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 locations 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
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>Definition of global elements</td>
<td>74</td>
</tr>
<tr>
<td>28</td>
<td>Global Assumptions on the Arena</td>
<td>74</td>
</tr>
<tr>
<td>29</td>
<td>Mapping Process</td>
<td>74</td>
</tr>
<tr>
<td>30</td>
<td>Composition of Output Event Mappings</td>
<td>74</td>
</tr>
<tr>
<td>31</td>
<td>Composition of Operation Mappings</td>
<td>75</td>
</tr>
<tr>
<td>32</td>
<td>Environment Process</td>
<td>75</td>
</tr>
<tr>
<td>33</td>
<td>Environment Process State</td>
<td>76</td>
</tr>
<tr>
<td>34</td>
<td>Input Trigger semantics</td>
<td>77</td>
</tr>
<tr>
<td>35</td>
<td>Operation Mapping Semantics</td>
<td>78</td>
</tr>
</tbody>
</table>
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 framework\(^1\) [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

\(^1\)robostar.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 RoboTool\(^2\). 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].

\(^2\)robostar.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.
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 competition\(^1\). 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.

\(^{1}\)www.mbzirc.com/ - Challenge 3 in 2020.
Chapter 2. RoboStar framework

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

Figure 2.1: Firefighting UAV

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 SimpleFirefighter 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().
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.
Chapter 3. RoboWorld: overview and metamodel

## 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 available\(^1\), 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

\(^1\)robostar.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
3.1 Document structure: overview

## 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
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`. 

---

Figure 3.3: Firefighter UAV RoboWorld - mapping for `spray`

Figure 3.4: Firefighter UAV RoboWorld - mapping for `searchFire()`
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
have been visited \((\text{indexSP} > \text{size(searchPattern(indexSP)))}\) then the operation \(\text{searchFire()}\) finishes.

Before finishing, \(\text{searchFire()}\) 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 \(\text{searchFire()}\) 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 \(\text{RWDocument}\). 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 \(\text{RWSentence}\) representing the forms of sentences allowed in RoboWorld, and Conditions, which are \(\text{RWSentences}\) prefixed by a subjunction. \(\text{RWSentences}\) 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 \(\text{RobotSentence}\), or by a p-model, represented by the class \(\text{RobotPModel}\). The attribute \(\text{pmodel}\) of \(\text{RobotPModel}\) has type \(\text{PModel}\). This is a class in the RoboSim metamodel\(^2\) that represents a specialised form of block

\(^2\)robostar.cs.york.ac.uk/publications/techreports/reports/physmod-reference.pdf
3.2 Metamodel

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 studies\(^3\): 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-
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 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: NounBl, representing a Noun, UnitBl, representing a unit, or a QualifiedBl, 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 and 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

<table>
<thead>
<tr>
<th>RW1</th>
<th>The values “arenas” and “robots” are not valid for the attribute noun of a BasicBI.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RW2</td>
<td>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.</td>
</tr>
<tr>
<td>RW3</td>
<td>The names in the signatures of the OperationMappings must be precisely those of the operations of the robotic platform in the associated RoboChart module.</td>
</tr>
<tr>
<td>RW4</td>
<td>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.</td>
</tr>
<tr>
<td>RW5</td>
<td>The name of the pmodel in a RobotPModel is “robot”.</td>
</tr>
<tr>
<td>RW6</td>
<td>The name of the pmodel in an ElementPModel matches the value of its name.</td>
</tr>
<tr>
<td>RW7</td>
<td>In the input of an InputEventMapping for an event that is typeless in the associated RoboChart module, there are no sentences.</td>
</tr>
<tr>
<td>RW8</td>
<td>The sentences that define a DiagrammaticOutput must define a unit of time.</td>
</tr>
<tr>
<td>RW9</td>
<td>If the name of an OutputEventMapping is n, and its output is a DiagrammaticOutput, then the name of the ROOperation in opd is nmapping.</td>
</tr>
<tr>
<td>RW10</td>
<td>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 ROOperation in opd has a parameter of type T.</td>
</tr>
<tr>
<td>RW11</td>
<td>The signature of an OperationMapping whose output is a DiagrammaticOutput matches the signature of the ROOperation in opd.</td>
</tr>
</tbody>
</table>

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 mapping (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.
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
4.2 Background on the Grammatical Framework

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
abstract ToyRoboWorld = {
flags startcat = Clause ;
}

-- categories
Determiner ; Noun ; Verb ;
NounPhrase ; VerbPhrase ; Clause ;

-- lexicon
a_SgDeterminer : Determiner ; a_PlDeterminer : Determiner ;
the_SgDeterminer : Determiner ; the_PlDeterminer : Determiner ;
robot_Noun : Noun ; wheel_Noun : Noun ; have_Verb : Verb ;

-- functions
mkNounPhrase : Determiner -> Noun -> NounPhrase ;
mkVerbPhrase : Verb -> NounPhrase -> VerbPhrase ;
mkClause : NounPhrase -> VerbPhrase -> Clause ;
}

Listing 1: Abstract grammar of ToyRoboWorld

concrete ToyRoboWorldEng of ToyRoboWorld = {

param Number = Sg | Pl ;
param VerbForm = VPresent Number ;

-- categories
Determiner = {s : Str ; n : Number} ;
Noun = {s : Number => Str} ;
Verb = {s : VerbForm => Str } ;
NounPhrase = {s : Str ; n : Number} ;
VerbPhrase = {v : Verb ; np : NounPhrase} ;
Clause = Str ;

-- lexicon
a_SgDeterminer = {s = "a" ; n = Sg} ;
a_PlDeterminer = {s = "" ; n = Pl} ;
the_SgDeterminer = {s = "the" ; n = Sg} ;
the_PlDeterminer = {s = "the" ; n = Pl} ;
robot_Noun = {s = table {Sg => "robot" ; Pl => "robots"}} ;
wheel_Noun = {s = table {Sg => "wheel" ; Pl => "wheels"}} ;
have_Verb = {s = table {VPresent Sg => "has" ; VPresent Pl => "have"}} ;

-- functions
mkNounPhrase det noun = {s = det.s ++ (noun.s ! det.n) ; n = det.n} ;
mkVerbPhrase v np = {v = v ; np = np} ;
mkClause np vp = np.s ++ (vp.v.s ! (VPresent np.n)) ++ vp.np.s ;
}

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 Pl (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.
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 (Cl), 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.
4.4 Building blocks: ItemPhrases

Table 4.1: Well-formedness conditions of the dictionary

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>RoboChart keywords must not be included in the dictionary.</td>
</tr>
<tr>
<td>D2</td>
<td>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.</td>
</tr>
<tr>
<td>D3</td>
<td>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.</td>
</tr>
</tbody>
</table>

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...
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 \texttt{lin A adj}), the adverb is constructed by the RGL function \texttt{SyntaxEng.mkAdv}, turning, for example, “initial” into “initially”. The function \texttt{mkCN} is also from RGL and creates a common noun given another common noun (\texttt{item}) and an adverb (\texttt{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 \texttt{mkItemPhrase\_PronounIP} and \texttt{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 \texttt{RWClause}) are used to define \texttt{RWSentences}; they are instances of RGL clauses (\texttt{CatEng\_Cl}), and define the writing structures supported in RoboWorld.

There are 12 forms of \texttt{RWClause}, each defined by a \texttt{mK} function. An \texttt{RWClause} can be written in the active voice (using functions whose names start with \texttt{mkRWClause\_ActiveVoice\_}) or in the passive voice (using \texttt{mkRWClause\_PassiveVoice\_functions}).

In the active voice, \texttt{mkRWClause\_ActiveVoice\_TransitiveVerb\_ItemPhrase} is used to create \texttt{RWClauses} using transitive verbs. There is also support for modal and progressive verbs in the active voice (\texttt{mkRWClause\_ActiveVoice\_Modal} and \texttt{mkRWClause\_ActiveVoice\_Progressive\_functions}). The \texttt{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 \texttt{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 \texttt{RWClause} of the form considered. First, a verb phrase (\texttt{VP}) named \texttt{progressive} is declared. The function \texttt{mkVP} creates a verb phrase from the text embedded in the provided verb (\texttt{lin V2 v2}) and the second \texttt{ItemPhrase} (\texttt{itemPhrase2}). A type annotation \texttt{(< ... : V2>)} is applied to \texttt{lin V2 v2} to ensure the text is cast to the type \texttt{V2} (since verbs can have several types). Afterwards, the RGL function \texttt{progressiveVP} transforms this verb phrase, taking into account the progressive...
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 RWClauses, 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 RWClauses 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

form of its verb, whose value is assigned to the local variable progressive. Finally, when creating the clause, the function mkCl 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”. □
40 Chapter 4. RoboWorld: realisation in the Grammatical Framework

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 List-
ing 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.
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 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
5.1 Metamodel and well-formedness conditions

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
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 `sentences` 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
5.1 Metamodel and well-formedness conditions

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 BinaryBooleanExpressions, 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
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

```java
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.operationsMappings)
    }
```

Rule 2. Map ArenaAssumptions

```java
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*

\[
\text{updateArena}(\text{assumption} : \text{ArenaAssumption}, \text{arena} : \text{Arena}) : \text{Arena} = \\
\text{if findArenaDimensionInfo(assumption) then } \text{arena}.\text{dimension} = \text{getArenaDimensionInfo(assumption)} \text{ else if findArenaClosedInfo(assumption) then } \text{arena}.\text{closed} = \text{getArenaClosedInfo(assumption)} \text{ else if findArenaFloorInfo(assumption) then } \text{arena}.\text{hasFloor} = \text{getArenaFloorInfo(assumption)} \text{ else if findArenaRainInfo(assumption) then } \text{arena}.\text{hasRain} = \text{get ArenaRainInfo(assumption)}
\]

Rule 4. *Find dimensionality information*

\[
\text{findArenaDimensionInfo(assumption : ArenaAssumption) : Boolean} = \\
\text{if refersToArena(assumption.sentence.clause.itemPhrase)} \land \text{assumption.sentence.clause instanceof mkRWClause_ActiveVoice_ToBe_Adjective} \land \text{positiveSentence(assumptions.sentence)} \text{ then let adj} = ((\text{mkRWClause_ActiveVoice_ToBe_Adjective}) \text{assumption.sentence.clause}.\text{a}) \text{ within if adj} \in \text{dimensionAdjectives then true else false else if ... else false}
\]

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

\[
\begin{align*}
\text{mapInputEvents} & (\text{inEvents} : \text{Seq}(\text{InputEventMapping})) : \text{Seq}(\text{InputEventMappingIR}) = \\
& \quad \text{if } \#\text{inEvents} = 0 \text{ then } () \\
& \quad \text{else } (\text{mapInputEvent}(\text{head}(\text{inEvents}))) \mapsto \text{mapInputEvents}(\text{tail}(\text{inEvents}))
\end{align*}
\]

### Rule 6. Map InputEventMapping

\[
\begin{align*}
\text{mapInputEvent} & (\text{inEvent} : \text{InputEventMapping}) : \text{InputEventMappingIR} = \\
& \quad \text{new InputEventMappingIR} \{
\text{name} = \text{inEvent}.\text{name} \\
\text{input} = \text{mapInput}(\text{inEvent}.\text{input})
\}
\end{align*}
\]

---

**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 \text{mapRobot}, is similar. Regarding elements, the main difference is that \text{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 \text{InputEventMappings}. The simple definition of the function \text{mapInputEvents} is shown in Rule 5. It recursively applies another function \text{mapInputEvent} (see Rule 6) to each \text{InputEventMapping} in its argument, yielding the sequence of the obtained results.

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

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

We use a \text{where} clause to define variables global to the rule called \text{conditions} and \text{communications}, used to define the homonymous attributes of the resulting \text{InputSometimesIR}. The definition of \text{conditions} applies, via the use of \text{map}, a function defined by a \(\lambda\) expression, to each sentence of the sequence \text{sentences} of the \text{conditions} of \text{input}. The result of the \text{map} is the sequence of the results. The function defined by the \(\lambda\) expression has argument \(x\) and specifies an instance of the IR class \text{Constraint}, whose \text{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
5.2 Generation from RoboWorld documents

**Rule 7. Map InputSometimes**

```java
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**

```java
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 boolean expression 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 `DeterminerIR` 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
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 \textit{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 \textit{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 \textit{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 \textit{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 \textit{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 ≜ (Environment [getSetChannels ∪ eventHappenedChannels ∪ \{proceed\}] Mapping)
\setminus_{\text{getSetChannels ∪ eventHappenedChannels ∪ \{proceed\}}}

Figure 6.2: The RWDocument process for the firefighter example

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 ([...]) 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 \[ and \] symbols, contains the events that require synchronisation between Environment and Mapping. The same set is indicated after \ 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, CyPhyCircus events represent communications over channels; fat brackets \{...\} 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


```plaintext
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 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 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
6.1 Formal semantics: overview

\[
\begin{align*}
arena & : \text{ArenaProperty} \\
\text{robotInit} & : \text{RobotProperty} \\
\text{potentialFires} & : \mathbb{P} \text{FireProperty} \\
\text{groundLocations} & : \mathbb{P} \text{Position} \\
\text{groundLocations} & = \{x,y,z \in \mathbb{R} \mid (x,y,z) \in \text{arena.locations} \wedge z = 0\} \\
\text{arena.xwidth} & = 50.0 \\
\text{arena.ywidth} & = 60.0 \\
\text{arena.zwidth} & \geq \text{building.zwidth} + 1.0 \\
\text{timeStep} & : \mathbb{R}
\end{align*}
\]

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, \text{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 ≜ begin

EnvironmentState ≜ [ visible robot : RobotProperty; visible fires : seq FireProperty; time : R; stepTimer : R; EventTimes ]

state EnvironmentState

EnvironmentStateInit

EnvironmentState′ 

robot′ = robotInit ∧ ran fires′ ⊆ potential Fires ∧ time′ = 0.0 ∧ stepTimer′ = 0.0 ∧ EventTimesInit

RobotMovementAction ≜ ···

CollisionDetection ≜ RobotGroundCollision □ RobotBuildingCollision □ RobotFireCollision

InputTriggers ≜ fire Detected InputEventMapping || noFire InputEventMapping || ···

Communication ≜ ((GetRobotPosition □ GetRobotVelocity □ ···); Communication) □ proceed → Skip

···

InputEventBuffers ≜ fire Detected Buffer || noFire Buffer || critical Buffer || landed Buffer

OutputEventBuffers ≜ spray Buffer || takeOff Buffer || goToBuilding Buffer || go Home Buffer || searchFire Buffer

EventBuffers ≜ InputEventBuffers || OutputEventBuffers

EnvironmentLoop ≜ (EnvironmentStateInit); μX •

RobotMovementAction;

(stepTimer < timeStep) & CollisionDetection

□

(stepTimer ≥ timeStep) & InputTriggers; Communication; stepTimer := 0.0 ) ; X

channelset trigger Channels ≜ { fireDetected Triggered, noFire Triggered, critical Triggered, landed Triggered }

• (EnvironmentLoop [ trigger Channels ] EventBuffers) \ trigger Channels

der 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 (•) to define its behaviour. This is in contrast, for example, with a process like RWDocument 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.
RobotMovementAction \triangleq (\text{RobotMovement}) \bigtriangleup \bigtriangleup \\
\left( \begin{array}{c}
\text{robot.position} \in \text{groundLocations} \land \text{robot.velocity} \cdot 3 < 0 \\
\lor \ldots \lor \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 \( X (\mu X) \). In the body of the recursion, EnvironmentLoop performs RobotMovementAction, sketched in Figure 6.7. Afterwards, EnvironmentLoop proceeds to a choice (\( \sqsupset \)) that depends on whether stepTimer < 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} \equiv 
\text{\begin{align*}
\text{if} \left( \exists \text{fire} : \text{ran fires} \bullet \neg \left( \text{distance(} \text{fire}.\text{position, robot}.\text{position}) > 0.5 \right) \right) & -\rightarrow \\
& \text{fireDetectedTriggered!True} -\rightarrow \text{fireDetectedOccurred}, \text{fireDetectedTimer} := \text{True}, \text{time} \\
\text{\mid} \neg \left( \exists \text{fire} : \text{ran fires} \bullet \neg \left( \text{distance(} \text{fire}.\text{position} - \text{robot}.\text{position}) > 0.5 \right) \right) & -\rightarrow \\
& \text{fireDetectedTriggered!False} -\rightarrow \text{Skip}
\end{align*}}
\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 Λ declaration of the state to specify evolution according to a set of given differential equations. The body of RobotMovement has, for instance, differential equations describing the movement of the robot and the evolution of timers. 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 (△) 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 ≥ 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 (\(\lor\)), 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.
6.1 Formal semantics: overview

In fireDetected_Buffer, we have a choice based on whether there is a fire 1 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 \text{Mapping} \triangleq \text{spray\_OutputEventMapping} \triangleright \text{proceed} \triangleright \text{takeOff\_OperationMapping} \\
\triangleright \text{proceed} \triangleright \text{goToBuilding\_OperationMapping} \triangleright \text{proceed} \triangleright \text{goHome\_OperationMapping} \\
\triangleright \text{proceed} \triangleright \text{searchFire\_OperationMapping}

process \text{goToBuilding\_OperationMapping} \triangleq \text{begin}

\text{goToBuilding\_Semantics} \triangleq \text{goToBuildingCall} \\
\rightarrow \text{getRobotPosition?robotPos} \rightarrow \text{getBuildingPosition?buildingPos} \\
\rightarrow (\text{setRobotVelocity!}(1.0 \ast ((\text{buildingPos} - \text{robotPos})/\text{norm}(\text{buildingPos} - \text{robotPos})))) \rightarrow \text{Skip}; \\
\text{proceed} \rightarrow \text{goToBuilding\_Semantics}

\text{goToBuilding\_Monitor} \triangleq \text{goToBuildingCall} \rightarrow \text{goToBuildingHappened} \rightarrow \text{goToBuilding\_Monitor}

\triangleright \text{goToBuilding\_Semantics} \triangleright \text{goToBuilding\_Monitor}

\text{end}

Figure 6.10: The Mapping and \text{goToBuilding\_OperationMapping} processes for the firefighter UAV example

\text{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 \text{goToBuildingCall} for the operation \text{goToBuilding}.

As shown in Figure 6.10, the \text{goToBuilding\_Semantics} action captures the semantics corresponding to the mapping definition “when the operation \text{goToBuilding} is called, the velocity of the robot is set to 1.0 m/s towards the building”. After \text{goToBuildingCall}, it obtains from Environment the position of the robot (\text{robotPos}) and of the building (\text{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 \text{robotPos} to \text{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, \text{goToBuilding\_Monitor}, after \text{goToBuildingCall}, communicates \text{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
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 $\mathcal{R}_\mathbf{w}$ that characterises the CyPhyCircus section that includes all definitions needed to specify the top process $\text{RWDocument}$ that captures the behaviours of the robot and environment elements allowed by the assumptions and mappings in a well-formed instance $\mathbf{rw}$ of the IR class $\text{RWIntermediateRepresentation}$ 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 $\mathbf{rw}$ 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($\mathbf{rw}$)` specified by Rule 32. The definitions of the `Mapping` processes are characterised by `for` iterations over the `outputEventMappings` and `operationMappings` of $\mathbf{rw}$. 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\}. 

formalisation of the semantics, via generative rules that define semantic functions.
Rule 10. Semantics of RoboWorld Documents

\[
[rw : RWIntermediateRepresentation]_{\mathcal{FR}} = \\
\quad \text{typeDefinitions}(rw.arena, rw.robot, rw.elements) \\
\quad \text{eventTriggeredChannelDefinitions}(rw.inputEventMappings, rw.variableMappings) \\
\quad \text{eventHappenedChannelDefinitions}(rw.outputEventMappings, rw.operationMappings) \\
\quad \text{getSetChannelDefinitions}(rw.robot, rw.elements) \\
\quad \text{channel} \ \proceed \\
\quad \text{elementGlobalAssumptions}(rw.arena, rw.robot, rw.elements) \\
\quad \mid \\
\quad \text{timeStep} : \mathbb{R} \\
\quad \text{processEnvironment} \triangleq \text{environmentProcess}(rw) \\
\quad \quad \text{for} \ \text{output} \ \text{in} \ rw.outputEventMappings \ \text{do} \\
\quad \quad \quad \text{outputMappingDefinition}(\text{output}) \\
\quad \quad \text{end for} \\
\quad \text{for} \ \text{operation} \ \text{in} \ rw.operationMappings \ \text{do} \\
\quad \quad \text{operationMappingDefinition}(\text{operation}) \\
\quad \text{end for} \\
\quad \text{processMapping} \triangleq \text{mappingProcess}(rw.outputEventMappings, rw.operationMappings) \\
\quad \text{processRWDocument} \triangleq (\text{Environment} \left[ \text{communicationEvents} \right] \text{Mapping}) \ \backslash \ \text{communicationEvents} \\
\quad \text{where} \\
\quad \quad \text{communicationEvents} = \\
\quad \quad \quad \text{getSetEvents}(rw.robot, rw.elements) \\
\quad \quad \quad \cup \text{eventHappenedSignals}(rw.outputEventMappings, rw.operationMappings) \\
\quad \quad \quad \cup \{ \proceed \}
\]

Rule 11. Type Definitions

\[
\text{typeDefinitions}(arena : \text{Arena}, robot : \text{Element}, elements : \text{Seq}(<\text{Element}>) = \\
\quad \text{arenaTypeDefinitions}(arena) \\
\quad \text{robotTypeDefinitions}(robot) \\
\quad \text{for} \ \text{element} \ \text{in} \ \text{elementTypes} \ \text{do} \\
\quad \quad \text{elementTypeDefinitions}(\text{element}) \\
\quad \text{end for}
\]

The rule for typeDefinitions is Rule 11. It calls \text{arenaTypeDefinitions()} to generate the ArenaProperty schema and the types it uses, and \text{robotTypeDefinitions()} to generate the RobotProperty schema and the types it uses. It then iterates over elementTypes, generating the types for each element in \text{elementTypeDefinitions()}. As an example of one of these, we show the rule for \text{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 \text{elementTypeDefinitions()}. In our firefighter example, this generates the HomeProperty schema for the home region. Afterwards, a call to \text{attributeTypeDefinitions()} generates type definitions for any named types required by the attributes. Although the arena in our example has no attributes, \text{attributeTypeDefinitions()} is used to generate types for attributes of other elements, so it generates \text{Tank_of_waterType}, for example. After other types are generated, the ArenaProperty schema is defined. The \text{width}, \text{windSpeed} and \text{locations} 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 \text{gradient}, \text{windSpeed} and \text{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)

if arena.dimension = ThreeD then
   xwidth, ywidth, zwidth : R
else if arena.dimension = TwoD then
   xwidth, ywidth : R
else
   xwidth : R
end if

gradient, windSpeed : R

locations : 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
Chapter 6. RoboWorld: semantics

**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 an0 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

```plaintext
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 : 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)
    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

```plaintext
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**

```
rule17 = 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(component.region.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**

```
rule18 = 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
```

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

\[
elementLocationsDefinition(elementShape : shape, dimension : Dimension) =
\]
\[
  \text{if } elementShape \neq \text{null then}
  \quad \text{if } dimension = \text{ThreeD then}
  \quad \text{if } elementShape \text{ instanceof Box then}
  \quad \quad \text{locations} = \text{boxLocations} \text{ position orientation} x \text{width} y \text{width} z \text{width}
  \quad \text{else if } elementShape \text{ instanceof Cylinder then}
  \quad \quad \text{locations} = \text{cylinderLocations} \text{ position orientation} \text{radius} \text{depth}
  \quad \text{else}
  \quad \quad \text{locations} = \text{sphereLocations} \text{ position orientation} \text{radius}
  \quad \text{end if}
  \text{else}
  \text{if } elementShape \text{ instanceof Box then}
  \text{if } dimension = \text{TwoD then}
  \text{locations} = \text{squareLocations} \text{ position orientation} x \text{width} y \text{width}
  \text{else}
  \quad \text{locations} = \text{lineLocations} \text{ position orientation} x \text{width}
  \text{end if}
  \text{else}
  \text{locations} = \text{lineLocations} \text{ position orientation} \text{radius}
  \text{end if}
\text{end if}
\]

Rule 20. **Input event trigger channel definitions**

\[
eventTriggeredChannelDefinitions(inputs : \text{Seq(InputEventMappingIR)}, variables : \text{Seq(VariableMappingIR)}) =
\]
\[
  \text{for } input \text{ in } inputs \text{ do}
  \quad \text{if } input\text{HasCommunications}(input) \text{ then}
  \quad \quad \text{channel } input\text{name}\text{Triggered} : B \times input\text{CommunicationTypes}(input)
  \quad \text{else}
  \quad \quad \text{channel } input\text{name}\text{Triggered} : B
  \quad \text{end if}
  \text{end for}
\]

Rule 21. **Output event happened channel definitions**

\[
eventHappenedChannelDefinitions(outputEvents : \text{Seq(OutputEventMappingIR)}, operations : \text{Seq(OperationMappingIR)}) =
\]
\[
  \text{for } outputEvent \text{ in } outputEvents \text{ do}
  \quad \text{channel } outputEvent\text{name}\text{Happened} : B
  \text{end for}
  \text{for } operation \text{ in } operations \text{ do}
  \quad \text{channel } operation\text{signature}\text{name}\text{Happened} : B
  \text{end for}
\]
Chapter 6. RoboWorld: semantics

Rule 22. Variable get/set channel definitions

```plaintext
getSetChannelDefinitions(\text{robot} : \text{Element}, \text{elements} : \text{Seq}(\text{Element})) =

\text{if \ robot \ instanceof \ ElementDefinition}
  \text{robotGetSetChannelDefinitions((ElementDefinition) \ robot)}
\text{else}
\text{\quad robotPModelGetSetChannelDefinitions((ElementPModel) \ robot)}
\text{end if}
\text{for \ element \ in \ elements \ do}
  \text{if \ element \ instanceof \ ElementDefinition}
    \text{if \ not \ (element \ instanceof \ Region) \ then}
      \text{elementGetSetChannelDefinitions((ElementDefinition) \ element)}
    \text{end if}
  \text{else}
    \text{elementPModelGetSetChannelDefinitions((ElementPModel) \ element)}
  \text{end if}
\text{end for}
```

Rule 23. Robot variable get/set channel definitions

```plaintext
robotGetSetChannelDefinitions(\text{robot} : \text{ElementDefinition}) =

\text{channel getRobotPosition : Position}
\text{channel getRobotVelocity : Velocity}
\text{channel getRobotAcceleration : Acceleration}
\text{channel getRobotOrientation : Orientation}
\text{channel getRobotAngularVelocity : AngularVelocity}
\text{channel getRobotAngularAcceleration : AngularAcceleration}
\text{channel setRobotPosition : Position}
\text{channel setRobotVelocity : Velocity}
\text{channel setRobotAcceleration : Acceleration}
\text{channel setRobotOrientation : Orientation}
\text{channel setRobotAngularVelocity : AngularVelocity}
\text{channel setRobotAngularAcceleration : AngularAcceleration}
\text{for \ attribute \ in \ \text{robot}.attributes \ do}
  \text{channel getRobot\_titleCase(attribute.name) : attribute.type}
  \text{channel setRobot\_titleCase(attribute.name) : attribute.type}
\text{end for}
\text{for \ component \ in \ \text{robot}.components \ do}
  \text{componentGetSetChannelDefinitions(component, "Robot")}
\text{end for}
```

Rule 24. ElementDefinition variable get/set channel definitions

```plaintext
elementGetSetChannelDefinitions(\text{element} : \text{ElementDefinition}) =

\text{channel getRobotPosition : Position}
\text{channel getRobotOrientation : Orientation}
\text{for \ attribute \ in \ \text{element}.attributes \ do}
  \text{channel getRobot\_titleCase(attribute.name) : attribute.type}
  \text{channel setRobot\_titleCase(attribute.name) : attribute.type}
\text{end for}
\text{for \ component \ in \ \text{element}.components \ do}
  \text{componentGetSetChannelDefinitions(component, "Robot")}
\text{end for}
```
6.2 Semantics generation: transformation rules

**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 ∘ titleCase(attribute.name) : qualifiedNameID x attribute.type
        end for
        for component in element.components do
            componentGetSetChannelDefinitions(component, qualifiedName)
        end for
    else
        for attribute in element.attributes do
            channel getQualifiedName ∘ titleCase(attribute.name) : attribute.type
        end for
        for component in element.components do
            componentGetSetChannelDefinitions(component, qualifiedName)
        end for
    end if

where

qualifiedName = namePrefix ∘ 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.
Chapter 6. RoboWorld: semantics

Rule 27. Definition of global elements

\[\text{elementGlobalDefinitions}(\text{arena}: \text{Arena}, \text{robot}: \text{Element}, \text{elements}: \text{Seq}(	ext{Element})) = \]

\[
\begin{array}{l}
\text{arena}: \text{ArenaProperty} \\
\text{robotInit}: \text{RobotProperty} \\
\text{for element} \in \text{elements} \text{ do} \\
\hspace{1em} \text{if element.plurality} = \text{SINGULAR} \text{ then} \\
\hspace{2em} \text{if hasAttributes(element) then} \\
\hspace{3em} \text{element.name}^\text{Init}: \text{titleCase(element.name)}\text{Property} \\
\hspace{3em} \text{else} \\
\hspace{4em} \text{element.name}: \text{titleCase(element.name)}\text{Property} \\
\hspace{2em} \text{else} \\
\hspace{3em} \text{if hasAttributes(element) then} \\
\hspace{4em} \text{potential}\text{titleCase(plural(element.name))}: \text{titleCase(element.name)}\text{Property} \\
\hspace{4em} \text{else} \\
\hspace{5em} \text{plural(element.name)}: \text{seq}\text{titleCase(element.name)}\text{Property} \\
\hspace{2em} \text{end if} \\
\hspace{1em} \text{end if} \\
\end{array}
\]

Rule 28. Global Assumptions on the Arena

\[\text{arenaGlobalAssumptions}(\text{arena}: \text{Arena}) = \]

\[
\begin{array}{l}
\text{shapeConstraints(arena.shape)} \\
\text{if arena.gradient} \neq \text{null} \text{ then} \\
\hspace{1em} \text{for constraint} \in \text{arena.gradient.properties} \text{ do} \\
\hspace{2em} \text{generateConstraint(constraint)} \\
\hspace{1em} \text{end for} \\
\hspace{1em} \text{end if} \\
\text{...} \\
\text{for region} \in \text{arena.components} \text{ do} \\
\hspace{1em} \text{elementGlobalAssumptions(region)} \\
\text{end for} \\
\end{array}
\]

Rule 29. Mapping Process

\[\text{mappingProcess(outEvents}: \text{Seq}(	ext{OutputEventMappingIR}), \text{operations}: \text{Seq}(	ext{OperationMappingIR})) = \]

\[
\text{composeOutputEventMappings(outEvents)} \parallel \text{composeOperationMappings(operations)}
\]

Rule 30. Composition of Output Event Mappings

\[\text{composeOutputEventMappings(outEvents}: \text{Seq}(	ext{OutputEventMappingIR})) = \]

\[
\begin{array}{l}
\text{if} \#\text{outEvents} = 0 \text{ then} \\
\hspace{1em} \text{Skip} \\
\text{else if} \#\text{outEvents} = 1 \text{ then} \\
\hspace{1em} \text{[head(outEvents)]} \parallel \text{.} \parallel \\
\text{else} \\
\hspace{1em} \text{[head(outEvents)]} \parallel \text{.} \parallel \text{[composeOutputEventMappings(tail(outEvents))]} \\
\end{array}
\]
Rule 31. Composition of Operation Mappings

\[
\text{composeOperationMappings}(\text{operations} : \text{Seq(OperationMappingIR)}) = \\
\begin{cases}
\text{Skip} & \text{if } \#\text{operations} = 0 \\
\left[\text{head}(\text{operations})\right]_{\sigma, \delta} & \text{else if } \#\text{operations} = 1 \\
\left[\text{head}(\text{operations})\right]_{\sigma, \delta} || \text{composeOperationMappings}((\text{tail}(\text{operations})) & \text{else}
\end{cases}
\]

Rule 32. Environment Process

\[
\text{environmentProcess}(\text{rw} : \text{RWIntermediateRepresentation}) = \\
\begin{align*}
\text{begin} & \\
\text{eventTimes}(\text{rw}\cdot\text{inputEventMappings}, \text{rw}\cdot\text{outputEventMappings}, \text{rw}\cdot\text{operationMappings}) & \\
\text{environmentState}(\text{rw}\cdot\text{robot}, \text{rw}\cdot\text{elements}) & \\
\text{robotMovementAction}(\text{rw}\cdot\text{arena}, \text{rw}\cdot\text{robot}, \text{rw}\cdot\text{elements}) & \\
\text{collisionDetectionAction}(\text{rw}\cdot\text{arena}, \text{rw}\cdot\text{robot}, \text{rw}\cdot\text{elements}) & \\
\text{inputTriggersAction}(\text{rw}\cdot\text{inputEventMappings}) & \\
\text{communicationAction}(\text{rw}\cdot\text{arena}, \text{rw}\cdot\text{robot}, \text{rw}\cdot\text{elements}) & \\
\text{inputEventBuffers}(\text{rw}\cdot\text{inputEventMappings}, \text{rw}\cdot\text{variableMappings}) & \\
\text{outputEventBuffers}(\text{rw}\cdot\text{outputEventMappings}, \text{rw}\cdot\text{operationMappings}) & \\
\text{EventBuffers} \equiv \text{InputEventBuffers} || \text{OutputEventBuffers} & \\
\text{EnvironmentLoop} \equiv (\text{EnvironmentStateInit}) ; \ \mu X \bullet \\
\left( \text{RobotMovementAction}; (\text{time} < \text{timeStep}) \& \text{CollisionDetection} \right. & \\
\left. (\text{time} \geq \text{timeStep}) \& \text{InputTriggers}; \text{Communication}; \text{stepTimer} := 0 \right); X & \\
\text{triggerChannelsSet}(\text{rw}\cdot\text{inputEventMappings}, \text{rw}\cdot\text{variableMappings}) & \\
\bullet (\text{EnvironmentLoop} \parallel [\text{triggerChannels} \left[ \text{EventBuffers} \right] \text{triggerChannels} & \\
\text{end}
\end{align*}
\]
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

\[
\text{InputTrigger(inputEvent : InputEventMappingIR) =}
\]

\[
\begin{align*}
\text{inputEvent.name} & \_\text{InputEventMapping} = \\
\text{if } \text{inputEvent.input \text{ instanceof InputSometimesIR} } \text{then} & \\
\text{if } \text{generateConstraintConjunction((InputSometimesIR)inputEvent.input).conditions} & \\
\quad \text{if } \#(\text{InputSometimesIR)inputEvent.input).communications = 0 \text{ then} \\
\quad \quad \text{inputEvent.name} & \text{Triggered}! \text{True} \\
\quad \text{else } & \ldots \\
\quad \text{end if} \\
\text{inputEvent.name} & \text{Occurred}, \text{inputEvent.name} \text{Timer} := \text{True}, \text{time} \\
\quad [\text{~ (generateConstraintConjunction((InputSometimesIR)inputEvent.input).conditions)}] & \\
\quad \text{if } \#(\text{InputSometimesIR)inputEvent.input).communications = 0 \text{ then} \\
\quad \quad \text{inputEvent.name} & \text{Triggered}! \text{False} \quad \text{Skip} \\
\quad \text{else } & \ldots \\
\quad \text{end if} \\
\text{fi} \\
\text{else if } & \ldots \\
\text{end if}
\end{align*}
\]

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 \text{False}. After the communication, we have a \text{Skip} action, instead of an assignment.

The function \text{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 \text{name} of the signature of the argument \text{operation} with \_\text{OperationMapping}. As defined in Rule 35, this process does not have a state, and its main action is always the parallel composition of \_\text{Semantics} and \_\text{Monitor} actions (also named after the \text{operation}) defined in Rule 35 (see Figure 6.10 for an example).

The body of the \_\text{Semantics} action begins with a communication on the \text{operation’s Call} channel. The parameters of the \text{operation} are iterated over in a \text{for} loop, with each parameter added as an input (?) in the communication. After the communication, the semantics depends on the type of the \text{output} for the \text{operation}; here, we show the case for \text{OutputAlwaysIR}, which has statements but not conditions. In this case, communications with the \text{Environment} on \text{get} channels are included (to obtain the values of any state components required by the statements). For example, for \text{goToBuilding_Semantics} in Figure 6.10, these are communications on \text{getRobotPosition} and \text{getBuildingPosition} so that the positions of the robot and building are available. After the needed variables are obtained, the semantics of each of the \text{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 \text{proceed} and a recursion of the \_\text{Semantics} action are specified.

The body of the \_\text{Monitor} action also begins with a communication on the \text{operation’s Call} channel, specified in the same way as for the \_\text{Semantics} action. This is followed by a communication with the \text{Environment} on the \text{Happened} channel for the \text{operation} and a recursion
**Rule 35. Operation Mapping Semantics**

operationMappingDefinition(operation : OperationMappingIR)

```plaintext
process operation.signature.name_Semantics \triangleq begin

operation.signature.name_Semantics \triangleq operation.signature.name_Call(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 \triangleq operation.signature.name_Call(for param in operation.signature.parameters do ?param end for) \rightarrow
  operation.signature.name_Happened \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.
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
Chapter 7. RoboWorld in RoboTool: authoring RoboWorld documents

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

![Dictionary Editor](image)

Figure 7.2: Combination of structural and surface editing

(a) New ElementAssumption  
(b) Types of ElementAssumptions  
(c) Types of Sentences

![Combination of Editing Paradigms](image)

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 the 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 and 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 if 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

abstract RoboWorld =
  RoboWorldLexicon,
  Numeral
**
{
  cat -- closed categories
    Unit ;

  fun -- functions of closed categories
  -- SI base units
Chapter A. Complete RoboWorld Grammar

m_Unit : Unit ;
meter_Unit : Unit ;
s_Unit : Unit ;
second_Unit : Unit ;
mole_Unit : Unit ;
a_Unit : Unit ;
ampere_Unit : Unit ;
k_Unit : Unit ;
kelvin_Unit : Unit ;
cd_Unit : Unit ;
candela_Unit : Unit ;
kg_Unit : Unit ;
kilogram_Unit : Unit ;

-- Other units
mm_Unit : Unit ;
millimeter_Unit : Unit ;
min_Unit : Unit ;
minute_Unit : Unit ;
ms_Unit : Unit ;
rads_Unit : Unit ;

---------------------------------------------------------------------

cat -- ItemPhrase

BasicItem ;
CompoundItem ;
Item ;
ItemPhrase ;
ItemPhraseList ;

---------------------------------------------------------------------

fun -- ItemPhrase

-- velocity
mkBasicItem_single_noun : Cat.N -> BasicItem ;
-- odometer value
mkBasicItem_two_nouns : Cat.N -> Cat.N -> BasicItem

-- angular velocity
mkBasicItem_QualifiedBI : Cat.A -> BasicItem -> BasicItem

-- m/s
mkBasicItem_Unit : Unit -> BasicItem ;

-- m/s upwards
mkCompoundItem_AdverbCI : Item -> Adv -> CompoundItem ;
-- object initially
mkCompoundItem_AdverbCI_from_adjective : Item -> A -> CompoundItem ;
-- distance from the robot to the nest
mkCompoundItem_PrepositionCI_single_ItemPhrase : BasicItem -> Prep ->
ItemPhrase -> CompoundItem;
-- location except the source and the nest
mkCompoundItem_PrepositionCI_and_list_of_ItemPhrases : BasicItem -> Prep
-> ItemPhraseList -> CompoundItem;
-- location except the source or the nest
mkCompoundItem_PrepositionCI_or_list_of_ItemPhrases : BasicItem -> Prep ->
ItemPhraseList -> CompoundItem;

-- angular velocity
mkItem_from_BasicItem : BasicItem -> Item;
-- angular velocity of the robot
mkItem_from_CompoundItem : CompoundItem -> Item;

-- it
mkItemPhrase_PronounIP : Cat.Pron -> ItemPhrase;
-- the angular velocity
mkItemPhrase_DeterminedIP : Cat.Det -> Item -> ItemPhrase
⇀ ;
-- 1 position
mkItemPhrase_QuantifiedIP_with_digits : Cat.Digits -> Item -> ItemPhrase
⇀ ;
-- one position
mkItemPhrase_QuantifiedIP_with_text : Cat.Numeral -> Item -> ItemPhrase
⇀ ;
-- 0.5 m
mkItemPhrase_QuantifiedIP_with_float : Float -> Item -> ItemPhrase
⇀ ;
-- at least 1 position
mkItemPhrase_AdN_QuantifiedIP_with_digits : AdN -> Cat.Digits -> Item ->
ItemPhrase;
-- at least one position
mkItemPhrase_AdN_QuantifiedIP_with_text : AdN -> Cat.Numeral -> Item ->
ItemPhrase;
-- at least 0.5 m
mkItemPhrase_AdN_QuantifiedIP_with_float : AdN -> Float -> Item ->
ItemPhrase;
-- no obstacles
mkItemPhrase_QuantifiedIP_with_plural_Quant : Cat.Quant -> Item ->
ItemPhrase;
-- this obstacle
mkItemPhrase_QuantifiedIP_with_singular_Quant : Cat.Quant -> Item ->
ItemPhrase;
-- 0.0
mkItemPhrase_Float_Literal : Float -> ItemPhrase;

-- [an x-width of 1 m, an y-width of 1 m]
mkItemPhraseList_binary : ItemPhrase -> ItemPhrase -> ItemPhraseList;
-- [an x-width of 1 m, an y-width of 1 m, an z-width of 1 m]
mkItemPhraseList_many : ItemPhrase -> ItemPhraseList -> ItemPhraseList;
Chapter A. Complete RoboWorld Grammar

---
cat -- RWClause

RWClause ;

---
fun -- RWClause

-- the odometer of the robot is reset
mkRWClause_PassiveVoice_IntransitiveVerb : ItemPhrase -> V -> RWClause ;
-- the velocity of the robot is set to 1 m/s upward
mkRWClause_PassiveVoice_TransitiveVerb_Preposition_ItemPhrase : ItemPhrase
  <-> V2 -> Prep -> ItemPhrase -> RWClause ;

-- the arena is three-dimensional
mkRWClause_ActiveVoice_ToBe_Adjective : ItemPhrase -> A -> RWClause
  <-> ;
-- the tank of water is either full or empty
mkRWClause_ActiveVoice_ToBe_Conj_Adjective_Adjective : ItemPhrase -> Conj
  <-> A -> A -> RWClause ;
-- the gradient of the ground is 0.0
mkRWClause_ActiveVoice_ToBe_ItemPhrase : ItemPhrase -> ItemPhrase ->
  RWClause ;
-- the robot is in the origin initially
mkRWClause_ActiveVoice_ToBe_Preposition_ItemPhrase : ItemPhrase -> Prep -> ItemPhrase
  <-> RWClause ;
-- the distance from the target to the origin is greater than 1 m
mkRWClause_ActiveVoice_ToBe_Comparison_ItemPhrase : ItemPhrase -> A -> ItemPhrase
  <-> RWClause ;
-- the robot places an object in the nest
mkRWClause_ActiveVoice_TransitiveVerb_ItemPhrase : ItemPhrase -> V2 -> ItemPhrase
  <-> ItemPhrase -> RWClause ;
-- the robot may carry 1 object
mkRWClause_ActiveVoice_Modal_TransitiveVerb_ItemPhrase : ItemPhrase -> VV
  <-> V2 -> ItemPhrase -> RWClause ;
-- the nest may contain up to 5 objects
mkRWClause_ActiveVoice_Modal_TransitiveVerb_Prep_ItemPhrase : ItemPhrase
  <-> VV -> V2 -> Prep -> ItemPhrase -> RWClause ;
-- it is raining
mkRWClause_ActiveVoice_Progressive_IntransitiveVerb : ItemPhrase -> V -> ItemPhrase
  <-> RWClause ;
-- the robot is carrying an object
mkRWClause_ActiveVoice_Progressive_TransitiveVerb_ItemPhrase : ItemPhrase
  <-> V2 -> ItemPhrase -> RWClause ;
---
fun -- Conditions

-- when the distance from the robot to the source is less than 1 m,

-- the distance from the robot to the nest is more than 2 m and the robot
→ is carrying an object

\[
\text{mkConditions_Subj_RWSentences} : \text{Subj} \rightarrow \text{RWSentences} \rightarrow \text{Conditions} ;
\]

------------------------------------------
cat -- ArenaAssumption

\[
\text{ArenaAssumption} ;
\]

------------------------------------------
fun -- ArenaAssumption

-- some locations of the arena except the source and the nest contain 1
→ obstacles

\[
\text{mkArenaAssumption_RWSentence} : \text{RWSentence} \rightarrow \text{ArenaAssumption} ;
\]

------------------------------------------
cat -- RobotAssumption

\[
\text{RobotAssumption} ;
\]

------------------------------------------
fun -- RobotAssumption

-- the robot is a point mass

\[
\text{mkRobotAssumption_RWSentence} : \text{RWSentence} \rightarrow \text{RobotAssumption} ;
\]

-- the robot is defined by a diagram

\[
\text{mkRobotAssumption_PModel} : \text{RobotAssumption} ;
\]

------------------------------------------
cat -- ElementAssumption

\[
\text{ElementAssumption} ;
\]

------------------------------------------
fun -- ElementAssumption

-- the source has an x-width of 0.25 m and a y-width of 0.25 m

\[
\text{mkElementAssumption_RWSentence} : \text{RWSentence} \rightarrow \text{ElementAssumption} ;
\]

-- the room is defined by a diagram

\[
\text{mkElementAssumption_PModel} : \text{Item} \rightarrow \text{ElementAssumption} ;
\]

------------------------------------------
cat -- InputEventMapping

\[
\text{InputEventMapping} ;
\]

------------------------------------------
fun -- InputEventMapping

A.1 RoboWorld.gf

-- when the distance from the robot to an obstacle is less than 1 m the event obstacle occurs
mkInputEventMapping_InputSometimes : String -> Conditions ->
  InputEventMapping ;
-- when the distance from the robot to an obstacle is less than 1 m
-- the event obstacle occurs and it communicates the linear velocity of the robot
mkInputEventMapping_InputSometimes_RWSentences : String -> Conditions ->
  RWSentences -> InputEventMapping ;
-- the event angularSpeed is always available
mkInputEventMapping_InputAlways : String -> InputEventMapping ;
-- the event angularSpeed is always available and it communicates the angular velocity of the robot
mkInputEventMapping_InputAlways_RWSentences : String -> RWSentences ->
  InputEventMapping ;
-- the event transferred never happens
mkInputEventMapping_InputNever : String -> InputEventMapping ;

-- OutputEventMapping
OutputEventMapping ;

-- when the event takeoff occurs if it is raining the velocity of the robot is set to 2.0 m/s upward
mkOutputEventMapping_Sometimes : String -> Conditions -> RWSentences ->
  OutputEventMapping ;
-- when the event takeoff occurs the velocity of the robot is set to 1 m/s upward
mkOutputEventMapping_OutputAlways : String -> RWSentences ->
  OutputEventMapping ;
-- when the event takeoff occurs nothing happens
mkOutputEventMapping_NoOutput : String -> OutputEventMapping ;
-- the event spray is defined by a diagram where one time unit is 1.0 s
mkOutputEventMapping_DiagrammaticOutput : String -> Float -> Unit ->
  OutputEventMapping ;
-- when the event spray occurs if the tank of water is full the effect is defined by a diagram where one time unit is 1.0 s
mkOutputEventMapping_DiagrammaticOutput_Conditions : String -> Conditions
  -> Float -> Unit -> OutputEventMapping ;

-- OperationMapping
OperationMapping ;
fun -- OperationMapping

-- when the operation Store() is called
-- as soon as the distance from the robot to the source is less than 1.0 m
\n-> the robot places an object in the nest

mkOperationMapping_Sometimes : String -> Conditions -> RWSentences ->
-> OperationMapping ;

-- when the operation move(ls,as) is called
-- the velocity of the robot is set to ls m/s towards the orientation of
\n-> the robot
-- and the angular velocity of the robot is set to as rad/s

mkOperationMapping_OutputAlways : String -> RWSentences ->
-> OperationMapping ;

-- when the operation Transfer() is called nothing happens

mkOperationMapping_NoOutput : String -> OperationMapping ;

-- the operation turnBack() is defined by a diagram where one time unit is
\n-> 1.0 s

mkOperationMapping_DiagrammaticOutput : String -> Float -> Unit ->
-> OperationMapping ;

-- when the operation turnBack() is called
-- if it is raining the effect is defined by a diagram where one time unit
\n-> is 1.0 s

mkOperationMapping_DiagrammaticOutput_Conditions : String -> Conditions ->
-> Float -> Unit -> OperationMapping ;

----------------------------------------

cat -- VariableMapping

VariableMapping ;

----------------------------------------

fun -- VariableMapping

-- when the robot is on the floor the variable dist is incremented

mkVariableMapping_Conditions_RWSentences : Conditions -> RWSentence ->
-> VariableMapping ;

----------------------------------------

-- Help functions for RoboWorld plugin

fun _special_N : N;
fun _special_A : A;
fun _special_AdN : AdN;
fun _special_Adv : Adv;
fun _special_AdV : AdV;
fun _special_Conj : Conj;
fun _special_Quant : Quant;
fun _special_Prep : Prep;
fun _special_Pron : Pron;
fun _special_Subj : Subj;
fun _special_V : V;
A.2 RoboWorldEng.gf

--- Concrete grammar of RoboWorld: a CNL for robotic systems

---

-- Authors:
-- * James Baxter <james.baxter@york.ac.uk>
-- (Department of Computer Science, University of York, UK)
-- * Gustavo Carvalho <ghpc@cin.ufpe.br> [corresponding author]
-- (Centro de Informática, Universidade Federal de Pernambuco, BR)
-- * Ana Cavalcanti <ana.cavalcanti@york.ac.uk>,
-- (Department of Computer Science, University of York, UK)
---

concrete RoboWorldEng of RoboWorld =
RoboWorldLexiconEng,
NumeralEng
**
open
SyntaxEng,
(ResEng = ResEng),
ParadigmsEng,
SymbolicEng,
ExtraEng,
Prelude,
MorphoEng,
Chapter A. Complete RoboWorld Grammar

```
ParamX
in {

-----------------------------
lincat -- closed categories
-----------------------------

Unit = CatEng.N ;

-----------------------------
lin -- functions of closed categories
-----------------------------

-- SI base units
m_Unit = mkN "m" "m" ;
meter_Unit = mkN "meter" "meters" ;
s_Unit = mkN "s" "s" ;
second_Unit = mkN "second" "seconds" ;
mole_Unit = mkN "mole" "moles" ;
a_Unit = mkN "A" "A" ;
ampere_Unit = mkN "ampere" "amperes" ;
k_Unit = mkN "K" "K" ;
kelvin_Unit = mkN "kelvin" "kelvins" ;
cd_Unit = mkN "cd" "cd" ;
candela_Unit = mkN "candela" "candelas" ;
kg_Unit = mkN "kg" "kg" ;
kilogram_Unit = mkN "kilogram" "kilograms" ;

-- Other units
mm_Unit = mkN "mm" "mm" ;
millimeter_Unit = mkN "millimeter" "millimeters" ;
min_Unit = mkN "min" "min" ;
minute_Unit = mkN "minute" "minutes" ;
ms_Unit = mkN "m/s" "m/s" ;
rads_Unit = mkN "rad/s" "rad/s" ;

-----------------------------
lincat -- ItemPhrase
-----------------------------

BasicItem = CatEng.CN ;
CompoundItem = CatEng.CN ;
Item = CatEng.CN ;
ItemPhrase = CatEng.NP ;
ItemPhraseList = ListNP ;

-----------------------------
lin -- ItemPhrase
-----------------------------

mkBasicItem_single_noun n =
  -- mkCN : N -> CN / velocity
  mkCN (lin N n) ;
```
mkBasicItem_two_nouns n1 n2 =
    let
    -- odometer
    -- mkN : Str -> N -> N / odometer, value
    n : N = mkN str (lin N n2) ;
    in
    mkCN n ;

mkBasicItem_QualifiedBI adj cn =
    -- mkCN : A -> CN -> CN / angular, velocity
    mkCN <lin A adj : A> <cn : CN> ;

mkBasicItem_Unit unit =
    -- mkCN : N -> CN / m/s
    mkCN unit ;

mkCompoundItem_AdverbCI item adv =
    -- mkCN : CN -> Adv -> CN / m/s, upwards
    mkCN item adv ;

mkCompoundItem_AdverbCI_from_adjective item adj =
    let
    -- mkAdv : A -> Adv / initial
    adv : CatEng.Adv = SyntaxEng.mkAdv (lin A adj) in
    -- mkCN : CN -> Adv -> CN / object, initially
    mkCN item adv ;

mkCompoundItem_PrepositionCI_single_ItemPhrase basic prep itemPhrase =
    let
    -- mkAdv : Prep -> NP -> Adv / from, the robot to the nest
    adv : CatEng.Adv = SyntaxEng.mkAdv (lin Prep prep) in
    -- mkCN : CN -> Adv -> CN / distance, from the robot to
    -- the nest
    mkCN basic adv ;

mkCompoundItem_PrepositionCI_and_list_of_ItemPhrases basic prep list =
    let
    -- mkNP : Conj -> ListNP -> NP / and, [the source, the
    -- nest]
    npAndList : NP = mkNP and_Conj list ;
    -- mkAdv : Prep -> NP -> Adv / except, the source and the
    -- nest
    adv : CatEng.Adv = SyntaxEng.mkAdv (lin Prep prep) in
    npAndList ;
Chapter A. Complete RoboWorld Grammar

-- mkCN : CN -> Adv -> CN | location, except the source and the nest
mkCN basic adv;

mkCompoundItem_PrepositionCI_or_list_of_ItemPhrases basic prep list =
let
  -- mkNP : Conj -> ListNP -> NP | or, [the source, the nest]
npAndList : NP = mkNP or_Conj list ;
  -- mkAdv : Prep -> NP -> Adv | except, the source or the nest
  npAndList ;
in
  -- mkCN : CN -> Adv -> CN | location, except the source or the nest
mkCN basic adv ;

mkItem_from_BasicItem basicItem =
basicItem ; -- angular velocity

mkItem_from_CompoundItem compoundItem =
compoundItem ; -- angular velocity of the robot

mkItemPhrase_PronounIP pron =
  -- mkNP : Pron -> NP | it
mkNP <lin Pron pron : Pron> ;

mkItemPhrase_DeterminedIP det item =
  -- mkNP : Det -> CN -> NP | the, angular velocity
mkNP <lin Det det : Det> <item : CN> ;

mkItemPhrase_QuantifiedIP_with_digits digits item =
let
  -- 1
  det : Det = (mkDet <(lin Digits digits) : Digits>) ;
in
  -- mkNP : Det -> CN -> NP | 1, position
mkNP det item ;

mkItemPhrase_QuantifiedIP_with_text numeral item =
let
  -- one
  det : Det = (mkDet <(lin Numeral numeral) : Numeral>) ;
in
  -- mkNP : Det -> CN -> NP | one, position
mkNP det item ;

mkItemPhrase_QuantifiedIP_with_float float item =
let
  -- mkSymb : Str -> Symb | 0.5
  sym : Symb = mkSymb float.s ;
  -- symb : Symb -> Card | 0.5
  card : Card = symb sym ;
  -- mkDet : Card -> Det | 0.5
  det : Det = mkDet card ;
  \in
  -- mkNP : Det -> CN -> NP | 0.5, m
  mkNP det item ;

mkItemPhrase_AdN QuantifiedIP with digits adn digits item =
  let
    -- mkCard : Digits -> Card | 1
    card : Card = mkCard <(lin Digits digits) : Digits> ;
    -- mkCard : AdN -> Card -> Card | at least, 1
    adnCard : Card = mkCard <(lin AdN adn) : AdN> card ;
    -- mkDet : Card -> Det | at least 1
    det : Det = mkDet adnCard ;
    \in
    -- mkNP : Det -> CN -> NP | at least 1, position
    mkNP det item ;
  \n
mkItemPhrase_AdN QuantifiedIP with text adn numeral item =
  let
    -- mkCard : Digits -> Card | one
    card : Card = mkCard <(lin Numeral numeral) : Numeral> ;
    -- mkCard : AdN -> Card -> Card | at least, one
    adnCard : Card = mkCard <(lin AdN adn) : AdN> card ;
    -- mkDet : Card -> Det | at least one
    det : Det = mkDet adnCard ;
    \in
    -- mkNP : Det -> CN -> NP | at least one, position
    mkNP det item ;
  \n
mkItemPhrase_AdN QuantifiedIP with float adn float item =
  let
    -- mkSymb : Str -> Symb | 0.5
    sym : Symb = mkSymb float.s ;
    -- symb : Symb -> Card | 0.5
    card : Card = symb sym ;
    -- mkCard : AdN -> Card -> Card | at least 0.5
    adnCard : Card = mkCard <(lin AdN adn) : AdN> card ;
    -- mkDet : Card -> Det | 0.5
    det : Det = mkDet adnCard ;
    \in
    -- mkNP : Det -> CN -> NP | at least 0.5, m
    mkNP det item ;
  \n
mkItemPhrase QuantifiedIP with plural Quant quant item =
let
  -- mkDet : Quant -> Num -> Det | no, 'plNum'
  det : Det = mkDet (lin Quant quant) : Quant> plNum
in
  -- mkNP : Det -> CN -> NP | no, obstacle
  mkNP det item ;

mkItemPhrase_QuantifiedIP_with_singular_Quant quant item =
  let
    -- mkDet : Quant -> Num -> Det | this, 'sgNum'
    det : Det = mkDet (lin Quant quant) : Quant> sgNum
  in
    -- mkNP : Det -> CN -> NP | this, obstacle
    mkNP det item ;

mkItemPhrase_Float_Literal float =
  -- symb : Float -> NP | 0.0
  symb float ;

mkItemPhraseList_binary itemPhrase1 itemPhrase2 =
  -- mkListNP : NP -> NP -> ListNP | an x-width of 1 m, a y-width of
  -> 1 m
  mkListNP itemPhrase1 itemPhrase2 ;

mkItemPhraseList_many itemPhrase itemPhraseList =
  -- mkListNP : NP -> ListNP -> ListNP | an x-width of 1 m, [a
  -> y-width of 1 m, a z-width of 1 m]
  mkListNP itemPhrase itemPhraseList ;

--------------------------------------------------------------------------------

lincat -- RWClause

RWClause = CatEng.Cl ;

--------------------------------------------------------------------------------

lin -- RWClause

mkRWClause_PassiveVoice_IntransitiveVerb itemPhrase v =
  let
    -- mkV2 : V -> V2 | reset
    -- passiveVP : V2 -> VP |
    ⇝ reset
    vp : VP = passiveVP (mkV2 (lin V v) : V>)
    ⇝ ;
  in
    -- mkCl : NP -> VP -> Cl | the odometer of the robot, is
    ⇝ reset
    mkCl itemPhrase vp ;
mkRWClause_PassiveVoice_TransitiveVerb_Preposition_ItemPhrase itemPhrase1
  v2 prep itemPhrase2 =
    let
      -- passiveVP : V2 -> VP | set
      passiveVerb : VP = passiveVP ((lin V2 v2) : V2) ;
      -- mkAdv : Prep -> NP -> Adv | to, 1 m/s upward
        itemPhrase2 ;
      -- mkVP : VP -> Adv -> VP | is set, to 1 m/s upward
      vp : VP = mkVP passiveVerb adv ;
    in
      -- mkCl : NP -> VP -> Cl | the velocity of the robot, is
                      set to 1 m/s upward
      mkCl itemPhrase1 vp ;

mkRWClause_ActiveVoice_ToBe_Adjective itemPhrase adj =
  -- mkCl : NP -> A -> Cl | the arena, three-dimensional
  mkCl itemPhrase (lin A adj) ;

mkRWClause_ActiveVoice_ToBe_Conj_Adjective_Adjective itemPhrase conj adj1 adj2
  let
    -- mkAP : A -> AP | full
    ap1 : AP = <lin AP (mkAP <lin A adj1 : A>) : AP> ;
    -- mkAP : A -> AP | empty
    ap2 : AP = <lin AP (mkAP <lin A adj2 : A>) : AP> ;
    -- mkAP : Conj -> AP -> AP -> AP | either ... or ...,
      full, empty
    ap : AP = mkAP <lin Conj conj : Conj> ap1 ap2 ;
  in
    -- mkCl : NP -> AP -> Cl | the tank of water, either full
      or empty
    mkCl itemPhrase ap ;

mkRWClause_ActiveVoice_ToBe_ItemPhrase itemPhrase1 itemPhrase2 =
  -- mkCl : NP -> NP -> Cl | the gradient of the ground, 0.0
  mkCl itemPhrase1 itemPhrase2 ;

mkRWClause_ActiveVoice_ToBe_Preposition_ItemPhrase itemPhrase1 prep
  itemPhrase2 =
    let
      -- mkAdv : Prep -> NP -> Adv | at the origin initially
        itemPhrase2 ;
    in
      -- mkCl : NP -> Adv -> Cl | the robot, at the origin
        initially
      mkCl itemPhrase1 adv ;
**Chapter A. Complete RoboWorld Grammar**

```plaintext
mkRWClause_ActiveVoice_ToBe_Comparison_ItemPhrase itemPhrase1 adj
  → itemPhrase2 =
  let
    -- mkAP : A -> AP | great, 1 m
    ap : AP = mkAP (lin A adj) itemPhrase2 ;
  in
    -- mkCl : NP -> AP -> Cl | the distance from the target to
    → the origin, greater than 1 m
    mkCl itemPhrase1 ap ;

mkRWClause_ActiveVoice_TransitiveVerb_ItemPhrase itemPhrase1 v2
  → itemPhrase2 =
  let
    -- mkVP : V2 -> NP -> VP | place, an object in the nest
    vp : VP = mkVP <(lin V2 v2) : V2> itemPhrase2 ;
  in
    -- mkCl : NP -> VP -> Cl | the robot, place an object in
    → the nest
    mkCl itemPhrase1 vp ;

mkRWClause_ActiveVoice_Modal_TransitiveVerb_ItemPhrase itemPhrase1 vv v2
  → itemPhrase2 =
  let
    -- mkVP : V2 -> NP -> VP | carry, 1 object
    vp : VP = mkVP <(lin V2 v2) : V2> itemPhrase2 ;
  in
    -- mkCl : NP -> VV -> VP -> Cl | the robot, may, carry 1
    → object
    mkCl itemPhrase1 <(lin VV vv) : VV> vp ;

mkRWClause_ActiveVoice_Modal_TransitiveVerb_Prep_ItemPhrase itemPhrase1 vv prep
  → itemPhrase2 =
  let
    -- ss : Str -> SS | up to
    preDet : Predet = <(lin Predet (ss prep.s)) : Predet> ;
    -- mkNP : Predet -> NP | up to, 5 objects
    np : NP = mkNP preDet itemPhrase2 ;
    -- mkVP : V2 -> NP -> VP | contain, up to 5 objects
    vp : VP = mkVP <(lin V2 v2) : V2> np ;
  in
    -- mkCl : NP -> VV -> VP -> Cl | the nest, may, contain up
    → to 5 objects
    mkCl itemPhrase1 <(lin VV vv) : VV> vp ;

mkRWClause_ActiveVoice_Progressive_IntransitiveVerb itemPhrase v =
  let
    -- mkVP : V -> VP | rain
    progressiveVP : VP -> VP | rain
    progressive : VP = progressiveVP (mkVP <(lin V v) : V>) ;
  in
```
```
-- mkCl : NP -> VP -> Cl | it, is raining
mkCl itemPhrase progressive ;

mkRWClausie_ActiveVoice_Progressive_TransitiveVerb_ItemPhrase itemPhrase1 v2
   itemPhrase2 =
   let
      -- mkVP : V2 -> NP -> VP | carry, an object
      -- progressiveVP : VP -> VP | carry an object
      progressive : VP = progressiveVP (mkVP <(lin V2 v2) : V2>
         itemPhrase2) ;
   in
      -- mkCl : NP -> VP -> Cl | the robot, is carrying an object
      mkCl itemPhrase1 progressive ;

-- lincat -- RWSentence

RWSentence = CatEng.S ;
RWSentenceList = ListS ;
RWSentences = CatEng.S ;

-- lin -- Sentence

mkRWSentence_Prefix_AdverbFromAdjective a sentence =
   let
      -- mkAdv : A -> Adv | initial
   in
      -- mkS : Adv -> S -> S | initial, the robot is in the
      <- origin
      mkS adv sentence ;

mkRWSentence_Prefix_Adverb adv sentence =
   -- mkS : Adv -> S -> S | then, the velocity of the robot is set to
   <- 1.0 m/s upward
   mkS <(lin Adv adv) : Adv> <sentence : S> ;

mkRWSentence_PresentTense_PositivePolarity clause =
   -- mkS : Cl -> S | it is raining
   mkS clause ;

mkRWSentence_PresentTense_NegativePolarity clause =
   -- mkS : Pol -> Cl -> S | UncNeg, it is not raining
   mkS UncNeg clause ;

mkRWSentence_PastTense_PositivePolarity clause =
   -- mkS : Tense -> Cl -> S | pastTense, it was raining
   mkS pastTense clause ;
```
mkRWSentence_PastTense_NegativePolarity clause =
    -- mkS : Tense -> Pol -> Cl -> S | pastTense, UncNeg, it was not raining
    mkS pastTense UncNeg clause ;

mkRWSentenceList_binary sentence1 sentence2 =
    -- mkListS : S -> S -> ListS | the odometer of the robot is reset, the velocity of the robot is set to 1 m/s upward
    mkListS sentence1 sentence2 ;

mkRWSentenceList_many sentence sentenceList =
    -- mkListS : S -> ListS -> ListS | the robot places an object in the nest,
    -- [the odometer of the robot is reset, the velocity of the robot is set to 1 m/s upward]
    mkListS sentence sentenceList ;

mkRWSentences_single_sentence sentence =
    sentence ; -- the velocity of the robot is set to 1 m/s upward

mkRWSentences_and_list_of_sentences sentenceList =
    -- mkS : Conj -> ListS -> S | and_Conj, [the event spray occurred in 3 minutes before, the operation takeOff was called in 20 minutes before]
    mkS and_Conj sentenceList ;

mkRWSentences_or_list_of_sentences sentenceList =
    -- mkS : Conj -> ListS -> S | or_Conj,
    -- [the event spray occurred in 3 minutes before, the operation takeOff was called in 20 minutes before]
    mkS or_Conj sentenceList ;

-----------------------------------------------------------------------------------------------

lincat -- Conditions

Conditions = CatEng.Adv ;

-----------------------------------------------------------------------------------------------

lin -- Conditions

mkConditions_Subj_RWSentences subj sentences =
    -- mkAdv : Subj -> S -> Adv | when,
    -- the distance from the robot to the source is less than 1 m,
    -- the distance from the robot to the nest is more than 2 m and the robot is carrying an object
    SyntaxEng.mkAdv <(lin Subj subj) : Subj> <sentences : S> ;
param OutputType = OutputEvent | Operation ;

-- auxiliary functions

is_defined_by_diagram : VP =
  let
    -- mkCN : N -> CN | diagram
    -- mkNP : Det -> CN -> NP | a, diagram
    np : NP = mkNP RoboWorldLexiconEng.a_Det (mkCN diagram_N)
    in
    -- passiveVP : V2 -> NP -> VP | define, a diagram
    passiveVP <(lin V2 define_V2) : V2> <(lin NP np) : NP> ;

the_Item_is_defined_by_diagram : Item -> S = \item ->
  let
    -- is_defined_by_diagram : VP
    vp : VP = is_defined_by_diagram ;
    in
    -- mkNP : Det -> CN -> NP | the, robot
    -- mkCl : NP -> VP -> Cl | the robot, defined by a diagram
    -- mkS : Cl -> S | the robot is defined by a diagram
    mkS (mkCl (mkNP <RoboWorldLexiconEng.the_Det : Det> <item : CN>) vp) ;

the_event_str : Str -> NP = \str ->
  let
    -- symb : Str -> NP | obstacle
    -- mkCN : N -> NP -> CN | event, obstacle
    cn : CN = mkCN RoboWorldLexiconEng.event_N (symb str) ;
    in
    -- mkNP : Det -> CN -> NP | the, event obstacle
    mkNP RoboWorldLexiconEng.the_Det cn ;

the_operation_str : Str -> NP = \str ->
  let
    -- symb : Str -> NP | takeoff
    -- mkCN : N -> NP -> CN | operation, takeoff
    cn : CN = mkCN RoboWorldLexiconEng.operation_N (symb str)
    in
    -- mkNP : Det -> CN -> NP | the, operation takeoff
    mkNP RoboWorldLexiconEng.the_Det cn ;

the_event_str_is_always_available : Str -> Cl = \str ->
let

-- the_event_str : Str -> NP | angularSpeed
np : NP = the_event_str str ;
-- mkVP : A -> VP | available
-- mkVP : Adv -> VP -> VP | always, to be available
vp : VP = mkVP always_Adv (mkVP available_A) ;
in
-- mkCl : NP -> VP -> Cl | the event angularSpeed, to be always available
mkCl np vp ;

when_the_event_Str_occurs : Str -> CatEng.Adv = \str ->
let

-- the_event_str : Str -> NP | takeoff
np : NP = the_event_str str ;
-- mkVP : V -> VP | occur
-- mkCl : NP -> VP -> Cl | the event takeoff, occur
-- mkS : Cl -> S | the event takeoff
  occurs
s : S = mkS (mkCl np (mkVP RoboWorldLexiconEng.occur_V)) ;
in
-- mkAdv : Subj -> S -> Adv | when, the event takeoff
  occurs
SyntaxEng.mkAdv RoboWorldLexiconEng.when_Subj s ;

when_the_operation_Str_is_called : Str -> CatEng.Adv = \str ->
let

-- the_event_str : Str -> NP | takeoff
np : NP = the_operation_str str ;
-- passiveVP : V2 -> VP | call
-- mkCl : NP -> VP -> Cl | the operation takeoff, to be called
-- mkS : Cl -> S | the operation takeoff is called
s : S = mkS (mkCl np (passiveVP
  RoboWorldLexiconEng.call_V2)) ;
in
-- mkAdv : Subj -> S -> Adv | when, the operation takeoff
  is called
SyntaxEng.mkAdv RoboWorldLexiconEng.when_Subj s ;

nothing_happens : S =
-- mkVP : V -> VP | happen
-- mkCl : NP -> VP -> Cl | nothing, happen
-- mkS : Cl -> S | nothing happens
mkS (mkCl nothing_NP (mkVP happen_V)) ;

where_one_time_unit_is_Str_Unit : Str -> RoboWorldEng.Unit -> CatEng.Adv =
  \str,unit ->
let

-- mkNumeral : Unit -> Numeral | n1.Unit
-- mkDet : Numeral -> Det | one
-- mkCN : N -> CN | unit
-- mkVP : CN -> NP | unit
-- mkCN : N -> NP -> CN | time, unit
-- mkNP : Det -> CN -> NP | one, time unit

one_time_unit : NP = mkNP (mkDet (mkNumeral n1_Unit))
  ∪ (mkCN time_N (mkNP (mkCN unit_1_N)))

-- mkSymb : Str -> Symb | 1.0
-- symb : Symb -> Card | 1.0

card : Card = symb (mkSymb str) ;

-- mkDet : Card -> Det | 1.0
-- mkNP : Det -> CN -> NP | 1.0, s

str_unit : NP = mkNP (mkDet card) <(lin CN unit) : CN> ;
-- mkComp : NP -> Comp | 1.0 s

comp : Comp = mkComp str_unit ;
-- mkVP : Comp -> VP | to be 1.0 s

vp : VP = mkVP comp ;
-- mkCl : NP -> VP -> Cl | one time unit, to be 1.0
  ⊣ s
-- mkS : Cl -> S | one time unit is 1.0 s

s : S = mkS (mkCl one_time_unit vp) ;

-- mkAdv : Subj -> S -> Adv | where, one time unit is 1.0
  ⊣ s

SyntaxEng.mkAdv where_Subj s ;

outputSentencePrefix_Adv = table {
  OutputEvent => when_the_event_Str_occurs ;
  Operation => when_the_operation_Str_is_called
} ;

outputSentencePrefix_NP = table {
  OutputEvent => the_event_str ;
  Operation => the_operation_str
} ;

output_sometimes : OutputType -> Str -> Conditions -> RWSentences -> S =
  \outputType, str, conditions, sentences ->
  let
    outputSentencePrefix_Adv / when the event str occurs OR
    when the operation str is called

    adv : CatEng.Adv = (outputSentencePrefix_Adv ! outputType)
      ⊣ str ;

    -- mkS : Adv -> S -> S | if it is raining, the velocity of
    -- the robot is set to 2.0 m/s upward

    s : S = mkS <conditions : Adv> <sentences : S> ;

    in
      -- mkS : Adv -> S -> S, when the event takeoff occurs,
-- if it is raining the velocity of the robot is set to 2.0 m/s upward
mkS adv s;

output_always : OutputType -> Str -> RWSentences -> S = \outputType, str,
\sentences ->
  let
    -- outputSentencePrefix_Adv / when the event str occurs OR
    -- when the operation str is called
    adv : CatEng.Adv = (outputSentencePrefix_Adv ! outputType)
    \str ;
    in
    -- mkS : Adv -> S -> S, when the event takeoff occurs, the
    -- velocity of the robot is set to 1.0 m/s upward
    mkS <adv : Adv> <sentences : S> ;

no_output : OutputType -> Str -> S = \outputType, str ->
  let
    -- outputSentencePrefix_Adv / when the event str occurs OR
    -- when the operation str is called
    adv : CatEng.Adv = (outputSentencePrefix_Adv ! outputType)
    \str ;
    in
    -- mkS : Adv -> S -> S, when the event takeoff occurs, nothing happens
    mkS adv nothing_happens ;

diagrammatic_output : OutputType -> Str -> Str -> RoboWorldEng.Unit -> S =
\outputType, str, float, unit ->
  let
    -- outputSentencePrefix_NP / the event str OR the
    -- operation str
    np : NP = (outputSentencePrefix_NP ! outputType) str ;
    -- is_defined_by_diagram : VP
    defined_by : VP = is_defined_by_diagram ;
    -- where_one_time_unit_is_Str_Unit : Str -> Unit -> Adv
    adv : CatEng.Adv = where_one_time_unit_is_Str_Unit float
    \(lin Unit unit) ;
    -- mkVP : VP -> Adv -> VP ; to be defined by a diagram,
    -- where one time unit is 1.0 s
    vp : VP = mkVP defined_by adv ;
    in
    -- mkCl : NP -> VP -> Cl / the event spray, to be defined
    -- by a diagram where one time unit is 1.0 s
    -- mkS : Cl -> S / the event spray is defined by a diagram
    -- where one time unit is 1.0 s
    mkS (mkCl np vp) ;
diagrammatic_output_conditions : OutputType -> Str -> Conditions -> Str ->
RoboWorldEng.Unit -> S =
\outputType, str, conditions, float, unit ->
let

-- mkCN : N -> CN | effect
-- mkNP : Det -> CN -> NP | the, effect
np : NP = mkNP RoboWorldEng.the_Det (mkCN
RoboWorldEng.effect_N);

-- is_defined_by_diagram : VP
defined_by : VP = is_defined_by_diagram ;

-- where_one_time_unit_is_Str_Unit : Str -> Unit -> Adv
adv : CatEng.Adv = where_one_time_unit_is_Str_Unit
float
(lin Unit unit);

-- mkVP : VP -> Adv -> VP ; to be defined by a diagram,

vp : VP = mkVP defined_by adv ;

-- mkCl : NP -> VP -> Cl | the effect, to be defined by a

s1 = mkS (mkCl np vp);

-- mkS : Adv -> S -> S | if the tank of water is full, the
effect is defined by a diagram where one time unit is
s2 : S = mkS <conditions : Adv> <s1 : S> ;

-- outputSentencePrefix_Adv / when the event str occurs OR
-- when the operation str is called
adv2 : CatEng.Adv = (outputSentencePrefix_Adv !
outputType) str ;

in

-- mkS : Adv -> S -> S | when the event spray occurs,

mkS adv2 s2 ;

-----------------------------------------------

lin

-- ArenaAssumption

ArenaAssumption = CatEng.S ;

-----------------------------------------------

lin

-- RobotAssumption

mkArenaAssumption_RWSentence sentence =
sentence ; -- some locations of the arena except the source and
the nest contain 1 obstacles

-----------------------------------------------

lin

-- RobotAssumption

RobotAssumption = CatEng.S ;
lin -- RobotAssumption

mkRobotAssumption_RWSentence sentence =
  sentence ; -- the robot is a point mass

mkRobotAssumption_PModel =
  let
    -- mkBasicItem_single_noun : N -> BasicItem | robot
    -- mkItem_from_BasicItem : BasicItem -> Item | robot
    item : Item = mkItem_from_BasicItem
    in
      -- the Item_is_defined_by_diagram : Item -> S | robot
      the_Item_is_defined_by_diagram item;

lincat -- ElementAssumption

ElementAssumption = CatEng.S ;

lin -- InputEventMapping

mkInputEventMapping_InputSometimes str conditions =
  let
    -- the_event_str : Str -> NP | obstacle
    np : NP = the_event_str str.s ;
    -- mkVP : V -> VP | occur
    vp : VP = mkVP RoboWorldLexiconEng.occur_V ;
    -- mkCl : NP -> VP -> Cl | the event obstacle, occur
    -- mkS : Cl -> S | the event obstacle occurs
s : S = mkS (mkCl np vp) ;

in

-- mkS : Adv -> S -> S | when the distance from the robot
→ to an obstacle is less than 1 m the event obstacle
→ occurs
mkS conditions s ;

mkInputEventMapping_InputSometimes_RWSentences str conditions sentences =

let

-- the_event_str : Str -> NP | obstacle
np : NP = the_event_str str.s ;
-- mkVP : V -> VP | occur
vp : VP = mkVP RoboWorldLexiconEng.occur_V ;
-- mkCl : NP -> VP -> Cl | the event obstacle, occur
-- mkS : Cl -> S | the event obstacle occurs
s : S = mkS (mkCl np vp) ;

in

-- mkListS : S -> S -> ListS | the event obstacle occurs,
→ it communicates the linear velocity of the robot
-- mkS : Conj -> List -> S | and, [the event obstacle
→ occurs, it communicates the linear velocity of the
→ robot]
-- mkS : Adv -> S -> S | when the distance from the robot
→ to an obstacle is less than 1 m,
→ the event obstacle occurs and it
→ communicates the linear velocity of the robot
mkS conditions (mkS and_Conj (mkListS s sentences)) ;

mkInputEventMapping_InputAlways str =

-- the_event_str_is_always_available : Str -> Cl | angularSpeed
-- mkS : Cl -> S | the event angularSpeed is always available
mkS (the_event_str_is_always_available str.s) ;

mkInputEventMapping_InputAlways_RWSentences str sentences =

let

-- the_event_str_is_always_available : Str -> Cl | angularSpeed
-- mkS : Cl -> S | the event angularSpeed is always available
s : S = mkS (the_event_str_is_always_available str.s) ;

in

-- mkListS : S -> S -> ListS | the event angularSpeed is
→ always available,
→ it communicates the angular
→ velocity of the robot
-- mkS : Conj -> List -> S | and,
--
→ [the event angularSpeed is always available,
→ it communicates the angular
→ velocity of the robot]
Chapter A. Complete RoboWorld Grammar

```
mkInputEventMapping_InputNever str =
  let
    -- the_event_str : Str -> NP, transferred
    np : NP = the_event_str str.s ;
    -- mkVP : V -> VP | happen
    -- mkVP : AdV -> VP -> VP | never, happen
    vp : VP = mkVP never_AdV (mkVP happen_V) ;
    in
      -- mkCl -> NP -> VP -> Cl | the event transferred, never
      -- happen
      -- mkS : Cl -> S | the event transferred never happens
      mkS (mkCl np vp) ;

--------------------------------------------------------------------------------

OutputEventMapping = CatEng.S ;

--------------------------------------------------------------------------------

mkOutputEventMapping_Sometimes eventName conditions sentences =
  -- output_sometimes : OutputType -> Str -> Conditions ->
  -- RWSentences -> S | OutputEvent, takeoff, if it is raining, the velocity of the
  -- robot is set to 2.0 m/s upward
  output_sometimes OutputEvent eventName.s (lin Conditions
  -- conditions) (lin RWSentences sentences) ;

mkOutputEventMapping_OutputAlways eventName sentences =
  -- output_always : OutputType -> Str -> RWSentences -> S |
  -- OutputEvent, takeoff, the velocity of the robot is set to 1.0
  -- m/s upward
  output_always OutputEvent eventName.s (lin RWSentences sentences) ;

mkOutputEventMapping_NoOutput eventName =
  -- no_output : OutputType -> Str -> S |
  -- OutputEvent, takeoff
  no_output OutputEvent eventName.s ;

mkOutputEventMapping_DiagrammaticOutput eventName float unit =
  -- diagrammatic_output : OutputType -> Str -> Str ->
  -- RoboWorldEng.Unit -> S |
  -- OutputEvent, spray, 1.0, s
  diagrammatic_output OutputEvent eventName.s float.s (lin Unit
  -- unit) ;
```
mkOutputEventMapping_DiagrammaticOutput_Conditions eventName conditions =
  float unit =
  -- diagrammatic_output_conditions : OutputType -> Str ->
  -- Conditions -> Str -> RoboWorldEng.Unit -> S |
  -- OutputEvent, spray, if the tank of water is full, 1.0, s
  diagrammatic_output_conditions OutputEvent eventName.s (lin
  -- Conditions conditions) float.s (lin Unit unit);

lincat -- OperationMapping

OperationMapping = CatEng.S;

lin -- OperationMapping

mkOperationMapping_Sometimes eventName conditions sentences =
  -- output_sometimes : OutputType -> Str -> Conditions ->
  RWSentences -> S |
  -- Operation, Store(), as soon as the distance from the robot to
  -- the source is less than 1.0 m,
  -- the robot places an object in the nest
  output_sometimes Operation eventName.s (lin Conditions conditions)
  (lin RWSentences sentences);

mkOperationMapping_OutputAlways eventName sentences =
  -- output_always : OutputType -> Str -> RWSentences -> S |
  -- Operation, move(ls,as), the velocity of the robot is set to ls
  -- m/s towards the orientation of the robot
  -- and the angular velocity of the robot is set to as rad/s
  output_always Operation eventName.s (lin RWSentences sentences);

mkOperationMapping_NoOutput eventName =
  -- no_output : OutputType -> Str -> S |
  -- Operation, Transfer()
  no_output Operation eventName.s;

mkOperationMapping_DiagrammaticOutput eventName float unit =
  -- diagrammatic_output : OutputType -> Str -> Str ->
  -- RoboWorldEng.Unit -> S |
  -- Operation, turnBack(), 1.0, s
  diagrammatic_output Operation eventName.s float.s (lin Unit unit)
  ;

mkOperationMapping_DiagrammaticOutput_Conditions eventName conditions =
  float unit =
  -- diagrammatic_output_conditions : OutputType -> Str ->
  -- Conditions -> Str -> RoboWorldEng.Unit -> S |
  -- Operation, turnBack(), if it is raining, 1.0, s
Chapter A. Complete RoboWorld Grammar

diagrammatic_output_conditions Operation eventName.s (lin
  ← Conditions conditions) float.s (lin Unit unit) ;

------------------------------------------------------------------------
lincat -- VariableMapping

  VariableMapping = CatEng.S ;

------------------------------------------------------------------------
lin -- VariableMapping

  mkVariableMapping_Conditions_RWSentences conditions sentence =
  -- mkS : Adv -> S -> S | when the robot is on the floor, the
  <- variable dist is incremented
  mkS conditions sentence ;

------------------------------------------------------------------------
-- Help functions for RoboWorld plugin
lin _special_N = mkN "[N]" "[N]";
lin _special_A = mka "[A]" "[A]";
lin _special_AdN = ParadigmsEng.mkAdN "[AdN]";
lin _special_Adv = ParadigmsEng.mkAdv "[Adv]";
lin _special_AdV = mkAdv "[AdV]";
lin _special_Conj = mkConj "[Conj]";
lin _special_Quant = ParadigmsEng.mkQuant "[Quant]" "[Quant]" "[Quant]" "[Quant]" <->;
lin _special_Prep = mkPrep "[Prep]";
  -> P3 nonhuman;
lin _special_Subj = mkSubj "[Subj]";
lin _special_V = mkV "[V]" "[V]" "[V]" "[V]" "[V]";
lin _special_V2 = mkV2 (mkV "[V2]" "[V2]" "[V2]" "[V2]" "[V2]");
lin _special_VV = mkVV (mkV "[VV]");
lin _special_empty_V = mkV "" "$" "$" "$" "$";
lin _special_Unit = mkN "[Unit]" "[Unit]";
lin _special_BasicItem = mkCN (mkN "[BasicItem]" "[BasicItem]");
lin _special_CompoundItem = mkCN (mkN "[CompoundItem]" "[CompoundItem]");
lin _special_Item = mkCN (mkN "[Item]" "[Item]");
lin _special_ItemPhrase = mkNP (mkN "[ItemPhrase]" "[ItemPhrase]");
lin _special_ItemPhraseList = mkListNP (mkNP (mkN "[ItemPhrase]" "[ItemPhrase]"))
  -> (mkNP (mkN "[ItemPhrase]" "[ItemPhrase]"));
lin _special_RWSentence = mkS (mkCl (mkNP (mkN "[RWSentence]" "[RWSentence]"))
  _special_empty_V);
lin _special_RWSentenceList = mkListS (mkS (mkCl (mkNP (mkN "[RWSentenceList]"
  "[RWSentenceList]")) _special_empty_V));
lin _special_RWSentences = mkS (mkCl (mkNP (mkN "[RWSentences]" "[RWSentences]"))
  _special_empty_V);
A.3 RoboWorldLexicon.gf

```
abstract RoboWorldLexicon =

  Cat

  **

  \{ 

    -- A
    fun a_Det : Det;
    fun after_Prep : Prep;
    fun angular_A : A;
    fun aPl_Det : Det;
    fun arena_N : N;
    fun asap_Subj : Subj;
    fun at_Prep : Prep;
```
Chapter A. Complete RoboWorld Grammar

fun at_least_AdN : AdN;
fun available_A : A;

-- B
fun before_Adv : Adv;
fun block_V2 : V2;
fun box_N : N;

-- C
fun can_VV : VV;
fun call_V : V;
fun call_V2 : V2;
fun carry_V2 : V2;
fun circle_N : N;
fun closed_A : A;
fun communicate_V2 : V2;
fun contain_V2 : V2;
fun cylinder_N : N;

-- D
fun define_V2 : V2;
fun depth_N : N;
fun diagram_N : N;
fun dimension_N : N;
fun direction_N : N;
fun distance_N : N;
fun downward_Adv : Adv;
fun downwards_Adv : Adv;

-- E
fun effect_N : N;
fun either7or_DConj : Conj;
fun event_N : N;

-- F
fun floor_N : N;
fun from_Prep : Prep;

-- G
fun gradient_N : N;
fun great_A : A;
fun ground_N : N;

-- H
fun have_V : V;
fun have_V2 : V2;
fun happen_V : V;
fun height_N : N;

-- I
fun if_Subj : Subj;
fun in_Prep : Prep;
fun initial_A : A;
fun inside_Prep : Prep;
fun it_Pron : Pron;

-- J

-- K

-- L
fun less_than_A : A;

-- M
fun magnitude_N : N;
fun mass_N : N;
fun may_1_VV : VV; -- be possible
fun minus_Prep : Prep;
fun movement_N : N;

-- N
fun never_AdV : AdV;
fun neither7nor_DConj : Conj;
fun no_Quant : Quant;

-- O
fun obstacle_N : N;
fun occur_V : V;
fun occurrence_N : N;
fun odometer_N : N;
fun of_Prep : Prep;
fun on_Prep : Prep;
fun one_dimensional_A : A;
fun operation_N : N;
fun orientation_N : N;
fun output_N : N;

-- P
fun place_V2 : V2;
fun plus_Prep : Prep;
fun point_N : N;
fun position_N : N;
fun pose_N : N;

-- Q
fun quarter_N : N;

-- R
fun receive_V2 : V2;
fun region_N : N;
fun reset_V : V;
fun robot_N : N;

-- S
fun sequence_N : N;
fun send_V2 : V2;
fun set_V2 : V2;
fun speed_N : N;
fun sphere_N : N;
fun square_N : N;

-- T
fun take_V2 : V2;
fun the_Det : Det;
fun then_Adv : Adv;
fun thePl_Det : Det;
fun three_dimensional_A : A;
fun through_Prep : Prep;
fun time_N : N;
fun times_Prep : Prep;
fun to_Prep : Prep;
fun towards_Prep : Prep;
fun two_dimensional_A : A;

-- U
fun under_Prep : Prep;
fun unit_1_N : N;
fun up_to_Prep : Prep;
fun upward_Adv : Adv;
fun upwards_Adv : Adv;

-- V
fun value_N : N;
fun variable_N : N;
fun velocity_N : N;

-- X
fun x_position_N : N;
fun x_width_N : N;

-- W
fun when_Subj : Subj;
fun where_Subj : Subj;
fun width_N : N;
fun wind_N : N;
fun within_Prep : Prep;

-- Y
fun y_position_N : N;
fun y_width_N : N;
fun z_position_N : N;
fun z_width_N : N;

{ 

-- Z 

-- A

lin _1D_A = mkA "1D" "IRREG";
lin _2D_A = mkA "2D" "IRREG";
lin _3D_A = mkA "3D" "IRREG";

lin a_Det = mkDeterminer singular "a" | mkDeterminer singular "an";
lin after_PreP = mkPrep "after";
lin angular_A = compoundA (mkA "angular");
lin aPl_Det = mkDeterminer plural "";
lin arena_N = mkN "arena" "arenas";
lin asap_Subj = mkSubj "as soon as" ;
lin at_PreP = mkPrep "at";
lin at_least_AdN = mkAdN "at least";
lin available_A = compoundA (mkA "available");

-- B

lin before_Adv = mkAdv "before";

-- B
lin block_V2 = mkV2 (mkV "block" "blocks" "blocked" "blocking");
lin box_N = mkN "box" "boxes";

-- C
lin can_VV = {
    s = table {
        VVF VInf => ["be able to"] ;
        VVF VPres => "can" ;
        VVF VPart => ["been able to"] ;
        VVF VPresPart => ["being able to"] ;
        VVF VPast => "could" ;
        VPastNeg => "couldn't" ;
        VVPresNeg => "can't"
    } ;
    p = [] ;
    typ = VVAux
};
lin call_V = mkV "call" "calls" "called" "called" "calling";
lin call_V2 = mkV2 (mkV "call" "calls" "called" "called" "calling");
lin carry_V2 = mkV2 (mkV "carry" "carries" "carried" "carried" "carrying";
    "carrying") ;
lin circle_N = mkN "circle" "circles";
lin closed_A = mkA "closed" "closed";
lin communicate_V2 = mkV2 (mkV "communicate" "communicates" "communicated" "communicated" "communicating";
    "communicating") ;
lin contain_V2 = mkV2 (mkV "contain" "contains" "contained" "contained" "containing";
    "containing") ;
lin cylinder_N = mkN "cylinder" "cylinders";

-- D
lin define_V2 = mkV2 (mkV "define" "defines" "defined" "defined" "defining";
    "defining") ;
lin depth_N = mkN "depth" "depths";
lin diagram_N = mkN "diagram" "diagrams";
lin dimension_N = mkN "dimension" "dimensions";
lin direction_N = mkN "direction" "directions";
lin distance_N = mkN "distance" "distances";
lin downward_Adv = mkAdv "downward";
lin downwards_Adv = mkAdv "downwards";

-- E
lin effect_N = mkN "effect" "effects";
lin either7or_DConj = mkConj "either" "or" singular ;
lin event_N = mkN "event" "events";

-- F
lin floor_N = mkN "floor" "floors";
lin from_Prep = mkPrep "from";
lin gradient_N = mkN "gradient" "gradients";
lin great_A = mkA "great" "greater";
lin ground_N = mkN "ground" "grounds";

-- H
lin have_V = IrregEng.have_V;
lin have_V2 = mkV2 (IrregEng.have_V);
lin happen_V = mkV "happen" "happens" "happened" "happened" "happening";
lin height_N = mkN "height" "heights";

-- I
lin if_Subj = mkSubj "if";
lin in_Prep = mkPrep "in";
lin initial_A = compoundA (mkA "initial");
lin inside_Prep = mkPrep "inside";
lin it_Pron = mkPron "it" "it" "its" "its" singular P3 nonhuman;

-- J

-- K

-- L
lin less_than_A = mkA "less" "less";

-- M
lin magnitude_N = mkN "magnitude";
lin mass_N = mkN "mass" "masses";
lin may_1_VV = {
  s = table {
    VVF VInf => ["be possible to"]
    VVF VPres => "may"
    VVF VPPart => ["been possible to"]
    VVF VPastPart => ["being possible to"]
    VVF VPast => "might"
    VVPastNeg => "mightn’t"
    VVPresNeg => "may not"
  };
  p = []
  typ = VVAux
}
lin minus_Prep = mkPrep "minus";
lin movement_N = mkN "movement" "movement";

-- N
lin never_AdV = mkAdV "never";
lin neither7nor_DConj = mkConj "neither" "nor" singular;
lin no_Quant = mkQuant "no" "no" "none" "none";

-- O
lin obstacle_N = mkN "obstacle" "obstacles";
lin odometer_N = mkN "odometer" "odometers";
lincurr_V = mkV "occur" "occurs" "occurred" "occurred" "occurring";
lincurrence_N = mkN "occurrence" "occurrences";
lino_of_Prep = mkPrep "of";
lino_on_Prep = mkPrep "on";
lino_one_dimensional_A = compoundA (mkA "one-dimensional");
lino_operation_N = mkN "operation" "operations";
linoOrientation_N = mkN "orientation";
linooutput_N = mkN "output" "IRREG";

-- P
linplace_V2 = mkV2 (mkV "place" "places" "placed" "placed" "placing");
lino_plus_Prep = mkPrep "plus";
lino_point_N = mkN "point" "points";
lino_position_N = mkN "position" "positions";
lino_pose_N = mkN "pose" "poses";

-- Q
linquarter_N = mkN "quarter" "quarters";

-- R
linreceive_V2 = mkV2 (mkV "receive" "receives" "received" "received" 
→ "receiving");
linoregion_N = mkN "region" "regions";
linreset_V = mkV "reset" "resets" "reset" "reset" "resetting";
linrobot_N = mkN "robot";

-- S
linsquence_N = mkN "sequence" "sequences";
linsend_V2 = mkV2 (IrregEng.send_V);
linset_V2 = mkV2 (IrregEng.set_V);
linspeed_N = mkN "speed" "speeds";
linsphere_N = mkN "sphere" "spheres";
linsquare_N = mkN "square" "squares";

-- T
linake_V2 = mkV2 (mkV "take" "takes" "took" "taken" "taking");
linthe_Det = mkDeterminer singular "the";
linthen_Adv = mkAdv "then";
linthePl_Det = mkDeterminer plural "the";
lino_three_dimensional_A = compoundA (mkA "three-dimensional");
lino_through_Prep = mkPrep "through";
lino_time_N = mkN "time" "times";
linto_Prep = mkPrep "to";
lintowards_Prep = mkPrep "towards";
lintwo_dimensional_A = compoundA (mkA "two-dimensional");

-- U
linunder_Prep = mkPrep "under";
lin unit_1_N = mkN "unit" "units";
lin up_to_Prep = mkPrep "up to";
lin upward_Adv = mkAdv "upward";
lin upwards_Adv = mkAdv "upwards";

-- Y
lin value_N = mkN "value" "values";
lin variable_N = mkN "variable" "variables";
lin velocity_N = mkN "velocity";

-- X
lin x_position_N = mkN "x-position" "x-positions";
lin x_width_N = mkN "x-width" "x-widths";

-- W
lin when_Subj = mkSubj "when";
lin where_Subj = mkSubj "where";
lin width_N = mkN "width" "widths";
lin wind_N = mkN "wind" "winds";
lin within_Prep = mkPrep "within";

-- Y
lin y_position_N = mkN "y-position" "y-positions";
lin y_width_N = mkN "y-width" "y-widths";

-- Z
lin z_position_N = mkN "z-position" "z-positions";
lin z_width_N = mkN "z-width" "z-widths";

}
B. Complete RoboWorld MetaModel

```java
import _='ecore.xml.type': 'http://www.eclipse.org/emf/2003/XMLType';

  RoboWorldMM'
{
      {,
        class RCOperation;
        class RCIntegerExp;
      }

    package PhysMod : PhysMod = 'http://www.robocalc.circus/PhysMod'
    {,
      class PModel;
      class Instantiation;
    }

    package GF : GF = 'http://www.grammaticalframework.org'
    {,
      datatype Quantifier : 'java.lang.String' { serializable };
      datatype Preposition : 'java.lang.String' { serializable };
      datatype Pronoun : 'java.lang.String' { serializable };
      datatype Noun : 'java.lang.String' { serializable };
      datatype Determiner : 'java.lang.String' { serializable };
      datatype Adjective : 'java.lang.String' { serializable };
      datatype Adverb : 'java.lang.String' { serializable };
      datatype Subjunction : 'java.lang.String' { serializable };
    }

    abstract class ItemPhrase;
    class PronounIP extends ItemPhrase
```
Chapter B. Complete RoboWorld MetaModel

```java
{  
    attribute pronoun : GF::Pronoun[1];
}
class DeterminedIP extends ItemPhrase
{
    property item : Item[1];
    attribute determiner : GF::Determiner[1];
}
class QualifiedBI extends BasicItem
{
    property basicitem : BasicItem[1];
    attribute adjective : GF::Adjective[1];
}
class QuantifiedIP extends ItemPhrase
{
    property item : Item[1];
    attribute number : GF::Quantifier[1];
}
class OperationMapping
{
    property output : Output[1] { composes };
    property signature : Signature[1] { composes };
}
class Signature
{
    property parameters : Identifier[*|1] { ordered composes };
    property name : Identifier[1] { composes };
}
class InputEventMapping
{
    property input : Input[1] { composes };
    property name : Identifier[1];
}
class OutputEventMapping
{
    property name : Identifier[1];
    property output : Output[1] { composes };
}
class VariableMapping
{
    property name : Identifier[1];
    property conditions : Conditions[1] { composes };
    property update : RWSentence[1] { composes };
}
abstract class RobotAssumption;
class RWDocument
{
    property inputEventMappings : InputEventMapping[*|1] { ordered composes };
    property outputEventMappings : OutputEventMapping[*|1] { ordered composes };
}
property operationMappings : OperationMapping[*|1] { ordered composes };
property variableMappings : VariableMapping[*|1] { ordered composes };
property robotAssumptions : RobotAssumption[*|1] { ordered composes };
property elementAssumptions : ElementAssumption[*|1] { ordered composes };
property arenaAssumptions : ArenaAssumption[*|1] { ordered composes };
}
}

class ArenaAssumption
{
    property sentence : RWSentence[1] { composes };
}

abstract class BasicItem extends Item;
class NounBI extends BasicItem
{
    attribute noun : GF::Noun[1];
}
class Identifier
{
    attribute identifier : String[?];
}
abstract class CompoundItem extends Item
{
    property item : Item[1];
}
class AdverbCI extends CompoundItem
{
    attribute adverb : GF::Adverb[1];
}
class PrepositionCI extends CompoundItem
{
    property itemphrases : ItemPhrase[*|1] { ordered };
    attribute preposition : GF::Preposition[1];
    attribute conjunctionType : ConjunctionType[1];
}
abstract class Item extends ItemPhrase;
abstract class ElementAssumption;
abstract class RWSentence
{
    property itemphrase : ItemPhrase[1];
}
class RobotSentence extends RobotAssumption
{
    property sentence : RWSentence[1] { composes };
}
class RobotPModel extends RobotAssumption
property pmodel : PhysMod::PModel[1];
property instantiations : PhysMod::Instantiation[*|1] { ordered };}

abstract class Input;
class InputSometimes extends Input {
  property conditions : Conditions[1];
  property sentences : RWSentence[*|1] { ordered !unique };
}
class Conditions {
  property sentences : RWSentence[+|1] { ordered };
  attribute subjunction : GF::Subjunction[1];
}
class InputAlways extends Input {
  property sentences : RWSentence[*|1] { ordered !unique };
}
class InputNever extends Input;
class NoOutput extends Output , RoboWorldIR::OutputIR;
class OutputSometimes extends Output {
  property conditions : Conditions[1];
  property sentences : RWSentence[+|1] { ordered !unique };
}

abstract class Output;
class UnitBI extends BasicItem {
  attribute unit : Unit[1];
}
class FloatLiteralIP extends ItemPhrase {
  attribute value : _::ecore.xml.type::Float[1];
}
class OutputAlways extends Output , RoboWorldIR::OutputIR {
  property sentences : RWSentence[+|1] { ordered };
}
class DiagrammaticOutput extends Output {
  property opd : RoboChart::RCOperation[1];
  property sizetu : RoboChart::RCIntegerExp[1];
  property conditions : Conditions[?];
  attribute timeunit : Unit[1];
}
class ElementSentence extends ElementAssumption {
  property sentence : RWSentence[1] { composes };
}
class ElementPModel extends ElementAssumption
{
    property pmodel : PhysMod::PModel[1];
    property instantiations : PhysMod::Instantiation[*|1] { ordered };
    property name : Item[1] { composes };
}

datatype Unit : 'java.lang.String' { serializable };
enum ConjunctionType { serializable }
{
    literal AND;
    literal OR = 1;
}
C. Complete RoboWorld IR Metamodel

C.1 RoboWorldIR.ecore

```ecore
import ExpressionIR : 'ExpressionIR.ecore#';
import RoboWorldMM : 'RoboWorldMM.ecore#';
import GF : 'RoboWorldMM.ecore#/GF';
import PhysMod : 'RoboWorldMM.ecore#/PhysMod';
import RoboChart : 'RoboWorldMM.ecore#/RoboChart';

package RoboWorldIR : RoboWorldIR = 'http://www.robocalc.circus/
                  RoboWorldIR'

  { 
    class OperationMappingIR 
    { 
      property signature : RoboWorldMM::Signature[1] { composes }; 
      property output : OutputIR[1] { composes }; 
    } 
    class InputEventMappingIR 
    { 
      property input : InputIR[1] { composes }; 
      property name : RoboWorldMM::Identifier[1] { composes }; 
    } 
    class OutputEventMappingIR 
    { 
      property name : RoboWorldMM::Identifier[1] { composes }; 
      property output : OutputIR[1] { composes }; 
    } 
    class RWIntermediateRepresentation 
    { 
```
Chapter C. Complete RoboWorld IR Metamodel

```plaintext
property inputEventMappings : InputEventMappingIR[*|1] { ordered composes };  
property outputEventMappings : OutputEventMappingIR[*|1] { ordered composes };  
property operationMappings : OperationMappingIR[*|1] { ordered composes };  
property variableMappings : VariableMappingIR[*|1] { ordered composes };  
property arena : Arena[1] { composes };  
property robot : Element[1] { composes };  
property elements : Element[*|1] { ordered composes };  
}  
class VariableMappingIR  
{  
  property conditions : ExpressionIR::Constraint[+|1] { ordered composes };  
  property name : RoboWorldMM::Identifier[1] { composes };  
}  
enum Dimension { serializable }  
{  
  literal OneD : '1D' = 1;  
  literal TwoD : '2D' = 2;  
  literal ThreeD : '3D' = 3;  
}  
class Conditions;  
abstract class Element  
{  
  property name : RoboWorldMM::Identifier[1] { composes };  
  property number : NumericProperty[?] { composes };  
  attribute plurality : Plurality[1];  
}  
class ElementPModel extends Element  
{  
  property pmodel : RoboWorldMM::PhysMod::PModel[1] { composes };  
  property instantiations : RoboWorldMM::PhysMod::Instantiation[*|1] { ordered composes };  
}  
class ElementDescription extends Element  
{  
  property components : ElementDescription[*|1] { ordered composes };  
  property properties : ExpressionIR::Constraint[*|1] { ordered composes };  
  property attributes : Attribute[*|1] { ordered composes };  
  property shape : Shape[?] { composes };  
}  
abstract class InputIR;  
class InputSometimesIR extends InputIR  
{  
  property conditions : ExpressionIR::Constraint[+|1] { ordered  
```
C.1 RoboWorldIR.ecore

```eclipselang
70 property communications : Statement[*|1] { ordered composes };
71 }
class InputAlwaysIR extends InputIR
72 {
73 property communications : Statement[+|1] { ordered composes };
74 }
class InputNeverIR extends InputIR;
abstract class OutputIR;
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 }
class Region extends ElementDescription
83 {
84 attribute dimension : Dimension[1];
85 attribute closed : Boolean[1] = 'false';
86 }
class NumericProperty
89 {
90 property properties : ExpressionIR::Constraint[+|1] { ordered composes };
91 }
abstract class Action;
class OutputAlwaysIR extends OutputIR
94 {
95 property statements : Statement[+|1] { ordered composes };
96 }
class NoOutputIR extends OutputIR;
class Attribute
99 {
100 property name : RoboWorldMM::Identifier[1] { composes };
101 property type : Type[1] { composes };
102 }
abstract class Type;
class DiagrammaticOutputIR extends OutputIR
105 {
106 property opd : RoboWorldMM::RoboChart::RCDoperation[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 }
abstract class Shape;
class Box extends Shape
```
Chapter C. Complete RoboWorld IR Metamodel

```plaintext
{
    property xwidth : NumericProperty[?] { composes };
    property ywidth : NumericProperty[?] { composes };
    property zwidth : NumericProperty[?] { composes };
}
class Sphere extends Shape
{
    property radius : NumericProperty[?] { composes };
}
class Cylinder extends Shape
{
    property radius : NumericProperty[?] { composes };
    property depth : NumericProperty[?] { composes };
}
class Assign extends Action
{
    property assignto : ExpressionIR::ItemPhraseIR[1] { composes };
    property value : ExpressionIR::ItemPhraseIR[1] { composes };
}
class Put extends Action
{
    property element : ExpressionIR::ElementReference[1] { composes };
    property into : ExpressionIR::ElementReference[1] { composes };
}
class Take extends Action
{
    property element : ExpressionIR::ElementReference[1] { composes };
    property from : ExpressionIR::ElementReference[1] { composes };
}
class Communicate extends Action
{
    property value : ExpressionIR::ItemPhraseIR[1] { composes };
}
class Statement
{
    property sentence : RWSentence[1] { composes };
    property action : Action[?] { composes };
}
class Arena extends Region
{
    attribute hasFloor : Boolean[1];
    property gradient : NumericProperty[?] { composes };
    attribute hasRain : Boolean[1];
    property windSpeed : NumericProperty[?] { composes };
}
enum Plurality { serializable }
{
    literal SINGULAR : 'SINGULAR';
    literal PLURAL = 1;
    literal UNCOUNTABLE : 'UNCOUNTABLE' = 2;
}
class RWSentence {
    attribute text : String[?];
}
datatype _'Sequence' { serializable };
class _'Tuple' extends Type {
    property types : Type[+|1] { ordered composes };
} class _'Real' extends Type;
class Enumeration extends Type {
    property variants : RoboWorldMM::Identifier[*|1] { ordered composes 
    →};
}
→ RoboWorldMM'
{
    package PhysMod : PhysMod = 'http://www.robocalc.circus/PhysMod'
    {
        class PModel;
        class Instantiation;
    }
→ '
    {
        class RCIntegerExp;
        class RCOperation;
    }
class Identifier;
class Signature;
}

C.2 ExpressionIR.ecore

import RoboWorldIR : 'RoboWorldIR.ecore#';
import RoboWorldMM : 'RoboWorldMM.ecore#';
import _'ecore.xml.type' : 'http://www.eclipse.org/emf/2003/XMLType';
import GF : 'RoboWorldMM.ecore#/GF';
package ExpressionIR : ExpressionIR = 'http://www.robocalc.circus/
→ ExpressionIR'
{
    abstract class BooleanExpression;
    abstract class UnaryBooleanExpression extends BooleanExpression
Chapter C. Complete RoboWorld IR Metamodel

```
  property pred : BooleanExpression[1] { composes };
}
abstract class BinaryBooleanExpression extends BooleanExpression
{
  property leftPred : BooleanExpression[1] { composes };
  property rightPred : BooleanExpression[1] { composes };
}
class NotExpression extends UnaryBooleanExpression;
abstract class QuantifierExpression extends UnaryBooleanExpression
{
  property element : RoboWorldIR::Element[1];
  property name : RoboWorldMM::Identifier[1] { composes };
  property variable : RoboWorldMM::Identifier[1] { composes };
}
class UniversalExpression extends QuantifierExpression;
class ExistentialExpression extends QuantifierExpression;
class AndExpression extends BinaryBooleanExpression;
class OrExpression extends BinaryBooleanExpression;
abstract class ComparisonExpression extends BooleanExpression
{
  property left : ItemPhraseIR[1] { composes };
  property right : ItemPhraseIR[1] { composes };
}
class LessThan extends ComparisonExpression;
class Equal extends ComparisonExpression;
class GreaterThan extends ComparisonExpression;
abstract class Expression;
class Distance extends PrimitiveExpression
{
  property from : ElementReference[1] { composes };
  property to : ElementReference[1] { composes };
}
class Towards extends PrimitiveExpression
{
  property towards : ElementReference[1] { composes };
  property base : ElementReference[1] { composes };
}
abstract class BinaryExpression extends Expression
{
  property right : Expression[1];
  property left : Expression[1];
}
class Multiplication extends BinaryExpression;
class Addition extends BinaryExpression;
class Negation extends Expression
{
  property expression : Expression[1] { composes };
}
class NounBIIR extends BasicItemIR
{
class PrepositionCIIR extends CompoundItemIR {
  property itemphrases : ItemPhraseIR[*|1] { ordered composes }
  attribute preposition : RoboWorldMM::GF::Preposition[1];
  attribute conjunctionType : RoboWorldMM::ConjunctionType[1];
}

abstract class ItemIR extends ItemPhraseIR;

class AdverbCIIR extends CompoundItemIR {
  attribute adverb : RoboWorldMM::GF::Adverb[1];
}

abstract class CompoundItemIR extends ItemIR {
  property item : ItemIR[1] { composes }
}

class QualifiedBIIR extends BasicItemIR {
  property basicitem : BasicItemIR[1] { composes }
  attribute adjective : RoboWorldMM::GF::Adjective[1];
}

class QuantifiedIPIR extends ItemPhraseIR {
  property item : ItemIR[1] { composes }
  attribute number : RoboWorldMM::GF::Quantifier[1];
}

abstract class ItemPhraseIR {
  property expression : Expression[?];
}

abstract class BasicItemIR extends ItemIR;

class DeterminedIPIR extends ItemPhraseIR {
  property item : ItemIR[1] { composes }
  attribute determiner : RoboWorldMM::GF::Determiner[1];
}

class NumericLiteral extends PrimitiveExpression {
  attribute value : _'ecore.xml.type'::Double[1];
}

class EnumLiteral extends PrimitiveExpression {
  property value : RoboWorldMM::Identifier[1] { composes }
}

class TimeSince extends PrimitiveExpression {
  property event : RoboWorldMM::Identifier[?];
}

class Constraint
{  
    property booleanexpression : BooleanExpression(?){composes};  
    property sentence : RoboWorldIR::RWSentence[1]{composes};  
  }

class PronounIPIR extends ItemPhraseIR  
{  
    attribute pronoun : RoboWorldMM::GF::Pronoun[1];  
    property referent : ItemPhraseIR[?]{composes};  
  }

class FloatLiteralIR extends ItemPhraseIR  
{  
    attribute value : _'ecore.xml.type'::Double[1];  
  }

class UnitBIIR extends BasicItemIR  
{  
    attribute unit : RoboWorldMM::Unit[1];  
  }

abstract class PrimitiveExpression extends Expression;  
abstract class EntityFieldExpression extends PrimitiveExpression  
{  
    property elementref : ElementReference[1]{composes};  
    property componentrefs : ElementReference[*|1] {ordered composes ▶};  
  }

class LessThanOrEqual extends ComparisonExpression;  
class GreaterThanOrEqual extends ComparisonExpression;  
class In extends BooleanExpression  
{  
    property set : ItemPhraseIR[1]{composes};  
    property element : ItemPhraseIR[1]{composes};  
  }

class MayExpression extends UnaryBooleanExpression;  
enum ElementProperty {serializable}  
{  
    literal XWIDTH : 'XWIDTH';  
    literal YWIDTH : 'YWIDTH' = 1;  
    literal ZWIDTH : 'ZWIDTH' = 2;  
    literal RADIUS = 3;  
    literal DEPTH = 4;  
    literal SIZE = 5;  
    literal XPOSITION : 'XPOSITION' = 6;  
    literal YPOSITION : 'YPOSITION' = 7;  
    literal ZPOSITION : 'ZPOSITION' = 8;  
    literal POSITION = 9;  
    literal YAW = 10;  
    literal PITCH = 11;  
    literal ROLL = 12;  
    literal VELOCITY = 13;  
    literal ACCELERATION = 14;  
    literal ANGULARVELOCITY = 15;
literal YAWVELOCITY = 16;
literal PITCHVELOCITY = 17;
literal ROLLVELOCITY = 18;
literal ANGULARACCELERATION = 19;
literal YAWACCELERATION = 20;
literal PITCHACCELERATION = 21;
literal ROLLACCELERATION = 22;
literal POSE = 23;
literal ORIENTATION = 24;

abstract class ElementReference
{
  property element : RoboWorldIR::Element[1];
  property name : RoboWorldMM::Identifier[1] { composes };
}
class UniqueElement extends ElementReference;
class SomeElement extends ElementReference
{
  property constraint : Constraint[?] { composes };
  property variable : RoboWorldMM::Identifier[1] { composes };
}
class AllElements extends ElementReference
{
  property constraint : Constraint[?] { composes };
  property variable : RoboWorldMM::Identifier[1] { composes };
}
class PotentialElement extends ElementReference;
class QuantifiedElement extends ElementReference
{
  property variable : RoboWorldMM::Identifier[1] { composes };
}
class AttributeExpression extends EntityFieldExpression
{
  property attribute : RoboWorldIR::Attribute[1];
  property name : RoboWorldMM::Identifier[1] { composes };
}
class PropertyExpression extends EntityFieldExpression
{
  attribute property : ElementProperty[1];
}
class ElementBody extends PrimitiveExpression
{
  property elementref : ElementReference[1] { composes };
}
class ElementSurface extends PrimitiveExpression
{
  property elementref : ElementReference[1] { composes };
}
class ArenaGradient extends PrimitiveExpression;
class ArenaWindSpeed extends PrimitiveExpression;
class Range extends BinaryExpression;
class TupleLiteral extends Expression
{
    property expression : Expression[+|1] { ordered };
}
class SequenceLiteral extends Expression
{
    property expression : Expression[+|1] { ordered composes };
}
class Ground extends PrimitiveExpression;
class VectorLiteral extends PrimitiveExpression
{
    attribute values : _'ecore.xmi.type':Double[2..*|1] { ordered };
}
class On extends BooleanExpression
{
    property on : ItemPhraseIR[1] { composes };
    property object : ItemPhraseIR[1] { composes };
}
class Subset extends ComparisonExpression;
D. Firefighter *CyPhyCircus* Semantics

### D.1 Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>$\mathbb{R} \times \mathbb{R} \times \mathbb{R}$</td>
</tr>
<tr>
<td>Velocity</td>
<td>$\mathbb{R} \times \mathbb{R} \times \mathbb{R}$</td>
</tr>
<tr>
<td>Acceleration</td>
<td>$\mathbb{R} \times \mathbb{R} \times \mathbb{R}$</td>
</tr>
<tr>
<td>Orientation</td>
<td>$\mathbb{R} \times \mathbb{R} \times \mathbb{R}$</td>
</tr>
<tr>
<td>AngularVelocity</td>
<td>$\mathbb{R} \times \mathbb{R} \times \mathbb{R}$</td>
</tr>
<tr>
<td>AngularAcceleration</td>
<td>$\mathbb{R} \times \mathbb{R} \times \mathbb{R}$</td>
</tr>
</tbody>
</table>

**HomeProperty**

- $x\text{width}, y\text{width}, z\text{width} : \mathbb{R}$
- position : Position
- orientation : Orientation
- locations : $\mathbb{P}$ Position

```
locations = boxLocs(position orientation xwidth ywidth zwidth)
```

**ArenaProperty**

- $x\text{width}, y\text{width}, z\text{width} : \mathbb{R}$
- gradient, windSpeed : $\mathbb{R}$
- locations : $\mathbb{P}$ Position
- home : HomeProperty

```
locations = \{ x : 0..xwidth; y : 0..ywidth; z : 0..zwidth \}
```
Chapter D. Firefighter CyPhyCircus Semantics

Tank of waterType ::= full | empty

<table>
<thead>
<tr>
<th>RobotProperty</th>
</tr>
</thead>
<tbody>
<tr>
<td>position : Position</td>
</tr>
<tr>
<td>velocity : Velocity</td>
</tr>
<tr>
<td>acceleration : Acceleration</td>
</tr>
<tr>
<td>orientation : Orientation</td>
</tr>
<tr>
<td>angularVelocity : AngularVelocity</td>
</tr>
<tr>
<td>angularAcceleration : AngularAcceleration</td>
</tr>
<tr>
<td>tank_of_water : Tank_of_waterType</td>
</tr>
<tr>
<td>searchPattern : seq Position</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BuildingProperty</th>
</tr>
</thead>
<tbody>
<tr>
<td>xwidth, ywidth, zwidth : R</td>
</tr>
<tr>
<td>position : Position</td>
</tr>
<tr>
<td>orientation : Orientation</td>
</tr>
<tr>
<td>locations : P Position</td>
</tr>
</tbody>
</table>

locations = boxLocs position orientation xwidth ywidth zwidth

StatusType ::= burning | extinguished

<table>
<thead>
<tr>
<th>FireProperty</th>
</tr>
</thead>
<tbody>
<tr>
<td>xwidth, ywidth, zwidth : R</td>
</tr>
<tr>
<td>position : Position</td>
</tr>
<tr>
<td>orientation : Orientation</td>
</tr>
<tr>
<td>locations : P Position</td>
</tr>
<tr>
<td>status : StatusType</td>
</tr>
</tbody>
</table>

locations = boxLocs position orientation xwidth ywidth zwidth

D.2 Channels

D.2.1 Software channels

InOut ::= in | out
D.2 Channels

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
\textbf{D.3 Global Constants}

\begin{itemize}
  \item \texttt{arena} : \texttt{ArenaProperty}
  \item \texttt{robotInit} : \texttt{RobotProperty}
  \item \texttt{building} : \texttt{BuildingProperty}
  \item \texttt{potentialFires} : \mathbb{P} \texttt{FireProperty}
  \item \texttt{groundLocations} : \mathcal{P}(\mathbb{R} \times \mathbb{R} \times \mathbb{R})
  \item \texttt{groundLocations} = \{ (x, y, z) \in \texttt{arena.locations} \land z = 0 \}
  \item \texttt{arena.xwidth} = 50.0
  \item \texttt{arena.ywidth} = 60.0
  \item \texttt{arena.zwidth} \geq \texttt{building.zwidth} + 1.0
  \item \texttt{arena.gradient} = 0.0
  \item \texttt{arena.windSpeed} < 8.0
  \item \texttt{arena.home.xwidth} = 1.0
  \item \texttt{arena.home.ywidth} = 1.0
  \item \texttt{locsOnLocs arena.home.locations groundLocations} = \texttt{True}
  \item \texttt{arena.home.locations} \subseteq \texttt{arena.locations}
  \item \texttt{robotInit.position} \in \texttt{arena.home.locations}
  \item \texttt{robotInit.position} \in \texttt{arena.locations}
\end{itemize}
D.4 Environment


\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}

\forall \text{fire} : \text{potentialFires} \exists \text{fire} . \text{fire.xwidth} = 0.036
\forall \text{fire} : \text{potentialFires} \exists \text{fire} . \text{fire.ywidth} = 0.0
\forall \text{fire} : \text{potentialFires} \exists \text{fire} . \text{fire.zwidth} = 0.060

\forall \text{fire} : \text{potentialFires} \exists \text{locOnLocs fire.locations groundLocations} = \text{True}
\forall \text{locOnLocs fire.locations building.locations} = \text{True}
\forall \text{fire.position} . 3 \in 5.0 \ldots 18.0
\forall \text{fire} : \text{potentialFires} \exists \text{fire.locations} \subseteq \text{arena.locations}

D.4 Environment

timeStep : \mathbb{R}

\text{process \text{Environment} } \equiv \text{begin}

D.4.1 Environment State
\begin{align*}
\text{EventTimesInit} & \quad \text{EventTimes}' \\
\text{fireDetectedTime}' &= 0.0 \\
\text{fireDetectedOccurred}' &= \text{False} \\
\text{fireLostTime}' &= 0.0 \\
\text{fireLostOccurred}' &= \text{False} \\
\text{criticalTime}' &= 0.0 \\
\text{criticalOccurred}' &= \text{False} \\
\text{landedTime}' &= 0.0 \\
\text{landedOccurred}' &= \text{False} \\
\text{sprayTime}' &= 0.0 \\
\text{sprayOccurred}' &= \text{False} \\
\text{takeOffTime}' &= 0.0 \\
\text{takeOffOccurred}' &= \text{False} \\
\text{goToBuildingTime}' &= 0.0 \\
\text{goToBuildingOccurred}' &= \text{False} \\
\text{searchFireTime}' &= 0.0 \\
\text{searchFireOccurred}' &= \text{False} \\
\text{goHomeTime}' &= 0.0 \\
\text{goHomeOccurred}' &= \text{False}
\end{align*}
**D.4 Environment**

**EnvironmentState**

\[\text{visible robot} : \text{RobotProperty} \\
\text{visible fires} : \text{seqFireProperty} \\
time : \mathbb{R} \\
\text{stepTimer} : \mathbb{R} \]

**EventTimes**

\[\text{state EnvironmentState} \]

**EnvironmentStateInit**

\[\text{EnvironmentState}' \]

\[\text{robot}' = \text{robotInit} \]

\[\text{ranfires}' \subseteq \text{potentialFires} \]

\[\text{time}' = 0.0 \]

\[\text{stepTimer}' = 0.0 \]

**EventTimesInit**

---

**D.4.2 Robot Movement**

**RobotMovement**

\[\Delta \text{EnvironmentState} \]

\[\text{drobot.position } \frac{dt}{d} = \text{robot.velocity} \]

\[\text{drobot.velocity } \frac{dt}{d} = \text{robot.acceleration} \]

\[\text{drobot.acceleration} = 0 \]

\[\text{drobot.orientation} = \text{robot.angularVelocity} \]

\[\text{drobot.angularVelocity} = \text{robot.angularAcceleration} \]

\[\text{dtime} = 1 \]

\[\text{dstepTimer} = 1 \]

**RobotMovementAction \( \cong \) (RobotMovement)**

\[\left( \text{robot.position } \subseteq \text{groundLocations} \land \text{robot.velocity} . 3 < 0 \right) \]

\[\lor \left( \text{robot.position } \subseteq \text{building.locations} \land \left( \text{robot.velocity } \cdot \left( \text{building.position - robot.position} \right) > 0 \right) \right) \]

\[\lor \left( \exists \text{fire} : \text{ranfires} . \left( \text{robot.position } \subseteq \text{fire.locations} \land \left( \text{robot.velocity } \cdot \left( \text{fire.position - robot.position} \right) > 0 \right) \right) \right) \]

\[\lor \left( \text{time } \geq \text{timeStep} \right) \]
D.4.3 Collision Detection

CollisionDetection ≝
  RobotGroundCollision
  □
  RobotBuildingCollision
  □
  RobotFireCollision

\[
\begin{align*}
\text{StopRobot} & \quad \Delta \text{EnvironmentState} \\
\text{robot}'.velocity &= (0, 0, 0) \\
\text{robot}'.acceleration &= (0, 0, 0) \\
\text{robot}'.position &= \text{robot}.position \\
\text{robot}'.orientation &= \text{robot}.orientation \\
\text{robot}'.angularVelocity &= \text{robot}.angularVelocity \\
\text{robot}'.angularAcceleration &= \text{robot}.angularAcceleration \\
\text{robot}'.tank_of_water &= \text{robot}.tank_of_water \\
\text{robot}'.searchPattern &= \text{robot}.searchPattern \\
\text{fires}' &= \text{fires} \\
\text{time}' &= \text{time} \\
\Sigma \text{EventTimes}
\end{align*}
\]

\[
\begin{align*}
\text{RobotGroundCollision} & \triangleq \\
\left( \text{robot}.position.3 = 0 \land \text{robot}.velocity.3 < 0 \right) & \land & \left( \text{StopRobot} \right)
\end{align*}
\]

\[
\begin{align*}
\text{RobotBuildingCollision} & \triangleq \\
\left( \text{robot}.position \in \text{building}.locations \land \left( \text{robot}.velocity \cdot \left( \text{building}.position - \text{robot}.position \right) > 0 \right) \right) & \land & \left( \text{StopRobot} \right)
\end{align*}
\]

\[
\begin{align*}
\text{RobotFireCollision} & \triangleq \\
\left( \exists \text{fire} : \text{ranfires} \bullet \\
\text{robot}.position \in \text{fire}.locations \land \left( \text{robot}.velocity \cdot \left( \text{fire}.position - \text{robot}.position \right) > 0 \right) \right) & \land & \left( \text{StopRobot} \right)
\end{align*}
\]

D.4.4 Communication Actions that occur on the time step

InputTriggers ≝ fireDetected\_InputEventMapping; \.fireLost\_InputEventMapping; \.critical\_InputEventMapping; \.landed\_InputEventMapping
fireDetected InputEventMapping ≜
  if (∃ fire1 : ran Fires • ¬ (norm (fire1.position − robot.position) > 0.5)) ⟷
      fireDetectedTriggered!True
      ⟷ fireDetectedOccurred, fireDetectedTime := True, time
          [] ¬ (∃ fire1 : ran Fires • ¬ (norm (fire1.position − robot.position) > 0.5)) ⟷
          fireDetectedTriggered!False ⟷ Skip fi

fireLost InputEventMapping ≜
  if (∃ fire1 : ran Fires • norm (fire1.position − robot.position) > 0.5) ⟷
      fireLostTriggered!True
      ⟷ fireLostOccurred, fireLostTime := True, time
          [] ¬ (∃ fire1 : ran Fires • norm (fire1.position − robot.position) > 0.5) ⟷
          fireLostTriggered!False ⟷ Skip fi

critical InputEventMapping ≜
  if (sprayOccurred = True ∧ sprayTime ≥ 180.0)
      ∨ (takeOffOccurred = True ∧ takeOffTime ≥ 1200.0) ⟷
          criticalTriggered!True
          ⟷ criticalOccurred, criticalTime := True, time
          [] ¬ (sprayOccurred = True ∧ sprayTime ≥ 180.0)
      ∨ (takeOffOccurred = True ∧ takeOffTime ≥ 1200.0) ⟷
          criticalTriggered!False ⟷ Skip fi

landed InputEventMapping ≜
  if (robot.position.3 = 0.0) ⟷
      landedTriggered!True
      ⟷ landedOccurred, landedTime := True, time
          [] ¬ (robot.position.3 = 0.0) ⟷
          landedTriggered!False ⟷ Skip fi

Communication ≜ Skip

D.4.5 Input Event Buffers

InputEventBuffers ≜ fireDetected_Buffer || fireLost_Buffer
                   || critical_Buffer || landed_Buffer
### Chapter D. Firefighter CyPhyCircus Semantics

**D.4.6 Output Event Buffers**

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

spray_Buffer $\triangleq$ sprayHappened

$\rightarrow$ sprayOccurred, sprayTime := True, time;

spray_Buffer

takeOff_Buffer $\triangleq$ takeOffHappened

$\rightarrow$ takeOffOccurred, takeOffTime := True, time;

takeOff_Buffer

goToBuilding_Buffer $\triangleq$ goToBuildingHappened

$\rightarrow$ goToBuildingOccurred, goToBuildingTime := True, time;

goToBuilding_Buffer

fireDetected_Buffer $\triangleq$ var fireDetectedTrig $\triangleq$ False;

\[
\begin{aligned}
&\text{fireDetectedTriggered?b $\rightarrow$ fireDetectedTrig := b} \\
&\quad \square \ (\text{fireDetectedTrig = True}) \ & \text{fireDetected.in $\rightarrow$ Skip} \\
\end{aligned}
\]

; fireDetected_Buffer

fireLost_Buffer $\triangleq$ var fireLostTrig $\triangleq$ False;

\[
\begin{aligned}
&\text{fireLostTriggered?b $\rightarrow$ fireLostTrig := b} \\
&\quad \square \ (\text{fireLostTrig = True}) \ & \text{fireLost.in $\rightarrow$ Skip} \\
\end{aligned}
\]

; fireLost_Buffer

critical_Buffer $\triangleq$ var criticalTrig $\triangleq$ False;

\[
\begin{aligned}
&\text{criticalTriggered?b $\rightarrow$ criticalTrig := b} \\
&\quad \square \ (\text{criticalTrig = True}) \ & \text{critical.in $\rightarrow$ Skip} \\
\end{aligned}
\]

; critical_Buffer

landed_Buffer $\triangleq$ var landedTrig $\triangleq$ False;

\[
\begin{aligned}
&\text{landedTriggered?b $\rightarrow$ landedTrig := b} \\
&\quad \square \ (\text{landedTrig = True}) \ & \text{landed.in $\rightarrow$ Skip} \\
\end{aligned}
\]

; landed_Buffer
searchFire_Buffer ≡ searchFireHappened
→ searchFireOccurred, searchFireTime := \textbf{True}, time;
searchFire_Buffer

goHome_Buffer ≡ goHomeHappened
→ goHomeOccurred, goHomeTime := \textbf{True}, time;
goHome_Buffer

\textbf{D.4.7 Environment main action}

\begin{align*}
\text{EnvironmentLoop} & \equiv \left( \text{EnvironmentStateInit} \right); \mu X \bullet \\
& \quad \text{RobotMovementAction}; \\
& \quad \left( \\
& \quad \quad \left( \text{time} < \text{timeStep} \right) \& \text{CollisionDetection} \\
& \quad \quad \left( \text{time} \geq \text{timeStep} \right) \& \text{InputTriggers}; \text{Communication} \bigtriangleup \text{time} := 0 \\
& \quad \right); X
\end{align*}

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

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

\bullet (\text{EnvironmentLoop} \mid \text{EnvVars} \mid \text{triggerChannels} \mid \bigcirc \text{InputEventBuffers})
\setminus \text{triggerChannels}

\textbf{end}

\textbf{D.5 Mapping}

\textbf{channel} lock

\textbf{channel} getRobot : RobotProperty
\textbf{channel} setRobot : RobotProperty
D.5.1 spray Output Event Mapping

```
process spray_OutputEventMapping ≜ begin

  sprayMappingDiagram ≜ ⋯

spray_Conditions ≜ var robotTank_of_water : Tank_of_waterType ⋮
  μX ⋮
  getRobotTank_of_water?x : (x ≠ robotTank_of_water)
    → robotTank_of_water := x ⋮ X
  □

  spray?b → X
```
\[ spray_{\text{Semantics}} \triangleq \left( \begin{array}{l}
(spray\text{mappingDiagram} [\varnothing | \{\text{spray}\} | \varnothing] \text{spray\text{Conditions}}) \\
[\varnothing | \{\text{tck}\} | \varnothing] \\
\text{ConvertTocks} \\
[\varnothing | \{\text{getRobot, setRobot}\} | \varnothing] \\
\text{ConvertRobotChannels} \\
\{\text{getRobot, setRobot}\} ; \text{spray\text{Semantics}} \\
\end{array} \right) \]

\[ \square \]

\[ \text{proceed} \rightarrow spray_{\text{Semantics}} \]

\[ \text{ConvertTocks} \triangleq \text{tck} \rightarrow \]

\[ (\text{var} \text{proceedCount} : \mathbb{N} \bullet \text{proceedCount} := 0 ; \mu X \bullet \\
\quad \text{if} \text{proceedCount} < 1.0/\text{timeStep} \rightarrow \text{proceed} \rightarrow X \\
\quad \{\text{proceedCount} \geq 1.0/\text{timeStep} \rightarrow \text{ConvertTocks} \}
\]
\(\text{ConvertRobotChannels} \equiv \text{var} \ \text{robotPos} : \text{Position}; \ \text{robotVel} : \text{Velocity};\)
\(\text{robotAcc} : \text{Acceleration} \bullet \)
\(\text{var} \ \text{robotOri} : \text{Orientation}; \ \text{robotAngVel} : \text{AngularVelocity};\)
\(\text{robotAngAcc} : \text{AngularAcceleration} \bullet \)
\(\text{var} \ \text{robotTank_of_water} : \text{Tank_of_waterType} \bullet \)
\(\text{var} \ \text{robotSearchPattern} : \text{seqPosition} \bullet \)
\(\mu X \bullet \)
\(\text{getRobotPosition?} x : (x \neq \text{robotPos}) \rightarrow \text{robotPos} := x; X \)
\(\square \)
\(\text{getRobotVelocity?} x : (x \neq \text{robotVel}) \rightarrow \text{robotVel} := x; X \)
\(\square \)
\(\text{getRobotAcceleration?} x : (x \neq \text{robotAcc}) \rightarrow \text{robotAcc} := x; X \)
\(\square \)
\(\text{getRobotOrientation?} x : (x \neq \text{robotOri}) \rightarrow \text{robotOri} := x; X \)
\(\square \)
\(\text{getRobotAngularVelocity?} x : (x \neq \text{robotAngVel}) \rightarrow \text{robotAngVel} := x; X \)
\(\square \)
\(\text{getRobotAngularAcceleration?} x : (x \neq \text{robotAngAcc}) \rightarrow \text{robotAngAcc} := x; X \)
\(\square \)
\(\text{getRobotTank_of_water?} x : (x \neq \text{robotTank_of_water}) \rightarrow \text{robotTank_of_water} := x; X \)
\(\square \)
\(\text{getRobotSearchPattern?} x : (x \neq \text{robotSearchPattern}) \rightarrow \text{robotSearchPattern} := x; X \)
\(\square \)
\((\text{var} \ \text{robotProperty} : \text{RobotProperty} \bullet (\text{PackRobotProperty})\); \)
\(\text{getRobot!robotProperty} \rightarrow \text{Skip}; X \)
\(\square \)
\(\text{setRobot?robotProperty} \rightarrow (\text{UnpackRobotProperty}); \)
\(\text{setRobotPosition!robotPos} \rightarrow \)
\(\text{setRobotVelocity!robotVel} \rightarrow \)
\(\text{setRobotAcceleration!robotAcc} \rightarrow \)
\(\text{setRobotOrientation!robotOri} \rightarrow \)
\(\text{setRobotAngularVelocity!robotAngVel} \rightarrow \)
\(\text{setRobotAngularAcceleration!robotAngAcc} \rightarrow \)
\(\text{setRobotTank_of_water!robotTank_of_water} \rightarrow \)
\(\text{setRobotSearchPattern!robotSearchPattern} \rightarrow X \)

\(\text{spray}_\text{Monitor} \equiv \text{spray}? b \rightarrow \text{sprayHappened} \rightarrow \text{Skip} \)

\(\bullet \text{spray}_\text{Semantics} \parallel \parallel \text{spray} \parallel \parallel \text{spray}_\text{Monitor} \)

\(\text{end} \)

**D.5.2 takeoff Operation Mapping**

\(\text{process}_{\text{takeOff OperationMapping}} \equiv \text{begin} \)

\(\text{takeOff}_\text{Semantics} \equiv \mu X \bullet \text{takeOffCall} \rightarrow \text{setRobotVelocity!(0,0,1.0)} \rightarrow X \)
\begin{align*}
take_{\text{Off}} \text{ Monitor} & \triangleq \mu X \bullet take_{\text{Off Call}} \rightarrow take_{\text{Off Happened}} \rightarrow X \\
\bullet take_{\text{Off Semantics}} & \parallel \{ take_{\text{Off Call}} \} \parallel \emptyset take_{\text{Off Monitor}}
\end{align*}

\textbf{D.5.3 \textit{goToBuilding Operation Mapping}}

\textbf{process} \textit{goToBuilding\_OperationMapping} \equiv \textit{begin}

\textit{goToBuilding\_Semantics} \equiv \textit{goToBuildingCall} \\
\quad \rightarrow \textit{getRobotPosition}\text{?robotPos} \rightarrow \textit{getBuildingPosition}\text{?buildingPos} \\
\quad \rightarrow (\textit{setRobotVelocity}!((1.0 \ast ((\text{buildingPos} - \text{robotPos})/\text{norm}(\text{buildingPos} - \text{robotPos})))) \\
\quad \rightarrow \text{Skip})
\quad \text{proceed} \rightarrow \textit{goToBuilding\_Semantics}

\textit{goToBuilding\_Monitor} \equiv \textit{goToBuildingCall} \rightarrow \textit{goToBuildingHappened} \rightarrow \textit{goToBuilding\_Monitor}

\bullet \textit{goToBuilding\_Semantics} \parallel \{ \emptyset \parallel \{ \textit{goToBuildingCall} \} \parallel \emptyset \} \textit{goToBuilding\_Monitor}

\textbf{end}

\textbf{D.5.4 \textit{goHome Operation Mapping}}

\textbf{process} \textit{goHome\_OperationMapping} \equiv \textit{begin}

\textit{goHome\_Semantics} \equiv \textit{goHomeCall} \\
\quad \rightarrow \textit{getRobotPosition}\text{?robotPos} \\
\quad \rightarrow (\textit{setRobotVelocity}!((1.0 \ast ((\text{arena\_home\_position} - \text{robotPos})/\text{norm}(\text{arena\_home\_position} - \text{robotPos})))) \\
\quad \rightarrow \text{Skip})
\quad \text{proceed} \rightarrow \textit{goHome\_Semantics}

\textit{goHome\_Monitor} \equiv \textit{goHomeCall} \rightarrow \textit{goHomeHappened} \rightarrow \textit{goHome\_Monitor}

\bullet \textit{goHome\_Semantics} \parallel \{ \emptyset \parallel \{ \textit{goHomeCall} \} \parallel \emptyset \} \textit{goHome\_Monitor}

\textbf{end}

\textbf{D.5.5 \textit{searchFire Operation Mapping}}

\textbf{process} \textit{searchFire\_OperationMapping} \equiv \textit{begin}

\textit{searchFireDiagram} \equiv \text{...}
searchFire_Semantics ≜

\[
\left( \left( \begin{array}{c}
\text{searchFireDiagram} \\
\{ \emptyset \mid \{ \text{tock} \} \mid \emptyset \}
\end{array} \right) \setminus \{ \text{tock} \} \right) \\
\left( \begin{array}{c}
\text{ConvertTocks} \\
\{ \emptyset \mid \{ \text{getRobot, setRobot} \} \mid \emptyset \}
\end{array} \right) \\
\left( \{ \text{getRobot, setRobot} \} ; \text{searchFire_Semantics} \right)
\]

\[\mathcal{D} \]

\[\text{proceed} \rightarrow \text{searchFire_Semantics} \]

ConvertTocks ≜ tock \rightarrow

\[(\text{var proceedCount : } \mathbb{N} \mid \text{proceedCount} := 0 ; \mu X \mid
\begin{array}{l}
\text{if } \text{proceedCount} < 1.0/\text{timeStep} \rightarrow \text{proceed} \rightarrow X \\
\text{\quad } \parallel \text{proceedCount} \geq 1.0/\text{timeStep} \rightarrow \text{ConvertTocks}
\end{array}
\)]
ConvertRobotChannels ≡ 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
   μX •
     getRobotPosition?x: (x ≠ robotPos) −→ robotPos := x; X
     getRobotVelocity?x: (x ≠ robotVel) −→ robotVel := x; X
     getRobotAcceleration?x: (x ≠ robotAcc) −→ robotAcc := x; X
     getRobotOrientation?x: (x ≠ robotOri) −→ robotOri := x; X
     getRobotAngularVelocity?x: (x ≠ robotAngVel) −→ robotAngVel := x; X
     getRobotAngularAcceleration?x: (x ≠ robotAngAcc) −→ robotAngAcc := x; X
     getRobotSearchPattern?x: (x ≠ robotSearchPattern) −→ robotSearchPattern := x; X
     (var robotProperty : RobotProperty • (PackRobotProperty)) ;
     getRobot!robotProperty −→ Skip) ; X
     • setRobot?robotProperty −→ (UnpackRobotProperty);
       setRobotPosition!robotPos −→
       setRobotVelocity!robotVel −→
       setRobotAcceleration!robotAcc −→
       setRobotOrientation!robotOri −→
       setRobotAngularVelocity!robotAngVel −→
       setRobotAngularAcceleration!robotAngAcc −→
       setRobotTank_of_water!robotTank_of_water −→
       setRobotSearchPattern!robotSearchPattern −→ X

searchFire_Monitor ≡ searchFireCall −→ searchFireHappened −→ Skip

• searchFire_Semantics [ ⊥ | { searchFireCall } ] | ⊥ ] searchFire_Monitor

end

process Mapping ≡ spray_OutputEventMapping || goToBuilding OperationMapping
   || takeOff.OperationMapping || goHome.OperationMapping
   || searchFire.OperationMapping
D.6 Composition


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

process RoboWorldDocument ≜

(Environment \[^\{\text{getSetChannels} \cup \text{eventHappenedChannels}\}^\text{Mapping}\] \[^\text{getSetChannels} \cup \text{eventHappenedChannels}\]
Credits

\LaTeX\ style based on the The Legrand Orange Book Template by Mathias Legrand and Vel from \LaTeX\Templates.com. Licensed under CC BY-NC-SA 3.0.

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

\begin{itemize}
  \item Icon made by Iconnice from www.flaticon.com is licensed by CC 3.0 BY
  \item Icon made by Sarfraz Shoukat from www.flaticon.com is licensed by CC 3.0 BY
  \item Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
  \item Icon made by Dario Ferrando from www.flaticon.com is licensed by CC 3.0 BY
  \item Icon made by Lyolya from www.flaticon.com is licensed by CC 3.0 BY
  \item Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
  \item Icon made by Google from www.flaticon.com is licensed by CC 3.0 BY
  \item Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
  \item Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
\end{itemize}
Chapter D. Firefighter CyPhyCircus Semantics

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
Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
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
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


**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 existing 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>annotateConstraint</td>
<td>53</td>
</tr>
<tr>
<td>annotateStatement</td>
<td>54</td>
</tr>
<tr>
<td>arenaGlobalAssumptions</td>
<td>74</td>
</tr>
<tr>
<td>arenaTypeDefinitions</td>
<td>67</td>
</tr>
<tr>
<td>attributeTypeDefinitions</td>
<td>68</td>
</tr>
<tr>
<td>componentGetSetChannelDefinitions</td>
<td>73</td>
</tr>
<tr>
<td>composeOperationMappings</td>
<td>75</td>
</tr>
<tr>
<td>composeOutputEventMappings</td>
<td>74</td>
</tr>
<tr>
<td>elementDefinitionTypeDefinitions</td>
<td>70</td>
</tr>
<tr>
<td>elementGetSetChannelDefinitions</td>
<td>72</td>
</tr>
<tr>
<td>elementGlobalAssumptions</td>
<td>73, 74</td>
</tr>
<tr>
<td>elementLocationsDefinition</td>
<td>71</td>
</tr>
<tr>
<td>elementSizeParameters</td>
<td>70</td>
</tr>
<tr>
<td>elementTypeDefinitions</td>
<td>69</td>
</tr>
<tr>
<td>environmentProcess</td>
<td>75</td>
</tr>
<tr>
<td>environmentState</td>
<td>76</td>
</tr>
<tr>
<td>eventHappenedChannelDefinitions</td>
<td>71, 72</td>
</tr>
<tr>
<td>eventTriggeredChannelDefinitions</td>
<td>71</td>
</tr>
<tr>
<td>findDimensionalityInfo</td>
<td>51</td>
</tr>
<tr>
<td>inputTrigger</td>
<td>77</td>
</tr>
<tr>
<td>mapArena</td>
<td>50</td>
</tr>
<tr>
<td>mapInput</td>
<td>53</td>
</tr>
<tr>
<td>mapInputEvent</td>
<td>52</td>
</tr>
<tr>
<td>mapInputEvents</td>
<td>52</td>
</tr>
<tr>
<td>mappingProcess</td>
<td>74</td>
</tr>
<tr>
<td>mapRWDoc</td>
<td>50</td>
</tr>
<tr>
<td>operationMappingDefinition</td>
<td>78</td>
</tr>
<tr>
<td>robotElementDefinitionTypeDefinitions</td>
<td>69</td>
</tr>
<tr>
<td>robotGetSetChannelDefinitions</td>
<td>72</td>
</tr>
<tr>
<td>robotTypeDefinitions</td>
<td>68</td>
</tr>
</tbody>
</table>
roboWorldDocument, 66  updateArena, 51

typeDefinitions, 66