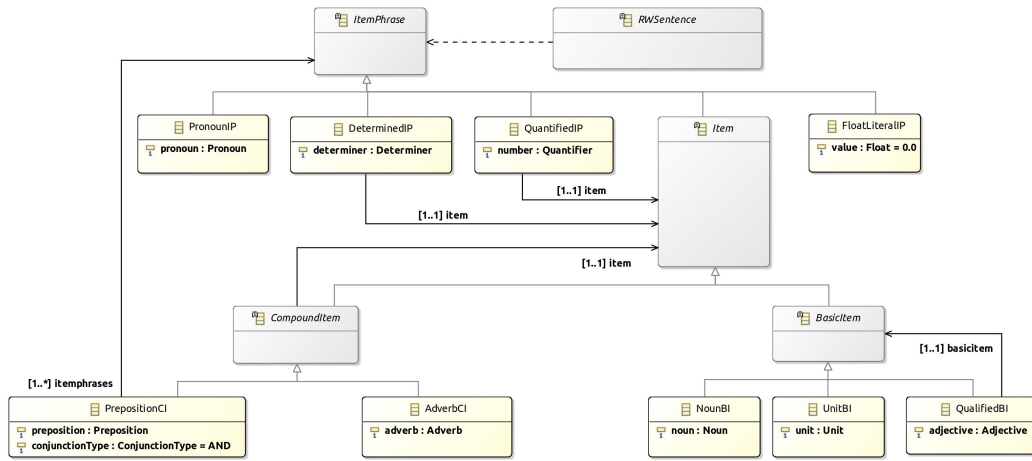


Figure 3.7: RoboWorld metamodel: sentences and item phrases

maticOutput. We refer to Figure 3.2, where we find the mapping for the event `spray`. Its effect is conditioned on the robot having a full tank of water. So, like in an instance of `OutputSometimes`, an attribute `conditions` records that restriction, namely, “if the tank of water is full”.

The state machine itself shown in Figure 3.4 is an instance of the class `RCOperation` from the RoboChart metamodel that defines the value of `opd` in the instance of `DiagrammaticOutput`. As illustrated, the diagrammatic definition is principally a state machine that defines the operation (or output) using the RoboChart notation. To support the definition of the state machine, we may need extra diagrams, like the interface used in Figure 3.2 to declare the variables required by the operation.

We recall that, as part of the mapping, we also define the value of the time unit. This is recorded in `sizetu`, whose type `RCIntegerExp` is a class of the RoboChart metamodel for integer expressions.

As mentioned before, the definitions of assumptions and mappings rely of `RWSentences`. Instances of `RWSentence` can represent a significant subset of the English sentences. Section 4 gives the details; the specification of `RWSentence` is not domain specific and is not further discussed in this section. As indicated in Figure 3.7, however, the definition of `RWSentence` depends on that of an `ItemPhrase`, which we present in Figure 3.7 and describe in what follows.

An `ItemPhrase` identifies an element of the environment; it is a restricted form of noun phrase, a concept of the English grammar. `ItemPhrase` has five direct subclasses. An `ItemPhrase` can be very concise, just a pronoun, represented by an instance of the class `PronounIP`. Its attribute `pronoun` is of a type `Pronoun`. We do not further discuss or define classes that correspond directly to categories of the English language, such as `Pronoun`, `Adverb`, and so on.

Another simple form of `ItemPhrase` is an instance of `FloatLiteralIP`, which is just a number. It has an attribute `value` of type `Float` whose default value is 0.0.

Other forms of `ItemPhrases` are constructed using a `Determiner`, in the case of the subclass `DeterminerIP`, or a `Quantifier`, in the case of `QuantifierIP`. The terms that can be determined or quantified are called `Items`, which can be basic or compound.

Example 8 A possible pronoun is “it”. In “the angular velocity”, we have a determined `ItemPhrase` created from the determiner “the” and the `BasicItem` “angular velocity”. Finally, in “1.0 rad/s upward”, we have a quantified `ItemPhrase` created from number 1 and `CompoundItem` “rad/s upward”. □

A `BasicItem` can be an instance of one of three classes: `NounBI`, representing a Noun, `UnitBI`, representing a unit, or a `QualifiedBI`, which qualifies a `basicitem` using an `Adjective`.

Example 9 Examples of `BasicItems` are “velocity”, “angular velocity”, and “m/s”. □

The notion of a `CompoundItem` allows the grouping of `Items` or `ItemPhrases` connected via a `Preposition` or modified by an `Adverb`, without creating ambiguity in the grammar. Every `CompoundItem` refers to an item. A `CompoundItem` can add a `preposition`, in the case of the subclass `PrepositionCI` of `CompoundItem`, to relate an item to one or more `ItemPhrases`. In the case of the subclass `AdverbCI`, the `CompoundItem` adds an `adverb`.

Example 10 In the `AdverbCI` “m/s upward”, we have the `BasicItem` “m/s” followed by the `Adverb` “upward”. In the `PrepositionCI` “distance from the robot to the nest”, we have the `BasicItem` “distance” followed by the `Preposition` “from” and an `ItemPhrase` “the robot to the nest”. The latter is a `DeterminedIP` that contains a `PrepositionCI` “robot to the nest”, itself another `PrepositionCI`. □

In Section 4, we describe a grammar that justifies the use of English sentences to describe instances of our metamodel. Not every instance of our metamodel, however, represents a valid RoboWorld document. So, we now present the well-formedness conditions that must be satisfied by an instance of the metamodel for a RoboWorld document.

3.3 Well-formedness conditions

Besides the expected restrictions of the English grammar, there are some general well-formedness conditions 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. These general restrictions are a form of well-typedness rules, and can be naturally enforced using the intermediate representation described in Section 5.

In this section, we concentrate on domain-specific well-formedness conditions related to the RoboWorld concepts, and the relationship between RoboWorld documents and RoboChart models,

Table 3.1: Well-formedness conditions of RoboWorld

RW1	The values “arenas” and “robots” are not valid for the attribute noun of a BasicBl.
RW2	The names in the InputEventMappings, OutputEventMappings, and VariableMappings must be precisely those of the input events, output events, and variables of the robotic platform in the associated RoboChart module.
RW3	The names in the signatures of the OperationMappings must be precisely those of the operations of the robotic platform in the associated RoboChart module.
RW4	The parameters in the signature of an OperationMapping must be precisely those (the same number, order and name) of the operation of the robotic platform in the associated RoboChart module.
RW5	The name of the pmodel in a RobotPModel is “robot”.
RW6	The name of the pmodel in an ElementPModel matches the value of its name.
RW7	In the input of an InputEventMapping for an event that is typeless in the associated RoboChart module, there are no sentences.
RW8	The sentences that define a DiagrammaticOutput must define a unit of time.
RW9	If the name of an OutputEventMapping is n , and its output is a DiagrammaticOutput, then the name of the RCOperation in opd is n mapping.
RW10	If the name of an OutputEventMapping is that of an event that has a type T in the associated RoboChart module, and the output of the OutputEventMapping is a DiagrammaticOutput, then the signature of its RCOperation in opd has a parameter of type T .
RW11	The signature of an OperationMapping whose output is a DiagrammaticOutput matches the signature of the RCOperation in opd .

if applicable (since RoboWorld can be used in conjunction with other design notations or even on its own). The conditions are presented in Table 3.1. In the next sections, we present additional well-formedness conditions. In Section 4, we present restrictions related to the vocabulary used in RWSentences. In Section 5, we present restrictions related to pre-defined terms (such as “linear velocity of the robot”) and to a form of well-typedness and scope of expressions (such as references to position should be consistent with the dimensionality of entities and regions).

If the RoboWorld document includes diagrams, for p-models or state machines, then they must also satisfy the well-formedness conditions defined in RoboSim and RoboChart [8, 9].

Here, RW1 is a well-formedness condition that indicates that presently RoboWorld considers single-robot applications, involving a single arena. Dealing with multiple robots requires little or no further work in terms of the grammar (see the next section) or intermediate representation (see Section 5). On the semantics, the impact is more significant. As for the restriction to a single arena, it is of little consequence, given that an arena can have several regions.

RW2-4 are concerned with the association between a RoboWorld document and a RoboChart module. In short, as indicated already, the mappings in the RoboWorld document must be for exactly the platform services defined in the corresponding RoboChart model. It is those services that define how the robot can perceive and affect the environment.

The name of a p-model used in a RoboWorld document must be consistent with the name used in the document. It is either just “robot” in the case of a p-model for the robot (RW5), or the name of the element being described by the p-model (RW6).

We recall that the sentences of an Input are used to define the values sent to the software based on the environment elements and their statuses. So, RW7 ensures that these sentences are present only if the input does require a value: it has a type.

The remaining RW8-11 ensure compatibility between the RoboWorld document and any state-machine diagrams to which it might refer. RW8 ensures that the RoboWorld document defines the value of the time unit. RW9-10 ensure that the name used in the RoboWorld document is that in the diagram, but we note that the diagram for an event n , such as `spray`, is supposed to be `nmapping` (`spraymapping`, in our example), to avoid conflict with the name of the event. RW11 ensures that, for operation mappings, the whole signature, not only the name, matches.

4. RoboWorld: realisation in the Grammatical Fram

As previously mentioned, the concrete syntax of RoboWorld is defined using the Grammatical Framework (GF) [10]. Along with the Resource Grammar Library (RGL), it provides native 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.

In the following sections, we detail how the RoboWorld metamodel is realised by grammars in GF. In Section 4.1, we present an overview of our approach. Before getting into details, in Section 4.2, we present background material on GF and RGL. In Section 4.3, we present the lexicon of RoboWorld and explain how it can be extended. The basic building blocks of a RoboWorld sentence are `ItemPhrases`, which are discussed in Section 4.4. Afterwards, we describe how sentences can be written, considering different writing structures (Section 4.5), tenses, and polarities (Section 4.6). Finally, we address the writing of assumptions and mapping definitions (Section 4.7).

4.1 RoboWorld in GF: overview

In GF, we have a notion of module, which may describe an abstract or concrete grammar, but also helper functions. Modules with helper functions are called resource modules. Abstract and concrete grammars can extend other abstract and concrete grammars, and concrete grammars implement abstract ones. Additionally, resource modules can be opened, that is, imported, by other modules.

Figure 4.1 shows the structure of our realisation of the RoboWorld metamodel in GF, indicating

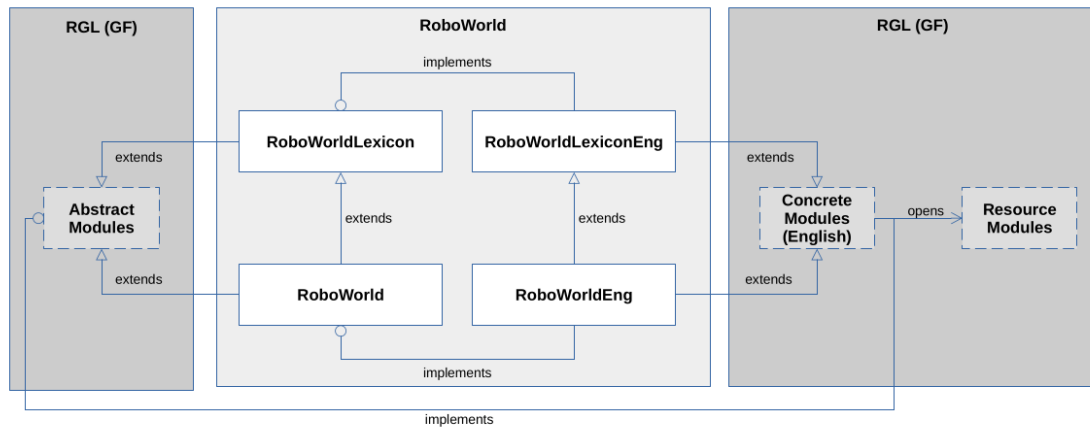


Figure 4.1: Architecture of RoboWorld realisation in GF

also how RoboWorld GF modules relate to the RGL of GF. In this figure, a module is represented as a box, whereas (to be more succinct) a collection of RGL modules is depicted as a dashed box. The RoboWorld metamodel presented in the previous section is realised by the abstract grammar called RoboWorld, which encodes the previously discussed structure. The concrete grammar RoboWorldEng describes how sentences in English correspond to elements of the metamodel.

In Figure 4.1, the collections of RGL modules used in our realisation of RoboWorld are shown on the left and on the right. RGL is concerned with morphology and syntax rules of languages. The RGL abstract grammars that we use, shown on the left in Figure 4.1, cover terms such as noun phrases and clauses, for instance, which are common to many languages. On the right, Figure 4.1 shows RGL modules that implement the abstract modules in the English language.

In the middle box in Figure 4.1, we show the grammars that we have defined specifically for RoboWorld. As indicated above, 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 and RoboWorldLexiconEng define the RoboWorld lexicon, that is, vocabulary. All these grammars use RGL grammars to cater for general concepts. They can be found in Appendix A.

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 separation between abstract and concrete grammars, along with the support provided by RGL, allows us to provide concrete implementations for RoboWorld considering other languages (and their particularities), such as Portuguese, French, and others. As said before, RGL takes into account more than 35 different languages. Here, we restrict ourselves to the English language.

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.

Before presenting the details of the grammars, we provide next an overview of GF.

4.2 Background on the Grammatical Framework

In GF, grammars are normally defined using functions to cater for context-sensitive languages. 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`.

Example 11 The following clauses are valid in ToyRoboWorld: “the robot has a wheel”, “the robot has wheels”, “the robots have wheels”. □

In Listing 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 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 and shown in Listing 2, defines how to implement the aforementioned abstract concepts in English, covering expected spellings and grammatical rules. 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).

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

```

1  abstract ToyRoboWorld = {
2    flags startcat = Clause ;
3    -----
4    cat -- categories
5      Determiner ; Noun ; Verb ;
6      NounPhrase ; VerbPhrase ; Clause ;
7    -----
8    fun -- lexicon
9      a_SgDeterminer : Determiner ; a_PlDeterminer : Determiner ;
10     the_SgDeterminer : Determiner ; the_PlDeterminer : Determiner ;
11     robot_Noun : Noun ; wheel_Noun : Noun ; have_Verb : Verb ;
12   -----
13   fun -- functions
14     mkNounPhrase : Determiner -> Noun -> NounPhrase ;
15     mkVerbPhrase : Verb -> NounPhrase -> VerbPhrase ;
16     mkClause : NounPhrase -> VerbPhrase -> Clause ;
17 }

```

Listing 1: Abstract 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 }

```

Listing 2: Concrete grammar of ToyRoboWorld

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).

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.

Example 12 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 Listing 2), and “wheels” is the inflection form associated with number `P1` (see Line 21 in Listing 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.

The Resource Grammar Library

As mentioned, RGL is the standard GF library; it covers a morphological and grammatical structure that is far from trivial, catering currently for 38 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.

```

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 }

```

Listing 3: Excerpts of the RoboWorld lexicon

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.

4.3 RoboWorld lexicon

Listing 3 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 (`C1`), for example, among many others. `CatEng` is its implementation for English. On Line 1 of Listing 3 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` (Line 11) and opens resource modules (for instance, `MorphoEng` and `IrregEng`) to deal with morphology rules and irregular inflections (Line 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.

Table 4.1: Well-formedness conditions of the dictionary

D1	RoboChart keywords must not be included in the dictionary.
D2	The identifiers used in RoboChart to denote the name of variables and constants must be in the dictionary, both as nouns and adjectives, and with inflection form IRREG.
D3	The identifiers used in RoboChart to denote the name of input and output events and of operations must be in the dictionary as nouns and with inflection form IRREG.

```

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  ...

```

Listing 4: Excerpts of the RoboWorld grammar: BasicItem and CompoundItem

It is possible to extend the RoboWorld lexicon to cover application-specific vocabulary. Hereafter, 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 (see Section 7).

When enriching the dictionary, the well-formedness conditions in Table 4.1 need to be observed. They ensure that RoboChart keywords are not used for any other purpose (D1), and the names of the robotic platform services are in the dictionary (D2 and D3), and therefore can be used in sentences. These words only need to be used in the singular form, so IRREG is to be used as their plural inflection to mark that they do not have a plural form. Identifiers that represent values (that is, the names of variables and constants) may also be used as an adjective (D2). For example, in “the linear velocity of the robot is set to *lv* m/s”, *lv* plays the role of an adjective.

4.4 Building blocks: ItemPhrases

Generally speaking, sentences in RoboWorld relate ItemPhrases by means of verbs. The realisation of ItemPhrase in GF closely mimics the metamodel presented in Figure 3.7. 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 reflect the RoboWorld metamodel and identify the expected forms of common nouns and noun phrases. For instance, in Listing 4, 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.

As said before, we use RGL to make RoboWorld more flexible and expressive. For example, according to the metamodel, an AdverbCI is a CompoundItem that modifies an Item by an adverb (see Figure 3.7). In the GF-realisation, we expect both adverbs (Adv) and adjectives (A) – see

```

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

```

Listing 5: Linearisation of `mkCompoundItem_AdverbCI_from_adjective`

Listing 4, Lines 6 and 7. In the second case, we use an RGL function to create an adverb from a given adjective (see Listing 5). In the linearisation of `mkCompoundItem_AdverbCI_from_adjective` there, 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 initially” used, for instance, in “The home region has 5 objects initially”.

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`.

4.5 Writing structures: `RWClauses`

RoboWorld clauses (defined by the category `RWClause`) are used to define `RWSentences`; they are instances of RGL clauses (`CatEng.Cl`), and define the writing structures supported in RoboWorld.

There are 12 forms of `RWClause`, each defined by a `mK` function. An `RWClause` can be written in the active voice (using functions whose names start with `mkRWClause_ActiveVoice_`) or in the passive voice (using `mkRWClause_PassiveVoice_` functions).

In the active voice, `mkRWClause_ActiveVoice_TransitiveVerb_ItemPhrase` is used to create `RWClauses` using transitive verbs. There is also support for modal and progressive verbs in the active voice (`mkRWClause_ActiveVoice_Modal_` and `mkRWClause_ActiveVoice_Progressive_` functions). The `mkRWClause_ActiveVoice_ToBe_` functions give a special treatment to clauses written using the verb “to be”. In the passive voice, we can use intransitive and transitive verbs. The latter expects a preposition followed by an `ItemPhrase`.

The linearisation of the aforementioned functions use RGL functions to ensure agreement between elements. In Listing 6 we give an example linearisation, along with an example `RWClause` of the form considered. First, a verb phrase (VP) named `progressive` is declared. The function `mkVP` creates a verb phrase from the text embedded in the provided verb (`lin V2 v2`) and the second `ItemPhrase` (`itemPhrase2`). A type annotation (`< ... : V2>`) is applied to `lin V2 v2` to ensure the text is cast to the type `V2` (since verbs can have several types). Afterwards, the RGL function `progressiveVP` transforms this verb phrase, taking into account the progressive

```

1  -- the robot is carrying an object
2  mkRWClause_ActiveVoice_Progressive_TransitiveVerb_ItemPhrase
3  itemPhrase1 v2 itemPhrase2 =
4      let progressive : VP =
5          progressiveVP (mkVP <(lin V2 v2) : V2> itemPhrase2) ;
6      in mkC1 itemPhrase1 progressive ;

```

Listing 6: Linearisation of mkRWClause_ActiveVoice_Progressive_TransitiveVerb_ItemPhrase

```

1  ...
2  mkRWSentence_PresentTense_PositivePolarity : RWClause -> RWSentence ;
3  mkRWSentence_PresentTense_NegativePolarity : RWClause -> RWSentence ;
4  mkRWSentence_PastTense_PositivePolarity : RWClause -> RWSentence ;
5  mkRWSentence_PastTense_NegativePolarity : RWClause -> RWSentence ;
6  ...

```

Listing 7: Excerpts of the RoboWorld grammar: RWSentence

form of its verb, whose value is assigned to the local variable `progressive`. Finally, when creating the clause, the function `mkC1` inserts the copula (that is, the verb “to be”, in this case), ensuring number agreement.

Example 13 The following clause is not valid since there is no number agreement between the first `ItemPhrase` and the copula: “the robots is carrying an object”.

□

4.6 Tenses and polarities: RWSentences

RoboWorld sentences (`RWSentence`) are instances of RGL sentences (`CatEng.S`). Here, we deal with verb tenses (present and past) and polarity (positive and negative sentences) – see Listing 7. Since these possibilities apply to arbitrary `RWClause`s, the 12 different writing structures discussed in Section 4.5 are lifted to $12 \times 4 = 48$ different types of sentences supported by the RoboWorld language. Additionally, an arbitrary `RWSentence` can be further modified by prefixing an adverb (for instance, “initially, the robot is in the origin”), thus, there is support for $2 \times 48 = 96$ different writing structures. Moreover, if a single new structure for writing `RWClause`s is added to the language, the number of different types of sentences automatically increases by 8.

The transformations between tenses and polarities are entirely handled by RGL – see Listing 8. Given an arbitrary `RWClause` (`clause`), it suffices to call `mkS`, providing the arguments `pastTense` and `UncNeg`, to transform `clause` into the past tense and the negative polarity.

```

1  mkRWSentence_PastTense_NegativePolarity clause =
2      mkS pastTense UncNeg clause ;

```

Listing 8: Linearisation of mkRWSentence_PastTense_NegativePolarity

```

1   ...
2   param OutputType = OutputEvent | Operation ;
3   ...
4   oper output_always : OutputType -> Str -> RWSentences -> S =
5   \outputType, str, sentences ->
6     let adv : CatEng.Adv = (outputSentencePrefix_Adv ! outputType) str ;
7     in mkS <adv : Adv> <sentences : S> ;
8   ...
9   mkOutputEventMapping_OutputAlways eventName sentences =
10    output_always OutputEvent eventName.s (lin RWSentences sentences) ;
11  ...
12  mkOperationMapping_OutputAlways eventName sentences =
13    output_always Operation eventName.s (lin RWSentences sentences) ;
14  ...

```

Listing 9: Linearisation of (mkOutputEventMapping | mkOperationMapping)_OutputAlways

4.7 Writing assumptions and mapping definitions

The GF realisation of RoboWorld assumptions and mapping definitions closely relates to their metamodel given in Figures 3.5 and 3.6; it is almost a one-to-one relation (that is, with one function in GF to represent each type of assumption or mapping definition).

4.7.1 Assumptions

ArenaAssumptions, RobotAssumptions and ElementAssumptions are essentially RWSentences: any valid RWSentence is accepted. For a RobotPModel or ElementPModel, we need to use a restricted form of sentence that, for example, includes “is defined by a diagram”.

4.7.2 Mapping definitions

To promote reuse, mapping definitions are realised in GF with the aid of helper functions (see Listing 9). To distinguish them from other types of functions, they are called operations (oper) in GF. Specific restrictions on the use of recursion apply to operations.

As illustrated in Listing 9, with the use of operations, the definitions of OutputEventMapping and OperationMapping are almost the same. For the functions mkOutputEventMapping_OutputAlways and mkOperationMapping_OutputAlways, it suffices to provide the operation output_always with a different argument (OutputEvent or Operation), which indicates whether the sentence being constructed relates to an output event or to an operation.

The definition of output_always is also in Listing 9. Its arguments (Line 5) include, besides the outputType just mentioned, the name str of the output event or operation, and the sentences of the mapping. The definition uses a variable adv to record an adverb (CatEng.Adv) (Line 6). It is specified using another operation outputSentencePrefix_Adv, which produces the fragments “when the event ... occurs” or “when the operation ... is called”, depending on

outputType. The ellipses here are replaced with (str). The result of outputSentencePrefix_Adv ! outputType is a function, determined by outputType, that is applied to str. With mkS <adv : Adv> <sentences : S>, we get a sentence combining adv with the mapping sentences.

In conclusion, RoboWorld is a flexible and expressive subset of the English language, yet controlled. The intermediate representation presented next can, therefore, be generated automatically.

5. Intermediate representation

We define the semantics of a RoboWorld document in terms of an intermediate representation (IR) of that document. With this representation, we insulate the semantics specification presented in the next section 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 dictionary or of the concrete grammar. With the IR, such extensions, which are important to make the language more flexible, do not affect the semantics definition.

In the IR, information about the arena, the robot, and the other elements is grouped, and structured using notions of expressions and actions, although the original sentences are still recorded. Two sets of rules formalise how an IR can be automatically generated for a given RoboWorld document.

In Section 5.1, we present the IR, via the definition of its metamodel and well-formedness conditions. In Section 5.2, we present the rules to generate the IR for a RoboWorld document. For a RoboWorld document to be considered well formed, besides satisfying the conditions in Tables 3.1 and 4.1, it must also be the case that the application of the rules in Section 5.2 to that document generates a valid IR according to the conditions discussed in Section 5.1. Some of the well-formedness conditions are guaranteed by the rules, and some need to be checked.

5.1 Metamodel and well-formedness conditions

Figure 5.1 presents the top classes of the metamodel for our IR. Here, a document is represented by an instance of `RWIntermediateRepresentation`. In contrast with the metamodel (see

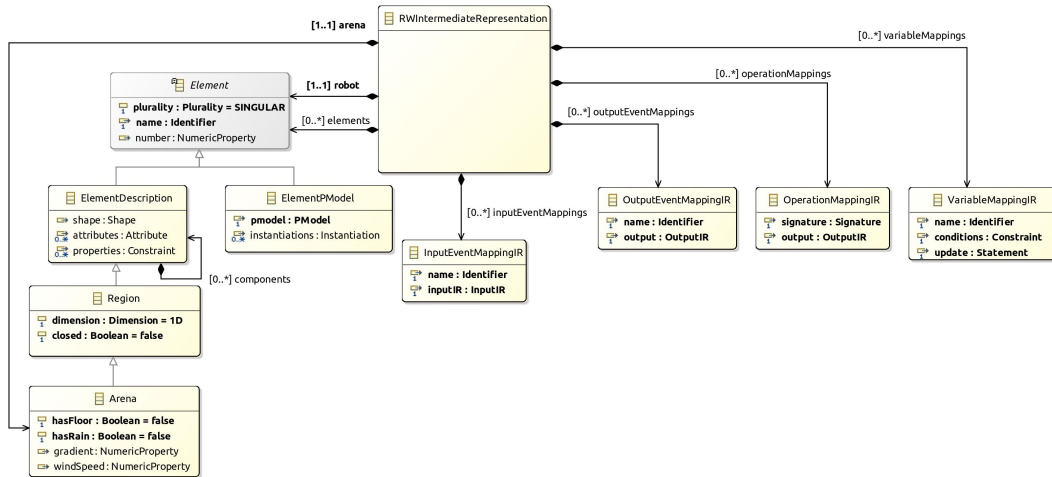


Figure 5.1: RoboWorld IR: top classes

Figure 3.5), its attributes do not record assumptions (just) in terms of sentences, but in terms of a richer collection of objects reflecting primitive and declared concepts in a RoboWorld document. These objects, including those that represent the arena and the robot, are all instances of an abstract class *Element*.

In the robotics domain, arenas and robots are clearly different concepts, and the notion of an element in RoboWorld covers everything else, including regions and entities, such as obstacles, robot components, and so on. In the IR, however, we provide a uniform view of all concepts of interest to provide an internal model that is more convenient to give semantics. This is achieved without affecting the domain-specific terminology used in RoboWorld documents.

The arena is represented by an instance of the class *Arena*, which in the IR is a *Region*. In turn, a *Region* is represented by an instance of *ElementDescription*. The *Element* abstract class has subclasses *ElementDescription*, to represent elements described using controlled natural language, and *ElementPModel*, to represent elements described by a p-model.

As an *Element*, the *Arena* has a plurality: it must be *SINGULAR*, since we have just one arena. Table 5.1 presents this well-formedness condition (IR1) and others for the IR. Figure 5.2 sketches the IR for our example. In general, the plurality of an *Element* can also be *PLURAL* for objects representing a set of instances of an element, such as fires, or *UNCOUNTABLE* (for example, smoke).

An *Element* also has a unique name (IR4), an *Identifier* that can be derived from an *Item* used in the RoboWorld document. For example, in the RoboTool implementation of the rules to generate the IR (see Section 5.2), the identifier used for the ‘tank of water’ is *tank_of_water*. An *Element* also has a pose, for *Elements* with a body, and a number of instances, for elements with plurality *PLURAL* (IR5). For the arena, the name must be “arena” (IR1).

In an *ElementDescription*, if it has a body, an attribute *shape* can record information using

Table 5.1: Some well-formedness conditions of RoboWorld’s IR

IR1	The plurality of the arena is SINGULAR, its name is “arena”, its shape is a Box, and its components, if any, are Regions.
IR2	If an Arena has a gradient, then hasFloor is true.
IR3	The plurality of the robot is SINGULAR, its name is “robot”, and it cannot be an instance of Region.
IR4	The names of the Elements and Attributes are unique.
IR5	The number of an element whose plurality is SINGULAR or UNCOUNTABLE is null.
IR6	If the arena dimension is 1D, then the shape of every ElementDescription is either null or an instance of Box with null ywidth and zwidth.
IR7	If the arena dimension is 2D, then the shape of every ElementDescription is either null, or an instance of Box with null ywidth and zwidth, or an instance of Sphere.
IR8	An ElementReference to an element whose plurality is SINGULAR must be an instance of UniqueElement.
IR9	An ElementReference to an element whose plurality is PLURAL must not be an instance of UniqueElement.
IR10	An ElementReference to an element whose plurality is UNCOUNTABLE must be an instance of UniqueElement or PotentialElement.
IR11	In an Assign, if the expressions of the assignto and of value are not null, then their types are equal.

objects that represent common geometric forms (boxes, cylinders, and so on). The not unexpected definition of the class Shape is omitted here, but all classes omitted here are in Appendix C. The shape of the arena is always a Box (IR1), but regions of the arena may have any shape. Moreover, if the arena is two-dimensional or one-dimensional, the Box degenerates to a square or a line.

In addition, to cater for application-specific elements, we can define attributes, more general properties, and components of an element. For the arena, however, components must be Regions (IR1). The class Attribute represents an attribute by recording its unique name (IR4) and type, the latter represented by a class Type that reflects the typing system of the RoboStar notations, which is based on that of the Z notation [17] for convenience of support for proof.

ElementPModel is similar to the homonymous class in the metamodel (see Figure 3.5).

A Region has a dimension and may be closed or not. An Arena may have a floor, as recorded by the Boolean attribute hasFloor. The definition of the gradient of the floor is optional, and can be present only if hasFloor is true (IR2). Our example in Figure 5.2 shows the gradient attribute, a NumericProperty characterised by a Constraint. The class NumericProperty has a single attribute properties containing one or more Constraints, a class whose definition is shown in Figure 5.3.

The Boolean attribute hasRain records whether it is raining. Finally, it is possible to record

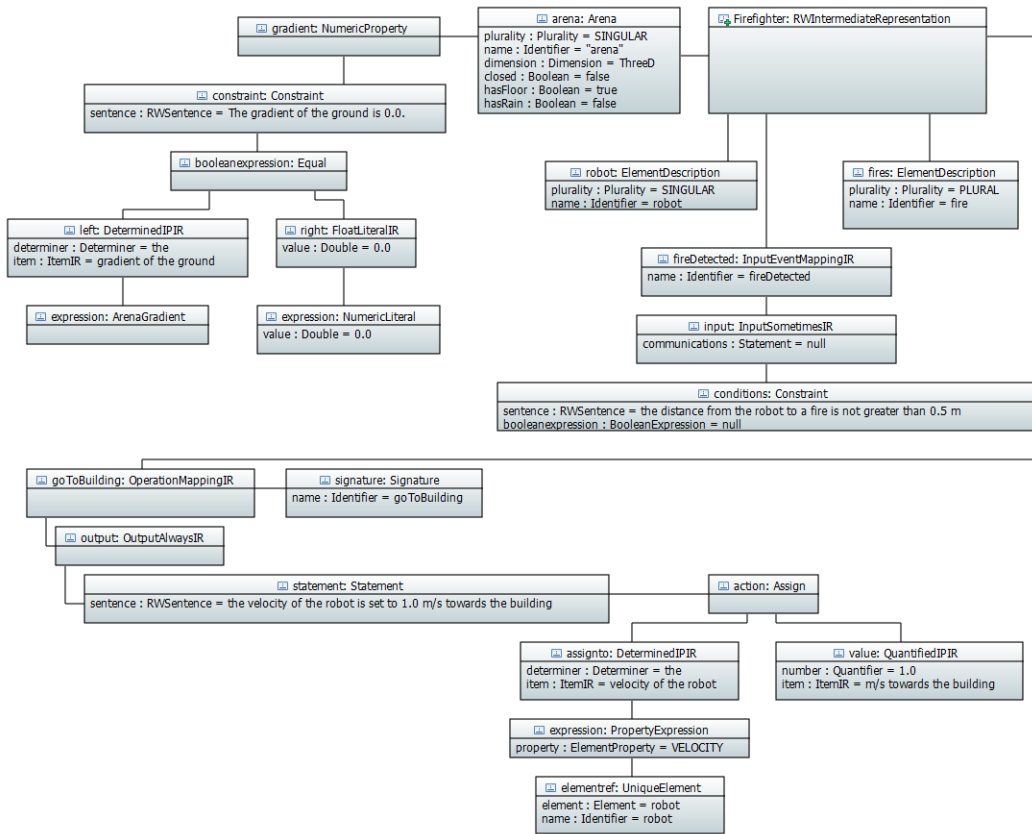


Figure 5.2: Partial sketch IR for RoboWorld document in Figures 3.1 and 3.2

the speed of the wind in `windSpeed`, which is yet another `NumericProperty`.

The robot is an `Element` with name “robot”. Its plurality has to be `SINGULAR`. It can be given by an `ElementDescription` or `ElementPModel`, but not by a `Region` (`IR3`).

For each mapping class of the metamodel (see Figure 3.5), there is a similar class in the IR. The differences are in the use of classes `InputIR` and `OutputIR`, instead of `Input` and `Output`, and `Constraint` and `Statement`, in Figure 5.3, instead of `Conditions` and `RWSentence`.

`InputIR` and `OutputIR`, omitted here, are themselves very similar to `Input` and `Output`. The core differences are just that `Conditions` and `RWSentence` are also replaced with `Constraint` and `Statement`. Moreover, the sentences attribute of the `InputIR` subclasses are named `communications`, not `sentences`, reflecting the fact that they define communicated values. In Figure 5.2, we show the IR objects related to the input event `fireDetected`. Similarly, `OutputIR` subclasses have an attribute `statements` instead of `sentences` because they define updates. In Figure 5.2, we show the IR objects related to the call to the operation `goToBuilding`, which is recorded as an output.

So, the main new features are the classes `Constraint` and `Statement`. As shown in Figure 5.3, these classes record, besides the sentences in the RoboWorld document, additional attributes that

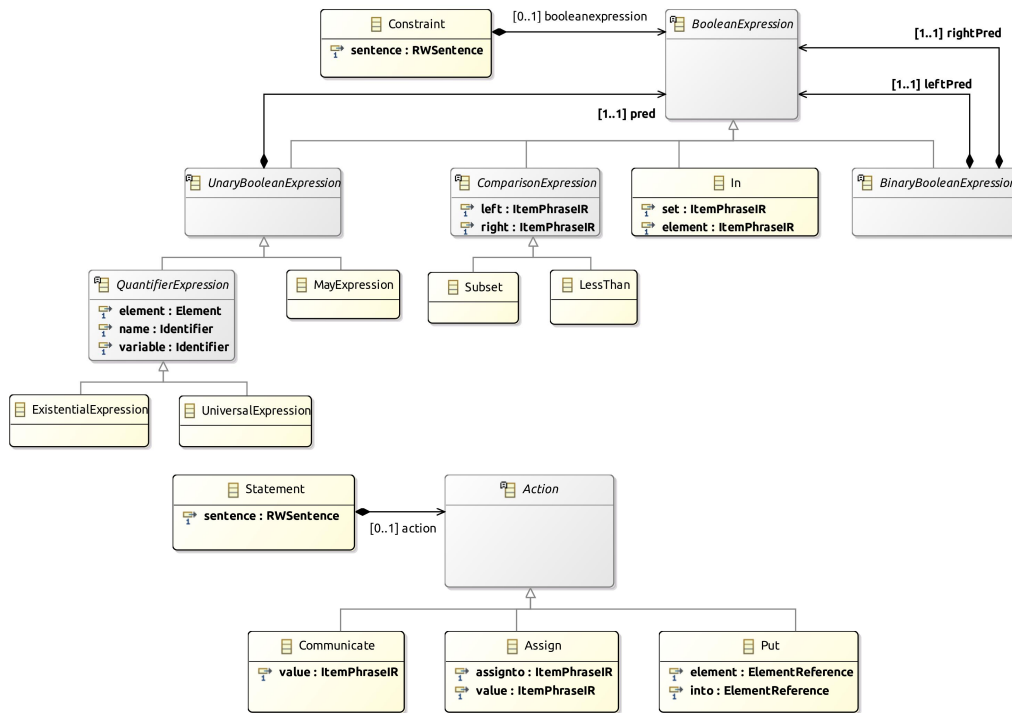


Figure 5.3: RoboWorld IR: constraints and statements

record the information in the sentences in a form suitable to define the semantics. Both Constraint and Statement have an attribute sentence, and also an extra attribute, booleanexpression in the case of Constraint and action in Statement. These extra attributes are annotations, which may or may not be present, depending on whether the meaning of the sentence can be captured by the RoboWorld semantics. This is determined by the rules to generate the IR presented in the next section.

As explained in the next section, there are two sets of rules: the first creates a basic IR, and the second defines an annotated version of that IR. For instance, in our example, the attribute booleanexpression of the constraint for the gradient of arena in the IR defined by the first set of IR generation rules is null. After the second set of rules is applied, we get the annotation in Figure 5.2.

The definition of the class BooleanExpression is in many ways as to be expected, and we show just some of its subclasses here. We have UnaryBooleanExpressions and Binary-BooleanExpressions, and note that in a QuantifierExpression we have an Identifier for the quantified variable, which ranges over the instances of the element. ComparisonExpressions include those based on the Subset and LessThan relations, among many others. The actual terms being compared are item phrases as represented in the IR: instances of the class ItemPhraseIR.

In Figure 5.2, the booleanexpression for the gradient constraint is an instance of the class Equal that represents equalities. It has attributes left and right whose types are ItemPhraseIR.

ItemPhraseIR is similar to ItemPhrase, but, like Constraint and Statement, it has an extra attribute `expression` to record the element described in a structured way. The type of expression is a class Expression with a rich set of subclasses omitted here. Some of these subclasses capture domain-specific expressions like TimeSince an event occurrence or the ArenaGradient.

As shown in Figure 5.2, the instance of Equal for the boolean expression of the gradient constraint has as its left attribute an instance of DeterminerIPIR, the IR version of DeterminerIP. For simplicity, we do not show the objects for the `item` attribute of `left`; we just indicate that it represents ‘gradient of the ground’. We show, however, the expression for `left`, which is an instance of ArenaGradient. This object has no attributes, but flags the meaning of the DeterminerIPIR. There can be many different ways to refer to the gradient of the floor of the arena (‘gradient of the ground’, as in the example, ‘gradient of the floor’, ‘gradient of the arena’, and so on). With the annotation, we simplify the definition of the semantics, which can be based on the presence of an instance of ArenaGradient, and not on the many forms that we can use to refer to this concept.

For the right attribute of the gradient constraint, we have an instance of FloatLiteralIR, the IR version of FloatLiteral. Its expression just records the value of the literal, but its presence does simplify the semantics, which can rely on the presence of an expression for all constraints.

The subclass PronounIPIR of ItemPhraseIR is similar to the subclass PronounIP of ItemPhrase, but has yet another attribute. Namely, it records, in an attribute `referent`, the ItemPhraseIR to which the pronoun refers. This is in addition to the `expression` attribute inherited from ItemPhraseIR. In the generation of the IR, the value of `referent` is used to indicate the element referenced by the pronoun. If its meaning is covered by the RoboWorld semantics, in addition, the value of `expression` is recorded to represent that element for the definition of the semantics.

Figure 5.3 shows just three forms of Actions. A communication (that is, an instance of Communicate) defines a value as an ItemPhraseIR. (This is the IR class that represents an expression.) An Assignment records its target `assignto` and assigning value as ItemPhraseIR. Finally, instances of a Put subclass of Action record that an element is put into another one.

The action attribute for the statement of the output for the operation goToBuilding is shown in Figure 5.2. It is an Assign, whose `assignto` attribute is a DeterminerIPIR whose expression is a reference to a property of an element (see Figure 5.1), represented by an instance of PropertyExpression. In this case, the value of the attribute `property` is one of several primitive properties, namely, VELOCITY. The element is identified by an `elementref`.

In Actions and Expressions, references to an element are represented by an instance of the class ElementReference shown in Figure 5.4. This is an abstract class with an attribute `element`; the subclasses reflect the several meanings that a reference to `element` may have. A reference to an element whose plurality is SINGULAR must be a UniqueElement (IR6). This is the case of the robot, in the example in Figure 5.2. For simplicity, we do not show the object for

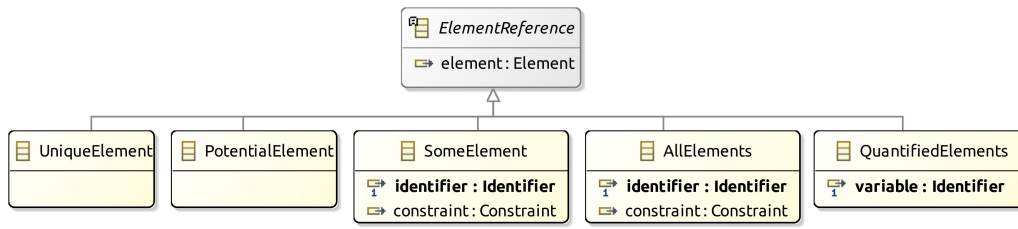


Figure 5.4: RoboWorld IR: element references

the robot as an `Element`.

For other elements, the different forms of `ElementReference` capture context information. For example, in “A fire can occur on the floor”, the reference “a fire” denotes a potential, but not necessary, instance of a fire. It is represented by an instance of `PotentialElement`. In “... the distance from the robot to a fire ...” we have a reference to some fire characterised by a `constraint`; this is represented by an instance of `SomeElement`. In the example below we have a mapping for an alternative typeless event `spray` for a firefighter.

Example 14 When the event `spray` occurs the fires within 3.0 m are extinguished.

□

Here, “the fires” refers to all fires satisfying a `constraint`, and it is represented by an instance of `AllElements`. Finally, `QuantifiedElements` records a reference to a quantified variable.

The well-formedness conditions IR7 and IR8 impose additional restrictions on the use of `ElementReferences` based on the plurality of an element. Finally, IR9 is an example of a well-formedness condition related to the types of `Expressions`. These are all conditions that need to be checked, after the application of the rules presented in the next section.

5.2 Generation from RoboWorld documents

The IR for a RoboWorld document can be automatically derived. As mentioned before, this is a two-step process. First, an IR is obtained from the provided document; afterwards, it is annotated. In the following sections, we cover these two steps.

5.2.1 Generating the intermediate representation

Our rules define functions. Each rule has a number and a name, followed by the function declaration: name, arguments, return type, and specification. The metanotation used for specification is functional and standard. It is distinguished from the target notation to describe objects of the IR by use of a grey font. The simple target notation is in italics. To define an object of a class `C`, we use the construct `new C{...}`, where we list, between curly brackets, the value of each attribute.

Rule 1. Map RWDocument

```
mapRWDoc(rwDoc : RWDocument) : RWIntermediateRepresentation =
```

```
  new RWIntermediateRepresentation {
    arena = mapArena(rwDoc.arenaAssumptions, new Arena{})
    robot = mapRobot(rwDoc.robotAssumptions, new Robot{})
    elements = mapElements(rwDoc.elementAssumptions,
      enumerateElements(rwDoc.arenaAssumptions, rwDoc.robotAssumptions, rwDoc.elementAssumptions))
    inputEventMappings = mapInputEvents(rwDoc.inputEventMappings)
    outputEventMappings = mapOutputEvents(rwDoc.outputEventMappings)
    operationMappings = mapOperations(rwDoc.operationMappings)
    variableMappings = mapInputEvents(rwDoc.variableMappings)
  }
```

Rule 2. Map ArenaAssumptions

```
mapArena(assumptions : Seq(ArenaAssumption), arena : Arena) : Arena =
```

```
  if #assumptions = 0 then arena
  else mapArena(tail(assumptions), updateArena(head(assumptions), arena))
```

Attributes not listed have arbitrary values.

Rule 1 defines the function `mapRWDoc` whose application to a document, represented by the argument `rwDoc` whose type `RWDocument` is defined in the RoboWorld metamodel (see Figure 3.5), produces an instance of `RWIntermediateRepresentation` (see Figure 5.1). So, it is this rule that defines the overall mapping from a RoboWorld document to its IR.

Each attribute of the `RWIntermediateRepresentation` object defined by Rule 1 is specified by the application of a separate `map` function, defined by other rules. Each function takes the relevant assumptions or mappings of `rwDoc` as argument. The functions `mapArena` and `mapRobot` used to define `arena` and `robot` take default instances of `Arena` and `Robot`, that is `new Arena{}` and `new Robot{}` (see Figure 5.1) as additional arguments. For `mapElements`, an additional argument is defined by the application of the function `enumerateElements`, which characterises the sequence of all the `Elements` declared in the assumptions made in `rwDoc`.

For illustration, we present here the definition of `mapArena` in Rule 2. It is defined recursively, iterating over its `assumptions` argument: the sequence (`Seq`) of `ArenaAssumptions` in the document. When that sequence is not empty (that is, its size `#assumptions` is different from 0) we apply `mapArena` recursively to the sequence's `tail`, but providing an updated version of the second argument `arena` of `mapArena` to record the information in the head of `assumptions`. When all `ArenaAssumptions` have been considered, that is, `#assumptions = 0`, the result is just `arena`.

The function `updateArena` used in Rule 2 is defined by Rule 3; we show below an excerpt of its specification. Taking into account the information that can be recorded in the IR, we have defined a collection of boolean `find` functions that determine if a given `assumption` refers to a particular concept. For example, `findArenaDimensionInfo` determines whether `assumption`

Rule 3. Update Arena

```
updateArena(assumption : ArenaAssumption, arena : Arena) : Arena =
```

```
  if findArenaDimensionInfo(assumption) then arena.dimension = getArenaDimensionInfo(assumption)
  else if findArenaClosedInfo(assumption) then arena.closed = getArenaClosedInfo(assumption)
  else if findArenaFloorInfo(assumption) then arena.hasFloor = getArenaFloorInfo(assumption)
  else if findArenaRainInfo(assumption) then arena.hasRain = getArenaRainInfo(assumption)
  ...
```

Rule 4. Find dimensionality information

```
findArenaDimensionInfo(assumption : ArenaAssumption) : Boolean =
```

```
  if refersToArena(assumption.sentence.clause.itemPhrase)
    ∧ assumption.sentence.clause instanceof mkRWClause_ActiveVoice_ToBe_Adjective
    ∧ positiveSentence(assumption.sentence)
  then
    let adj = ((mkRWClause_ActiveVoice_ToBe_Adjective) assumption.sentence.clause).a
    within if adj ∈ dimensionAdjectives then true else false
  else if ...
  else false
```

refers to the arena dimensionality. In Rule 3, we use these `find` functions to determine whether the second argument `arena` of `updateArena` can be enriched with information from `assumption`.

If no `find` function identifies information recognised in the IR, the result of `updateArena` is just `arena`. Otherwise, the result is an updated version of `arena`, where one of its attributes is changed using a `get` function that retrieves the relevant information from `assumption`.

To define the `find` and `get` functions for the sentences in the assumptions, we rely on the control imposed by RoboWorld. Rule 4 defines `findArenaDimensionInfo`; the sketch of its specification presented below illustrates how we detect whether the sentence in a given assumption provides information about the arena dimension. First, we check whether the first `itemPhrase` of the clause embedded in the sentence of the assumption mentions the arena (using a boolean function `refersToArena`), whether this clause has been created by `mkRWClause_ActiveVoice_ToBe_Adjective` (see Section 4.5), and whether sentence has a positive polarity (using a boolean function `positiveSentence`).

If these conditions are met, we also verify whether the adjective embedded in the clause (`adj`) belongs to the set of dimension-related adjectives (`dimensionAdjectives`). Here, we use a `let-within` structure to define the variable `adj` local to the rule. Its value is the adjective (`a`) of the clause in the sentence of the given assumption. A cast `(mkRWClause_ActiveVoice_ToBe_Adjective)` ensures that the clause is of the right type and, therefore, we can refer to its adjective `a`. The set `dimensionAdjectives` includes, for example, ‘three-dimensional’, ‘two-dimensional’, and ‘3D’.

Sentences with other structures may also say something about the arena dimensionality. Therefore, Rule 4 also considers other conditions (as indicated below by the `else if ...`).

Rule 5. Map InputEventMappings

```
mapInputEvents(inEvents : Seq(InputEventMapping)) : Seq(InputEventMappingIR) =
  if #inEvents = 0 then {}
  else { mapInputEvent(head(inEvents)) } ∪ mapInputEvents(tail(inEvents))
```

Rule 6. Map InputEventMapping

```
mapInputEvent(inEvent : InputEventMapping) : InputEventMappingIR =
  new InputEventMappingIR {
    name = inEvent.name
    input = mapInput(inEvent.input)
  }
```

Example 15 Rule 4 yields true when applied to the following sentences: “the arena is three-dimensional”, “the arena has three dimensions”. □

The mapping process for the robot, defined by `mapRobot`, is similar. Regarding elements, the main difference is that `mapElements` considers each of the elements passed as its second argument.

Information about mappings is also extracted from the sentences. Here, we exemplify this process for `InputEventMappings`. The simple definition of the function `mapInputEvents` is shown in Rule 5. It recursively applies another function `mapInputEvent` (see Rule 6) to each `InputEventMapping` in its argument, yielding the sequence of the obtained results.

The function `mapInputEvent` defines an `InputEventMappingIR` for a `InputEventMapping`. The name of the `InputEventMappingIR` is that in the `InputEventMapping`. The value of the `input` attribute depends on the type of the `Input` in the `InputEventMapping`. So, it is defined by an overloaded function `mapInput` that considers the subclasses of `Input` (see Figure 3.6).

Rule 7 presents the definition of `mapInput` for instances `input` of `InputSometimes`. An `InputSometimes` has conditions and sentences, which are recorded as constraints and statements. To provide a concise definition, we rely on the standard `map` function from functional programming to apply anonymous functions that create *Constraints* and *Statements* from the respective sentences.

We use a `where` clause to define variables global to the rule called `conditions` and `communications`, used to define the homonymous attributes of the resulting *InputSometimesIR*. The definition of `conditions` applies, via the use of `map`, a function defined by a λ expression, to each sentence of the sequence `sentences` of the `conditions` of `input`. The result of the `map` is the sequence of the results. The function defined by the λ expression has argument `x` and specifies an instance of the IR class *Constraint*, whose `sentence` attribute has value `x`. It is the second set of rules, presented in the next section, that extracts further information from the sentences, if possible. The definition of

Rule 7. Map InputSometimes

```
mapInput(input : InputSometimes) : InputIR =

  new InputSometimesIR {
    conditions = conditions
    communications = communications
  }
  where
    conditions = map (λ x → new Constraint{sentence = x}) input.conditions.sentences
    communications = map (λ x → new Statement{sentence = x}) input.sentences
```

Rule 8. Annotate Constraint

```
annotateConstraint(constraint : Constraint) : Constraint =

  if positiveSentence(constraint.sentence)
  ∧ constraint.sentence.clause instanceof mkRWClause_ActiveVoice_ToBe_ItemPhrase then
    let cl = (mkRWClause_ActiveVoice_ToBe_ItemPhrase) constraint.sentence.clause
    within
      constraint.booleanexpression = new Equal {
        left = createItemPhraseIR(cl.itemPhrase1)
        right = createItemPhraseIR(cl.itemPhrase2)
      }
  else if ...
```

communications is similar, but considers the sentences of input and specifies a *Statement*.

The extraction of information from OutputEventMappings, OperationMappings and VariableMappings to derive an appropriate IR follows the ideas presented before.

5.2.2 Annotating the intermediate representation

As explained in the previous section, our first set of rules maps a document to an IR representation. In contrast, the second set of rules defines an IR-to-IR transformation. Its purpose is to enrich the IR, via the expression and action attributes of Constraints and Statements, to record information in a structured way.

A top rule defines a function that applies to an *RWIntermediateRepresentation* and, like Rule 1, uses other functions that deal with attributes of the IR classes, using yet more functions. It is the functions for *Constraint* and *Expression* that define the features of an enriched IR.

We present next part of Rule 8, which defines a function `annotateConstraint`. Specifically, we focus on the fragment that deals with positive sentences that have clauses created with the function `mkRWClause_ActiveVoice_ToBe_ItemPhrase`. An example of such a sentence is “the gradient of the ground is 0.0”, recorded in the constraint for the gradient of the arena in Figure 5.2. The `mk` function has two parameters of type *ItemPhrase*. For the example, the first *ItemPhrase* is “the gradient of the ground” and the second is “0.0”.

Rule 9. Annotate Statement

```

annotateStatement(statement : Statement) : Statement =

  if statement.sentence.clause instanceof mkRWClause_PassiveVoice_TransitiveVerb_Preposition_ItemPhrase
  ∧ positiveSentence(statement.sentence) then
    let cl = (mkRWClause_PassiveVoice_TransitiveVerb_Preposition_ItemPhrase) statement.sentence.clause
    within
      if cl.verb ∈ assignmentVerbs then
        statement.action = new Assign {
          assignto = createItemPhraseIR(cl.itemPhrase1)
          value = createItemPhraseIR(cl.itemPhrase2)
        }
      else if ...
    else if ...

```

Rule 8 annotates the *constraint* by setting its *booleanexpression* to an instance of an *Equal* expression with *left* and *right* attributes corresponding to the *ItemPhraseIR*s created from the first and second item phrases of the clause in the sentence of the given constraint. The local variable *cl* records that clause, cast to ensure it is created using *mkRWClause_ActiveVoice_ToBe_ItemPhrase*. In this case, *cl.itemPhrase1* and *cl.itemPhrase2* give the clause's instances of *ItemPhrase*. The function *createItemPhraseIR*, from our first set of rules, is used to translate these *ItemPhrases* to their representations in the IR: instances of *ItemPhraseIR*. For our example, as shown in Figure 5.2, we get a *DeterminerIPIR* and a *FloatLiteralIR* for the item phrases in the constraint of the arena.

We show below part of the Rule 9 definition for annotation of Statements. The case presented is for clauses of type *mkRWClause_PassiveVoice_TransitiveVerb_Preposition_ItemPhrase*, that are positive. An example is “the velocity of the robot is set to 1.0 m/s towards the building”. This function has four parameters: a first *ItemPhrase*, a two-place verb *V2*, a *Preposition*, and a second *ItemPhrase*. For the previous example, the arguments are “the velocity of the robot”, “set”, “to”, and “1.0 m/s towards the building”. For this kind of clause, the definition of Rule 9 considers two cases, depending on whether the verb (*cl.verb*) of the clause denotes an assignment notion (that is, it belongs to the set of *assignmentVerbs*) or not. If it does, the *action* attribute of the statement is annotated with an instance of *Assign*, whose value of the attribute *assignto* is an *ItemPhraseIR* for the first argument of the *mk* function (*cl.itemPhrase1*), and whose *value* is the *ItemPhraseIR* for the last argument (*cl.itemPhrase2*).

Next, we show how we use the IR to define a semantics for RoboWorld.

6. RoboWorld: semantics

In this section, we give an overview of the formal semantics of RoboWorld documents (Section 6.1), and present semantic functions that apply to the IR (Section 6.2). Together with the rules presented in Section 5.2, they can be used to generate the semantics of a RoboWorld document automatically.

6.1 Formal semantics: overview

The semantics of a RoboWorld document is a hybrid model, due to the continuous nature of the arena and movement. We thus specify formal semantics for RoboWorld using *CyPhyCircus*. Like in CSP, *CyPhyCircus* models define mechanisms via processes that can communicate with each other via atomic and instantaneous event. Like in *Circus*, however, *CyPhyCircus* processes include a state. As already said, *Circus* combines Z and CSP. Moreover, the state of a *CyPhyCircus* process can contain continuous variables and may or may not be encapsulated. The behaviour of a process is defined by an action, which, like in CSP, defines patterns of interaction, but, like in *Circus*, can also define data updates. We explain the constructs of *CyPhyCircus* as we use them.

The overall structure of the RoboWorld semantics and how it connects with the semantics of RoboChart is indicated in Figure 6.1. The semantics of a RoboWorld document is a *CyPhyCircus* process comprised of two further processes composed in parallel: an environment process, which represents the objects in the environment and handles triggering of events, and a mapping process, which contains the semantics for the output-event and operation mappings of the document. To define a model for a whole system, including the control software modelled in RoboChart, and the

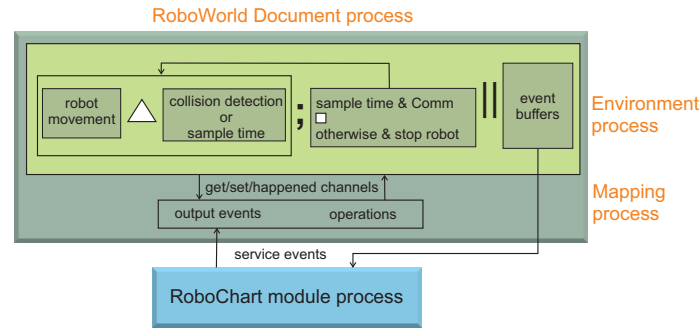


Figure 6.1: The structure of the RoboWorld semantics

robot and the environment as defined in a RoboWorld document, we can compose the RoboWorld process with the process that defines the semantics of the RoboChart module as shown at the bottom of Figure 6.1. The RoboWorld process communicates with the RoboChart process on *CyPhyCircus* (and CSP) events representing the services of the robotic platform.

The environment process is defined by the parallelism (represented by parallel bars in Figure 6.1) of two actions. The first action is a loop that (a) evolves the state; (b) communicates with the mapping process via get and set channels; and (c) buffers information about inputs. The body of the loop includes an action (indicated by the box labelled `robot movement` in Figure 6.1) that continuously evolves variables representing elements of the environment to capture the movement of the robot. This evolution can be interrupted (indicated by \triangle in Figure 6.1) by either the detection of a collision between the robot and an element of the environment, or by the time reaching a specified sample time. After the interruption, if it is due to reaching the sample time, the loop action checks if the conditions for each input event are fulfilled, communicates the result to the second parallel action (indicated by the box labelled `event buffers` in Figure 6.1) and then communicates with the mapping process to allow it to get and set the values of state variables, before starting again. If the interruption of `robot movement` is due to a collision, the robot is stopped: its velocity and acceleration are set to zero, and then the action loops back.

The action `event buffers` defines a set of buffers, one for each input or output event. A buffer for an input event records whether that event was detected on the time step, and provides that information to the RoboChart process. A buffer for an output event records the time in which it last happened. It takes that information from the mapping process via a `happened` channel. Buffering the inputs and outputs allows the evolution of the environment in `robot movement` to proceed independently from their communication to the RoboChart process, directly in the case of input events, or indirectly via the mapping process, in the case of output events.

The mapping process is defined by the interleaving of processes that accept output events and operation calls from the RoboChart process, and pass on the relevant information to the environment process. These processes capture the mapping definitions in the RoboWorld document.

Figures 6.2-6.10 sketch the semantics for the firefighter document presented in Figures 3.1 and


```

channelset getSetChannels == { getRobotPosition, getRobotVelocity, ... }
channelset eventHappenedChannels == { sprayHappened, takeOffHappened, ... }

process RWDocument  $\hat{=}$  (Environment [ getSetChannels  $\cup$  eventHappenedChannels  $\cup$  {proceed} ] Mapping)
    \ getSetChannels  $\cup$  eventHappenedChannels  $\cup$  {proceed}

```

Figure 6.2: The *RWDocument* process for the firefighter example

<i>ArenaProperty</i>
<i>xwidth</i> , <i>ywidth</i> , <i>zwidth</i> : \mathbb{R} <i>gradient</i> , <i>windSpeed</i> : \mathbb{R} <i>locations</i> : \mathbb{P} <i>Position</i> <i>home</i> : <i>HomeProperty</i>
<hr style="border: 0; border-top: 1px solid black; margin: 5px 0;"/> <i>locations</i> = { <i>x</i> : 0.0.. <i>xwidth</i> ; <i>y</i> : 0.0.. <i>ywidth</i> ; <i>z</i> : 0.0.. <i>zwidth</i> }

Figure 6.3: Example of a type declared in the semantics of the firefighter RoboWorld document

3.2. Figure 6.2 shows the definition of the overall *RWDocument* process that captures the semantics of the whole document. As already said, it is defined by a parallel composition ($\llbracket \dots \rrbracket$) of processes *Environment* (see Figure 6.6) and *Mapping* (see Figure 6.10). The union of the sets *getSetChannels*, *eventHappenedChannels* and {*proceed*}, indicated between the \llbracket and \rrbracket symbols, contains the events that require synchronisation between *Environment* and *Mapping*. The same set is indicated after \backslash to define that the events happen instantaneously and are not visible by the RoboChart process.

The sets *getSetChannels* and *eventHappenedChannels* are also defined in Figure 6.2. As their names indicate, these are events for communication with the *Mapping* process (*getSetChannels*) and with the buffers (*eventHappenedChannels*) as sketched in Figure 6.1. Like in CSP, *CyPhy-Circus* events represent communications over channels; fat brackets $\{\dots\}$ are used to define the set of all events representing communications on the channels listed. Examples of get, set, and happened channels are given in Figure 6.2, as part of the definition of *getSetChannels* and *eventHappenedChannels*.

The channel *proceed* is just a signal, that is, it does not communicate any values. It is used by *Mapping* to indicate to the *Environment* that it can proceed with the loop (see Figure 6.1) after all necessary communications over *getSetChannels* and *eventHappenedChannels* have finished.

To define the types of channels and state variables, the semantics declares types used to represent the properties of the elements in the environment. These are record types specified as Z schemas, written as a box with the name of the schema (record type) at the top, the components of the schema (fields of the record) and their types specified inside the box, and constraints on those components specified below a horizontal line. Figure 6.3 shows the type *ArenaProperty* used to record properties of the arena. The complete model is available in Appendix D.

The definition of *ArenaProperty* follows closely that of the class *Arena* in the IR. Some of the attributes of the IR *arena*, however, are used to define the semantics, but do not need to be

```

channel fireDetectedTriggered :  $\mathbb{B}$ 
channel sprayHappened
channel getRobotPosition : Position
channel setRobotTank_of_water : Tank_of_waterType

```

Figure 6.4: Some channels declared in the semantics of the firefighter RoboWorld document

reflected in *ArenaProperty*. For instance, we recall that the shape of the arena is always a *Box*. We do not, however, have a *shape* component in *ArenaProperty*, but the dimension attribute of the IR arena determines the attributes of the IR class *Box* that we include in *ArenaProperty*. For our example, we have a three-dimensional arena, and so components *xwidth*, *ywidth*, and *zwidth* of *ArenaProperty*, each of which is a real number, record the size of the arena.

Additionally, when the attribute *hasFloor* of arena is true, like in our example, *ArenaProperty* has a component recording the *gradient* of the ground. We always record the *windspeed*, but use *closed* and *hasRain* from arena to define the action that models the movement of the robot (see Figure 6.1). The component *locations* is a set (specified in \mathbb{Z} by \mathbb{P}) of *Positions*, representing all the positions inside the arena. *Position* is defined as the set of triples of real numbers, since the arena is three-dimensional. The *locations* set is derived from the size of the arena, and hence is defined in a constraint on the *ArenaProperty* schema. It includes the whole range of positions, with the values for each coordinate starting from 0.0 and going up to the size for each dimension.

Finally, *ArenaProperty* has a component for each region of the arena. In our example, we have a component *home*. Its type is defined by another schema, omitted here. Additional schemas define types to represent the robot, and, in our example, also the building and a fire.

The declaration of channels in *CyPhyCircus* is global to processes. Figure 6.4 shows the declaration of a few of the channels used in the RoboWorld semantics as indicated in Figure 6.1.

We declare channels used to indicate to the event buffers whether an input event has occurred. One of these is declared for each input, with the name of the channel formed from the name of the input appended with *Triggered*. Each of these channels communicates a boolean value (\mathbb{B}) indicating whether the event has been detected at the timestep.

There are also *Happened* channels, one for each output event and operation, to signal to the buffers when an output event has happened or an operation has been called. This allows the *Environment* to record the time since the occurrence of these events, which can be used in the trigger conditions for input events. For example, in the mapping of the input event *critical* in the firefighting UAV example, where the times since *spray* and *takeOff* are used (see Figure 3.2).

Finally, there are get and set channels for the properties of each element in the environment. Figure 6.4 shows the declarations of the get channel for the robot position, and of the set channel for the tank of water of the firefighter example. *Tank_of_waterType* is an enumeration determined by the type of the attribute *tank_of_water* of the robot in the IR. The arena and its regions

	<i>arena</i> : <i>ArenaProperty</i>
	<i>robotInit</i> : <i>RobotProperty</i>
	<i>potentialFires</i> : $\mathbb{P} \text{ FireProperty}$
	<i>groundLocations</i> : $\mathbb{P} \text{ Position}$
	<i>groundLocations</i> = $\{x, y, z : \mathbb{R} \mid (x, y, z) \in \text{arena.locations} \wedge z = 0\}$
	<i>arena.xwidth</i> = 50.0
	<i>arena.ywidth</i> = 60.0
	<i>arena.zwidth</i> \geq <i>building.zwidth</i> + 1.0
	<i>timeStep</i> : \mathbb{R}

Figure 6.5: Some global constants and constraints in the semantics of the firefighter

do not have channels for their properties, since they are always static and so their properties are defined as global constants. The properties for other elements have channels so that they can be handled in a uniform way, regardless of whether they are static or not.

The semantics also uses the channels for the services of the RoboChart robotic platform, which correspond to the input and output events, and to operation calls.

Global constants along with constraints on them capture environment assumptions. Examples are shown in Figure 6.5. The constants are specified using the Z notation for axiomatic definitions, indicated by a vertical line on the left without a full box. They have a similar structure to schemas, consisting of definitions and optional constraints separated by a horizontal line.

We declare global constants for the properties of each of the elements of the environment; their types are the *Property* records. The *arena*, for example, is unique and static, so its global constant records the values for its properties, making them globally accessible to the *Environment* and *Mapping* processes. The robot is not static, so its global constant, *robotInit*, just represents the initial values of its properties. Figure 6.5 also gives the example of the constant for the fires. Since they are plural and have dynamic attributes (since the status can change over time), the global constant is a set of potential *FireProperty* records, *potentialFires*. The actual fires are declared later in the state of *Environment*, so that their statuses can change, with the fires drawn from the *potentialFires* set.

Some global constants capture general properties. For instance, in Figure 6.5, *groundLocations* is a set of *Positions* defined to be the *locations* in the *arena* where the *z* component is equal to zero. This definition is standard and is included since the *arena* is defined to have a floor of gradient 0.0.

Additional axiomatic definitions capture the assumptions, potentially referring to properties of different elements. For instance, the penultimate definition in Figure 6.5 is concerned with *arena.zwidth* and *building.zwidth*. These constraints arise from the annotated Constraints in

process *Environment* \triangleq **begin**

EnvironmentState \equiv [**visible** *robot* : *RobotProperty*; **visible** *fires* : seq *FireProperty*;
 time : \mathbb{R} ; *stepTimer* : \mathbb{R} ; *EventTimes*]

state *EnvironmentState*

$\frac{\text{EnvironmentStateInit}}{\text{EnvironmentState}'}$
$\text{robot}' = \text{robotInit} \wedge \text{ran fires}' \subseteq \text{potentialFires} \wedge \text{time}' = 0.0 \wedge \text{stepTimer}' = 0.0 \wedge \text{EventTimesInit}$

RobotMovementAction \triangleq ...

CollisionDetection \triangleq *RobotGroundCollision* \square *RobotBuildingCollision* \square *RobotFireCollision*

InputTriggers \triangleq *fireDetected_InputEventMapping* \parallel *noFire_InputEventMapping* \parallel ...

Communication \triangleq ((*GetRobotPosition* \square *GetRobotVelocity* \square ...) ; *Communication*) \square *proceed* \rightarrow **Skip**

...

InputEventBuffers \triangleq *fireDetected_Buffer* \parallel *noFire_Buffer* \parallel *critical_Buffer* \parallel *landed_Buffer*

OutputEventBuffers \triangleq *spray_Buffer* \parallel *takeOff_Buffer* \parallel *goToBuilding_Buffer* \parallel *goHome_Buffer* \parallel *searchFire_Buffer*

EventBuffers \triangleq *InputEventBuffers* \parallel *OutputEventBuffers*

EnvironmentLoop \triangleq (*EnvironmentStateInit*) ; $\mu X \bullet$
 RobotMovementAction ;
 $\left(\begin{array}{l} (\text{stepTimer} < \text{timeStep}) \& \text{CollisionDetection} \\ \square \\ (\text{stepTimer} \geq \text{timeStep}) \& \text{InputTriggers ; Communication ; stepTimer} := 0.0 \end{array} \right) ; X$

channelset *triggerChannels* \equiv {*fireDetectedTriggered*, *noFireTriggered*, *criticalTriggered*, *landedTriggered*}

\bullet (*EnvironmentLoop* [*triggerChannels*] *EventBuffers*) \setminus *triggerChannels*

end

Figure 6.6: *Environment* process for the firefighter example

the IR, relying on their expressions. The constraint just mentioned corresponds to the assumption “the height of the arena is the height of the building plus at least 1.0 m” (see Figure 3.1).

Finally, a constant *timeStep* records the length of the time for the loop in the *Environment* process. The structure of *Environment* is shown in Figure 6.6.

Environment is defined as a basic process, which explicitly specifies a state and a main action at the end after a spot (\bullet) to define its behaviour. This is in contrast, for example, with a process like *RWDDocument* in Figure 6.2, which is defined in terms of other processes. Differently from a process, an action is local to a process and has access to the state of that process. Typically, a basic process definition includes various actions used to define its main action.

$$\begin{array}{c}
\text{RobotMovement} \\
\hline
\wedge \text{EnvironmentState} \\
\hline
\frac{d\text{robot.position}}{dt} = \text{robot.velocity} \dots \\
\frac{d\text{time}}{dt} = 1 \wedge \frac{d\text{stepTimer}}{dt} = 1 \\
\hline
\end{array}$$

$$\text{RobotMovementAction} \triangleq (\text{RobotMovement}) \triangleleft \left(\begin{array}{l} (\text{robot.position} \in \text{groundLocations} \wedge \text{robot.velocity} < 0) \\ \vee \dots \vee (\text{stepTimer} \geq \text{timeStep}) \end{array} \right)$$

Figure 6.7: The *RobotMovementAction* for the firefighter UAV

The state of the *Environment* process is given by a schema *EnvironmentState*, which defines components to record the state of the *robot* and other dynamic elements. These components are marked **visible**, so that the behaviour of *Environment* is characterised by the evolution of the values of these components over time, as well as occurrences of events. In our example, besides *robot*, we have a sequence of *fires*. The types *RobotProperty* and *FireProperty* are defined in Appendix D. There is no component to record the state of the building in *EnvironmentState* since the building contains no dynamic elements, whereas the fires have a status that may change.

EnvironmentState also contains encapsulated components. First, *time* is a clock recording the global time; it is used to determine when events occur. Second *stepTimer* is another clock that accounts for the time the environment evolves to detect when *timeStep* is reached. A schema *EventTimes*, which we omit here, is also defined with two components for each (input and output) event and operation. One component is a boolean recording whether the event happened or the operation was called, and another records the time of the occurrence or call. The *EventTimes* schema is included into *EnvironmentState* so that its components become components of *EnvironmentState*.

The main action of *Environment* is a parallel composition of an action *EnvironmentLoop*, defining the main loop for the environment, and an action *EventBuffers* (see Figure 6.1). This parallelism synchronises on the input-event *Triggered* channels, which are placed into a channel set *triggerChannels*, to signal to the buffers when an input event is detected at the *timeStep*. The *triggerChannels* are hidden, so that communications on these channels are internal to *Environment*.

EnvironmentLoop first initialises the state using another action *EnvironmentStateInit*, and then enters a loop, defined by a recursion that introduces a local name X (μX). In the body of the recursion, *EnvironmentLoop* performs *RobotMovementAction*, sketched in Figure 6.7. Afterwards, *EnvironmentLoop* proceeds to a choice (\square) that depends on whether $\text{stepTimer} < \text{timeStep}$ or not, that is, on the reason for interrupting *RobotMovementAction* (see Figure 6.1), and then recurses (X). Figures 6.7, 6.8, and 6.9 show actions of *Environment* that are omitted in Figure 6.6.

EnvironmentStateInit is a data operation, defined by a Z schema. Its declaration *EnvironmentState'* specifies dashed copies of the state components to represent the final state of the initialisation, that is, the initial values of the state components. The initial state of the *robot* is defined to be that

```

fireDetected_InputEventMapping  $\hat{=}$ 
  if( $\exists \text{fire1} : \text{ran fires} \bullet \neg (\text{distance}(\text{fire1.position}, \text{robot.position}) > 0.5)$ )  $\longrightarrow$ 
    fireDetectedTriggered!True  $\longrightarrow$  fireDetectedOccurred, fireDetectedTimer := True, time
  ||  $\neg (\exists \text{fire1} : \text{ran fires} \bullet \neg (\text{distance}(\text{fire1.position} - \text{robot.position}) > 0.5)) \longrightarrow$ 
    fireDetectedTriggered!False  $\longrightarrow$  Skip
fi

```

Figure 6.8: The *fireDetected_InputEventMapping* action for the firefighter example

specified in the global constant *robotInit*. The initial state of *fires* is defined by requiring that its range (elements, identified by *ran fires*) is a subset of the *potentialFires*. This ensures that all fires satisfy the constraints on *potentialFires*, without specifying the number of fires (which is undefined in the assumptions). The *time* and *stepTimer* components are initialised to 0.0, and the *EventTimes* components are initialised as defined in a separate schema *EventTimesInit* (omitted here): the timers are initialised to 0.0 and the boolean components to false.

RobotMovementAction specifies a state evolution using a special kind of schema, here with name *RobotMovement*, that is specifically available in *CyPhyCircus* (but not in *Z* or *Circus*). Such schemas are indicated by a Δ 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. For example, as shown in Figure 6.7, the *robot*'s *position* evolves with a derivative equal to its *velocity*; other equations are omitted. The *time* and *stepTimer* components evolve with a derivative of 1, so that it keeps track of the time in the environment. Every component in *EnvironmentState* not mentioned in the equations of *RobotMovement*, including the discrete components, remains the same throughout the evolution.

In *RobotMovementAction*, *RobotMovement* is interrupted (\triangle) by the detection of a collision or the *stepTimer* reaching the *timeStep*. The interruption condition is a disjunction covering four cases, two of which are shown in Figure 6.7. The first three cases are related to the robot colliding: with the ground, with the building, or with a fire. In each case, a collision is detected if the *robot*'s *position* is within the element it is colliding with, and the robot is moving towards that element. In Figure 6.7, we consider collision with the ground, so we require *robot.position* to belong to the *groundLocations*, and the third component of the *robot.velocity* vector (triple) to be negative, so that the robot is moving downwards. We note that the fires are treated as solid objects, since their dimensions are defined in the assumptions. The fourth disjunct of the interruption condition shown in Figure 6.7 is about the *stepTimer* reaching the *timeStep* (*stepTimer* \geq *timeStep*).

In *EnvironmentLoop*, a choice checks if the *stepTimer* has reached *timeStep*. If not, a *CollisionDetection* action offers another choice based on the three cases of collision described above. In all cases, the *robot* is simply stopped: its *velocity* and *acceleration* are set to 0.0.

If the *timeStep* is reached, trigger conditions for input events are checked in interleaving ($|||$), that is, independently, as defined by *InputTriggers* in Figure 6.6. For example, the conditions for *fireDetected* are checked by the action in *fireDetected_InputEventMapping*, in Figure 6.8.

$$\begin{aligned}
& fireDetected_Buffer \hat{=} \mathbf{var} fireDetectedTrig : \mathbb{B} \bullet fireDetectedTrig := \mathbf{False}; \\
& \mu X \bullet \left(\begin{array}{l} fireDetectedTriggered?b \longrightarrow fireDetectedTrig := b \\ \square \\ (fireDetectedTrig = \mathbf{True}) \& fireDetected.in \longrightarrow \mathbf{Skip} \end{array} \right); X \\
\\
& takeOff_Buffer \hat{=} takeOffHappened \longrightarrow takeOffOccurred, takeOffTime := \mathbf{True}, time; takeOff_Buffer
\end{aligned}$$

Figure 6.9: Some *_Buffer* actions for the firefighter example

In *fireDetected_InputEventMapping*, we have a choice based on whether there is a *fire1* such that the *distance* between its *position* and the *robot's position* is not greater than 0.5 (metres, since SI units are used in the semantics and already adopted in the IR). If the condition is fulfilled, **True** is signalled through the *fireDetectedTriggered* channel to communicate to *EventBuffers* the occurrence of *fireDetected* (as stated in the RoboWorld mapping – see Figure 3.2). Moreover, the state components for *fireDetected* (from *EventTimes*) are updated: the boolean *fireDetectedOccurred* is set to **True**, and the timer *fireDetectedTimer* is set to *time*. If the condition is not fulfilled, **False** is communicated on *fireDetectedTriggered* and the action terminates (**Skip**).

After *InputTriggers*, *Communication* repeatedly offers a choice of simple actions (omitted in Figure 6.6) that communicate (with the *Mapping* process) via the *getSetChannels* to get and set values for the state components. This is used by *Mapping* to capture the effect of output events and operations – see Figure 6.1. When *Mapping* is finished, for the current loop, it signals that via *proceed*. At that point the *stepTimer* is reset and *EnvironmentLoop* recurses.

EventBuffers is defined by the interleaving of two actions *InputEventBuffers* and *OutputEventBuffers*. These are themselves defined by the interleaving of a *_Buffer* action for each input or output event. These are similar, so we just present *fireDetected_Buffer* and *takeOff_Buffer* in Figure 6.9.

Regarding *fireDetected_Buffer*, it initialises the boolean state component for the event, here *fireDetectedTrig*, to **False**, then enters a recursion. In the body of the recursion it repeatedly offers a choice between accepting a new value from *EnvironmentLoop* via *fireDetectedTriggered* and storing it in *fireDetectedTrig*, and offering the *fireDetected.in* input (to the RoboChart process – see Figure 6.1) whenever *fireDetectedTrig* is **True**. Thus, the input event is offered after its triggering condition holds at the *timeStep*, until a *timeStep* where the condition for the event is no longer satisfied.

As illustrated in Figure 6.9 for *takeOff_Buffer*, the *_Buffer* action for an output event or operation call accepts a signal from the *Mapping* process via the *Happened* channel. Afterwards, it sets the corresponding state components for the event or operation, just like an input *_Buffer* action.

The *Mapping* process is defined by a parallelism of similar processes for each output event and operation synchronising on the channel *proceed*. The definition for our firefighter example is shown in Figure 6.10. For illustration, we show the process for the *goToBuilding* operation, called


```

process Mapping  $\hat{=}$  spray_OutputEventMapping [ [ proceed ] ] takeOff_OperationMapping
    [ [ proceed ] ] goToBuilding_OperationMapping [ [ proceed ] ] goHome_OperationMapping
    [ [ proceed ] ] searchFire_OperationMapping

process goToBuilding_OperationMapping  $\hat{=}$  begin

    goToBuilding_Semantics  $\hat{=}$  goToBuildingCall
         $\rightarrow$  getRobotPosition?robotPos  $\rightarrow$  getBuildingPosition?buildingPos
         $\rightarrow$  (setRobotVelocity!(1.0 * ((buildingPos - robotPos) / norm (buildingPos - robotPos))))  $\rightarrow$  Skip;
        proceed  $\rightarrow$  goToBuilding_Semantics

    goToBuilding_Monitor  $\hat{=}$  goToBuildingCall  $\rightarrow$  goToBuildingHappened  $\rightarrow$  goToBuilding_Monitor

    • goToBuilding_Semantics [ [ goToBuildingCall ] ] goToBuilding_Monitor

end

```

Figure 6.10: The *Mapping* and *goToBuilding_OperationMapping* processes for the firefighter UAV example

goToBuilding_OperationMapping, also in Figure 6.10.

The *_OperationMapping* and *_OutputEventMapping* processes are basic, but without state; their main actions are parallelisms of two other actions: a *_Semantics* action, to capture the mapping defined in the RoboWorld document, and a *_Monitor* action, to communicate with *EventBuffers*. They are both triggered by the *CyPhyCircus* event for the RoboWorld operation or event. In our example, this is the *CyPhyCircus* event *goToBuildingCall* for the operation *goToBuilding*.

As shown in Figure 6.10, the *goToBuilding_Semantics* action captures the semantics corresponding to the mapping definition “when the operation *goToBuilding* is called, the velocity of the robot is set to 1.0 m/s towards the building”. After *goToBuildingCall*, it obtains from *Environment* the position of the robot (*robotPos*) and of the building (*buildingPos*) via *get* channels. It then sets, via a *set* channel, the velocity of the robot to 1, multiplied by a normalised vector from the *robotPos* to *buildingPos*, representing 1.0 m/s towards the building.

When finished setting values as required to capture the mapping, a *_Semantics* action signals the *Environment* to *proceed* and recurses. Since all *Mapping* actions need to synchronise on *proceed*, the *Environment* proceeds only when all *_Semantics* actions are done.

A *_Monitor* action communicates with *EventBuffers* via *Happened* channels. In our example, *goToBuilding_Monitor*, after *goToBuildingCall*, communicates *goToBuildingHappened* to *EventBuffers* so that it can update timers, before recursing. The synchronisation between the *_Semantics* and the *_Monitor* actions ensures that they respond to the same event occurrence or operation call.

The semantics of a RoboWorld document can be generated automatically. Next, we discuss the

formalisation of the semantics, via generative rules that define semantic functions.

6.2 Semantics generation: transformation rules

In this section we present the rules for generating the semantics of a RoboWorld document from its IR presented in Section 5. The top-level Rule 10 defines the overall semantics as a *CyPhyCircus* section (that is, sequence of definitions). As in Section 5.2, the text in grey indicates terms of the metanotation describing how the output is constructed. The output of these rules is *CyPhyCircus*, describing the model, and is presented in black text.

Rule 10 defines the semantic function $\llbracket - \rrbracket_{RW}$ that characterises the *CyPhyCircus* section that includes all definitions needed to specify the top process *RWDocument* that captures the behaviours of the robot and environment elements allowed by the assumptions and mappings in a well-formed instance *rw* of the IR class *RWIntermediateRepresentation* given as argument.

The definition of Rule 10 uses functions defined by other rules to specify groups of definitions. The first, *typeDefinitions*, generates the property types for each element, such as *ArenaProperty* and *RobotProperty*. Afterwards, the channels are declared. The declarations for those signalling when an input has been triggered are defined by the function *eventTriggeredChannelDefinitions*, for those signalling when an operation or output has happened by *eventHappenedChannelDefinitions*, and for those for getting and setting the values of properties for each element by *getSetChannelDefinitions*. Finally, *proceed* is declared. Each function takes as argument the attributes of *rw* that contain the relevant information.

The constraints on elements are defined by an application of *elementGlobalAssumptions*. The declaration of *timeStep* is in the body of Rule 10 directly. It is followed by the definitions of the process *Environment*, of *Mapping* processes and of *Mapping* itself, and finally *RWDocument*. Each of these processes is characterised using further functions.

Environment is defined by *environmentProcess(rw)* specified by Rule 32. The definitions of the *Mapping* processes are characterised by for iterations over the *outputEventMappings* and *operationMappings* of *rw*. For each output or operation in these attributes, a process definition characterised by *outputMappingDefinition(output)* or *operationMappingDefinition(operation)* is included (see Figure 6.10 for an example). *Mapping* itself is characterised by *mappingProcess*, which also takes the attributes *outputEventMappings* and *operationMappings* as arguments, to define the parallelism of the *Mapping* processes (see Figure 6.10).

Finally, the *RWDocument* process is defined as the parallel composition of *Environment* and *Mapping*. The synchronisation set, *communicationEvents*, is defined in the *where* clause as the union of three sets: the get and set channels, defined by *getSetEvents*, the signals that output events have happened or operations have been called, defined by *eventHappenedSignals*, and *{proceed}*.

Rule 10. Semantics of RoboWorld Documents

```

[[rw : RWIntermediateRepresentation]] $\mathcal{RW}$  =

  typeDefinitions(rw.arena, rw.robot, rw.elements)
  eventTriggeredChannelDefinitions(rw.inputEventMappings, rw.variableMappings)
  eventHappenedChannelDefinitions(rw.outputEventMappings, rw.operationMappings)
  getSetChannelDefinitions(rw.robot, rw.elements)
  channel proceed
  elementGlobalAssumptions(rw.arena, rw.robot, rw.elements)

  |   timeStep :  $\mathbb{R}$ 
  process Environment  $\hat{=}$  environmentProcess(rw)
  for output in rw.outputEventMappings do
    outputMappingDefinition(output)
  end for
  for operation in rw.operationMappings do
    operationMappingDefinition(operation)
  end for
  process Mapping  $\hat{=}$  mappingProcess(rw.outputEventMappings, rw.operationMappings)
  process RWDocument  $\hat{=}$  (Environment [[communicationEvents]] Mapping) \ communicationEvents
where
  communicationEvents =
    getSetEvents(rw.robot, rw.elements)
     $\cup$  eventHappenedSignals(rw.outputEventMappings, rw.operationMappings)
     $\cup$  {proceed}

```

Rule 11. Type Definitions

```

typeDefinitions(arena : Arena, robot : Element, elements : Seq(Element)) =

  arenaTypeDefinitions(arena)
  robotTypeDefinitions(robot)
  for element in elementTypes do
    elementTypeDefinitions(element)
  end for

```

The rule for `typeDefinitions` is Rule 11. It calls `arenaTypeDefinitions()` to generate the *ArenaProperty* schema and the types it uses, and `robotTypeDefinitions()` to generate the *RobotProperty* schema and the types it uses. It then iterates over `elementTypes`, generating the types for each element in `elementTypeDefinitions()`.

As an example of one of these, we show the rule for `arenaTypeDefinitions()` (Rule 12), the rules for generating other type definitions are similar. This rule takes the `arena` passed to it, and first generates type definitions for each of the regions in the components of the `arena` using `elementTypeDefinitions()`. In our firefighter example, this generates the *HomeProperty* schema for the home region. Afterwards, a call to `attributeTypeDefinitions()` generates type definitions for any named types required by the attributes. Although the `arena` in our example has no attributes, `attributeTypeDefinitions()` is used to generate types for attributes of other elements, so it generates *Tank_of_waterType*, for example. After other types are generated, the *ArenaProperty* schema is defined. The *xwidth*, *ywidth* and *zwidth* components are generated depending on the dimension of the `arena`. Note that the shape does not need to be considered here, since it is always a box for the `arena`. The *gradient*, *windSpeed* and *locations* components, which all

Rule 12. Type Definitions for the Arena

```
arenaTypeDefinitions(arena : Arena) =
```

```
  for region in arena.components do
    elementTypeDefinitions(region)
  end for
  attributeTypeDefinitions(arena.attributes)
```

```
  ArenaProperty _____
```

```
    if arena.dimension = ThreeD then
      xwidth, ywidth, zwidth :  $\mathbb{R}$ 
    else if arena.dimension = TwoD then
      xwidth, ywidth :  $\mathbb{R}$ 
    else
      xwidth :  $\mathbb{R}$ 
    end if
    gradient, windSpeed :  $\mathbb{R}$ 
    locations :  $\mathbb{P}$  Position
    for region in arena.components do
      region.name : titleCase(region.name) Property
    end for
    for attribute in arena.attributes do
      attribute.name : attributeType(attribute.type)
    end for
```

```
    if arena.dimension = ThreeD then
      locations = {x : 0.0..xwidth; y : 0.0..ywidth; z : 0.0..zwidth}
    else if arena.dimension = TwoD then
      locations = {x : 0.0..xwidth; y : 0.0..ywidth}
    else
      locations = {x : 0.0..xwidth}
    end if
```

Rule 13. Type Definitions Needed for Attributes

```
attributeTypeDefinitions(attributes : Seq(Attribute)) =
```

```
  for attribute in attributes do
    if attribute.type instanceof Enumeration then
      titleCase(attribute.name)Type ::= ((Enumeration)attribute.type).variants[0]
      for variant in ((Enumeration)attribute.type).variants[1..] do
        | variant
      end for
    else if attribute.type instanceof Record then
      titleCase(attribute.name)Type _____
      for field in ((Record)attribute.type).fields do
        field.name : attributeType(field.type)
      end for
    end if
  end for
```

Rule 14. Type Definitions for the Robot

```
robotTypeDefinitions(robot : Element, dimension : Dimension) =
```

```
  if robot instanceof ElementDefinition then
    robotElementDefinitionTypeDefinitions((ElementDefinition)robot, dimension)
  else
    robotElementPModelTypeDefinitions((ElementPModel)robot, dimension)
  end if
```

arenas have are then generated. At the end of the schema, components for each region in the components of the arena and each attribute in the attributes of the arena. The types for the regions are the schemas formed from the name of the region with the first letter capitalised and appended with *Property*. The type for the attributes is generated by a function *attributeType*, converting the type representation in the IR into *CyPhyCircus* text and referencing the types declared in *attributeTypeDefinitions()* if needed.

The *attributeTypeDefinitions()* rule is Rule 13. It receives a sequence of attributes, and iterates over them, checking for any whose type is an Enumeration or a Record. For an Enumeration type, a Z notation free type is generated representing an enumeration, consisting of its variants separated by vertical bars. For a Record type, a Z schema is generated, with each of its fields as the components. In both cases, the name of the new type is formed from taking the name of the attribute, with the first letter capitalised, and appending *Type*.

After types are defined, channels are declared, in the function *eventTriggeredChannelDefinitions*, defined in Rule 20. It generates channels for the robot and each element, other than *Regions* (which are static, so always defined as global constants).

The rule for *elementGlobalAssumptions()*, which generates the global assumptions on elements following the channel definitions, is shown in Rule 26. It receives the arena, robot and elements from the IR, and first generates definitions of global constants recording the values of the properties for each of these, either the actual values for a static element or the initial values for

Rule 15. Type Definitions for an ElementDefinition Robot

```
robotElementDefinitionTypeDefinitions(robot : ElementDefinition, dimension : Dimension) =
```

```
  for component in robot.components do
    elementTypeDefinitions(component)
  end for
  attributeTypeDefinitions(robot.attributes)
```

RobotProperty

```
    elementSizeParameters(robot.shape, dimension)
    if robot.shape! = null then
      locations : P Position
    end if
    position : Position
    velocity : Velocity
    acceleration : Acceleration
    orientation : Orientation
    angularVelocity : AngularVelocity
    angularAcceleration : AngularAcceleration
    for component in robot.components do
      component.name : titleCase(component.name) Property
    end for
    for attribute in robot.attributes do
      attribute.name : attributeType(attribute.type)
    end for
    elementLocationsDefinition(robot.shape, dimension)
```

Rule 16. Type Definitions for an Element

```
elementTypeDefinitions(element : Element, dimension : Dimension) =
```

```
  if element instanceof ElementDefinition then
    elementDefinitionTypeDefinitions((ElementDefinition)element, dimension)
  else
    elementPModelTypeDefinitions((ElementPModel)element, dimension)
  end if
```

Rule 17. Type Definitions for an ElementDefinition Element

```
elementDefinitionTypeDefinitions(element : ElementDefinition, dimension : Dimension) =
```

```

  for component in robot.components do
    elementTypeDefinitions(component)
  end for
  attributeTypeDefinitions(element.attributes)

  RobotProperty
  elementSizeParameters(element.shape, dimension)
  if element.shape! = null then
    locations : P Position
  end if
  position : Position
  orientation : Orientation
  for component in element.components do
    region.name : titleCase(componentregion.name) Property
  end for
  for attribute in element.attributes do
    attribute.name : attributeType(attribute.type)
  end for

  elementLocationsDefinition(element.shape, dimension)

```

Rule 18. Fields for Size Parameters of an Element

```
elementSizeParameters(elementShape : Shape, dimension : Dimension) =
```

```

  if elementShape! = null then
    if dimension = ThreeD then
      if elementShape instanceof Box then
        xwidth, ywidth, zwidth : R
      else if elementShape instanceof Cylinder then
        radius, depth : R
      else
        radius : R
      end if
    else
      if elementShape instanceof Box then
        if dimension = TwoD then
          xwidth, ywidth : R
        else
          xwidth : R
        end if
      else
        radius : R
      end if
    end if
  end if

```

Rule 19. Definition of *locations* Set for an Element

elementLocationsDefinition(elementShape : shape, dimension : Dimension) =

```

    if elementShape! = null then
      if dimension = ThreeD then
        if elementShape instanceof Box then
          locations = boxLocations position orientation xwidth ywidth zwidth
        else if elementShape instanceof Cylinder then
          locations = cylinderLocations position orientation radius depth
        else
          locations = sphereLocations position orientation radius
        end if
      else
        if elementShape instanceof Box then
          if dimension = TwoD then
            locations = squareLocations position orientation xwidth ywidth
          else
            locations = lineLocations position orientation xwidth
          end if
        else
          locations = lineLocations position orientation radius
        end if
      end if
    end if
  end if

```

Rule 20. Input event trigger channel definitions

eventTriggeredChannelDefinitions(inputs : Seq(InputEventMappingIR),
variables : Seq(VariableMappingIR)) =

```

    for input in inputs do
      if inputHasCommunications(input) then
        channel input.nameTriggered :  $\mathbb{B}$   $\times$  inputCommunicationTypes(input)
      else
        channel input.nameTriggered :  $\mathbb{B}$ 
      end if
    end for

```

Rule 21. Output event happened channel definitions

eventHappenedChannelDefinitions(outputEvents : Seq(OutputEventMappingIR),
operations : Seq(OperationMappingIR)) =

```

    for outputEvent in outputEvents do
      channel outputEvent.nameHappened :  $\mathbb{B}$ 
    end for
    for operation in operations do
      channel operation.signature.nameHappened :  $\mathbb{B}$ 
    end for

```

Rule 22. Variable get/set channel definitions

```
getSetChannelDefinitions(robot : Element, elements : Seq(Element)) =
```

```

  if robot instanceof ElementDefinition
    robotGetSetChannelDefinitions((ElementDefinition)robot)
  else
    robotPModelGetSetChannelDefinitions((ElementPModel)robot)
  end if
  for element in elements do
    if element instanceof ElementDefinition
      if not (element instanceof Region) then
        elementGetSetChannelDefinitions((ElementDefinition)element)
      end if
    else
      elementPModelGetSetChannelDefinitions((ElementPModel)element)
    end if
  end for

```

Rule 23. Robot variable get/set channel definitions

```
robotGetSetChannelDefinitions(robot : ElementDefinition) =
```

```

  channel getRobotPosition : Position
  channel getRobotVelocity : Velocity
  channel getRobotAcceleration : Acceleration
  channel getRobotOrientation : Orientation
  channel getRobotAngularVelocity : AngularVelocity
  channel getRobotAngularAcceleration : AngularAcceleration
  channel setRobotPosition : Position
  channel setRobotVelocity : Velocity
  channel setRobotAcceleration : Acceleration
  channel setRobotOrientation : Orientation
  channel setRobotAngularVelocity : AngularVelocity
  channel setRobotAngularAcceleration : AngularAcceleration
  for attribute in robot.attributes do
    channel getRobotTitleCase(attribute.name) : attribute.type
    channel setRobotTitleCase(attribute.name) : attribute.type
  end for
  for component in robot.components do
    componentGetSetChannelDefinitions(component, "Robot")
  end for

```

Rule 24. ElementDefinition variable get/set channel definitions

```
elementGetSetChannelDefinitions(element : ElementDefinition) =
```

```

  channel getRobotPosition : Position
  channel getRobotOrientation : Orientation
  for attribute in robot.attributes do
    channel getRobotTitleCase(attribute.name) : attribute.type
    channel setRobotTitleCase(attribute.name) : attribute.type
  end for
  for component in robot.components do
    componentGetSetChannelDefinitions(component, "Robot")
  end for

```


Rule 25. Component variable get/set channel definitions

```

componentGetSetChannelDefinitions(element : ElementDefinition, namePrefix : Identifier) =

  if; element.plurality == PLURAL then
    for attribute in element.attributes do
      channel getqualifiedName  $\wedge$  titleCase(attribute.name)
        : qualifiedNameID  $\times$  attribute.type
    end for
    for component in element.components do
      componentGetSetChannelDefinitions(component, qualifiedName)
    end for
  else
    for attribute in element.attributes do
      channel getqualifiedName  $\wedge$  titleCase(attribute.name) : attribute.type
    end for
    for component in element.components do
      componentGetSetChannelDefinitions(component, qualifiedName)
    end for
  end if
where
  qualifiedName = namePrefix  $\wedge$  titleCase(element.name)

```

Rule 26. Global assumptions about elements

```

elementGlobalAssumptions(arena : Arena, robot : Element, elements : Seq(Element)) =

  elementGlobalDefinitions(arena, robot, elements)
  arenaGlobalAssumptions(arena)
  elementGlobalAssumptions(robot)
  for element in elements do
    elementGlobalAssumptions(element)
  end for

```

a dynamic element.

The function `environmentProcess` (Rule 32) defines the body of the *Environment* process, included between **begin** and **end**. The state and its initialisation are specified by two functions. The *EventTimes* and *EventsTimesInit* schemas are specified by the function `eventTimes`, which takes the `inputEventMappings`, the `outputEventMappings`, and the `operationMappings` as arguments, so it can identify the needed timers. The *EnvironmentState* and *EnvironmentStateInit* schemas are specified by `environmentState`, which receives the `robot` and `elements` as arguments.

The next functions applied in Rule 32 define the actions of *Environment*. The *RobotMovement* schema and the *RoboMovementAction* are characterised by `robotMovementAction`. *CollisionDetection* and the actions that it combines in choice, by `collisionDetectionAction`. *InputTriggers* and the *_InputEventMapping* actions are specified by `inputTriggersAction`. *Communication* is specified by `communicationAction`. The *_Buffer* actions and their compositions in *InputEventBuffers* and *OutputEventBuffers* are characterised in `inputEventBuffers` and `outputEventBuffers`. *EventBuffers* has the same definition for all documents (in terms of the actions mentioned above), so it is specified directly in Rule 32. Each function application takes the relevant attributes of `rw` as arguments.

Rule 27. Definition of global elements

elementGlobalDefinitions(arena : Arena, robot : Element, elements : Seq(Element)) =

```

    arena : ArenaProperty
    robotInit : RobotProperty
    for element in elements do
        if element.plurality = SINGULAR then
            if hasAttributes(element) then
                element.nameInit : titleCase(element.name)Property
            else
                element.name : titleCase(element.name)Property
            end if
        else
            if hasAttributes(element) then
                potentialTitleCase(plural(element.name)) : P titleCase(element.name)Property
            else
                plural(element.name) : seq titleCase(element.name)Property
            end if
        end if
    end for

```

Rule 28. Global Assumptions on the Arena

arenaGlobalAssumptions(arena : Arena) =

```

    shapeConstraints(arena.shape)
    if arena.gradient ≠ null then
        for constraint in arena.gradient.properties do
            generateConstraint(constraint)
        end for
    end if
    ...
    for region in arena.components do
        elementGlobalAssumptions(region)
    end for

```

Rule 29. Mapping Process

mappingProcess(outEvents : Seq(OutputEventMappingIR), operations : Seq(OperationMappingIR)) =

composeOutputEventMappings(outEvents) ||| composeOperationMappings(operations)

Rule 30. Composition of Output Event Mappings

composeOutputEventMappings(outEvents : Seq(OutputEventMappingIR)) =

```

    if #outEvents = 0 then
        Skip
    else if #outEvents = 1 then
        [[head(outEvents)]]OE.M
    else
        [[head(outEvents)]]OE.M ||| composeOutputEventMappings(tail(outEvents))
    end if

```

Rule 31. Composition of Operation Mappings

$\text{composeOperationMappings}(\text{operations} : \text{Seq}(\text{OperationMappingIR})) =$

```

  if #operations = 0 then
    Skip
  else if #operations = 1 then
     $\llbracket \text{head}(\text{operations}) \rrbracket_{\mathcal{O.M}}$ 
  else
     $\llbracket \text{head}(\text{operations}) \rrbracket_{\mathcal{O.M}} \parallel \text{composeOperationMappings}(\text{tail}(\text{operations}))$ 

```

Rule 32. Environment Process

$\text{environmentProcess}(\text{rw} : \text{RWIntermediateRepresentation}) =$

```

  begin
    eventTimes(rw.inputEventMappings, rw.outputEventMappings, rw.operationMappings)
    environmentState(rw.robot, rw.elements)
    robotMovementAction(rw.arena, rw.robot, rw.elements)
    collisionDetectionAction(rw.arena, rw.robot, rw.elements)
    inputTriggersAction(rw.inputEventMappings)
    communicationAction(rw.arena, rw.robot, rw.elements)
    inputEventBuffers(rw.inputEventMappings, rw.variableMappings)
    outputEventBuffers(rw.outputEventMappings, rw.operationMappings)
     $\text{EventBuffers} \triangleq \text{InputEventBuffers} \parallel \text{OutputEventBuffers}$ 
     $\text{EnvironmentLoop} \triangleq (\text{EnvironmentStateInit}) ; \mu X \bullet$ 
       $\left( \begin{array}{l} \text{RobotMovementAction;} \\ (time < timeStep) \ \& \ \text{CollisionDetection} \\ \square \\ (time \geq timeStep) \ \& \ \text{InputTriggers;} \ \text{Communication;} \ \text{stepTimer} := 0 \end{array} \right) ; X$ 
    triggerChannelsSet(rw.inputEventMappings, rw.variableMappings)
     $\bullet (\text{EnvironmentLoop} \llbracket \text{triggerChannels} \rrbracket \text{EventBuffers}) \setminus \text{triggerChannels}$ 
  end

```

Rule 33. Environment Process State

```
environmentState(arena : Arena, robot : Robot, elements : Seq(Element)) =
```

```

  EnvironmentState
  robot : RobotProperty
  for element in elements do
    if hasAttributes(element) then
      if element.plurality = PLURAL then
        element.name : seq titleCase(element.name)Property
      else
        element.name : titleCase(element.name)Property
      end if
    end if
  end for
  time : ℝ
  stepTimer : ℝ
  EventTimes

```

```
state EnvironmentState
```

EnvironmentLoop, which like *EventBuffers* is the same for all RoboWorld documents, is also defined in Rule 32. The *triggerChannels* set is defined by *triggerChannelsSet* and used as the synchronisation set for the main action specified after the • also directly in Rule 32.

The definition of *inputTriggersAction* uses applications of *inputTrigger*, defined in Rule 34, to specify the *_InputEventMapping* actions. As shown in Rule 34, it defines an action whose name is formed from the *name* of the *inputEvent* argument appended with *_InputEventMapping*.

The body of the action depends on the type of the *input* of the *inputEvent*; we show here the case for where it is an instance of *InputSometimesIR*, which, as already said, represents a conditional event. In this case, the action is a *CyPhyCircus* conditional (*if...fi*), with two branches: the first is guarded by the conjunction of the *conditions* for the *input*, specified by *generateConstraintConjunction*, and the second is guarded by the negation of that conjunction.

In the first branch, where the *conditions* hold, the occurrence of the *input* event is signalled via a channel named from the *name* of the *inputEvent*, appended with *Triggered*. Any values communicated by the *input* event must be sent to the buffer, so the actual communication depends on the number of *communications* for the *input* (*#inputEvent.input.communications*). Here, we show the case for when there are no *communications* so that only the value **True** is communicated to indicate the occurrence of the event. After the communication, an assignment is included. In this (multiple) assignment, a variable whose name is formed from the *name* of the *inputEvent*, appended with *Occurred*, is set to **True** (because the event has occurred), and a variable whose name is formed from the *name* of the *inputEvent*, appended with *Timer*, is set to the current *time*.

Rule 34. Input Trigger semantics

```

inputTrigger(inputEvent : InputEventMappingIR) =

  inputEvent.name _InputEventMapping  $\hat{=}$ 
    if inputEvent.input instanceof InputSometimesIR then
      if generateConstraintConjunction(((InputSometimesIR)inputEvent.input).conditions)  $\longrightarrow$ 
        if #((InputSometimesIR)inputEvent.input).communications = 0 then
          inputEvent.name Triggered!True  $\longrightarrow$ 
        else ...
      end if
      inputEvent.name Occurred, inputEvent.name Timer := True, time
    []  $\neg$  (generateConstraintConjunction(((InputSometimesIR)inputEvent.input).conditions))  $\longrightarrow$ 
      if #((InputSometimesIR)inputEvent.input).communications = 0 then
        inputEvent.name Triggered!False  $\longrightarrow$  Skip
      else ...
    end if
  fi
else if ...
end if

```

In the second branch, where the conjunction of the `conditions` do not hold, a communication to signal to the event buffer is generated, as in the first branch, but communicating the value **False**. After the communication, we have a **Skip** action, instead of an assignment.

The function `operationMappingDefinition` (used in Rule 10) is specified in Rule 35 to give the semantics for an operation mapping. It is a process named by appending the `name` of the `signature` of the argument `operation` with `_OperationMapping`. As defined in Rule 35, this process does not have a state, and its main action is always the parallel composition of `_Semantics` and `_Monitor` actions (also named after the `operation`) defined in Rule 35 (see Figure 6.10 for an example).

The body of the `_Semantics` action begins with a communication on the `operation`'s `Call` channel. The `parameters` of the `operation` are iterated over in a **for** loop, with each parameter added as an input (?) in the communication. After the communication, the semantics depends on the type of the `output` for the `operation`; here, we show the case for `OutputAlwaysIR`, which has statements but not conditions. In this case, communications with the *Environment* on `get` channels are included (to obtain the values of any state components required by the statements). For example, for `goToBuilding_Semantics` in Figure 6.10, these are communications on `getRobotPosition` and `getBuildingPosition` so that the positions of the robot and building are available. After the needed variables are obtained, the semantics of each of the `statements` is defined, composed in sequence (;). The sequential composition is wrapped in brackets so that all the variables and operation parameters are in scope for the semantics of all statements. Afterwards, a communication on `proceed` and a recursion of the `_Semantics` action are specified.

The body of the `_Monitor` action also begins with a communication on the `operation`'s `Call` channel, specified in the same way as for the `_Semantics` action. This is followed by a communication with the *Environment* on the `Happened` channel for the `operation` and a recursion

Rule 35. Operation Mapping Semantics

```
operationMappingDefinition(operation : OperationMappingIR)
```

```
process operation.signature.name_OperationMapping  $\hat{=}$  begin
```

```
operation.signature.name_Semantics  $\hat{=}$ 
```

```
operation.signature.nameCall(for param in operation.signature.parameters do ?param end for)  $\rightarrow$ 
```

```
if operation.output instanceof OutputAlwaysIR then
```

```
  getNeededVariablesStatement(((OutputAlwaysIR)operation.output).statements)
```

```
  (outputStatementSemantics(((OutputAlwaysIR)operation.output).statements[0])
```

```
  for statement in ((OutputAlwaysIR)operation.output).statements[1..] do
```

```
    ; outputStatementSemantics(statement)
```

```
  end for)
```

```
else if ...
```

```
end if ; proceed  $\rightarrow$  operation.signature.name_Semantics
```

```
operation.signature.name_Monitor  $\hat{=}$ 
```

```
operation.signature.nameCall(for param in operation.signature.parameters do ?param end for)  $\rightarrow$ 
```

```
operation.signature.nameHappened  $\rightarrow$  operation.signature.name_Monitor
```

```
• operation.signature.name_Semantics [ [ operation.signature.nameCall ] ] operation.signature.name_Monitor
```

```
end
```

of the *_Monitor* action. In the main action of the *_OperationMapping* process (after the •), the *_Semantics* and *_Monitor* actions synchronise on the *operation*'s *Call* channel.

We next present the RoboWorld tool.

7. RoboWorld in RoboTool: authoring RoboWorld d

Tool support for authoring RoboWorld documents is provided by a specific plug-in for RoboTool. It is developed in Java using the Eclipse Rich Client Platform (RCP) for developing general-purpose applications. Here, we provide an overview of the main distinguishing features of this plug-in, namely, extending the RoboWorld language to deal with project-specific vocabulary, in addition to the support provided to edit sentences adhering to the underlying grammar of RoboWorld.

In Figure 7.1, we show the main screen of the RoboWorld plug-in. As an Eclipse-based application, files are organised into projects, listed on the left panel. The highlighted project is the one for the firefighter example. When the user clicks on any `.env` file, the RoboWorld Editor opens. It has two tabs: Dictionary and RoboWorld Document. As the names suggest, the former allows editing the project-specific dictionary, and the latter writing assumptions and mappings. In Figure 7.1, we show the Dictionary. Using a tabular representation, we can extend the RoboWorld lexicon by adding words that are specific to the selected project. For that, it suffices to provide its category (for instance, N for nouns or A for adjectives, and so on), along with its inflection forms.

Whenever a new word is added to the dictionary, the plug-in automatically extends the RoboWorld lexicon for this project, as explained in Section 4.3, and recompiles all related grammars, according to the structure discussed in Section 4.1. This process is completely hidden from the user, who does not need to understand the underlying details, for instance, the GF syntax. Nevertheless, as we can see in the left-side of Figure 7.1, the underlying grammars (that is, `.gf` files) are listed within the project such that advanced users can still inspect their contents.

According to [12], there are two predominant paradigms when writing sentences to adhere to a CNL: structural and surface editing. In structural editing, the user mostly follow a structural

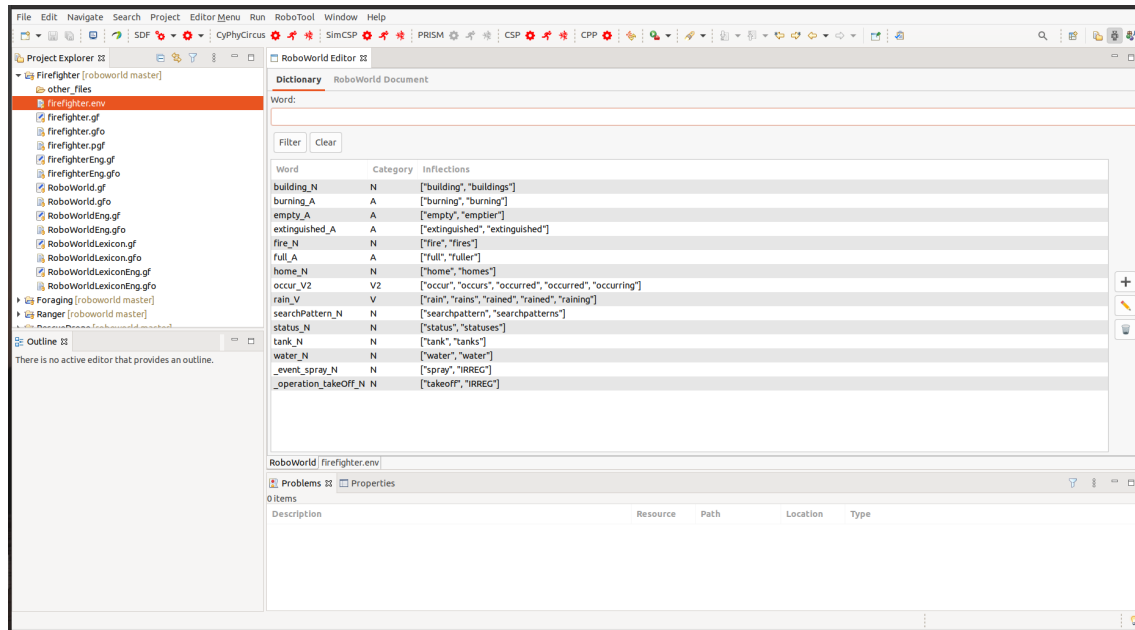


Figure 7.1: RoboWorld plug-in in RoboTool: dictionary editor

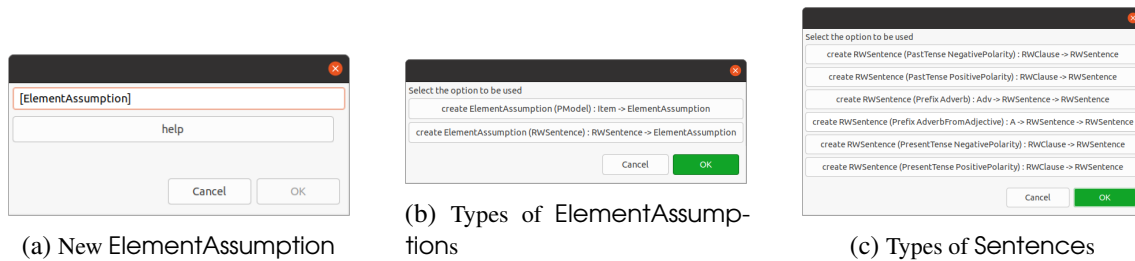


Figure 7.2: Combination of structural and surface editing

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.

The RoboWorld plug-in combines both paradigms. Depending on their expertise, users can adopt one paradigm or use a mix of both. At one side of the spectrum, sentences can be written freely, with the support of a typical syntax complete feature. At the other side, we can write sentences by selecting the desired structure among those supported (see Figure 7.2). The list of supported structures is dynamically built. If the dictionary is updated, the new words are listed. If the grammar evolves, the plug-in deals automatically with new versions. This is achieved by a dynamic integration between our plug-in and the underlying grammars, supported by the GF API.

In Figure 7.2, we illustrate our combination of the editing paradigms. Figure 7.2a is shown when we start writing a new element assumption. In the text field, between square brackets, we have the type of sentence being created: `ElementAssumption`. The user can then write the sentence freely, by just overwriting the text initially shown. However, we can select `ElementAssumption`

and click on Help. Figure 7.2b is then shown, indicating that there are two possible ways of describing an element assumption: using PModels or writing RWSentences. If we select the second possibility, Figure 7.2c is shown, listing the different ways of creating RWSentences (see Section 4.6). This guide goes until the lowest level of the grammar, when words (for instance, nouns, adjectives, and so on) are defined. At any point, if the user knows how to write a term of a specific grammatical category, this can be done by overwriting the text between square brackets.

Less experienced users initially benefit from the guide to write sentences, but with time the number of interactions with the writing guidance is likely to be reduced. The flexible combination of surface and structural editing supported by RoboTool suits users with different experience levels.

8. Conclusions

We have presented RoboWorld, a controlled natural language for documenting operational requirements of robotic systems. We have described the overall structure of a RoboWorld document using a metamodel, which is defined using elements of the English grammar, such as *Sentence*, *Noun*, and so on. A concrete grammar, defined using the Grammatical Framework, specifies in more detail the subset of the English language that is currently accepted. RoboWorld is a very flexible language, with an open vocabulary to define, for example, elements of the environment. Parsing creates an intermediate representation, and two sets of model-to-model transformation rules define a precise hybrid process-algebraic semantics written in *CyPhyCircus* for RoboWorld.

The concrete grammar is very powerful, allowing and enforcing correct use of inflections, for example. The parsing to an intermediate representation groups together the sentences that are relevant to each of the concepts primitive to RoboWorld: arena, robot, any additional entities, and so on. The first set of model-to-model rules enrich the intermediate representation to expose further structure in the sentences. They carry out a form of pre-processing to simplify the second transformation, from the intermediate representation to *CyPhyCircus*.

The intermediate representation can be a gateway to consider semantics in several notations. We have suggested here the translation of the *CyPhyCircus* models to hybrid automata for reasoning with a model checker. Another possibility is the direct generation of a hybrid automata semantics, which may be more suitable for model checking. Such a semantics might avoid the state explosion arising from the use of networks of automata to reflect the structure of processes. An automata model requires restrictions on the use of data types in the RoboWorld document, and is limited in terms of integration with richer (reactive or probabilistic, for example) semantics. It is, however,

appealing in terms of automated reasoning in the scope of what it can cover.

Use of RoboWorld can support several aspects of the design and verification of robotic systems, over and above the obvious advantage of documenting assumptions about the environment that are otherwise left explicit. RoboWorld sentences can be used to check the validity of different models and generated simulation code. For testing, this documentation can be used to prevent the generation of infeasible or useless test cases or, at least, eliminate such tests. Finally, operational requirements have an important role in proof, allowing us to establish properties that do not hold in any environment. In this paper, we have focussed on this latter form of application. We will, however, consider all above applications of RoboWorld in future work.

Our first line of future work, however, will push the limits of RoboWorld by considering additional case studies. RoboWorld is already very flexible: its vocabulary can be extended, and we cater for 96 different structures for writing sentences. Our tool takes advantage of well-established technology: the GF framework has been under development and use for more than 20 years. The support for document writing is in line with well accepted practice in the area [12]. We can either write documents in free form, or guided by a set of dialogues that enforce the required structure of sentences. We can benefit, nevertheless, of a usability study.

Regarding the semantics, *CyPhyCircus* is a hybrid process algebra, and the challenges of automated reasoning using hybrid models are many. Scalability requires theorem proving, and we can benefit from Isabelle/UTP, unique in that it builds on a widely used theorem prover and the UTP to support very rich hybrid, reactive, and concurrent models.

Automation can benefit from integrated use of theorem proving and model checking. To translate *CyPhyCircus* processes or actions to a hybrid automata notation accepted by model checkers, however, use of networks of hybrid automata is necessary. It avoids the construction of large models arising from flattening, and make the argument for soundness of translation much more direct. So, model checkers that are restricted to linear equations or do not support networks of automata are not powerful enough [15, 24]. In this respect, use CORA, as illustrated here, is a very promising option, which we will work to integrate with Isabelle/UTP to enhance proof automation.

A. Complete RoboWorld Grammar

A.1 RoboWorld.gf

```
1  -----
2  -- Abstract grammar of RoboWorld: a CNL for robotic systems
3  --
4  -- Authors:
5  -- * James Baxter <james.baxter@york.ac.uk>
6  --   (Department of Computer Science, University of York, UK)
7  -- * Gustavo Carvalho <ghpc@cin.ufpe.br> [corresponding author]
8  --   (Centro de Informática, Universidade Federal de Pernambuco, BR)
9  -- * Ana Cavalcanti <ana.cavalcanti@york.ac.uk>,
10 --   (Department of Computer Science, University of York, UK)
11 -----
12 abstract RoboWorld =
13     RoboWorldLexicon,
14     Numeral
15     **
16 {
17
18     -----
19     cat -- closed categories
20
21         Unit ;
22
23     -----
24     fun -- functions of closed categories
25
26         -- SI base units
```

```

27         m_Unit : Unit ;
28         meter_Unit : Unit ;
29         s_Unit : Unit ;
30         second_Unit : Unit ;
31         mole_Unit : Unit ;
32         a_Unit : Unit ;
33         ampere_Unit : Unit ;
34         k_Unit : Unit ;
35         kelvin_Unit : Unit ;
36         cd_Unit : Unit ;
37         candela_Unit : Unit ;
38         kg_Unit : Unit ;
39         kilogram_Unit : Unit ;
40
41         -- Other units
42         mm_Unit : Unit ;
43         millimeter_Unit : Unit ;
44         min_Unit : Unit ;
45         minute_Unit : Unit ;
46         ms_Unit : Unit ;
47         rads_Unit : Unit ;
48
49 -----
50 cat -- ItemPhrase
51
52         BasicItem ;
53         CompoundItem ;
54         Item ;
55         ItemPhrase ;
56         ItemPhraseList ;
57
58 -----
59 fun -- ItemPhrase
60
61         -- velocity
62         mkBasicItem_single_noun : Cat.N -> BasicItem ;
63         -- odometer value
64         mkBasicItem_two_nouns : Cat.N -> Cat.N -> BasicItem
65         ↪ ;
66         -- angular velocity
67         mkBasicItem_QualifiedBI : Cat.A -> BasicItem -> BasicItem
68         ↪ ;
69         -- m/s
70         mkBasicItem_Unit : Unit -> BasicItem ;
71
72         -- m/s upwards
73         mkCompoundItem_AdverbCI : Item -> Adv -> CompoundItem ;
74         -- object initially
75         mkCompoundItem_AdverbCI_from_adjective : Item -> A -> CompoundItem ;
76         -- distance from the robot to the nest

```

```

75     mkCompoundItem_PrepositionCI_single_ItemPhrase : BasicItem -> Prep ->
76         ↪ ItemPhrase -> CompoundItem ;
77     -- location except the source and the nest
78     mkCompoundItem_PrepositionCI_and_list_of_ItemPhrases : BasicItem -> Prep
79         ↪ -> ItemPhraseList -> CompoundItem ;
80     -- location except the source or the nest
81     mkCompoundItem_PrepositionCI_or_list_of_ItemPhrases : BasicItem -> Prep ->
82         ↪ ItemPhraseList -> CompoundItem ;
83
84     -- angular velocity
85     mkItem_from_BasicItem : BasicItem -> Item ;
86     -- angular velocity of the robot
87     mkItem_from_CompoundItem : CompoundItem -> Item ;
88
89     -- it
90     mkItemPhrase_PronounIP : Cat.Pron -> ItemPhrase ;
91     -- the angular velocity
92     mkItemPhrase_DeterminedIP : Cat.Det -> Item -> ItemPhrase
93         ↪ ;
94     -- 1 position
95     mkItemPhrase_QuantifiedIP_with_digits : Cat.Digits -> Item -> ItemPhrase
96         ↪ ;
97     -- one position
98     mkItemPhrase_QuantifiedIP_with_text : Cat.Numeral -> Item -> ItemPhrase
99         ↪ ;
100     -- 0.5 m
101     mkItemPhrase_QuantifiedIP_with_float : Float -> Item -> ItemPhrase
102         ↪ ;
103     -- at least 1 position
104     mkItemPhrase_AdN_QuantifiedIP_with_digits : AdN -> Cat.Digits -> Item ->
105         ↪ ItemPhrase ;
106     -- at least one position
107     mkItemPhrase_AdN_QuantifiedIP_with_text : AdN -> Cat.Numeral -> Item ->
108         ↪ ItemPhrase ;
109     -- at least 0.5 m
110     mkItemPhrase_AdN_QuantifiedIP_with_float : AdN -> Float -> Item ->
111         ↪ ItemPhrase ;
112     -- no obstacles
113     mkItemPhrase_QuantifiedIP_with_plural_Quant : Cat.Quant -> Item ->
114         ↪ ItemPhrase ;
115     -- this obstacle
116     mkItemPhrase_QuantifiedIP_with_singular_Quant : Cat.Quant -> Item ->
117         ↪ ItemPhrase ;
118     -- 0.0
119     mkItemPhrase_Float_Literal : Float -> ItemPhrase ;
120
121     -- [an x-width of 1 m, an y-width of 1 m]
122     mkItemPhraseList_binary : ItemPhrase -> ItemPhrase -> ItemPhraseList ;
123     -- [an x-width of 1 m, an y-width of 1 m, an z-width of 1 m]
124     mkItemPhraseList_many : ItemPhrase -> ItemPhraseList -> ItemPhraseList ;

```

```

113
114 -----
115 cat -- RWClause
116
117     RWClause ;
118
119 -----
120 fun -- RWClause
121
122     -- the odometer of the robot is reset
123     mkRWClause_PassiveVoice_IntransitiveVerb : ItemPhrase -> V -> RWClause ;
124     -- the velocity of the robot is set to 1 m/s upward
125     mkRWClause_PassiveVoice_TransitiveVerb_Preposition_ItemPhrase : ItemPhrase
126     ↪ -> V2 -> Prep -> ItemPhrase -> RWClause ;
127
128     -- the arena is three-dimensional
129     mkRWClause_ActiveVoice_ToBe_Adjective : ItemPhrase -> A -> RWClause
130     ↪ ;
131     -- the tank of water is either full or empty
132     mkRWClause_ActiveVoice_ToBe_Conj_Adjective_Adjective : ItemPhrase -> Conj
133     ↪ -> A -> A -> RWClause ;
134     -- the gradient of the ground is 0.0
135     mkRWClause_ActiveVoice_ToBe_ItemPhrase : ItemPhrase -> ItemPhrase ->
136     ↪ RWClause ;
137     -- the robot is in the origin initially
138     mkRWClause_ActiveVoice_ToBe_Preposition_ItemPhrase : ItemPhrase -> Prep ->
139     ↪ ItemPhrase -> RWClause ;
140     -- the distance from the target to the origin is greater than 1 m
141     mkRWClause_ActiveVoice_ToBe_Comparison_ItemPhrase : ItemPhrase -> A ->
142     ↪ ItemPhrase -> RWClause ;
143
144     -- the robot places an object in the nest
145     mkRWClause_ActiveVoice_TransitiveVerb_ItemPhrase : ItemPhrase -> V2 ->
146     ↪ ItemPhrase -> RWClause ;
147
148     -- the robot may carry 1 object
149     mkRWClause_ActiveVoice_Modal_TransitiveVerb_ItemPhrase : ItemPhrase -> VV
150     ↪ -> V2 -> ItemPhrase -> RWClause ;
151     -- the nest may contain up to 5 objects
152     mkRWClause_ActiveVoice_Modal_TransitiveVerb_Prep_ItemPhrase : ItemPhrase
153     ↪ -> VV -> V2 -> Prep -> ItemPhrase -> RWClause ;
154
155     -- it is raining
156     mkRWClause_ActiveVoice_Progressive_IntransitiveVerb : ItemPhrase -> V ->
157     ↪ RWClause ;
158     -- the robot is carrying an object
159     mkRWClause_ActiveVoice_Progressive_TransitiveVerb_ItemPhrase : ItemPhrase
160     ↪ -> V2 -> ItemPhrase -> RWClause ;
161
162 -----

```



```

152     cat -- RWSentence
153
154         RWSentence ;
155         RWSentenceList ;
156         RWSentences ;
157
158     -----
159     fun -- RWSentence
160
161         -- initially the robot is in the origin
162         mkRWSentence_Prefix_AdverbFromAdjective : A -> RWSentence -> RWSentence ;
163         -- then the velocity of the robot is set to 1.0 m/s upward
164         mkRWSentence_Prefix_Adverb : Adv -> RWSentence -> RWSentence ;
165
166         -- it is raining
167         mkRWSentence_PresentTense_PositivePolarity : RWClause -> RWSentence ;
168         -- it is not raining
169         mkRWSentence_PresentTense_NegativePolarity : RWClause -> RWSentence ;
170         -- it was raining
171         mkRWSentence_PastTense_PositivePolarity : RWClause -> RWSentence ;
172         -- it was not raining
173         mkRWSentence_PastTense_NegativePolarity : RWClause -> RWSentence ;
174
175         -- [the odometer of the robot is reset, the velocity of the robot is set
176         ↪ to 1 m/s upward]
177         mkRWSentenceList_binary : RWSentence -> RWSentence -> RWSentenceList ;
178         -- [the robot places an object in the nest, the odometer of the robot is
179         ↪ reset, the velocity of the robot is set to 1 m/s upward]
180         mkRWSentencetList_many : RWSentence -> RWSentenceList -> RWSentenceList ;
181
182         -- the velocity of the robot is set to 1 m/s upward
183         mkRWSentences_single_sentence : RWSentence -> RWSentences ;
184         -- the odometer of the robot is reset, and the velocity of the robot is
185         ↪ set to 1 m/s upward
186         mkRWSentences_and_list_of_sentences : RWSentenceList -> RWSentences
187         ↪ ;
188         -- the event spray occurred in 3 minutes before or the operation takeOff
189         ↪ was called in 20 minutes before
190         mkRWSentences_or_list_of_sentences : RWSentenceList -> RWSentences ;
191
192     -----
193     cat -- Conditions
194
195         Conditions ;
196
197     -----
198     fun -- Conditions
199
200         -- when the distance from the robot to the source is less than 1 m,

```

```

196      -- the distance from the robot to the nest is more than 2 m and the robot
197      ↪ is carrying an object
198      mkConditions_Subj_RWSentences : Subj -> RWSentences -> Conditions ;
199
200 -----
201 cat -- ArenaAssumption
202
203      ArenaAssumption ;
204
205 -----
206 fun -- ArenaAssumption
207
208      -- some locations of the arena except the source and the nest contain 1
209      ↪ obstacles
210      mkArenaAssumption_RWSentence : RWSentence -> ArenaAssumption ;
211
212 -----
213 cat -- RobotAssumption
214
215      RobotAssumption ;
216
217 -----
218 fun -- RobotAssumption
219
220      -- the robot is a point mass
221      mkRobotAssumption_RWSentence : RWSentence -> RobotAssumption ;
222      -- the robot is defined by a diagram
223      mkRobotAssumption_PModel : RobotAssumption ;
224
225 -----
226 cat -- ElementAssumption
227
228      ElementAssumption ;
229
230 -----
231 fun -- ElementAssumption
232
233      -- the source has an x-width of 0.25 m and a y-width of 0.25 m
234      mkElementAssumption_RWSentence : RWSentence -> ElementAssumption ;
235      -- the room is defined by a diagram
236      mkElementAssumption_PModel : Item -> ElementAssumption ;
237
238 -----
239 cat -- InputEventMapping
240
241      InputEventMapping ;
242
243 -----
244 fun -- InputEventMapping

```

```

244      -- when the distance from the robot to an obstacle is less than 1 m the
245      ↪ event obstacle occurs
246      mkInputEventMapping_InputSometimes : String -> Conditions ->
247      ↪ InputEventMapping ;
248      -- when the distance from the robot to an obstacle is less than 1 m
249      -- the event obstacle occurs and it communicates the linear velocity of
250      ↪ the robot
251      mkInputEventMapping_InputSometimes_RWSentences : String -> Conditions ->
252      ↪ RWSentences -> InputEventMapping ;
253      -- the event angularSpeed is always available
254      mkInputEventMapping_InputAlways : String -> InputEventMapping ;
255      -- the event angularSpeed is always available and it communicates the
256      ↪ angular velocity of the robot
257      mkInputEventMapping_InputAlways_RWSentences : String -> RWSentences ->
258      ↪ InputEventMapping ;
259      -- the event transferred never happens
260      mkInputEventMapping_InputNever : String -> InputEventMapping ;
261
262      -----
263      cat -- OutputEventMapping
264
265      OutputEventMapping ;
266
267      -----
268      fun -- OutputEventMapping
269
270      -- when the event takeoff occurs if it is raining the velocity of the
271      ↪ robot is set to 2.0 m/s upward
272      mkOutputEventMapping_Sometimes : String -> Conditions -> RWSentences ->
273      ↪ OutputEventMapping ;
274      -- when the event takeoff occurs the velocity of the robot is set to 1 m/s
275      ↪ upward
276      mkOutputEventMapping_OutputAlways : String -> RWSentences ->
277      ↪ OutputEventMapping ;
278      -- when the event takeoff occurs nothing happens
279      mkOutputEventMapping_NoOutput : String -> OutputEventMapping ;
280      -- the event spray is defined by a diagram where one time unit is 1.0 s
281      mkOutputEventMapping_DiagrammaticOutput : String -> Float -> Unit ->
282      ↪ OutputEventMapping ;
283      -- when the event spray occurs if the tank of water is full the effect is
284      ↪ defined by a diagram where one time unit is 1.0 s
285      mkOutputEventMapping_DiagrammaticOutput_Conditions : String -> Conditions
286      ↪ -> Float -> Unit -> OutputEventMapping ;
287
288      -----
289      cat -- OperationMapping
290
291      OperationMapping ;
292
293      -----

```

```

281 fun -- OperationMapping
282
283     -- when the operation Store() is called
284     -- as soon as the distance from the robot to the source is less than 1.0 m
285     ⇨ the robot places an object in the nest
286
287 mkOperationMapping_Sometimes : String -> Conditions -> RWSentences ->
288     ⇨ OperationMapping ;
289
290     -- when the operation move(ls,as) is called
291     -- the velocity of the robot is set to ls m/s towards the orientation of
292     ⇨ the robot
293
294     -- and the angular velocity of the robot is set to as rad/s
295
296 mkOperationMapping_OutputAlways : String -> RWSentences ->
297     ⇨ OperationMapping ;
298
299     -- when the operation Transfer() is called nothing happens
300
301 mkOperationMapping_NoOutput : String -> OperationMapping ;
302
303     -- the operation turnBack() is defined by a diagram where one time unit is
304     ⇨ 1.0 s
305
306 mkOperationMapping_DiagrammaticOutput : String -> Float -> Unit ->
307     ⇨ OperationMapping ;
308
309     -- when the operation turnBack() is called
310
311     -- if it is raining the effect is defined by a diagram where one time unit
312     ⇨ is 1.0 s
313
314 mkOperationMapping_DiagrammaticOutput_Conditions : String -> Conditions ->
315     ⇨ Float -> Unit -> OperationMapping ;
316
317 -----
318
319 cat -- VariableMapping
320
321     VariableMapping ;
322
323 -----
324
325 fun -- VariableMapping
326
327     -- when the robot is on the floor the variable dist is incremented
328
329 mkVariableMapping_Conditions_RWSentences : Conditions -> RWSentence ->
330     ⇨ VariableMapping ;
331
332 -----
333
334 -- Help functions for RoboWorld plugin
335
336 fun _special_N : N;
337
338 fun _special_A : A;
339
340 fun _special_AdN : AdN;
341
342 fun _special_Adv : Adv;
343
344 fun _special_AdV : AdV;
345
346 fun _special_Conj : Conj;
347
348 fun _special_Quant : Quant;
349
350 fun _special_Prep : Prep;
351
352 fun _special_Pron : Pron;
353
354 fun _special_Subj : Subj;
355
356 fun _special_V : V;

```

```

322     fun _special_V2 : V2;
323     fun _special_VV : VV;
324
325     fun _special_empty_V : V;
326     fun _special_Unit : Unit;
327     fun _special_BasicItem : BasicItem;
328     fun _special_CompoundItem : CompoundItem;
329     fun _special_Item : Item;
330     fun _special_ItemPhrase : ItemPhrase;
331     fun _special_ItemPhraseList : ItemPhraseList;
332     fun _special_RWSentence : RWSentence;
333     fun _special_RWSentenceList : RWSentenceList;
334     fun _special_RWSentences : RWSentences;
335     fun _special_Conditions : Conditions;
336     fun _special_ArenaAssumption : ArenaAssumption;
337     fun _special_RobotAssumption : RobotAssumption;
338     fun _special_ElementAssumption : ElementAssumption;
339     fun _special_InputEventMapping : InputEventMapping;
340     fun _special_OutputEventMapping : OutputEventMapping;
341     fun _special_OperationMapping : OperationMapping;
342     fun _special_VariableMapping : VariableMapping;
343
344 }

```

A.2 RoboWorldEng.gf

```

1  -----
2  -- Concrete grammar of RoboWorld: a CNL for robotic systems
3  --
4  -- Authors:
5  -- * James Baxter <james.baxter@york.ac.uk>
6  --   (Department of Computer Science, University of York, UK)
7  -- * Gustavo Carvalho <ghpc@cin.ufpe.br> [corresponding author]
8  --   (Centro de Informática, Universidade Federal de Pernambuco, BR)
9  -- * Ana Cavalcanti <ana.cavalcanti@york.ac.uk>,
10 --   (Department of Computer Science, University of York, UK)
11 -----
12 concrete RoboWorldEng of RoboWorld =
13     RoboWorldLexiconEng,
14     NumeralEng
15     **
16 open
17     SyntaxEng,
18     (ResEng = ResEng),
19     ParadigmsEng,
20     SymbolicEng,
21     ExtraEng,
22     Prelude,
23     MorphoEng,

```

```

24 ParamX
25 in {
26
27 -----
28 lincat -- closed categories
29
30     Unit = CatEng.N ;
31
32 -----
33 lin -- functions of closed categories
34
35     -- SI base units
36     m_Unit = mkN "m" "m" ;
37     meter_Unit = mkN "meter" "meters" ;
38     s_Unit = mkN "s" "s" ;
39     second_Unit = mkN "second" "seconds" ;
40     mole_Unit = mkN "mole" "moles" ;
41     a_Unit = mkN "A" "A" ;
42     ampere_Unit = mkN "ampere" "amperes" ;
43     k_Unit = mkN "K" "K" ;
44     kelvin_Unit = mkN "kelvin" "kelvins" ;
45     cd_Unit = mkN "cd" "cd" ;
46     candela_Unit = mkN "candela" "candelas" ;
47     kg_Unit = mkN "kg" "kg" ;
48     kilogram_Unit = mkN "kilogram" "kilograms" ;
49
50     -- Other units
51     mm_Unit = mkN "mm" "mm" ;
52     millimeter_Unit = mkN "millimeter" "millimeters" ;
53     min_Unit = mkN "min" "min" ;
54     minute_Unit = mkN "minute" "minutes" ;
55     ms_Unit = mkN "m/s" "m/s" ;
56     rads_Unit = mkN "rad/s" "rad/s" ;
57
58 -----
59 lincat -- ItemPhrase
60
61     BasicItem = CatEng.CN ;
62     CompoundItem = CatEng.CN ;
63     Item = CatEng.CN ;
64     ItemPhrase = CatEng.NP ;
65     ItemPhraseList = ListNP ;
66
67 -----
68 lin -- ItemPhrase
69
70     mkBasicItem_single_noun n =
71         -- mkCN : N -> CN / velocity
72         mkCN (lin N n) ;
73

```

```

74     mkBasicItem_two_nouns n1 n2 =
75         let
76             -- odometer
77             str : Str = (n1.s ! ResEng.Sg ! ResEng.Nom) ;
78             -- mkN : Str -> N -> N | odometer, value
79             n : N = mkN str (lin N n2) ;
80         in
81             mkCN n ;
82
83     mkBasicItem_QualifiedBI adj cn =
84         -- mkCN : A -> CN -> CN | angular, velocity
85         mkCN <lin A adj : A> <cn : CN> ;
86
87     mkBasicItem_Unit unit =
88         -- mkCN : N -> CN | m/s
89         mkCN unit ;
90
91     mkCompoundItem_AdverbCI item adv =
92         -- mkCN : CN -> Adv -> CN | m/s, upwards
93         mkCN <item : CN> <lin Adv adv : Adv> ;
94
95     mkCompoundItem_AdverbCI_from_adjective item adj =
96         let
97             -- mkAdv : A -> Adv | initial
98             adv : CatEng.Adv = SyntaxEng.mkAdv (lin A adj)
99         in
100             -- mkCN : CN -> Adv -> CN | object, initially
101             mkCN item adv ;
102
103     mkCompoundItem_PrepositionCI_single_ItemPhrase basic prep itemPhrase =
104         let
105             -- mkAdv : Prep -> NP -> Adv | from, the robot to the nest
106             adv : CatEng.Adv = SyntaxEng.mkAdv (lin Prep prep)
107             ⇨ itemPhrase
108         in
109             -- mkCN : CN -> Adv -> CN | distance, from the robot to
110             ⇨ the nest
111             mkCN basic adv ;
112
113     mkCompoundItem_PrepositionCI_and_list_of_ItemPhrases basic prep list =
114         let
115             -- mkNP : Conj -> ListNP -> NP | and, [the source, the
116             ⇨ nest]
117             npAndList : NP = mkNP and_Conj list ;
118             -- mkAdv : Prep -> NP -> Adv | except, the source and the
119             ⇨ nest
120             adv : CatEng.Adv = SyntaxEng.mkAdv (lin Prep prep)
121             ⇨ npAndList ;
122         in

```

```

118      -- mkCN : CN -> Adv -> CN / location, except the source
119      ↪ and the nest
120      mkCN basic adv ;
121
122      mkCompoundItem_PrepositionCI_or_list_of_ItemPhrases basic prep list =
123      let
124          -- mkNP : Conj -> ListNP -> NP / or, [the source, the
125          ↪ nest]
126          npAndList : NP = mkNP or_Conj list ;
127          -- mkAdv : Prep -> NP -> Adv / except, the source or the
128          ↪ nest
129          adv : CatEng.Adv = SyntaxEng.mkAdv (lin Prep prep)
130          ↪ npAndList ;
131      in
132          -- mkCN : CN -> Adv -> CN / location, except the source or
133          ↪ the nest
134          mkCN basic adv
135          ↪ ;
136
137      mkItem_from_BasicItem basicItem =
138      basicItem ; -- angular velocity
139
140      mkItem_from_CompoundItem compoundItem =
141      compoundItem ; -- angular velocity of the robot
142
143      mkItemPhrase_PronounIP pron =
144      -- mkNP : Pron -> NP / it
145      mkNP <lin Pron pron : Pron> ;
146
147      mkItemPhrase_DeterminedIP det item =
148      -- mkNP : Det -> CN -> NP / the, angular velocity
149      mkNP <lin Det det : Det> <item : CN> ;
150
151      mkItemPhrase_QuantifiedIP_with_digits digits item =
152      let
153          -- 1
154          det : Det = (mkDet <(lin Digits digits) : Digits>) ;
155      in
156          -- mkNP : Det -> CN -> NP / 1, position
157          mkNP det item ;
158
159      mkItemPhrase_QuantifiedIP_with_text numeral item =
160      let
161          -- one
162          det : Det = (mkDet <(lin Numeral numeral) : Numeral>) ;
163      in
164          -- mkNP : Det -> CN -> NP / one, position
165          mkNP det item ;
166
167      mkItemPhrase_QuantifiedIP_with_float float item =

```



```

162         let
163             -- mkSymb : Str -> Symb | 0.5
164             sym : Symb = mkSymb float.s ;
165             -- symb : Symb -> Card | 0.5
166             card : Card = symb sym ;
167             -- mkDet : Card -> Det | 0.5
168             det : Det = mkDet card ;
169         in
170             -- mkNP : Det -> CN -> NP | 0.5, m
171             mkNP det item ;
172
173     mkItemPhrase_AdN_QuantifiedIP_with_digits adn digits item =
174         let
175             -- mkCard : Digits -> Card | 1
176             card : Card = mkCard <(lin Digits digits) : Digits> ;
177             -- mkCard : AdN -> Card -> Card | at least, 1
178             adnCard : Card = mkCard <(lin AdN adn) : AdN> card ;
179             -- mkDet : Card -> Det | at least 1
180             det : Det = mkDet adnCard ;
181         in
182             -- mkNP : Det -> CN -> NP | at least 1, position
183             mkNP det item ;
184
185     mkItemPhrase_AdN_QuantifiedIP_with_text adn numeral item =
186         let
187             -- mkCard : Digits -> Card | one
188             card : Card = mkCard <(lin Numeral numeral) : Numeral> ;
189             -- mkCard : AdN -> Card -> Card | at least, one
190             adnCard : Card = mkCard <(lin AdN adn) : AdN> card ;
191             -- mkDet : Card -> Det | at least one
192             det : Det = mkDet adnCard ;
193         in
194             -- mkNP : Det -> CN -> NP | at least one, position
195             mkNP det item ;
196
197     mkItemPhrase_AdN_QuantifiedIP_with_float adn float item =
198         let
199             -- mkSymb : Str -> Symb | 0.5
200             sym : Symb = mkSymb float.s ;
201             -- symb : Symb -> Card | 0.5
202             card : Card = symb sym ;
203             -- mkCard : AdN -> Card -> Card | at least 0.5
204             adnCard : Card = mkCard <(lin AdN adn) : AdN> card ;
205             -- mkDet : Card -> Det | 0.5
206             det : Det = mkDet adnCard ;
207         in
208             -- mkNP : Det -> CN -> NP | at least 0.5, m
209             mkNP det item ;
210
211     mkItemPhrase_QuantifiedIP_with_plural_Quant quant item =

```

```

212         let
213             -- mkDet : Quant -> Num -> Det / no, 'plNum'
214             det : Det = mkDet <(lin Quant quant) : Quant> plNum
215         in
216             -- mkNP : Det -> CN -> NP / no, obstacle
217             mkNP det item ;
218
219     mkItemPhrase_QuantifiedIP_with_singular_Quant quant item =
220         let
221             -- mkDet : Quant -> Num -> Det / this, 'sgNum'
222             det : Det = mkDet <(lin Quant quant) : Quant> sgNum
223         in
224             -- mkNP : Det -> CN -> NP / this, obstacle
225             mkNP det item ;
226
227     mkItemPhrase_Float_Literal float =
228         -- symb : Float -> NP / 0.0
229         symb float ;
230
231     mkItemPhraseList_binary itemPhrase1 itemPhrase2 =
232         -- mkListNP : NP -> NP -> ListNP / an x-width of 1 m, a y-width of
233         --           ↪ 1 m
234         mkListNP itemPhrase1 itemPhrase2 ;
235
236     mkItemPhraseList_many itemPhrase itemPhraseList =
237         -- mkListNP : NP -> ListNP -> ListNP / an x-width of 1 m, [a
238         --           ↪ y-width of 1 m, a z-width of 1 m]
239         mkListNP itemPhrase itemPhraseList ;
240
241     -----
242     lincat -- RWClause
243
244         RWClause = CatEng.Cl ;
245
246     -----
247     lin -- RWClause
248
249         mkRWClause_PassiveVoice_IntransitiveVerb itemPhrase v =
250             let
251                 -- mkV2 : V -> V2 / reset
252                 -- passiveVP : V2 -> VP /
253                 --           ↪ reset
254                 vp : VP = passiveVP (mkV2 <(lin V v) : V>)
255                 --           ↪ ;
256             in
257                 -- mkCl : NP -> VP -> Cl / the odometer of the robot, is
258                 --           ↪ reset
259                 mkCl itemPhrase vp ;

```

```

256 mkRWClause_PassiveVoice_TransitiveVerb_Preposition_ItemPhrase itemPhrase1
    ↪ v2 prep itemPhrase2 =
257     let
258         -- passiveVP : V2 -> VP / set
259         passiveVerb : VP = passiveVP (<(lin V2 v2) : V2>) ;
260         -- mkAdv : Prep -> NP -> Adv / to, 1 m/s upward
261         adv : CatEng.Adv = SyntaxEng.mkAdv (lin Prep prep)
                ↪ itemPhrase2 ;
262         -- mkVP : VP -> Adv -> VP / is set, to 1 m/s upward
263         vp : VP = mkVP passiveVerb adv ;
264     in
265         -- mkCl : NP -> VP -> Cl / the velocity of the robot, is
                ↪ set to 1 m/s upward
266         mkCl itemPhrase1 vp ;
267
268 mkRWClause_ActiveVoice_ToBe_Adjective itemPhrase adj =
269     -- mkCl : NP -> A -> Cl / the arena, three-dimensional
270     mkCl itemPhrase (lin A adj) ;
271
272 mkRWClause_ActiveVoice_ToBe_Conj_Adjective_Adjective itemPhrase conj adj1
    ↪ adj2 =
273     let
274         -- mkAP : A -> AP / full
275         ap1 : AP = <lin AP (mkAP <lin A adj1 : A>) : AP> ;
276         -- mkAP : A -> AP / empty
277         ap2 : AP = <lin AP (mkAP <lin A adj2 : A>) : AP> ;
278         -- mkAP : Conj -> AP -> AP -> AP / either ... or ...,
                ↪ full, empty
279         ap : AP = mkAP <lin Conj conj : Conj> ap1 ap2 ;
280     in
281         -- mkCl : NP -> AP -> Cl / the tank of water, either full
                ↪ or empty
282         mkCl itemPhrase ap ;
283
284 mkRWClause_ActiveVoce_ToBe_ItemPhrase itemPhrase1 itemPhrase2 =
285     -- mkCl : NP -> NP -> Cl / the gradient of the ground, 0.0
286     mkCl itemPhrase1 itemPhrase2 ;
287
288 mkRWClause_ActiveVoice_ToBe_Preposition_ItemPhrase itemPhrase1 prep
    ↪ itemPhrase2 =
289     let
290         -- mkAdv : Prep -> NP -> Adv / at the origin initially
291         adv : CatEng.Adv = SyntaxEng.mkAdv (lin Prep prep)
                ↪ itemPhrase2 ;
292     in
293         -- mkCl : NP -> Adv -> Cl / the robot, at the origin
                ↪ initially
294         mkCl itemPhrase1 adv ;
295

```

```

296 mkRWClause_ActiveVoice_ToBe_Comparison_ItemPhrase itemPhrase1 adj
    ↪ itemPhrase2 =
297     let
298         -- mkAP : A -> NP -> AP / great, 1 m
299         ap : AP = mkAP (lin A adj) itemPhrase2 ;
300     in
301         -- mkCl : NP -> AP -> Cl / the distance from the target to
302         ↪ the origin, greater than 1 m
303         mkCl itemPhrase1 ap ;
304
305 mkRWClause_ActiveVoice_TransitiveVerb_ItemPhrase itemPhrase1 v2
    ↪ itemPhrase2 =
306     let
307         -- mkVP : V2 -> NP -> VP / place, an object in the nest
308         vp : VP = mkVP <(lin V2 v2) : V2> itemPhrase2 ;
309     in
310         -- mkCl : NP -> VP -> Cl / the robot, place an object in
311         ↪ the nest
312         mkCl itemPhrase1 vp ;
313
314 mkRWClause_ActiveVoice_Modal_TransitiveVerb_ItemPhrase itemPhrase1 vv v2
    ↪ itemPhrase2 =
315     let
316         -- mkVP : V2 -> NP -> VP / carry, 1 object
317         vp : VP = mkVP <(lin V2 v2) : V2> itemPhrase2 ;
318     in
319         -- mkCl : NP -> VV -> VP -> Cl / the robot, may, carry 1
320         ↪ object
321         mkCl itemPhrase1 <(lin VV vv) : VV> vp ;
322
323 mkRWClause_ActiveVoice_Modal_TransitiveVerb_Prep_ItemPhrase itemPhrase1 vv
    ↪ v2 prep itemPhrase2 =
324     let
325         -- ss : Str -> SS / up to
326         preDet : Predet = <(lin Predet (ss prep.s)) : Predet> ;
327         -- mkNP : Predet -> NP -> NP / up to, 5 objects
328         np : NP = mkNP preDet itemPhrase2 ;
329         -- mkVP : V2 -> NP -> VP / contain, up to 5 objects
330         vp : VP = mkVP <(lin V2 v2) : V2> np ;
331     in
332         -- mkCl : NP -> VV -> VP -> Cl / the nest, may, contain up
333         ↪ to 5 objects
334         mkCl itemPhrase1 <(lin VV vv) : VV> vp ;
335
336 mkRWClause_ActiveVoice_Progressive_IntransitiveVerb itemPhrase v =
337     let
338         -- mkVP : V -> VP / rain
339         -- progressiveVP : VP -> VP / rain
340         progressive : VP = progressiveVP (mkVP <(lin V v) : V>) ;
341     in

```

```

338         -- mkCl : NP -> VP -> Cl / it, is raining
339         mkCl itemPhrase progressive ;
340
341     mkRWClause_ActiveVoice_Progressive_TransitiveVerb_ItemPhrase itemPhrase1 v2
342     ⇨ itemPhrase2 =
343         let
344             -- mkVP : V2 -> NP -> VP / carry, an object
345             -- progressiveVP : VP -> VP / carry an object
346             progressive : VP = progressiveVP (mkVP <(lin V2 v2) : V2>
347                 ⇨ itemPhrase2) ;
348         in
349             -- mkCl : NP -> VP -> Cl / the robot, is carrying an object
350             mkCl itemPhrase1 progressive ;
351
352 -----
353 lincat -- RWSentence
354
355     RWSentence = CatEng.S ;
356     RWSentenceList = ListS ;
357     RWSentences = CatEng.S ;
358
359 -----
360 lin -- Sentence
361
362     mkRWSentence_Prefix_AdverbFromAdjective a sentence =
363         let
364             -- mkAdv : A -> Adv / initial
365             adv : CatEng.Adv = SyntaxEng.mkAdv <(lin A a) : A> ;
366         in
367             -- mkS : Adv -> S -> S / initial, the robot is in the
368             ⇨ origin
369             mkS adv sentence ;
370
371     mkRWSentence_Prefix_Adverb adv sentence =
372         -- mkS : Adv -> S -> S / then, the velocity of the robot is set to
373         ⇨ 1.0 m/s upward
374         mkS <(lin Adv adv) : Adv> <sentence : S> ;
375
376     mkRWSentence_PresentTense_PositivePolarity clause =
377         -- mkS : Cl -> S / it is raining
378         mkS clause ;
379
380     mkRWSentence_PresentTense_NegativePolarity clause =
381         -- mkS : Pol -> Cl -> S / UncNeg, it is not raining
382         mkS UncNeg clause ;
383
384     mkRWSentence_PastTense_PositivePolarity clause =
385         -- mkS : Tense -> Cl -> S / pastTense, it was raining
386         mkS pastTense clause ;
387
388

```

```

384     mkRWSentence_PastTense_NegativePolarity clause =
385         -- mkS : Tense -> Pol -> Cl -> S / pastTense, UncNeg, it was not
386         ↪ raining
387     mkS pastTense UncNeg clause ;
388
389     mkRWSentenceList_binary sentence1 sentence2 =
390         -- mkListS : S -> S -> ListS / the odometer of the robot is reset,
391         ↪ the velocity of the robot is set to 1 m/s upward
392     mkListS sentence1 sentence2 ;
393
394     mkRWSentencetList_many sentence sentenceList =
395         -- mkListS : S -> ListS -> ListS / the robot places an object in
396         ↪ the nest,
397         --
398         ↪
399         ↪ [the odometer of the robot is reset, the velocity of the robot
400         ↪ is set to 1 m/s upward]
401     mkListS sentence sentenceList ;
402
403     mkRWSentences_single_sentence sentence =
404         sentence ; -- the velocity of the robot is set to 1 m/s upward
405
406     mkRWSentences_and_list_of_sentences sentenceList =
407         -- mkS : Conj -> ListS -> S / and_Conj, [the odometer of the robot
408         ↪ is reset, the velocity of the robot is set to 1 m/s upward]
409     mkS and_Conj sentenceList ;
410
411     mkRWSentences_or_list_of_sentences sentenceList =
412         -- mkS : Conj -> ListS -> S / or_Conj,
413         -- [the event
414         ↪ spray occurred in 3 minutes before,
415         -- the
416         ↪ operation takeOff was called in 20 minutes before]
417     mkS or_Conj sentenceList ;
418
419     -----
420     lincat -- Conditions
421
422     Conditions = CatEng.Adv ;
423
424     -----
425     lin -- Conditions
426
427     mkConditions_Subj_RWSentences subj sentences =
428         -- mkAdv : Subj -> S -> Adv / when,
429         -- the distance from the robot to the
430         ↪ source is less than 1 m,
431         -- the distance from the robot to the
432         ↪ nest is more than 2 m and the robot is carrying an object
433     SyntaxEng.mkAdv <(lin Subj subj) : Subj> <sentences : S> ;

```

```

423
424 -----
425 -- auxiliary parameter types
426
427 param OutputType = OutputEvent | Operation ;
428
429 -----
430 oper -- auxiliary functions
431
432     is_defined_by_diagram : VP =
433         let
434             -- mkCN : N -> CN | diagram
435             -- mkNP : Det -> CN -> NP | a, diagram
436             np : NP = mkNP RoboWorldLexiconEng.a_Det (mkCN diagram_N)
437             ↪ ;
438         in
439             -- passiveVP : V2 -> NP -> VP | define, a diagram
440             passiveVP <(lin V2 define_V2) : V2> <(lin NP np) : NP> ;
441
442     the_Item_is_defined_by_diagram : Item -> S = \item ->
443         let
444             -- is_defined_by_diagram : VP
445             vp : VP = is_defined_by_diagram ;
446         in
447             -- mkNP : Det -> CN -> NP | the, robot
448             -- mkCl : NP -> VP -> Cl | the robot, defined by a diagram
449             -- mkS : Cl -> S | the robot is defined by a diagram
450             mkS (mkCl (mkNP <RoboWorldLexiconEng.the_Det : Det> <item
451                 ↪ : CN>) vp) ;
452
453     the_event_str : Str -> NP = \str ->
454         let
455             -- symb : Str -> NP | obstacle
456             -- mkCN : N -> NP -> CN | event, obstacle
457             cn : CN = mkCN RoboWorldLexiconEng.event_N (symb str) ;
458         in
459             -- mkNP : Det -> CN -> NP | the, event obstacle
460             mkNP RoboWorldLexiconEng.the_Det cn ;
461
462     the_operation_str : Str -> NP = \str ->
463         let
464             -- symb : Str -> NP | takeoff
465             -- mkCN : N -> NP -> CN | operation, takeoff
466             cn : CN = mkCN RoboWorldLexiconEng.operation_N (symb str)
467             ↪ ;
468         in
469             -- mkNP : Det -> CN -> NP | the, operation takeoff
470             mkNP RoboWorldLexiconEng.the_Det cn ;
471
472     the_event_str_is_always_available : Str -> Cl = \str ->

```

```

470         let
471             -- the_event_str : Str -> NP / angularSpeed
472             np : NP = the_event_str str ;
473             -- mkVP : A -> VP / available
474             -- mkVP : Adv -> VP -> VP / always, to be available
475             vp : VP = mkVP always_Adv (mkVP available_A) ;
476         in
477             -- mkCl : NP -> VP -> Cl / the event angularSpeed, to be
478             --   always available
479             mkCl np vp ;
480
481 when_the_event_Str_occurs : Str -> CatEng.Adv = \str ->
482     let
483         -- the_event_str : Str -> NP / takeoff
484         np : NP = the_event_str str ;
485         -- mkVP : V -> VP / occur
486         -- mkCl : NP -> VP -> Cl / the event takeoff, occur
487         -- mkS : Cl -> S / the event takeoff
488         --   occurs
489         s : S = mkS (mkCl np (mkVP RoboWorldLexiconEng.occure_V)) ;
490     in
491         -- mkAdv : Subj -> S -> Adv / when, the event takeoff
492         --   occurs
493         SyntaxEng.mkAdv RoboWorldLexiconEng.when_Subj s ;
494
495 when_the_operation_Str_is_called : Str -> CatEng.Adv = \str ->
496     let
497         -- the_event_str : Str -> NP / takeoff
498         np : NP = the_operation_str str ;
499         -- passiveVP : V2 -> VP / call
500         -- mkCl : NP -> VP -> Cl / the operation takeoff, to be
501         --   called
502         -- mkS : Cl -> S / the operation takeoff is called
503         s : S = mkS (mkCl np (passiveVP
504             --   RoboWorldLexiconEng.call_V2)) ;
505     in
506         -- mkAdv : Subj -> S -> Adv / when, the operation takeoff
507         --   is called
508         SyntaxEng.mkAdv RoboWorldLexiconEng.when_Subj s ;
509
510 nothing_happens : S =
511     -- mkVP : V -> VP / happen
512     -- mkCl : NP -> VP -> Cl / nothing, happen
513     -- mkS : Cl -> S / nothing happens
514     mkS (mkCl nothing_NP (mkVP happen_V)) ;
515
516 where_one_time_unit_is_Str_Unit : Str -> RoboWorldEng.Unit -> CatEng.Adv =
517     -- \str,unit ->
518     let
519         -- mkNumeral : Unit -> Numeral / n1_Unit

```



```

513         -- mkDet : Numeral -> Det / one
514         -- mkCN : N -> CN / unit
515         -- mkNP : CN -> NP / unit
516         -- mkCN : N -> NP -> CN / time, unit
517         -- mkNP : Det -> CN -> NP / one, time unit
518         one_time_unit : NP = mkNP (mkDet (mkNumeral n1_Unit))
           ↪ (mkCN time_N (mkNP (mkCN unit_1_N))) ;
519     -- mkSymb : Str -> Symb / 1.0
520     -- symb : Symb -> Card / 1.0
521     card : Card = symb (mkSymb str) ;
522     -- mkDet : Card -> Det / 1.0
523     -- mkNP : Det -> CN -> NP / 1.0, s
524     str_unit : NP = mkNP (mkDet card) <(lin CN unit) : CN> ;
525     -- mkComp : NP -> Comp / 1.0 s
526     comp : Comp = mkComp str_unit ;
527     -- mkVP : Comp -> VP / to be 1.0 s
528     vp : VP = mkVP comp ;
529     -- mkCl : NP -> VP -> Cl / one time unit, to be 1.0
           ↪ s
530     -- mkS : Cl -> S / one time unit is 1.0 s
531     s : S = mkS (mkCl one_time_unit vp) ;
532     in
533         -- mkAdv : Subj -> S -> Adv / where, one time unit is 1.0
           ↪ s
534         SyntaxEng.mkAdv where_Subj s ;
535
536     outputSentencePrefix_Adv = table {
537         OutputEvent => when_the_event_Str_occurs ;
538         Operation => when_the_operation_Str_is_called
539     } ;
540
541     outputSentencePrefix_NP = table {
542         OutputEvent => the_event_str ;
543         Operation => the_operation_str
544     } ;
545
546     output_sometimes : OutputType -> Str -> Conditions -> RWSentences -> S =
           ↪ \outputType, str, conditions, sentences ->
547         let
548             -- outputSentencePrefix_Adv / when the event str occurs OR
           ↪ when the operation str is called
549             adv : CatEng.Adv = (outputSentencePrefix_Adv ! outputType)
           ↪ str ;
550             -- mkS : Adv -> S -> S / if it is raining, the velocity of
           ↪ the robot is set to 2.0 m/s upward
551             s : S = mkS <conditions : Adv> <sentences : S> ;
552         in
553             -- mkS : Adv -> S -> S, when the event takeoff occurs,

```

```

554      -- if it is raining the velocity of
555      ↪ the robot is set to 2.0 m/s
556      ↪ upward
557      mkS adv s ;
558
559      output_always : OutputType -> Str -> RWSentences -> S = \outputType, str,
560      ↪ sentences ->
561      let
562      -- outputSentencePrefix_Adv / when the event str occurs OR
563      ↪ when the operation str is called
564      adv : CatEng.Adv = (outputSentencePrefix_Adv ! outputType)
565      ↪ str ;
566      in
567      -- mkS : Adv -> S -> S, when the event takeoff occurs, the
568      ↪ velocity of the robot is set to 1.0 m/s upward
569      mkS <adv : Adv> <sentences : S> ;
570
571      no_output : OutputType -> Str -> S = \outputType, str ->
572      let
573      -- outputSentencePrefix_Adv / when the event str occurs OR
574      ↪ when the operation str is called
575      adv : CatEng.Adv = (outputSentencePrefix_Adv ! outputType)
576      ↪ str ;
577      in
578      -- mkS : Adv -> S -> S / when the event takeoff occurs,
579      ↪ nothing happens
580      mkS adv nothing_happens ;
581
582      diagrammatic_output : OutputType -> Str -> Str -> RoboWorldEng.Unit -> S =
583      ↪ \outputType, str, float, unit ->
584      let
585      -- outputSentencePrefix_NP / the event str OR the
586      ↪ operation str
587      np : NP = (outputSentencePrefix_NP ! outputType) str ;
588      -- is_defined_by_diagram : VP
589      defined_by : VP = is_defined_by_diagram ;
590      -- where_one_time_unit_is_Str_Unit : Str -> Unit -> Adv
591      adv : CatEng.Adv = where_one_time_unit_is_Str_Unit float
592      ↪ (lin Unit unit) ;
593      -- mkVP : VP -> Adv -> VP ; to be defined by a diagram,
594      ↪ where one time unit is 1.0 s
595      vp : VP = mkVP defined_by adv ;
596      in
597      -- mkCl : NP -> VP -> Cl / the event spray, to be defined
598      ↪ by a diagram where one time unit is 1.0 s
599      -- mkS : Cl -> S / the event spray is defined by a diagram
600      ↪ where one time unit is 1.0 s
601      mkS (mkCl np vp);

```

```

588     diagrammatic_output_conditions : OutputType -> Str -> Conditions -> Str ->
589     ↪ RoboWorldEng.Unit -> S =
590     \outputType, str, conditions, float, unit ->
591         let
592             -- mkCN : N -> CN | effect
593             -- mkNP : Det -> CN -> NP | the, effect
594             np : NP = mkNP RoboWorldEng.the_Det (mkCN
595             ↪ RoboWorldEng.effect_N) ;
596             -- is_defined_by_diagram : VP
597             defined_by : VP = is_defined_by_diagram ;
598             -- where_one_time_unit_is_Str_Unit : Str -> Unit -> Adv
599             adv : CatEng.Adv = where_one_time_unit_is_Str_Unit float
600             ↪ (lin Unit unit) ;
601             -- mkVP : VP -> Adv -> VP ; to be defined by a diagram,
602             ↪ where one time unit is 1.0 s
603             vp : VP = mkVP defined_by adv ;
604             -- mkCl : NP -> VP -> Cl | the effect, to be defined by a
605             ↪ diagram where one time unit is 1.0 s
606             s1 = mkS (mkCl np vp) ;
607             -- mkS : Adv -> S -> S | if the tank of water is full, the
608             ↪ effect is defined by a diagram where one time unit is
609             ↪ 1.0 s
610             s2 : S = mkS <conditions : Adv> <s1 : S> ;
611             -- outputSentencePrefix_Adv | when the event str occurs OR
612             ↪ when the operation str is called
613             adv2 : CatEng.Adv = (outputSentencePrefix_Adv !
614             ↪ outputType) str ;
615         in
616             -- mkS : Adv -> S -> S | when the event spray occurs,
617             -- if the tank of water is full the
618             ↪ effect is defined by a diagram where one time unit is
619             ↪ 1.0 s
620             mkS adv2 s2 ;
621
622 -----
623 lincat -- ArenaAssumption
624
625     ArenaAssumption = CatEng.S ;
626
627 -----
628 lin -- ArenaAssumption
629
630     mkArenaAssumption_RWSentence sentence =
631         sentence ; -- some locations of the arena except the source and
632         ↪ the nest contain 1 obstacles
633
634 -----
635 lincat -- RobotAssumption
636
637     RobotAssumption = CatEng.S ;

```

```

626
627 -----
628 lin -- RobotAssumption
629
630 mkRobotAssumption_RWSentence sentence =
631     sentence ; -- the robot is a point mass
632
633 mkRobotAssumption_PModel =
634     let
635         -- mkBasicItem_single_noun : N -> BasicItem | robot
636         -- mkItem_from_BasicItem : BasicItem -> Item | robot
637         item : Item = mkItem_from_BasicItem
638             ↪ (mkBasicItem_single_noun RoboWorldLexiconEng.robot_N);
639     in
640         -- the_Item_is_defined_by_diagram : Item -> S | robot
641         the_Item_is_defined_by_diagram item;
642 -----
643 lincat -- ElementAssumption
644
645     ElementAssumption = CatEng.S ;
646 -----
647
648 lin -- ElementAssumption
649
650 mkElementAssumption_RWSentence sentence =
651     sentence ; -- the source has an x-width of 0.25 m and a y-width of
652     ↪ 0.25 m
653
654 mkElementAssumption_PModel item =
655     -- the_Item_is_defined_by_diagram : Item -> S | room
656     the_Item_is_defined_by_diagram (lin Item (lin CN item))
657     ↪ ;
658 -----
659 lincat -- InputEventMapping
660
661     InputEventMapping = CatEng.S ;
662 -----
663 lin -- InputEventMapping
664
665 mkInputEventMapping_InputSometimes str conditions =
666     let
667         -- the_event_str : Str -> NP | obstacle
668         np : NP = the_event_str str.s ;
669         -- mkVP : V -> VP | occur
670         vp : VP = mkVP RoboWorldLexiconEng.occure_V ;
671         -- mkCl : NP -> VP -> Cl | the event obstacle, occur
672         -- mkS : Cl -> S | the event obstacle occurs

```

```

673         s : S = mkS (mkCl np vp) ;
674     in
675         -- mkS : Adv -> S -> S / when the distance from the robot
676         --   ↳ to an obstacle is less than 1 m the event obstacle
677         --   ↳ occurs
678         mkS conditions s ;
679
680 mkInputEventMapping_InputSometimes_RWSentences str conditions sentences =
681     let
682         -- the_event_str : Str -> NP / obstacle
683         np : NP = the_event_str str.s ;
684         -- mkVP : V -> VP / occur
685         vp : VP = mkVP RoboWorldLexiconEng.occure_V ;
686         -- mkCl : NP -> VP -> Cl / the event obstacle, occur
687         -- mkS : Cl -> S / the event obstacle occurs
688         s : S = mkS (mkCl np vp) ;
689     in
690         -- mkListS : S -> S -> ListS / the event obstacle occurs,
691         --   ↳ it communicates the linear velocity of the robot
692         -- mkS : Conj -> List -> S / and, [the event obstacle
693         --   ↳ occurs, it communicates the linear velocity of the
694         --   ↳ robot]
695         -- mkS : Adv -> S -> S / when the distance from the robot
696         --   ↳ to an obstacle is less than 1 m,
697         --   ↳ the event obstacle occurs and it
698         --   ↳ communicates the linear velocity of the robot
699         mkS conditions (mkS and_Conj (mkListS s sentences)) ;
700
701 mkInputEventMapping_InputAlways str =
702     -- the_event_str_is_always_available : Str -> Cl / angularSpeed
703     -- mkS : Cl -> S / the event angularSpeed is always available
704     mkS (the_event_str_is_always_available str.s) ;
705
706 mkInputEventMapping_InputAlways_RWSentences str sentences =
707     let
708         -- the_event_str_is_always_available : Str -> Cl /
709         --   ↳ angularSpeed
710         -- mkS : Cl -> S / the event angularSpeed is always
711         --   ↳ available
712         s : S = mkS (the_event_str_is_always_available str.s) ;
713     in
714         -- mkListS : S -> S -> ListS / the event angularSpeed is
715         --   ↳ always available,
716         --   ↳ it communicates the angular
717         --   ↳ velocity of the robot
718         -- mkS : Conj -> List -> S / and,
719         --   ↳ [the event angularSpeed is always available,
720         --   ↳ it communicates the angular
721         --   ↳ velocity of the robot]

```

```

710         mkS and_Conj (mkListS s sentences) ;
711
712     mkInputEventMapping_InputNever str =
713         let
714             -- the_event_str : Str -> NP, transferred
715             np : NP = the_event_str str.s ;
716             -- mkVP : V -> VP | happen
717             -- mkVP : Adv -> VP -> VP | never, happen
718             vp : VP = mkVP never_AdV (mkVP happen_V) ;
719         in
720             -- mkCl -> NP -> VP -> Cl | the event transferred, never
721             -- mkS : Cl -> S | the event transferred never happens
722             mkS (mkCl np vp) ;
723
724     -----
725     lincat -- OutputEventMapping
726
727         OutputEventMapping = CatEng.S ;
728
729     -----
730     lin -- OutputEventMapping
731
732     mkOutputEventMapping_Sometimes eventName conditions sentences =
733         -- output_sometimes : OutputType -> Str -> Conditions ->
734         --   ↳ RWSentences -> S |
735         -- OutputEvent, takeoff, if it is raining, the velocity of the
736         --   ↳ robot is set to 2.0 m/s upward
737         output_sometimes OutputEvent eventName.s (lin Conditions
738         --   ↳ conditions) (lin RWSentences sentences) ;
739
740     mkOutputEventMapping_OutputAlways eventName sentences =
741         -- output_always : OutputType -> Str -> RWSentences -> S |
742         -- OutputEvent, takeoff, the velocity of the robot is set to 1.0
743         --   ↳ m/s upward
744         output_always OutputEvent eventName.s (lin RWSentences sentences)
745         --   ↳ ;
746
747     mkOutputEventMapping_NoOutput eventName =
748         -- no_output : OutputType -> Str -> S |
749         -- OutputEvent, takeoff
750         no_output OutputEvent eventName.s ;
751
752     mkOutputEventMapping_DiagrammaticOutput eventName float unit =
753         -- diagrammatic_output : OutputType -> Str -> Str ->
754         --   ↳ RoboWorldEng.Unit -> S |
755         -- OutputEvent, spray, 1.0, s
756         diagrammatic_output OutputEvent eventName.s float.s (lin Unit
757         --   ↳ unit) ;

```

```

752     mkOutputEventMapping_DiagrammaticOutput_Conditions eventName conditions
753     ↪ float unit =
754         -- diagrammatic_output_conditions : OutputType -> Str ->
755         ↪ Conditions -> Str -> RoboWorldEng.Unit -> S |
756         -- OutputEvent, spray, if the tank of water is full, 1.0, s
757         diagrammatic_output_conditions OutputEvent eventName.s (lin
758         ↪ Conditions conditions) float.s (lin Unit unit) ;
759
760 -----
761 lincat -- OperationMapping
762
763     OperationMapping = CatEng.S ;
764
765 -----
766 lin -- OperationMapping
767
768     mkOperationMapping_Sometimes eventName conditions sentences =
769         -- output_sometimes : OutputType -> Str -> Conditions ->
770         ↪ RWSentences -> S |
771         -- Operation, Store(), as soon as the distance from the robot to
772         ↪ the source is less than 1.0 m,
773         -- the robot places an object in the nest
774         output_sometimes Operation eventName.s (lin Conditions conditions)
775         ↪ (lin RWSentences sentences) ;
776
777     mkOperationMapping_OutputAlways eventName sentences =
778         -- output_always : OutputType -> Str -> RWSentences -> S |
779         -- Operation, move(ls,as), the velocity of the robot is set to ls
780         ↪ m/s towards the orientation of the robot
781         -- and the angular velocity of the robot is set to as rad/s
782         output_always Operation eventName.s (lin RWSentences sentences) ;
783
784     mkOperationMapping_NoOutput eventName =
785         -- no_output : OutputType -> Str -> S |
786         -- Operation, Transfer()
787         no_output Operation eventName.s ;
788
789     mkOperationMapping_DiagrammaticOutput eventName float unit =
790         -- diagrammatic_output : OutputType -> Str -> Str ->
791         ↪ RoboWorldEng.Unit -> S |
792         -- Operation, turnBack(), 1.0, s
793         diagrammatic_output Operation eventName.s float.s (lin Unit unit)
794         ↪ ;
795
796     mkOperationMapping_DiagrammaticOutput_Conditions eventName conditions
797     ↪ float unit =
798         -- diagrammatic_output_conditions : OutputType -> Str ->
799         ↪ Conditions -> Str -> RoboWorldEng.Unit -> S |
800         -- Operation, turnBack(), if it is raining, 1.0, s

```

```

790      diagrammatic_output_conditions Operation eventName.s (lin
791      ↪ Conditions conditions) float.s (lin Unit unit) ;
792
793 -----
794 lincat -- VariableMapping
795
796     VariableMapping = CatEng.S ;
797
798 -----
799 lin -- VariableMapping
800
801     mkVariableMapping_Conditions_RWSentences conditions sentence =
802     -- mkS : Adv -> S -> S / when the robot is on the floor, the
803     ↪ variable dist is incremented
804     mkS conditions sentence ;
805
806 -----
807 -- Help functions for RoboWorld plugin
808 lin _special_N = mkN "[N]" "[N]";
809 lin _special_A = mkA "[A]" "[A]";
810 lin _special_AdN = ParadigmsEng.mkAdN "[AdN]";
811 lin _special_Adv = ParadigmsEng.mkAdv "[Adv]";
812 lin _special_AdV = mkAdV "[AdV]";
813 lin _special_Conj = mkConj "[Conj]";
814 lin _special_Quant = ParadigmsEng.mkQuant "[Quant]" "[Quant]" "[Quant]" "[Quant]"
815     ↪ ;
816 lin _special_Prep = mkPrep "[Prep]";
817 lin _special_Pron = MorphoEng.mkPron "[Pron]" "[Pron]" "[Pron]" "[Pron]" singular
818     ↪ P3 nonhuman;
819 lin _special_Subj = mkSubj "[Subj]";
820 lin _special_V = mkV "[V]" "[V]" "[V]" "[V]" "[V]";
821 lin _special_V2 = mkV2 (mkV "[V2]" "[V2]" "[V2]" "[V2]" "[V2]");
822 lin _special_VV = mkVV (mkV "[VV]");
823
824 lin _special_empty_V = mkV "" "" "" "" "";
825 lin _special_Unit = mkN "[Unit]" "[Unit]";
826 lin _special_BasicItem = mkCN (mkN "[BasicItem]" "[BasicItem]");
827 lin _special_CompoundItem = mkCN (mkN "[CompoundItem]" "[CompoundItem]");
828 lin _special_Item = mkCN (mkN "[Item]" "[Item]");
829 lin _special_ItemPhrase = mkNP (mkN "[ItemPhrase]" "[ItemPhrase]");
830 lin _special_ItemPhraseList = mkListNP (mkNP (mkN "[ItemPhrase]" "[ItemPhrase]"))
831     ↪ (mkNP (mkN "[ItemPhrase]" "[ItemPhrase]"));
832 lin _special_RWSentence = mkS (mkCl (mkNP (mkN "[RWSentence]" "[RWSentence]"))
833     ↪ _special_empty_V);
834 lin _special_RWSentenceList = mkListS (mkS (mkCl (mkNP (mkN "[RWSentenceList]"
835     ↪ "[RWSentenceList]")) _special_empty_V))
836     (mkS (mkCl (mkNP (mkN "[RWSentenceList]"
837     ↪ "[RWSentenceList]")) _special_empty_V));
838 lin _special_RWSentences = mkS (mkCl (mkNP (mkN "[RWSentences]" "[RWSentences]"))
839     ↪ _special_empty_V);

```



```

831     lin _special_Conditions = ParadigmsEng.mkAdv "[Conditions]";
832     lin _special_ArenaAssumption = mkS (mkCl (mkNP (mkN "[ArenaAssumption]"
833       ↪ "[ArenaAssumption]"))) _special_empty_V);
834     lin _special_RobotAssumption = mkS (mkCl (mkNP (mkN "[RobotAssumption]"
835       ↪ "[RobotAssumption]"))) _special_empty_V);
836     lin _special_ElementAssumption = mkS (mkCl (mkNP (mkN "[ElementAssumption]"
837       ↪ "[ElementAssumption]"))) _special_empty_V);
838     lin _special_InputEventMapping = mkS (mkCl (mkNP (mkN "[InputEventMapping]"
839       ↪ "[InputEventMapping]"))) _special_empty_V);
840     lin _special_OutputEventMapping = mkS (mkCl (mkNP (mkN "[OutputEventMapping]"
841       ↪ "[OutputEventMapping]"))) _special_empty_V);
842     lin _special_OperationMapping = mkS (mkCl (mkNP (mkN "[OperationMapping]"
843       ↪ "[OperationMapping]"))) _special_empty_V);
844     lin _special_VariableMapping = mkS (mkCl (mkNP (mkN "[VariableMapping]"
845       ↪ "[VariableMapping]"))) _special_empty_V);
846 }

```

A.3 RoboWorldLexicon.gf

```

1  -----
2  -- Abstract grammar of RoboWorldLexicon: a lexicon for robotic systems
3  --
4  -- Authors:
5  -- * James Baxter <james.baxter@york.ac.uk>
6  --   (Department of Computer Science, University of York, UK)
7  -- * Gustavo Carvalho <ghpc@cin.ufpe.br> [corresponding author]
8  --   (Centro de Informática, Universidade Federal de Pernambuco, BR)
9  -- * Ana Cavalcanti <ana.cavalcanti@york.ac.uk>,
10 --   (Department of Computer Science, University of York, UK)
11 -----
12 abstract RoboWorldLexicon =
13     Cat
14     **
15 {
16
17     --
18     fun _1D_A : A;
19     fun _2D_A : A;
20     fun _3D_A : A;
21
22     -- A
23     fun a_Det : Det;
24     fun after_Prep : Prep;
25     fun angular_A : A;
26     fun aPl_Det : Det;
27     fun arena_N : N;
28     fun asap_Subj : Subj;
29     fun at_Prep : Prep;

```

```

30     fun at_least_AdN : AdN;
31     fun available_A : A;
32
33     -- B
34     fun before_Adv : Adv;
35     fun block_V2 : V2;
36     fun box_N : N;
37
38     -- C
39     fun can_VV : VV;
40     fun call_V : V;
41     fun call_V2 : V2;
42     fun carry_V2 : V2;
43     fun circle_N : N;
44     fun closed_A : A;
45     fun communicate_V2 : V2;
46     fun contain_V2 : V2;
47     fun cylinder_N : N;
48
49     -- D
50     fun define_V2 : V2;
51     fun depth_N : N;
52     fun diagram_N : N;
53     fun dimension_N : N;
54     fun direction_N : N;
55     fun distance_N : N;
56     fun downward_Adv : Adv;
57     fun downwards_Adv : Adv;
58
59     -- E
60     fun effect_N : N;
61     fun either7or_DConj : Conj;
62     fun event_N : N;
63
64     -- F
65     fun floor_N : N;
66     fun from_Prep : Prep;
67
68     -- G
69     fun gradient_N : N;
70     fun great_A : A;
71     fun ground_N : N;
72
73     -- H
74     fun have_V : V;
75     fun have_V2 : V2;
76     fun happen_V : V;
77     fun height_N : N;
78
79     -- I

```

```

80     fun if_Subj : Subj;
81     fun in_Prep : Prep;
82     fun initial_A : A;
83     fun inside_Prep : Prep;
84     fun it_Pron : Pron;
85
86     -- J
87
88     -- K
89
90     -- L
91     fun less_than_A : A;
92
93     -- M
94     fun magnitude_N : N;
95     fun mass_N : N;
96     fun may_1_VV : VV; -- be possible
97     fun minus_Prep : Prep;
98     fun movement_N : N;
99
100    -- N
101    fun never_AdV : Adv;
102    fun neither7nor_DConj : Conj;
103    fun no_Quant : Quant;
104
105    -- O
106    fun obstacle_N : N;
107    fun occur_V : V;
108    fun occurrence_N : N;
109    fun odometer_N : N;
110    fun of_Prep : Prep;
111    fun on_Prep : Prep;
112    fun one_dimensional_A : A;
113    fun operation_N : N;
114    fun orientation_N : N;
115    fun output_N : N;
116
117    -- P
118    fun place_V2 : V2;
119    fun plus_Prep : Prep;
120    fun point_N : N;
121    fun position_N : N;
122    fun pose_N : N;
123
124    -- Q
125    fun quarter_N : N;
126
127    -- R
128    fun receive_V2 : V2;
129    fun region_N : N;

```

```

130     fun reset_V : V;
131     fun robot_N : N;
132
133     -- S
134     fun sequence_N : N;
135     fun send_V2 : V2;
136     fun set_V2 : V2;
137     fun speed_N : N;
138     fun sphere_N : N;
139     fun square_N : N;
140
141     -- T
142     fun take_V2 : V2;
143     fun the_Det : Det;
144     fun then_Adv : Adv;
145     fun thePl_Det : Det;
146     fun three_dimensional_A : A;
147     fun through_Prep : Prep ;
148     fun time_N : N;
149     fun times_Prep : Prep;
150     fun to_Prep : Prep;
151     fun towards_Prep : Prep;
152     fun two_dimensional_A : A;
153
154     -- U
155     fun under_Prep : Prep;
156     fun unit_1_N : N;
157     fun up_to_Prep : Prep;
158     fun upward_Adv : Adv;
159     fun upwards_Adv : Adv;
160
161     -- V
162     fun value_N : N;
163     fun variable_N : N;
164     fun velocity_N : N;
165
166     -- X
167     fun x_position_N : N;
168     fun x_width_N : N;
169
170     -- W
171     fun when_Subj : Subj;
172     fun where_Subj : Subj;
173     fun width_N : N;
174     fun wind_N : N;
175     fun within_Prep : Prep;
176
177     -- Y
178     fun y_position_N : N;
179     fun y_width_N : N;

```

```

180
181     -- Z
182     fun z_position_N : N;
183     fun z_width_N : N;
184
185 }

```

A.4 RoboWorldLexiconEng.gf

```

1  -----
2  -- Concrete grammar of RoboWorldLexicon: a lexicon for robotic systems
3  --
4  -- Authors:
5  -- * James Baxter <james.baxter@york.ac.uk>
6  --   (Department of Computer Science, University of York, UK)
7  -- * Gustavo Carvalho <ghpc@cin.ufpe.br> [corresponding author]
8  --   (Centro de Informática, Universidade Federal de Pernambuco, BR)
9  -- * Ana Cavalcanti <ana.cavalcanti@york.ac.uk>,
10 --   (Department of Computer Science, University of York, UK)
11 -----
12 concrete RoboWorldLexiconEng of RoboWorldLexicon =
13     CatEng
14     **
15 open
16     MorphoEng,
17     ResEng,
18     ParadigmsEng,
19     IrregEng,
20     Prelude
21 in {
22
23     --
24     lin _1D_A = mkA "1D" "IRREG";
25     lin _2D_A = mkA "2D" "IRREG";
26     lin _3D_A = mkA "3D" "IRREG";
27
28     -- A
29     lin a_Det = mkDeterminer singular "a" | mkDeterminer singular "an";
30     lin after_Prep = mkPrep "after";
31     lin angular_A = compoundA (mkA "angular");
32     lin aPl_Det = mkDeterminer plural "";
33     lin arena_N = mkN "arena" "arenas";
34     lin asap_Subj = mkSubj "as soon as" ;
35     lin at_Prep = mkPrep "at";
36     lin at_least_AdN = mkAdN "at least";
37     lin available_A = compoundA (mkA "available");
38
39     -- B
40     lin before_Adv = mkAdv "before";

```

```

41     lin block_V2 = mkV2 (mkV "block" "blocks" "blocked" "blocked" "blocking");
42     lin box_N = mkN "box" "boxes";
43
44     -- C
45     lin can_VV = {
46         s = table {
47             VVF VInf => ["be able to"] ;
48             VVF VPres => "can" ;
49             VVF VPPart => ["been able to"] ;
50             VVF VPresPart => ["being able to"] ;
51             VVF VPast => "could" ;
52             VVPastNeg => "couldn't" ;
53             VVPresNeg => "can't"
54         } ;
55         p = [] ;
56         typ = VVAux
57     } ;
58     lin call_V = mkV "call" "calls" "called" "called" "calling";
59     lin call_V2 = mkV2 (mkV "call" "calls" "called" "called" "calling");
60     lin carry_V2 = mkV2 (mkV "carry" "carries" "carried" "carried"
61         ↪ "carrying");
62     lin circle_N = mkN "circle" "circles";
63     lin closed_A = mkA "closed" "closed";
64     lin communicate_V2 = mkV2 (mkV "communicate" "communicates" "communicated"
65         ↪ "communicated" "communicating");
66     lin contain_V2 = mkV2 (mkV "contain" "contains" "contained" "contained"
67         ↪ "containing");
68     lin cylinder_N = mkN "cylinder" "cylinders";
69
70     -- D
71     lin define_V2 = mkV2 (mkV "define" "defines" "defined" "defined"
72         ↪ "defining");
73     lin depth_N = mkN "depth" "depths";
74     lin diagram_N = mkN "diagram" "diagrams";
75     lin dimension_N = mkN "dimension" "dimensions";
76     lin direction_N = mkN "direction" "directions";
77     lin distance_N = mkN "distance" "distances";
78     lin downward_Adv = mkAdv "downward";
79     lin downwards_Adv = mkAdv "downwards";
80
81     -- E
82     lin effect_N = mkN "effect" "effects";
83     lin either7or_DConj = mkConj "either" "or" singular ;
84     lin event_N = mkN "event" "events";
85
86     -- F
87     lin floor_N = mkN "floor" "floors";
88     lin from_Prep = mkPrep "from";
89
90     -- G

```

```

87     lin gradient_N = mkN "gradient" "gradients";
88     lin great_A = mkA "great" "greater";
89     lin ground_N = mkN "ground" "grounds";
90
91     -- H
92     lin have_V = IrregEng.have_V;
93     lin have_V2 = mkV2 (IrregEng.have_V);
94     lin happen_V = mkV "happen" "happens" "happened" "happened" "happening";
95     lin height_N = mkN "height" "heights";
96
97     -- I
98     lin if_Subj = mkSubj "if";
99     lin in_Prep = mkPrep "in";
100    lin initial_A = compoundA (mkA "initial");
101    lin inside_Prep = mkPrep "inside";
102    lin it_Pron = mkPron "it" "it" "its" "its" singular P3 nonhuman;
103
104    -- J
105
106    -- K
107
108    -- L
109    lin less_than_A = mkA "less" "less";
110
111    -- M
112    lin magnitude_N = mkN "magnitude" ;
113    lin mass_N = mkN "mass" "masses";
114    lin may_1_VV = {
115  s = table {
116          VVF VInf => ["be possible to"] ;
117          VVF VPres => "may" ;
118          VVF VPPart => ["been possible to"] ;
119          VVF VPresPart => ["being possible to"] ;
120          VVF VPast => "might" ;
121          VVPastNeg => "mightn't" ;
122          VVPresNeg => "may not"
123        } ;
124    p = [] ;
125    typ = VVAux
126  } ;
127  lin minus_Prep = mkPrep "minus";
128  lin movement_N = mkN "movement" "movement";
129
130  -- N
131  lin never_AdV = mkAdV "never" ;
132  lin neither7nor_DConj = mkConj "neither" "nor" singular;
133  lin no_Quant = mkQuant "no" "no" "none" "none" ;
134
135  -- O
136  lin obstacle_N = mkN "obstacle" "obstacles";

```

```

137     lin odometer_N = mkN "odometer" "odometers";
138     lin occur_V = mkV "occur" "occurs" "occurred" "occurred" "occurring";
139     lin occurrence_N = mkN "occurrence" "occurrences";
140     lin of_Prep = mkPrep "of";
141     lin on_Prep = mkPrep "on";
142     lin one_dimensional_A = compoundA (mkA "one-dimensional");
143     lin operation_N = mkN "operation" "operations";
144     lin orientation_N = mkN "orientation" ;
145     lin output_N = mkN "output" "IRREG";
146
147     -- P
148     lin place_V2 = mkV2 (mkV "place" "places" "placed" "placed" "placing");
149     lin plus_Prep = mkPrep "plus";
150     lin point_N = mkN "point" "points";
151     lin position_N = mkN "position" "positions";
152     lin pose_N = mkN "pose" "poses";
153
154     -- Q
155     lin quarter_N = mkN "quarter" "quarters";
156
157     -- R
158     lin receive_V2 = mkV2 (mkV "receive" "receives" "received" "received"
159     ↪ "receiving");
160     lin region_N = mkN "region" "regions";
161     lin reset_V = mkV "reset" "resets" "reset" "reset" "resetting";
162     lin robot_N = mkN "robot";
163
164     -- S
165     lin sequence_N = mkN "sequence" "sequences";
166     lin send_V2 = mkV2 (IrregEng.send_V);
167     lin set_V2 = mkV2 (IrregEng.set_V);
168     lin speed_N = mkN "speed" "speeds";
169     lin sphere_N = mkN "sphere" "spheres";
170     lin square_N = mkN "square" "squares";
171
172     -- T
173     lin take_V2 = mkV2 (mkV "take" "takes" "took" "taken" "taking");
174     lin the_Det = mkDeterminer singular "the";
175     lin then_Adv = mkAdv "then" ;
176     lin thePl_Det = mkDeterminer plural "the";
177     lin three_dimensional_A = compoundA (mkA "three-dimensional");
178     lin through_Prep = mkPrep "through";
179     lin time_N = mkN "time" "times";
180     lin times_Prep = mkPrep "times";
181     lin to_Prep = mkPrep "to";
182     lin towards_Prep = mkPrep "towards";
183     lin two_dimensional_A = compoundA (mkA "two-dimensional");
184
185     -- U
186     lin under_Prep = mkPrep "under";

```



```
186     lin unit_1_N = mkN "unit" "units";
187     lin up_to_Prep = mkPrep "up to";
188     lin upward_Adv = mkAdv "upward";
189     lin upwards_Adv = mkAdv "upwards";
190
191     -- V
192     lin value_N = mkN "value" "values";
193     lin variable_N = mkN "variable" "variables";
194     lin velocity_N = mkN "velocity";
195
196     -- X
197     lin x_position_N = mkN "x-position" "x-positions";
198     lin x_width_N = mkN "x-width" "x-widths";
199
200     -- W
201     lin when_Subj = mkSubj "when";
202     lin where_Subj = mkSubj "where";
203     lin width_N = mkN "width" "widths";
204     lin wind_N = mkN "wind" "winds";
205     lin within_Prep = mkPrep "within";
206
207     -- Y
208     lin y_position_N = mkN "y-position" "y-positions";
209     lin y_width_N = mkN "y-width" "y-widths";
210
211     -- Z
212     lin z_position_N = mkN "z-position" "z-positions";
213     lin z_width_N = mkN "z-width" "z-widths";
214
215 }
```


B. Complete RoboWorld MetaModel

```
1  import _'ecore.xml.type' : 'http://www.eclipse.org/emf/2003/XMLType';
2
3  package RoboWorldMM : RoboWorldMM = 'http://www.robocalc.circus/
   ⇨ RoboWorldMM'
4  {
5      package RoboChart : RoboChart = 'http://www.robocalc.circus/RoboChart
   ⇨ '
6      {
7          class RCOperation;
8          class RCIntegerExp;
9      }
10     package PhysMod : PhysMod = 'http://www.robocalc.circus/PhysMod'
11     {
12         class PModel;
13         class Instantiation;
14     }
15     package GF : GF = 'http://www.grammaticalframework.org'
16     {
17         datatype Quantifier : 'java.lang.String' { serializable };
18         datatype Preposition : 'java.lang.String' { serializable };
19         datatype Pronoun : 'java.lang.String' { serializable };
20         datatype Noun : 'java.lang.String' { serializable };
21         datatype Determiner : 'java.lang.String' { serializable };
22         datatype Adjective : 'java.lang.String' { serializable };
23         datatype Adverb : 'java.lang.String' { serializable };
24         datatype Subjunction : 'java.lang.String' { serializable };
25     }
26     abstract class ItemPhrase;
27     class PronounIP extends ItemPhrase
```

```

28  {
29      attribute pronoun : GF::Pronoun[1];
30  }
31  class DeterminedIP extends ItemPhrase
32  {
33      property item : Item[1];
34      attribute determiner : GF::Determiner[1];
35  }
36  class QualifiedBI extends BasicItem
37  {
38      property basicitem : BasicItem[1];
39      attribute adjective : GF::Adjective[1];
40  }
41  class QuantifiedIP extends ItemPhrase
42  {
43      property item : Item[1];
44      attribute number : GF::Quantifier[1];
45  }
46  class OperationMapping
47  {
48      property output : Output[1] { composes };
49      property signature : Signature[1] { composes };
50  }
51  class Signature
52  {
53      property parameters : Identifier[*|1] { ordered composes };
54      property name : Identifier[1] { composes };
55  }
56  class InputEventMapping
57  {
58      property input : Input[1] { composes };
59      property name : Identifier[1];
60  }
61  class OutputEventMapping
62  {
63      property name : Identifier[1];
64      property output : Output[1] { composes };
65  }
66  class VariableMapping
67  {
68      property name : Identifier[1];
69      property conditions : Conditions[1] { composes };
70      property update : RWSentence[1] { composes };
71  }
72  abstract class RobotAssumption;
73  class RWDocument
74  {
75      property inputEventMappings : InputEventMapping[*|1] { ordered
76          ↪ composes };
77      property outputEventMappings : OutputEventMapping[*|1] { ordered

```

```

    ⇨ composes };
77   property operationMappings : OperationMapping[*|1] { ordered
    ⇨ composes };
78   property variableMappings : VariableMapping[*|1] { ordered composes
    ⇨ };
79   property robotAssumptions : RobotAssumption[*|1] { ordered composes
    ⇨ };
80   property elementAssumptions : ElementAssumption[*|1] { ordered
    ⇨ composes };
81   property arenaAssumptions : ArenaAssumption[*|1] { ordered composes
    ⇨ };
82 }
83 class ArenaAssumption
84 {
85   property sentence : RWSentence[1] { composes };
86 }
87 abstract class BasicItem extends Item;
88 class NounBI extends BasicItem
89 {
90   attribute noun : GF::Noun[1];
91 }
92 class Identifier
93 {
94   attribute identifier : String[?];
95 }
96 abstract class CompoundItem extends Item
97 {
98   property item : Item[1];
99 }
100 class AdverbCI extends CompoundItem
101 {
102   attribute adverb : GF::Adverb[1];
103 }
104 class PrepositionCI extends CompoundItem
105 {
106   property itemphrases : ItemPhrase[+|1] { ordered };
107   attribute preposition : GF::Preposition[1];
108   attribute conjunctionType : ConjunctionType[1];
109 }
110 abstract class Item extends ItemPhrase;
111 abstract class ElementAssumption;
112 abstract class RWSentence
113 {
114   property itemphrase : ItemPhrase[1];
115 }
116 class RobotSentence extends RobotAssumption
117 {
118   property sentence : RWSentence[1] { composes };
119 }
120 class RobotPModel extends RobotAssumption

```

```

121 {
122     property pmodel : PhysMod::PModel[1];
123     property instantiations : PhysMod::Instantiation[*|1] { ordered };
124 }
125 abstract class Input;
126 class InputSometimes extends Input
127 {
128     property conditions : Conditions[1];
129     property sentences : RWSentence[*|1] { ordered !unique };
130 }
131 class Conditions
132 {
133     property sentences : RWSentence[+|1] { ordered };
134     attribute subjunction : GF::Subjunction[1];
135 }
136 class InputAlways extends Input
137 {
138     property sentences : RWSentence[*|1] { ordered !unique };
139 }
140 class InputNever extends Input;
141 class NoOutput extends Output,RoboWorldIR::OutputIR;
142 class OutputSometimes extends Output
143 {
144     property conditions : Conditions[1];
145     property sentences : RWSentence[+|1] { ordered !unique };
146 }
147 abstract class Output;
148 class UnitBI extends BasicItem
149 {
150     attribute unit : Unit[1];
151 }
152 class FloatLiteralIP extends ItemPhrase
153 {
154     attribute value : _'ecore.xml.type'::Float[1];
155 }
156 class OutputAlways extends Output,RoboWorldIR::OutputIR
157 {
158     property sentences : RWSentence[+|1] { ordered };
159 }
160 class DiagrammaticOutput extends Output
161 {
162     property opd : RoboChart::RCOperation[1];
163     property sizetu : RoboChart::RCIntegerExp[1];
164     property conditions : Conditions[?];
165     attribute timeunit : Unit[1];
166 }
167 class ElementSentence extends ElementAssumption
168 {
169     property sentence : RWSentence[1] { composes };
170 }

```

```
171 class ElementPModel extends ElementAssumption
172 {
173     property pmodel : PhysMod::PModel[1];
174     property instantiations : PhysMod::Instantiation[*|1] { ordered };
175     property name : Item[1] { composes };
176 }
177 datatype Unit : 'java.lang.String' { serializable };
178 enum ConjunctionType { serializable }
179 {
180     literal AND;
181     literal OR = 1;
182 }
183 }
```


C. Complete RoboWorld IR Metamodel

C.1 RoboWorldIR.ecore

```
1 import ExpressionIR : 'ExpressionIR.ecore#';
2 import RoboWorldMM : 'RoboWorldMM.ecore#';
3 import GF : 'RoboWorldMM.ecore#//GF';
4 import PhysMod : 'RoboWorldMM.ecore#//PhysMod';
5 import RoboChart : 'RoboWorldMM.ecore#//RoboChart';
6
7 package RoboWorldIR : RoboWorldIR = 'http://www.robocalc.circus/
  ↳ RoboWorldIR'
8 {
9   class OperationMappingIR
10  {
11    property signature : RoboWorldMM::Signature[1] { composes };
12    property output : OutputIR[1] { composes };
13  }
14   class InputEventMappingIR
15  {
16    property input : InputIR[1] { composes };
17    property name : RoboWorldMM::Identifier[1] { composes };
18  }
19   class OutputEventMappingIR
20  {
21    property name : RoboWorldMM::Identifier[1] { composes };
22    property output : OutputIR[1] { composes };
23  }
24   class RWIntermediateRepresentation
25  {
```

```

26     property inputEventMappings : InputEventMappingIR[*|1] { ordered
    ⇨ composes };
27     property outputEventMappings : OutputEventMappingIR[*|1] { ordered
    ⇨ composes };
28     property operationMappings : OperationMappingIR[*|1] { ordered
    ⇨ composes };
29     property variableMappings : VariableMappingIR[*|1] { ordered
    ⇨ composes };
30     property arena : Arena[1] { composes };
31     property robot : Element[1] { composes };
32     property elements : Element[*|1] { ordered composes };
33 }
34 class VariableMappingIR
35 {
36     property conditions : ExpressionIR::Constraint[+|1] { ordered
    ⇨ composes };
37     property update : Statement[1] { composes };
38     property name : RoboWorldMM::Identifier[1] { composes };
39 }
40 enum Dimension { serializable }
41 {
42     literal OneD : '1D' = 1;
43     literal TwoD : '2D' = 2;
44     literal ThreeD : '3D' = 3;
45 }
46 class Conditions;
47 abstract class Element
48 {
49     property name : RoboWorldMM::Identifier[1] { composes };
50     property number : NumericProperty[?] { composes };
51     attribute plurality : Plurality[1];
52 }
53 class ElementPModel extends Element
54 {
55     property pmodel : RoboWorldMM::PhysMod::PModel[1] { composes };
56     property instantiations : RoboWorldMM::PhysMod::Instantiation[*|1]
    ⇨ { ordered composes };
57 }
58 class ElementDescription extends Element
59 {
60     property components : ElementDescription[*|1] { ordered composes };
61     property properties : ExpressionIR::Constraint[*|1] { ordered
    ⇨ composes };
62     property attributes : Attribute[*|1] { ordered composes };
63     property shape : Shape[?] { composes };
64 }
65 abstract class InputIR;
66 class InputSometimesIR extends InputIR
67 {
68     property conditions : ExpressionIR::Constraint[+|1] { ordered

```

```

    ⇨ composes };
69   property communications : Statement[*|1] { ordered composes };
70 }
71 class InputAlwaysIR extends InputIR
72 {
73   property communications : Statement[+|1] { ordered composes };
74 }
75 class InputNeverIR extends InputIR;
76 abstract class OutputIR;
77 class OutputSometimesIR extends OutputIR
78 {
79   property conditions : ExpressionIR::Constraint[+|1] { ordered
    ⇨ composes };
80   property statements : Statement[*|1] { ordered composes };
81   attribute subjunction : RoboWorldMM::GF::Subjunction[1];
82 }
83 class Region extends ElementDescription
84 {
85   attribute dimension : Dimension[1];
86   attribute closed : Boolean[1] = 'false';
87 }
88 class NumericProperty
89 {
90   property properties : ExpressionIR::Constraint[+|1] { ordered
    ⇨ composes };
91 }
92 abstract class Action;
93 class OutputAlwaysIR extends OutputIR
94 {
95   property statements : Statement[+|1] { ordered composes };
96 }
97 class NoOutputIR extends OutputIR;
98 class Attribute
99 {
100   property name : RoboWorldMM::Identifier[1] { composes };
101   property type : Type[1] { composes };
102 }
103 abstract class Type;
104 class DiagrammaticOutputIR extends OutputIR
105 {
106   property opd : RoboWorldMM::RoboChart::RCOperation[1] { composes };
107   property sizetu : RoboWorldMM::RoboChart::RCIntegerExp[1] {
    ⇨ composes };
108   attribute tunit : RoboWorldMM::Unit[1];
109   property conditions : ExpressionIR::Constraint[*|1] { ordered
    ⇨ composes };
110   attribute subjunction : RoboWorldMM::GF::Subjunction[?];
111 }
112 abstract class Shape;
113 class Box extends Shape

```

```

114 {
115     property xwidth : NumericProperty[?] { composes };
116     property ywidth : NumericProperty[?] { composes };
117     property zwidth : NumericProperty[?] { composes };
118 }
119 class Sphere extends Shape
120 {
121     property radius : NumericProperty[?] { composes };
122 }
123 class Cylinder extends Shape
124 {
125     property radius : NumericProperty[?] { composes };
126     property depth : NumericProperty[?] { composes };
127 }
128 class Assign extends Action
129 {
130     property assignto : ExpressionIR::ItemPhraseIR[1] { composes };
131     property value : ExpressionIR::ItemPhraseIR[1] { composes };
132 }
133 class Put extends Action
134 {
135     property element : ExpressionIR::ElementReference[1] { composes };
136     property into : ExpressionIR::ElementReference[1] { composes };
137 }
138 class Take extends Action
139 {
140     property element : ExpressionIR::ElementReference[1] { composes };
141     property from : ExpressionIR::ElementReference[1] { composes };
142 }
143 class Communicate extends Action
144 {
145     property value : ExpressionIR::ItemPhraseIR[1] { composes };
146 }
147 class Statement
148 {
149     property sentence : RWSentence[1] { composes };
150     property action : Action[?] { composes };
151 }
152 class Arena extends Region
153 {
154     attribute hasFloor : Boolean[1];
155     property gradient : NumericProperty[?] { composes };
156     attribute hasRain : Boolean[1];
157     property windSpeed : NumericProperty[?] { composes };
158 }
159 enum Plurality { serializable }
160 {
161     literal SINGULAR : 'SINGULAR';
162     literal PLURAL = 1;
163     literal UNCOUNTABLE : 'UNCOUNTABLE' = 2;

```

```

164     }
165     class RWSentence
166     {
167         attribute text : String[?];
168     }
169     datatype _'Sequence' { serializable };
170     class _'Tuple' extends Type
171     {
172         property types : Type[+|1] { ordered composes };
173     }
174     class _'Real' extends Type;
175     class Enumeration extends Type
176     {
177         property variants : RoboWorldMM::Identifier[*|1] { ordered composes
178             ↪ };
179     }
180 }
181 package RoboWorldMM : RoboWorldMM = 'http://www.robocalc.circus/
182     ↪ RoboWorldMM'
183 {
184     package PhysMod : PhysMod = 'http://www.robocalc.circus/PhysMod'
185     {
186         class PModel;
187         class Instantiation;
188     }
189     package RoboChart : RoboChart = 'http://www.robocalc.circus/RoboChart
190     ↪ '
191     {
192         class RCIntegerExp;
193         class RCOperation;
194     }
195     class Identifier;
196     class Signature;
197 }

```

C.2 ExpressionIR.ecore

```

1  import RoboWorldIR : 'RoboWorldIR.ecore#/' ;
2  import RoboWorldMM : 'RoboWorldMM.ecore#/' ;
3  import _'ecore.xml.type' : 'http://www.eclipse.org/emf/2003/XMLType' ;
4  import GF : 'RoboWorldMM.ecore#//GF' ;
5
6  package ExpressionIR : ExpressionIR = 'http://www.robocalc.circus/
7  ↪ ExpressionIR'
8  {
9      abstract class BooleanExpression;
10     abstract class UnaryBooleanExpression extends BooleanExpression
11     {

```

```

11     property pred : BooleanExpression[1] { composes };
12 }
13 abstract class BinaryBooleanExpression extends BooleanExpression
14 {
15     property leftPred : BooleanExpression[1] { composes };
16     property rightPred : BooleanExpression[1] { composes };
17 }
18 class NotExpression extends UnaryBooleanExpression;
19 abstract class QuantifierExpression extends UnaryBooleanExpression
20 {
21     property element : RoboWorldIR::Element[1];
22     property name : RoboWorldMM::Identifier[1] { composes };
23     property variable : RoboWorldMM::Identifier[1] { composes };
24 }
25 class UniversalExpression extends QuantifierExpression;
26 class ExistentialExpression extends QuantifierExpression;
27 class AndExpression extends BinaryBooleanExpression;
28 class OrExpression extends BinaryBooleanExpression;
29 abstract class ComparisonExpression extends BooleanExpression
30 {
31     property left : ItemPhraseIR[1] { composes };
32     property right : ItemPhraseIR[1] { composes };
33 }
34 class LessThan extends ComparisonExpression;
35 class Equal extends ComparisonExpression;
36 class GreaterThan extends ComparisonExpression;
37 abstract class Expression;
38 class Distance extends PrimitiveExpression
39 {
40     property from : ElementReference[1] { composes };
41     property to : ElementReference[1] { composes };
42 }
43 class Towards extends PrimitiveExpression
44 {
45     property towards : ElementReference[1] { composes };
46     property base : ElementReference[1] { composes };
47 }
48 abstract class BinaryExpression extends Expression
49 {
50     property right : Expression[1];
51     property left : Expression[1];
52 }
53 class Multiplication extends BinaryExpression;
54 class Addition extends BinaryExpression;
55 class Negation extends Expression
56 {
57     property expression : Expression[1] { composes };
58 }
59 class NounBIIR extends BasicItemIR
60 {

```

```

61     attribute noun : RoboWorldMM::GF::Noun[1];
62 }
63 class PrepositionCIIR extends CompoundItemIR
64 {
65     property itemphrases : ItemPhraseIR[+|1] { ordered composes };
66     attribute preposition : RoboWorldMM::GF::Preposition[1];
67     attribute conjunctionType : RoboWorldMM::ConjunctionType[1];
68 }
69 abstract class ItemIR extends ItemPhraseIR;
70 class AdverbCIIR extends CompoundItemIR
71 {
72     attribute adverb : RoboWorldMM::GF::Adverb[1];
73 }
74 abstract class CompoundItemIR extends ItemIR
75 {
76     property item : ItemIR[1] { composes };
77 }
78 class QualifiedBIIR extends BasicItemIR
79 {
80     property basicitem : BasicItemIR[1] { composes };
81     attribute adjective : RoboWorldMM::GF::Adjective[1];
82 }
83 class QuantifiedIPIR extends ItemPhraseIR
84 {
85     property item : ItemIR[1] { composes };
86     attribute number : RoboWorldMM::GF::Quantifier[1];
87 }
88 abstract class ItemPhraseIR
89 {
90     property expression : Expression[?];
91 }
92 abstract class BasicItemIR extends ItemIR;
93 class DeterminedIPIR extends ItemPhraseIR
94 {
95     property item : ItemIR[1] { composes };
96     attribute determiner : RoboWorldMM::GF::Determiner[1];
97 }
98 class NumericLiteral extends PrimitiveExpression
99 {
100     attribute value : _'ecore.xml.type'::Double[1];
101 }
102 class EnumLiteral extends PrimitiveExpression
103 {
104     property value : RoboWorldMM::Identifier[1] { composes };
105 }
106 class TimeSince extends PrimitiveExpression
107 {
108     property event : RoboWorldMM::Identifier[?];
109 }
110 class Constraint

```

```

111  {
112      property booleanexpression : BooleanExpression[?] { composes };
113      property sentence : RoboWorldIR::RWSentence[1] { composes };
114  }
115  class PronounIPIR extends ItemPhraseIR
116  {
117      attribute pronoun : RoboWorldMM::GF::Pronoun[1];
118      property referent : ItemPhraseIR[?] { composes };
119  }
120  class FloatLiteralIR extends ItemPhraseIR
121  {
122      attribute value : _'ecore.xml.type'::Double[1];
123  }
124  class UnitBIIR extends BasicItemIR
125  {
126      attribute unit : RoboWorldMM::Unit[1];
127  }
128  abstract class PrimitiveExpression extends Expression;
129  abstract class EntityFieldExpression extends PrimitiveExpression
130  {
131      property elementref : ElementReference[1] { composes };
132      property componentrefs : ElementReference[*|1] { ordered composes
133      ↪ };
134  }
135  class LessThanOrEqual extends ComparisonExpression;
136  class GreaterThanOrEqual extends ComparisonExpression;
137  class In extends BooleanExpression
138  {
139      property set : ItemPhraseIR[1] { composes };
140      property element : ItemPhraseIR[1] { composes };
141  }
142  class MayExpression extends UnaryBooleanExpression;
143  enum ElementProperty { serializable }
144  {
145      literal XWIDTH : 'XWIDTH';
146      literal YWIDTH : 'YWIDTH' = 1;
147      literal ZWIDTH : 'ZWIDTH' = 2;
148      literal RADIUS = 3;
149      literal DEPTH = 4;
150      literal SIZE = 5;
151      literal XPOSITION : 'XPOSITION' = 6;
152      literal YPOSITION : 'YPOSITION' = 7;
153      literal ZPOSITION : 'ZPOSITION' = 8;
154      literal POSITION = 9;
155      literal YAW = 10;
156      literal PITCH = 11;
157      literal ROLL = 12;
158      literal VELOCITY = 13;
159      literal ACCELERATION = 14;
160      literal ANGULARVELOCITY = 15;

```



```

160     literal YAWVELOCITY = 16;
161     literal PITCHVELOCITY = 17;
162     literal ROLLVELOCITY = 18;
163     literal ANGULARACCELERATION = 19;
164     literal YAWACCELERATION = 20;
165     literal PITCHACCELERATION = 21;
166     literal ROLLACCELERATION = 22;
167     literal POSE = 23;
168     literal ORIENTATION = 24;
169 }
170 abstract class ElementReference
171 {
172     property element : RoboWorldIR::Element[1];
173     property name : RoboWorldMM::Identifier[1] { composes };
174 }
175 class UniqueElement extends ElementReference;
176 class SomeElement extends ElementReference
177 {
178     property constraint : Constraint[?] { composes };
179     property variable : RoboWorldMM::Identifier[1] { composes };
180 }
181 class AllElements extends ElementReference
182 {
183     property constraint : Constraint[?] { composes };
184     property variable : RoboWorldMM::Identifier[1] { composes };
185 }
186 class PotentialElement extends ElementReference;
187 class QuantifiedElement extends ElementReference
188 {
189     property variable : RoboWorldMM::Identifier[1] { composes };
190 }
191 class AttributeExpression extends EntityFieldExpression
192 {
193     property attribute : RoboWorldIR::Attribute[1];
194     property name : RoboWorldMM::Identifier[1] { composes };
195 }
196 class PropertyExpression extends EntityFieldExpression
197 {
198     attribute property : ElementProperty[1];
199 }
200 class ElementBody extends PrimitiveExpression
201 {
202     property elementref : ElementReference[1] { composes };
203 }
204 class ElementSurface extends PrimitiveExpression
205 {
206     property elementref : ElementReference[1] { composes };
207 }
208 class ArenaGradient extends PrimitiveExpression;
209 class ArenaWindSpeed extends PrimitiveExpression;

```

```
210 class Range extends BinaryExpression;
211 class TupleLiteral extends Expression
212 {
213     property expression : Expression[+|1] { ordered };
214 }
215 class SequenceLiteral extends Expression
216 {
217     property expression : Expression[+|1] { ordered composes };
218 }
219 class Ground extends PrimitiveExpression;
220 class VectorLiteral extends PrimitiveExpression
221 {
222     attribute values : _'ecore.xml.type'::Double[2..*|1] { ordered };
223 }
224 class On extends BooleanExpression
225 {
226     property on : ItemPhraseIR[1] { composes };
227     property object : ItemPhraseIR[1] { composes };
228 }
229 class Subset extends ComparisonExpression;
230 }
```

D. Firefighter *CyPhyCircus* Semantics

D.1 Types

$Position == \mathbb{R} \times \mathbb{R} \times \mathbb{R}$
 $Velocity == \mathbb{R} \times \mathbb{R} \times \mathbb{R}$
 $Acceleration == \mathbb{R} \times \mathbb{R} \times \mathbb{R}$
 $Orientation == \mathbb{R} \times \mathbb{R} \times \mathbb{R}$
 $AngularVelocity == \mathbb{R} \times \mathbb{R} \times \mathbb{R}$
 $AngularAcceleration == \mathbb{R} \times \mathbb{R} \times \mathbb{R}$

HomeProperty

$xwidth, ywidth, zwidth : \mathbb{R}$
 $position : Position$
 $orientation : Orientation$
 $locations : \mathbb{P} Position$

$locations = boxLocs\ position\ orientation\ xwidth\ ywidth\ zwidth$

ArenaProperty

$xwidth, ywidth, zwidth : \mathbb{R}$
 $gradient, windSpeed : \mathbb{R}$
 $locations : \mathbb{P} Position$
 $home : HomeProperty$

$locations = \{x : 0..xwidth; y : 0..ywidth; z : 0..zwidth\}$

$Tank_of_waterType ::= full \mid empty$

RobotProperty

$position : Position$
 $velocity : Velocity$
 $acceleration : Acceleration$
 $orientation : Orientation$
 $angularVelocity : AngularVelocity$
 $angularAcceleration : AngularAcceleration$
 $tank_of_water : Tank_of_waterType$
 $searchPattern : seq\ Position$

BuildingProperty

$xwidth, ywidth, zwidth : \mathbb{R}$
 $position : Position$
 $orientation : Orientation$
 $locations : \mathbb{P}\ Position$

 $locations = boxLocs\ position\ orientation\ xwidth\ ywidth\ zwidth$

$StatusType ::= burning \mid extinguished$

FireProperty

$xwidth, ywidth, zwidth : \mathbb{R}$
 $position : Position$
 $orientation : Orientation$
 $locations : \mathbb{P}\ Position$
 $status : StatusType$

 $locations = boxLocs\ position\ orientation\ xwidth\ ywidth\ zwidth$

D.2 Channels

D.2.1 Software channels

$InOut ::= in \mid out$

channel*fireDetected* : *InOut*
channel*fireLost* : *InOut*
channel*critical* : *InOut*
channel*spray* : *InOut* \times \mathbb{B}
channel*landed* : *InOut*
channel*takeOffCall*
channel*goToBuildingCall*
channel*searchFireCall*
channel*goHomeCall*

D.2.2 Input event triggered channels

channel*fireDetectedTriggered* : \mathbb{B}
channel*fireLostTriggered* : \mathbb{B}
channel*criticalTriggered* : \mathbb{B}
channel*landedTriggered* : \mathbb{B}

D.2.3 Output clock reset channels

channel*sprayHappened*
channel*takeOffHappened*
channel*goToBuildingHappened*
channel*searchFireHappened*
channel*goHomeHappened*

D.2.4 Variable get/set channels

channel*getRobotPosition* : *Position*
channel*getRobotVelocity* : *Velocity*
channel*getRobotAcceleration* : *Acceleration*
channel*getRobotOrientation* : *Orientation*
channel*getRobotAngularVelocity* : *AngularVelocity*
channel*getRobotAngularAcceleration* : *AngularAcceleration*
channel*getRobotTank_of_water* : *Tank_of_waterType*
channel*getRobotSearchPattern* : *seq Position*
channel*setRobotPosition* : *Position*
channel*setRobotVelocity* : *Velocity*
channel*setRobotAcceleration* : *Acceleration*
channel*setRobotOrientation* : *Orientation*
channel*setRobotAngularVelocity* : *AngularVelocity*
channel*setRobotAngularAcceleration* : *AngularAcceleration*
channel*setRobotTank_of_water* : *Tank_of_waterType*
channel*setRobotSearchPattern* : *seq Position*

channel *getBuildingPosition* : *Position*
channel *getBuildingOrientation* : *Orientation*
channel *setBuildingPosition* : *Position*
channel *setBuildingOrientation* : *Orientation*

channel *getNumFires* : \mathbb{N}
channel *getFirePosition* : $\mathbb{N} \times \textit{Position}$
channel *getFireOrientation* : $\mathbb{N} \times \textit{Orientation}$
channel *getFireStatus* : $\mathbb{N} \times \textit{StatusType}$
channel *setFirePosition* : $\mathbb{N} \times \textit{Position}$
channel *setFireOrientation* : $\mathbb{N} \times \textit{Orientation}$
channel *setFireStatus* : $\mathbb{N} \times \textit{StatusType}$

channel *proceed*

D.3 Global Constants

arena : *ArenaProperty*
robotInit : *RobotProperty*
building : *BuildingProperty*
potentialFires : $\mathbb{P} \textit{FireProperty}$

groundLocations : $\mathbb{P}(\mathbb{R} \times \mathbb{R} \times \mathbb{R})$
groundLocations = $\{x, y, z : \mathbb{R} \mid (x, y, z) \in \textit{arena.locations} \wedge z = 0\}$

arena.xwidth = 50.0
arena.ywidth = 60.0
arena.zwidth \geq *building.zwidth* + 1.0
arena.gradient = 0.0
arena.windSpeed < 8.0

arena.home.xwidth = 1.0
arena.home.ywidth = 1.0
locsOnLocs arena.home.locations groundLocations = **True**
arena.home.locations \subseteq *arena.locations*

robotInit.position \in *arena.home.locations*
robotInit.position \in *arena.locations*

$$\begin{aligned}
& \neg (\text{building.xwidth} < 10.0) \\
& \neg (\text{building.xwidth} > 30.0) \\
& \neg (\text{building.ywidth} < 10.0) \\
& \neg (\text{building.ywidth} > 40.0) \\
& \neg (\text{building.zwidth} < 6.0) \\
& \neg (\text{building.zwidth} > 20.0) \\
& \text{building.locations} \subseteq \text{arena.locations}
\end{aligned}$$

$$\begin{aligned}
& \forall \text{fire} : \text{potentialFires} \bullet \text{fire.xwidth} = 0.036 \\
& \forall \text{fire} : \text{potentialFires} \bullet \text{fire.ywidth} = 0.0 \\
& \forall \text{fire} : \text{potentialFires} \bullet \text{fire.zwidth} = 0.060 \\
& \forall \text{fire} : \text{potentialFires} \bullet \\
& \quad \text{locsOnLocs fire.locations groundLocations} = \mathbf{True} \\
& \quad \vee (\text{locsOnLocs fire.locations building.locations} = \mathbf{True} \\
& \quad \quad \wedge \text{fire.position.3} \in 5.0 \dots 18.0) \\
& \forall \text{fire} : \text{potentialFires} \bullet \text{fire.locations} \subseteq \text{arena.locations}
\end{aligned}$$

D.4 Environment

$$\text{timeStep} : \mathbb{R}$$

$$\mathbf{process} \text{Environment} \triangleq \mathbf{begin}$$

D.4.1 Environment State

EventTimes

$fireDetectedTime : \mathbb{R}$
 $fireDetectedOccurred : \mathbb{B}$
 $fireLostTime : \mathbb{R}$
 $fireLostOccurred : \mathbb{B}$
 $criticalTime : \mathbb{R}$
 $criticalOccurred : \mathbb{B}$
 $landedTime : \mathbb{R}$
 $landedOccurred : \mathbb{B}$

 $sprayTime : \mathbb{R}$
 $sprayOccurred : \mathbb{B}$
 $takeOffTime : \mathbb{R}$
 $takeOffOccurred : \mathbb{B}$
 $goToBuildingTime : \mathbb{R}$
 $goToBuildingOccurred : \mathbb{B}$
 $searchFireTime : \mathbb{R}$
 $searchFireOccurred : \mathbb{B}$
 $goHomeTime : \mathbb{R}$
 $goHomeOccurred : \mathbb{B}$

*EventTimesInit**EventTimes'*

$fireDetectedTime' = 0.0$
 $fireDetectedOccurred' = \mathbf{False}$
 $fireLostTime' = 0.0$
 $fireLostOccurred' = \mathbf{False}$
 $criticalTime' = 0.0$
 $criticalOccurred' = \mathbf{False}$
 $landedTime' = 0.0$
 $landedOccurred' = \mathbf{False}$

 $sprayTime' = 0.0$
 $sprayOccurred' = \mathbf{False}$
 $takeOffTime' = 0.0$
 $takeOffOccurred' = \mathbf{False}$
 $goToBuildingTime' = 0.0$
 $goToBuildingOccurred' = \mathbf{False}$
 $searchFireTime' = 0.0$
 $searchFireOccurred' = \mathbf{False}$
 $goHomeTime' = 0.0$
 $goHomeOccurred' = \mathbf{False}$

EnvironmentState

visible *robot* : *RobotProperty*
visible *fires* : *seqFireProperty*
time : \mathbb{R}
stepTimer : \mathbb{R}
EventTimes

state *EnvironmentState*

EnvironmentStateInit

EnvironmentState'

robot' = *robotInit*
ranfires' \subseteq *potentialFires*
time' = 0.0
stepTimer' = 0.0
EventTimesInit

D.4.2 Robot Movement

RobotMovement

\wedge *EnvironmentState*

$\frac{drobot.position}{dt} = robot.velocity$
 $\frac{drobot.velocity}{dt} = robot.acceleration$
 $\frac{drobot.acceleration}{dt} = 0$
 $\frac{drobot.orientation}{dt} = robot.angularVelocity$
 $\frac{drobot.angularVelocity}{dt} = robot.angularAcceleration$
 $\frac{drobot.angularAcceleration}{dt} = 0$
 $\frac{dtime}{dt} = 1$
 $\frac{dstepTimer}{dt} = 1$

RobotMovementAction \triangleq (*RobotMovement*)

$$\Delta \left(\begin{array}{l} (robot.position \in groundLocations \wedge robot.velocity.3 < 0) \\ \vee (robot.position \in building.locations \\ \wedge (robot.velocity \cdot (building.position - robot.position) > 0)) \\ \vee (\exists fire : ranfires \bullet \\ robot.position \in fire.locations \\ \wedge (robot.velocity \cdot (fire.position - robot.position) > 0)) \\ \vee (time \geq timeStep) \end{array} \right)$$

D.4.3 Collision Detection

$\text{CollisionDetection} \hat{=}$

$\text{RobotGroundCollision}$

\square

$\text{RobotBuildingCollision}$

\square

$\text{RobotFireCollision}$

StopRobot

$\Delta\text{EnvironmentState}$

$\text{robot}'.\text{velocity} = (0,0,0)$

$\text{robot}'.\text{acceleration} = (0,0,0)$

$\text{robot}'.\text{position} = \text{robot}.\text{position}$

$\text{robot}'.\text{orientation} = \text{robot}.\text{orientation}$

$\text{robot}'.\text{angularVelocity} = \text{robot}.\text{angularVelocity}$

$\text{robot}'.\text{angularAcceleration} = \text{robot}.\text{angularAcceleration}$

$\text{robot}'.\text{tank_of_water} = \text{robot}.\text{tank_of_water}$

$\text{robot}'.\text{searchPattern} = \text{robot}.\text{searchPattern}$

$\text{fires}' = \text{fires}$

$\text{time}' = \text{time}$

$\exists \text{EventTimes}$

$\text{RobotGroundCollision} \hat{=}$

$(\text{robot}.\text{position}.3 = 0 \wedge \text{robot}.\text{velocity}.3 < 0) \ \&$
 (StopRobot)

$\text{RobotBuildingCollision} \hat{=}$

$(\text{robot}.\text{position} \in \text{building}.\text{locations} \wedge (\text{robot}.\text{velocity} \cdot (\text{building}.\text{position} - \text{robot}.\text{position}) > 0)) \ \&$
 (StopRobot)

$\text{RobotFireCollision} \hat{=}$

$(\exists \text{fire} : \text{ran fires} \bullet$
 $\text{robot}.\text{position} \in \text{fire}.\text{locations} \wedge (\text{robot}.\text{velocity} \cdot (\text{fire}.\text{position} - \text{robot}.\text{position}) > 0)) \ \&$
 (StopRobot)

D.4.4 Communication Actions that occur on the time step

$\text{InputTriggers} \hat{=} \text{fireDetected_InputEventMapping};$

$\text{fireLost_InputEventMapping};$

$\text{critical_InputEventMapping};$

$\text{landed_InputEventMapping}$

$$\begin{aligned}
& fireDetected_InputEventMapping \hat{=} \\
& \quad \mathbf{if}(\exists fire1 : \mathbf{ranfires} \bullet \neg (norm(fire1.position - robot.position) > 0.5)) \longrightarrow \\
& \quad \quad fireDetectedTriggered!\mathbf{True} \\
& \quad \quad \longrightarrow fireDetectedOccurred, fireDetectedTime := \mathbf{True}, time \\
& \quad \square \neg (\exists fire1 : \mathbf{ranfires} \bullet \neg (norm(fire1.position - robot.position) > 0.5)) \longrightarrow \\
& \quad \quad fireDetectedTriggered!\mathbf{False} \longrightarrow \mathbf{Skip} \quad \mathbf{fi}
\end{aligned}$$

$$\begin{aligned}
& fireLost_InputEventMapping \hat{=} \\
& \quad \mathbf{if}(\exists fire1 : \mathbf{ranfires} \bullet norm(fire1.position - robot.position) > 0.5) \longrightarrow \\
& \quad \quad fireLostTriggered!\mathbf{True} \\
& \quad \quad \longrightarrow fireLostOccurred, fireLostTime := \mathbf{True}, time \\
& \quad \square \neg (\exists fire1 : \mathbf{ranfires} \bullet norm(fire1.position - robot.position) > 0.5) \longrightarrow \\
& \quad \quad fireLostTriggered!\mathbf{False} \longrightarrow \mathbf{Skip} \quad \mathbf{fi}
\end{aligned}$$

$$\begin{aligned}
& critical_InputEventMapping \hat{=} \\
& \quad \mathbf{if}(sprayOccurred = \mathbf{True} \wedge sprayTime \geq 180.0) \\
& \quad \quad \vee (takeOffOccurred = \mathbf{True} \wedge takeOffTime \geq 1200.0) \longrightarrow \\
& \quad \quad \quad criticalTriggered!\mathbf{True} \\
& \quad \quad \quad \longrightarrow criticalOccurred, criticalTime := \mathbf{True}, time \\
& \quad \square ((sprayOccurred = \mathbf{True} \wedge sprayTime \geq 180.0) \\
& \quad \quad \vee (takeOffOccurred = \mathbf{True} \wedge takeOffTime \geq 1200.0)) \longrightarrow \\
& \quad \quad \quad criticalTriggered!\mathbf{False} \longrightarrow \mathbf{Skip} \quad \mathbf{fi}
\end{aligned}$$

$$\begin{aligned}
& landed_InputEventMapping \hat{=} \\
& \quad \mathbf{if}(robot.position.3 = 0.0) \longrightarrow \\
& \quad \quad landedTriggered!\mathbf{True} \\
& \quad \quad \longrightarrow landedOccurred, landedTime := \mathbf{True}, time \\
& \quad \square \neg (robot.position.3 = 0.0) \longrightarrow \\
& \quad \quad landedTriggered!\mathbf{False} \longrightarrow \mathbf{Skip} \quad \mathbf{fi}
\end{aligned}$$

$$Communication \hat{=} \mathbf{Skip}$$

D.4.5 Input Event Buffers

$$\begin{aligned}
InputEventBuffers \hat{=} & fireDetected_Buffer \parallel fireLost_Buffer \\
& \parallel critical_Buffer \parallel landed_Buffer
\end{aligned}$$

$$\begin{aligned} \text{fireDetected_Buffer} \hat{=} & \text{var fireDetectedTrig} : \mathbb{B} \bullet \text{fireDetectedTrig} := \text{False}; \\ & \left(\begin{array}{l} \text{fireDetectedTriggered?}b \longrightarrow \text{fireDetectedTrig} := b \\ \square \\ (\text{fireDetectedTrig} = \text{True}) \ \& \ \text{fireDetected.in} \longrightarrow \text{Skip} \end{array} \right); \text{fireDetected_Buffer} \end{aligned}$$

$$\begin{aligned} \text{fireLost_Buffer} \hat{=} & \text{var fireLostTrig} : \mathbb{B} \bullet \text{fireLostTrig} := \text{False}; \\ & \left(\begin{array}{l} \text{fireLostTriggered?}b \longrightarrow \text{fireLostTrig} := b \\ \square \\ (\text{fireLostTrig} = \text{True}) \ \& \ \text{fireLost.in} \longrightarrow \text{Skip} \end{array} \right); \text{fireLost_Buffer} \end{aligned}$$

$$\begin{aligned} \text{critical_Buffer} \hat{=} & \text{var criticalTrig} : \mathbb{B} \bullet \text{criticalTrig} := \text{False}; \\ & \left(\begin{array}{l} \text{criticalTriggered?}b \longrightarrow \text{criticalTrig} := b \\ \square \\ (\text{criticalTrig} = \text{True}) \ \& \ \text{critical.in} \longrightarrow \text{Skip} \end{array} \right); \text{critical_Buffer} \end{aligned}$$

$$\begin{aligned} \text{landed_Buffer} \hat{=} & \text{var landedTrig} : \mathbb{B} \bullet \text{landedTrig} := \text{False}; \\ & \left(\begin{array}{l} \text{landedTriggered?}b \longrightarrow \text{landedTrig} := b \\ \square \\ (\text{landedTrig} = \text{True}) \ \& \ \text{landed.in} \longrightarrow \text{Skip} \end{array} \right); \text{landed_Buffer} \end{aligned}$$

D.4.6 Output Event Buffers

$$\begin{aligned} \text{OutputEventBuffers} \hat{=} & \text{spray_Buffer} \parallel \text{takeOff_Buffer} \parallel \text{goToBuilding_Buffer} \\ & \parallel \text{searchFire_Buffer} \parallel \text{goHome_Buffer} \end{aligned}$$

$$\begin{aligned} \text{spray_Buffer} \hat{=} & \text{sprayHappened} \\ & \longrightarrow \text{sprayOccurred}, \text{sprayTime} := \text{True}, \text{time}; \\ & \text{spray_Buffer} \end{aligned}$$

$$\begin{aligned} \text{takeOff_Buffer} \hat{=} & \text{takeOffHappened} \\ & \longrightarrow \text{takeOffOccurred}, \text{takeOffTime} := \text{True}, \text{time}; \\ & \text{takeOff_Buffer} \end{aligned}$$

$$\begin{aligned} \text{goToBuilding_Buffer} \hat{=} & \text{goToBuildingHappened} \\ & \longrightarrow \text{goToBuildingOccurred}, \text{goToBuildingTime} := \text{True}, \text{time}; \\ & \text{goToBuilding_Buffer} \end{aligned}$$

$$\begin{aligned} searchFire_Buffer &\hat{=} searchFireHappened \\ &\longrightarrow searchFireOccurred, searchFireTime := \mathbf{True}, time; \\ searchFire_Buffer \end{aligned}$$

$$\begin{aligned} goHome_Buffer &\hat{=} goHomeHappened \\ &\longrightarrow goHomeOccurred, goHomeTime := \mathbf{True}, time; \\ goHome_Buffer \end{aligned}$$

D.4.7 Environment main action

$$\begin{aligned} EnvironmentLoop &\hat{=} (EnvironmentStateInit); \mu X \bullet \\ &\quad RobotMovementAction; \\ &\quad \left(\begin{array}{l} (time < timeStep) \& CollisionDetection \\ \square \\ (time \geq timeStep) \& InputTriggers; Communication \triangle_0 time := 0 \end{array} \right); X \end{aligned}$$

channelset triggerChannels ==
 $\{\{fireDetectedTriggered, fireLostTriggered, criticalTriggered, landedTriggered\}$

nameset EnvVars == {robot, building, fires, time, sprayTime, takeOffTime}

$\bullet (EnvironmentLoop \llbracket EnvVars \mid triggerChannels \mid \emptyset \rrbracket InputEventBuffers)$
 $\backslash triggerChannels$

end

D.5 Mapping

channel tock

channel getRobot : RobotProperty

channel setRobot : RobotProperty

PackRobotProperty

$robotPos? : Position$; $robotVel? : Velocity$; $robotAcc? : Acceleration$
 $robotOri? : Orientation$; $robotAngVel? : AngularVelocity$
 $robotAngAcc? : AngularAcceleration$
 $robotTank_of_water? : Tank_of_waterType$
 $robotSearchPattern? : seqPosition$
 $robotProperty! : RobotProperty$

$robotProperty!.position = robotPos?$
 $robotProperty!.velocity = robotVel?$
 $robotProperty!.acceleration = robotAcc?$
 $robotProperty!.orientation = robotOri?$
 $robotProperty!.angularVelocity = robotAngVel?$
 $robotProperty!.angularAcceleration = robotAngAcc?$
 $robotProperty!.tank_of_water = robotTank_of_water?$
 $robotProperty!.searchPattern = robotSearchPattern?$

UnpackRobotProperty

$robotProperty? : RobotProperty$
 $robotPos! : Position$; $robotVel! : Velocity$; $robotAcc! : Acceleration$
 $robotOri! : Orientation$; $robotAngVel! : AngularVelocity$
 $robotAngAcc! : AngularAcceleration$
 $robotTank_of_water! : Tank_of_waterType$
 $robotSearchPattern! : seqPosition$

$robotPos! = robotProperty?.position$
 $robotVel! = robotProperty?.velocity$
 $robotAcc! = robotProperty?.acceleration$
 $robotOri! = robotProperty?.orientation$
 $robotAngVel! = robotProperty?.angularVelocity$
 $robotAngAcc! = robotProperty?.angularAcceleration$
 $robotTank_of_water! = robotProperty?.tank_of_water$
 $robotSearchPattern! = robotProperty?.searchPattern$

D.5.1 spray **Output Event Mapping**

process *spray_OutputEventMapping* $\hat{=}$ **begin**

spraymappingDiagram $\hat{=}$...

spray_Conditions $\hat{=}$ **var** *robotTank_of_water* : *Tank_of_waterType* •

μX •

*getRobotTank_of_water?**x* : ($x \neq robotTank_of_water$)

$\longrightarrow robotTank_of_water := x$; *X*

□

*spray?**b* $\longrightarrow X$

$$\begin{aligned}
& \text{spray_Semantics} \hat{=} \\
& \left(\left(\left(\begin{array}{l} (\text{spraymappingDiagram } \llbracket \emptyset \mid \{\text{spray}\} \mid \emptyset \rrbracket \text{ spray_Conditions}) \\ \llbracket \emptyset \mid \{\text{tock}\} \mid \emptyset \rrbracket \\ \text{ConvertTocks} \end{array} \right) \setminus \{\text{tock}\} \right) \right) \\
& \left(\left(\begin{array}{l} \llbracket \emptyset \mid \{\text{getRobot}, \text{setRobot}\} \mid \emptyset \rrbracket \\ \text{ConvertRobotChannels} \\ \setminus \{\text{getRobot}, \text{setRobot}\}; \text{spray_Semantics} \end{array} \right) \right) \\
& \square \\
& \text{proceed} \longrightarrow \text{spray_Semantics}
\end{aligned}$$

$$\begin{aligned}
& \text{ConvertTocks} \hat{=} \text{tock} \longrightarrow \\
& (\text{var } \text{proceedCount} : \mathbb{N} \bullet \text{proceedCount} := 0; \mu X \bullet \\
& \quad \text{if } \text{proceedCount} < 1.0/\text{timeStep} \longrightarrow \text{proceed} \longrightarrow X \\
& \quad \llbracket \text{proceedCount} \geq 1.0/\text{timeStep} \longrightarrow \text{ConvertTocks} \\
& \quad \text{fi})
\end{aligned}$$

$\text{ConvertRobotChannels} \hat{=} \text{var } \text{robotPos} : \text{Position}; \text{robotVel} : \text{Velocity};$
 $\quad \text{robotAcc} : \text{Acceleration} \bullet$
 $\text{var } \text{robotOri} : \text{Orientation}; \text{robotAngVel} : \text{AngularVelocity};$
 $\quad \text{robotAngAcc} : \text{AngularAcceleration} \bullet$
 $\text{var } \text{robotTank_of_water} : \text{Tank_of_waterType} \bullet$
 $\text{var } \text{robotSearchPattern} : \text{seq Position} \bullet$
 $\mu X \bullet$
 $\quad \text{getRobotPosition}?x : (x \neq \text{robotPos}) \longrightarrow \text{robotPos} := x; X$
 $\quad \square$
 $\quad \text{getRobotVelocity}?x : (x \neq \text{robotVel}) \longrightarrow \text{robotVel} := x; X$
 $\quad \square$
 $\quad \text{getRobotAcceleration}?x : (x \neq \text{robotAcc}) \longrightarrow \text{robotAcc} := x; X$
 $\quad \square$
 $\quad \text{getRobotOrientation}?x : (x \neq \text{robotOri}) \longrightarrow \text{robotOri} := x; X$
 $\quad \square$
 $\quad \text{getRobotAngularVelocity}?x : (x \neq \text{robotAngVel}) \longrightarrow \text{robotAngVel} := x; X$
 $\quad \square$
 $\quad \text{getRobotAngularAcceleration}?x : (x \neq \text{robotAngAcc}) \longrightarrow \text{robotAngAcc} := x; X$
 $\quad \square$
 $\quad \text{getRobotTank_of_water}?x : (x \neq \text{robotTank_of_water}) \longrightarrow \text{robotTank_of_water} := x; X$
 $\quad \square$
 $\quad \text{getRobotSearchPattern}?x : (x \neq \text{robotSearchPattern}) \longrightarrow \text{robotSearchPattern} := x; X$
 $\quad \square$
 $\quad (\text{var } \text{robotProperty} : \text{RobotProperty} \bullet (\text{PackRobotProperty});$
 $\quad \quad \text{getRobot!robotProperty} \longrightarrow \text{Skip}); X$
 $\quad \square$
 $\quad \text{setRobot?robotProperty} \longrightarrow (\text{UnpackRobotProperty});$
 $\quad \quad \text{setRobotPosition!robotPos} \longrightarrow$
 $\quad \quad \text{setRobotVelocity!robotVel} \longrightarrow$
 $\quad \quad \text{setRobotAcceleration!robotAcc} \longrightarrow$
 $\quad \quad \text{setRobotOrientation!robotOri} \longrightarrow$
 $\quad \quad \text{setRobotAngularVelocity!robotAngVel} \longrightarrow$
 $\quad \quad \text{setRobotAngularAcceleration!robotAngAcc} \longrightarrow$
 $\quad \quad \text{setRobotTank_of_water!robotTank_of_water} \longrightarrow$
 $\quad \quad \text{setRobotSearchPattern!robotSearchPattern} \longrightarrow X$

 $\text{spray_Monitor} \hat{=} \text{spray}?b \longrightarrow \text{sprayHappened} \longrightarrow \text{Skip}$

 $\bullet \text{ spray_Semantics } \llbracket \emptyset \mid \{ \text{spray} \} \mid \emptyset \rrbracket \text{ spray_Monitor}$

 end

D.5.2 takeoff Operation Mapping

$\text{process takeOff_OperationMapping} \hat{=} \text{begin}$

$\text{takeOff_Semantics} \hat{=} \mu X \bullet \text{takeOffCall} \longrightarrow \text{setRobotVelocity}!(0, 0, 1.0) \longrightarrow X$

$$takeOff_Monitor \triangleq \mu X \bullet takeOffCall \longrightarrow takeOffHappened \longrightarrow X$$

$$\bullet takeOff_Semantics \llbracket \emptyset \mid \{ \{ takeOffCall \} \mid \emptyset \} \rrbracket takeOff_Monitor$$

end

D.5.3 goToBuilding Operation Mapping

process *goToBuilding_OperationMapping* \triangleq **begin**

$$goToBuilding_Semantics \triangleq goToBuildingCall$$

$$\longrightarrow getRobotPosition?robotPos \longrightarrow getBuildingPosition?buildingPos$$

$$\longrightarrow (setRobotVelocity!(1.0 * ((buildingPos - robotPos) / norm(builtingPos - robotPos))))$$

$$\longrightarrow \mathbf{Skip};$$

$$proceed \longrightarrow goToBuilding_Semantics$$

$$goToBuilding_Monitor \triangleq goToBuildingCall \longrightarrow goToBuildingHappened \longrightarrow goToBuilding_Monitor$$

$$\bullet goToBuilding_Semantics \llbracket \emptyset \mid \{ \{ goToBuildingCall \} \mid \emptyset \} \rrbracket goToBuilding_Monitor$$

end

D.5.4 goHome Operation Mapping

process *goHome_OperationMapping* \triangleq **begin**

$$goHome_Semantics \triangleq goHomeCall$$

$$\longrightarrow getRobotPosition?robotPos$$

$$\longrightarrow (setRobotVelocity!(1.0 * ((arena.home.position - robotPos) / norm(arena.home.position - robotPos))))$$

$$\longrightarrow \mathbf{Skip};$$

$$proceed \longrightarrow goHome_Semantics$$

$$goHome_Monitor \triangleq goHomeCall \longrightarrow goHomeHappened \longrightarrow goHome_Monitor$$

$$\bullet goHome_Semantics \llbracket \emptyset \mid \{ \{ goHomeCall \} \mid \emptyset \} \rrbracket goHome_Monitor$$

end

D.5.5 searchFire Operation Mapping

process *searchFire_OperationMapping* \triangleq **begin**

$$searchFireDiagram \triangleq \dots$$

$$\begin{aligned}
searchFire_Semantics &\triangleq \\
&\left(\left(\left(\begin{array}{c} searchFireDiagram \\ \llbracket \emptyset \mid \{tock\} \mid \emptyset \rrbracket \\ ConvertTocks \end{array} \right) \setminus \{tock\} \right) \right. \\
&\quad \left. \left(\begin{array}{c} \llbracket \emptyset \mid \{getRobot, setRobot\} \mid \emptyset \rrbracket \\ ConvertRobotChannels \\ \setminus \{getRobot, setRobot\}; searchFire_Semantics \end{array} \right) \right) \\
&\square \\
&proceed \longrightarrow searchFire_Semantics
\end{aligned}$$

$$\begin{aligned}
ConvertTocks &\triangleq tock \longrightarrow \\
&(\text{var } proceedCount : \mathbb{N} \bullet proceedCount := 0; \mu X \bullet \\
&\quad \text{if } proceedCount < 1.0/timeStep \longrightarrow proceed \longrightarrow X \\
&\quad \llbracket proceedCount \geq 1.0/timeStep \longrightarrow ConvertTocks \\
&\quad \text{fi})
\end{aligned}$$

```

ConvertRobotChannels  $\hat{=}$  var robotPos : Position; robotVel : Velocity;
    robotAcc : Acceleration •
var robotOri : Orientation; robotAngVel : AngularVelocity;
    robotAngAcc : AngularAcceleration •
var robotTank_of_water : Tank_of_waterType •
var robotSearchPattern : seq Position •
 $\mu X$  •
    getRobotPosition?x : (x  $\neq$  robotPos)  $\longrightarrow$  robotPos := x; X
    □
    getRobotVelocity?x : (x  $\neq$  robotVel)  $\longrightarrow$  robotVel := x; X
    □
    getRobotAcceleration?x : (x  $\neq$  robotAcc)  $\longrightarrow$  robotAcc := x; X
    □
    getRobotOrientation?x : (x  $\neq$  robotOri)  $\longrightarrow$  robotOri := x; X
    □
    getRobotAngularVelocity?x : (x  $\neq$  robotAngVel)  $\longrightarrow$  robotAngVel := x; X
    □
    getRobotAngularAcceleration?x : (x  $\neq$  robotAngAcc)  $\longrightarrow$  robotAngAcc := x; X
    □
    getRobotTank_of_water?x : (x  $\neq$  robotTank_of_water)  $\longrightarrow$  robotTank_of_water := x; X
    □
    getRobotSearchPattern?x : (x  $\neq$  robotSearchPattern)  $\longrightarrow$  robotSearchPattern := x; X
    □
    (var robotProperty : RobotProperty • (PackRobotProperty);
        getRobot!robotProperty  $\longrightarrow$  Skip); X
    □
    setRobot?robotProperty  $\longrightarrow$  (UnpackRobotProperty);
        setRobotPosition!robotPos  $\longrightarrow$ 
        setRobotVelocity!robotVel  $\longrightarrow$ 
        setRobotAcceleration!robotAcc  $\longrightarrow$ 
        setRobotOrientation!robotOri  $\longrightarrow$ 
        setRobotAngularVelocity!robotAngVel  $\longrightarrow$ 
        setRobotAngularAcceleration!robotAngAcc  $\longrightarrow$ 
        setRobotTank_of_water!robotTank_of_water  $\longrightarrow$ 
        setRobotSearchPattern!robotSearchPattern  $\longrightarrow$  X

searchFire_Monitor  $\hat{=}$  searchFireCall  $\longrightarrow$  searchFireHappened  $\longrightarrow$  Skip

• searchFire_Semantics  $\llbracket \emptyset \mid \{ \} \text{searchFireCall} \} \mid \emptyset \rrbracket$  searchFire_Monitor

end

processMapping  $\hat{=}$  spray_OutputEventMapping  $\lll$  goToBuilding_OperationMapping
     $\lll$  takeOff_OperationMapping  $\lll$  goHome_OperationMapping
     $\lll$  searchFire_OperationMapping

```

D.6 Composition

channelset *getSetChannels* == { *getRobotPosition*, *getRobotVelocity*, *getRobotAcceleration*, *getRobotOrientation*, *getRobotAngularVelocity*, *setRobotPosition*, *setRobotVelocity*, *setRobotAcceleration*, *setRobotOrientation*, *setRobotAngularVelocity*, *setRobotAngularAcceleration*, *getBuildingPosition*, *getBuildingOrientation*, *setBuildingPosition*, *setBuildingOrientation*, *getNumFires*, *getFirePosition*, *getFireOrientation*, *getFireStatus*, *setFirePosition*, *setFireOrientation*, *setFireStatus* }











channelset *eventHappenedChannels* == { *sprayHappened*, *takeOffHappened*, *goToBuildingHappened*, *searchFireHappened*, *goHomeHappened* }









process *RoboWorldDocument* $\hat{=}$
 (Environment [*getSetChannels* \cup *eventHappenedChannels*] Mapping)
 \ *getSetChannels* \cup *eventHappenedChannels*

Credits

L^AT_EX style based on the *The Legrand Orange Book Template* by Mathias Legrand and Vel from LaTeXTemplates.com. Licensed under [CC BY-NC-SA 3.0](https://creativecommons.org/licenses/by-nc-sa/3.0/).

Icons used in RoboTool and this report have been obtained from www.flaticon.com. Individual credits are given below.

-  Icon made by [Iconnice](#) from www.flaticon.com is licensed by [CC 3.0 BY](https://creativecommons.org/licenses/by/3.0/)
-  Icon made by [Sarfraz Shoukat](#) from www.flaticon.com is licensed by [CC 3.0 BY](https://creativecommons.org/licenses/by/3.0/)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](https://creativecommons.org/licenses/by/3.0/)
-  Icon made by [Dario Ferrando](#) from www.flaticon.com is licensed by [CC 3.0 BY](https://creativecommons.org/licenses/by/3.0/)
-  Icon made by [Lyolya](#) from www.flaticon.com is licensed by [CC 3.0 BY](https://creativecommons.org/licenses/by/3.0/)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](https://creativecommons.org/licenses/by/3.0/)
-  Icon made by [Google](#) from www.flaticon.com is licensed by [CC 3.0 BY](https://creativecommons.org/licenses/by/3.0/)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](https://creativecommons.org/licenses/by/3.0/)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](https://creativecommons.org/licenses/by/3.0/)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](https://creativecommons.org/licenses/by/3.0/)

-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
-  Icon made by [Revicon](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
- F** Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
- O** Icon made by [Icomoon](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
- π** Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
- P** Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
- ®** Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
- ®** Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
-  Icon made by [Freepik](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)
-  Icon made by [Popcic](#) from www.flaticon.com is licensed by [CC 3.0 BY](#)

Bibliography

- [1] A. Burns, I. J. Hayes, and C. B. Jones. “Deriving specifications of control programs for cyber physical systems”. In: *The Computer Journal* 63.5 (2020), pages 774–790 (cited on page 10).
- [2] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. “A Refinement Strategy for *Circus*”. In: *Formal Aspects of Computing* 15.2 - 3 (2003), pages 146–181. DOI: [10.1007/s00165-003-0006-5](https://doi.org/10.1007/s00165-003-0006-5). URL: [papers/CSW03.pdf](https://papers.csw03.pdf) (cited on page 16).
- [3] A. L. C. Cavalcanti, J. Baxter, and G. Carvalho. “RoboWorld: Where Can My Robot Work?” In: *Software Engineering and Formal Methods*. Edited by R. Calinescu and C. S. Păsăreanu. Lecture Notes in Computer Science. Springer, 2021, pages 3–22. DOI: https://doi.org/10.1007/978-3-030-92124-8_1 (cited on page 11).
- [4] A. L. C. Cavalcanti et al. “Verified simulation for robotics”. In: *Science of Computer Programming* 174 (2019), pages 1–37. DOI: doi.org/10.1016/j.scico.2019.01.004. URL: papers/CSMRCD19.pdf (cited on page 20).
- [5] A. L. C. Cavalcanti et al. “RoboStar Technology: A Robotist’s Toolbox for Combined Proof, Simulation, and Testing”. In: *Software Engineering for Robotics*. Edited by A. L. C. Cavalcanti et al. Springer International Publishing, 2021, pages 249–293. DOI: [10.1007/978-3-030-66494-7_9](https://doi.org/10.1007/978-3-030-66494-7_9). URL: papers/CBBCFMRS21.pdf (cited on page 9).
- [6] A. L. C. Cavalcanti et al., editors. *Software Engineering for Robotics*. Springer International Publishing, 2021. ISBN: 978-3-030-66493-0. DOI: [10.1007/978-3-030-66494-7](https://doi.org/10.1007/978-3-030-66494-7) (cited on page 9).

- [7] A. Miyazawa et al. “RoboChart: modelling and verification of the functional behaviour of robotic applications”. In: *Software & Systems Modeling* 18.5 (2019), pages 3097–3149. DOI: doi.org/10.1007/s10270-018-00710-z. URL: rdcu.be/bh7dI (cited on page 11).
- [8] A. Miyazawa et al. *RoboChart: Modelling, Verification and Simulation for Robotics*. Technical report. Available at www.cs.york.ac.uk/robostar/notations/. York, UK: University of York, Department of Computer Science, 2020 (cited on page 29).
- [9] A. Miyazawa et al. *RoboSim Physical Modelling: Diagrammatic Physical Robot Models*. Technical report. Available at robostar.cs.york.ac.uk/notations/. York, UK: University of York, Department of Computer Science, 2020 (cited on pages 20, 29).
- [10] A. Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, 2011 (cited on pages 10, 11, 31).
- [11] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011 (cited on page 15).
- [12] B. Luteberget. “Automated Reasoning for Planning Railway Infrastructure”. PhD thesis. University of Oslo, 2019 (cited on pages 79, 84).
- [13] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998 (cited on page 9).
- [14] Ana Cavalcanti et al. “Verified simulation for robotics”. In: *Science of Computer Programming* 174 (2019), pages 1–37. ISSN: 0167-6423. DOI: <https://doi.org/10.1016/j.scico.2019.01.004>. URL: <http://www.sciencedirect.com/science/article/pii/S0167642318301655>.
- [15] G. Frehse et al. “SpaceEx: Scalable Verification of Hybrid Systems”. In: *Computer Aided Verification*. Edited by G. Gopalakrishnan and S. Qadeer. Volume 6806. Lecture Notes in Computer Science. Springer, 2011, pages 379–395 (cited on page 84).
- [16] J. Baxter, P. Ribeiro, and A. L. C. Cavalcanti. “Sound reasoning in tock-CSP”. In: *Acta Informatica* 59 (2022), pages 125–162. DOI: [10.1007/s00236-020-00394-3](https://doi.org/10.1007/s00236-020-00394-3) (cited on page 16).
- [17] J. C. P. Woodcock and J. Davies. *Using Z - Specification, Refinement, and Proof*. Prentice-Hall, 1996 (cited on pages 16, 45).
- [18] J. H. Y. Munive, G. Struth, and S. Foster. “Differential Hoare Logics and Refinement Calculi for Hybrid Systems with Isabelle/HOL”. In: *18th International Conference on Relational and Algebraic Methods in Computer Science*. Volume 12062. Lecture Notes in Computer Science. Springer, 2020, pages 169–186 (cited on pages 11, 16).
- [19] M. Kwiatkowska, G. Norman, and D. Parker. “Probabilistic symbolic model checking with PRISM: a hybrid approach”. In: *International Journal on Software Tools for Technology Transfer* 6.2 (2004), pages 128–142 (cited on page 10).

- [20] Alvaro Miyazawa et al. “RoboChart: Modelling and verification of the functional behaviour of robotic applications”. In: *Software & Systems Modeling* (Jan. 2019). ISSN: 1619-1374. DOI: [10.1007/s10270-018-00710-z](https://doi.org/10.1007/s10270-018-00710-z). URL: doi.org/10.1007/s10270-018-00710-z.
- [21] S. Foster et al. “Automating Verification of State Machines with Reactive Designs and Isabelle/UTP”. In: *Formal Aspects of Component Software*. Edited by K. Bae and P. C. Ölveczky. Cham: Springer, 2018, pages 137–155. DOI: [10.1007/978-3-030-02146-7_7](https://doi.org/10.1007/978-3-030-02146-7_7). URL: [papers/FBCMW18.pdf](https://papers.fbcmw18.pdf) (cited on pages 9, 10).
- [22] S. Foster et al. “Unifying theories of reactive design contracts”. In: *Theoretical Computer Science* 802 (2020), pages 105–140. DOI: [10.1016/j.tcs.2019.09.017](https://doi.org/10.1016/j.tcs.2019.09.017). URL: [papers/FCCWZ20.pdf](https://papers.fccwz20.pdf) (cited on pages 9, 11).
- [23] T. Gibson-Robinson et al. “FDR3 - A Modern Refinement Checker for CSP”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2014, pages 187–201 (cited on page 10).
- [24] X. Chen, E. Ábrahám, and S. Sankaranarayanan. “Flow*: An Analyzer for Non-linear Hybrid Systems”. In: *Computer Aided Verification*. Edited by N. Sharygina and H. Veith. Springer, 2013, pages 258–263 (cited on page 84).

Index of Semantic Rules

In this index you'll find the list of semantic functions in alphabetic order, and page where they are defined. Timed versions of existig semantic rules are indexed by a **timed** item under the entry for the semantic function. Semantic functions exclusive to the timed model are identified by a **timed** annotation in parenthesis after the rule name. Rules whose names are abbreviation (e.g., S) are annotated with the full name in parenthesis.

annotateConstraint, 53
annotateStatement, 54
arenaGlobalAssumptions, 74
arenaTypeDefinitions, 67
attributeTypeDefinitions, 68

componentGetSetChannelDefinitions, 73
composeOperationMappings, 75
composeOutputEventMappings, 74

elementDefinitionTypeDefinitions, 70
elementGetSetChannelDefinitions, 72
elementGlobalAssumptions, 73, 74
elementLocationsDefinition, 71
elementSizeParameters, 70
elementTypeDefinitions, 69
environmentProcess, 75
environmentState, 76

eventHappenedChannelDefinitions, 71, 72
eventTriggeredChannelDefinitions, 71

findDimensionalityInfo, 51

inputTrigger, 77

mapArena, 50
mapInput, 53
mapInputEvent, 52
mapInputEvents, 52
mappingProcess, 74
mapRWDoc, 50

operationMappingDefinition, 78

robotElementDefinitionTypeDefinitions, 69
robotGetSetChannelDefinitions, 72
robotTypeDefinitions, 68

roboWorldDocument, [66](#)

updateArena, [51](#)

typeDefinitions, [66](#)