RoboWorld: Verification of Robotic Systems with Environment in the Loop

JAMES BAXTER, University of York, UK
GUSTAVO CARVALHO, Universidade Federal de Pernambuco, Brazil
ANA CAVALCANTI, University of York, UK
FRANCISCO RODRIGUES JÚNIOR, Universidade Federal de Pernambuco, Brazil

A robot affects and is affected by its environment, so that typically its behaviour depends on properties of that environment. For verification, we need to formalise those properties. Modelling the environment is very challenging, if not impossible, but we can capture assumptions. Here, we present RoboWorld, a domain-specific controlled natural language with a process algebraic semantics that can be used to define (a) operational requirements, and (b) environment interactions of a robot. RoboWorld is part of the RoboStar framework for verification of robotic systems. In this article, we define RoboWorld's syntax and hybrid semantics, and illustrate its use for capturing operational requirements, for automatic test generation, and for proof. We also present a tool that supports the writing of RoboWorld documents. Since RoboWorld is a controlled natural language, it complements the other RoboStar notations in being accessible to roboticists, while at the same time benefitting from a formal semantics to support rigorous verification (via testing and proof).

CCS Concepts: • **Software and its engineering** → **Formal language definitions**; • **Computer systems organization** → **Robotics**; • **Theory of computation** → *Semantics and reasoning;*

Additional Key Words and Phrases: Model-driven engineering, domain-specific languages, controlled natural languages, hybrid systems, process algebra

ACM Reference format:
James Baxter, Gustavo Carvalho, Ana Cavalcanti, and Francisco Rodrigues Júnior. 2023. RoboWorld: Verification of Robotic Systems with Environment in the Loop. *Form. Asp. Comput.* 35, 4, Article 26 (November 2023), 46 pages.
https://doi.org/10.1145/3625563

1 INTRODUCTION

Recent advances in Engineering and Artificial Intelligence promise a transformative impact on our society, as robots become ubiquitous in homes, offices, and public spaces, to facilitate and enrich our lives. Development of software for robots operating in these complex environments, however,

The work reported here is funded by the Royal Academy of Engineering grant CiET1718/45, UK EPSRC grants EP/M025756/1 and EP/R025479/1, and UKRI TAS Verifiability Node EP/V026801/1. The work is also partially supported by CNPq grant 465614/2014-0, CAPES grant 88887.136410/2017-00, and FACEPE grants APQ-0399-1.03/17 and PRONEX APQ/0388-1.03/14.

Authors' addresses: J. Baxter and A. Cavalcanti, Department of Computer Science, University of York, Deramore Lane, York, YO10 5GH, United Kingdom; e-mails: James.Baxter@york.ac.uk, Ana.Cavalcanti@york.ac.uk; G. Carvalho and F. Rodrigues Jünior, Centro de Informática, Universidade Federal de Pernambuco, Av. Jornalista Aníbal Fernandes, s/n - Cidade Universitária, Recife - PE, 50740-560, Brazil; e-mails: {ghpc, fwrj}@cin.ufpe.br.

0934-5043/2023/11-ART26 $15.00
https://doi.org/10.1145/3625563

is a challenge. Roboticists often have in mind restrictions that must be satisfied for their robots to operate well: they make assumptions about temperature, wind, layout of rooms, and weight of the robot, for example. Rarely, however, these restrictions are recorded precisely or at all. The usual code-centric approach adopted in software development for robotics often leads to tests that take these restrictions into account, but no record beyond the test base, if any, is normally produced.

Model-driven, as opposed to code-centric, software engineering has been advocated for robotics [13]. Many domain-specific languages support modelling and automated generation of code for simulation and deployment. A few have a formal semantics. The RoboStar framework [10] is distinctive in its design and simulation notations with semantics that can be automatically generated.[1] The semantics is provided using *CyPhyCircus*, a state-rich and hybrid version of a process algebra for refinement [20, 37] cast in the **Unifying Theories of Programming** (**UTP**) of Hoare and He [25], and formalised in the theorem prover Isabelle [19]. Based on such semantics, RoboStar can support automatic generation also of test suites and verification by proof.

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

It is possible to define a single model covering the design of the control software and environment assumptions. With a separate model of the control software, however, we can give a platform and environment-independent account of the design, encouraging reuse across applications. Moreover, simpler discrete-time verification techniques are enough to reason about software models, but not adequate for a system model that caters for the continuous behaviour of the platform and environment. In our approach, we define a design model in terms of a set of services that need to be provided by a platform; no assumption is made in the software model about the nature or the realisation of these services. In an additional document, we capture separately and clearly the assumptions that need to be made at the system, not the software, level.

In this article, we present and formalise RoboWorld, a **controlled natural language** (**CNL**) for writing these additional documents that describe operational requirements of a robotic system for use in simulation, test generation, and proof. The RoboWorld requirements cover aspects of the arena (that is, area) in which the robot is expected to work and of the robotic platform. As a CNL, RoboWorld provides an accessible notation to describe operational requirements, and communicate them to stakeholders who must ensure they are met (such as end users). On the other hand, RoboWorld's formal semantics enables its use in rigorous verification techniques, with automatic generation insulating roboticists from the need to deal with the mathematical notations used to describe the semantics when generating tests or even carrying out proofs.

In [11], we have provided an overview of the RoboWorld syntax, semantics, and tool support using a couple of examples. As a general discussion of RoboWorld, [11] does not present the metamodel of RoboWorld, nor details of its realisation as a natural language. The RoboWorld semantics and our approach to its generation via an intermediate representation are also not addressed. Additionally, here, we consider a more interesting and challenging example (a firefighting UAV). Therefore, here we provide a comprehensive definition of RoboWorld: metamodel, grammar, well-formedness conditions, formal semantics, a tool, and an application in automated testing.

Defining a (constructive) model of all but the simplest of environments of a service robot is not feasible due to their highly complex physical properties, and their often dynamic and unpredictable

[1] robostar.cs.york.ac.uk

nature. Simulations, for example, do provide constructive accounts of an environment, but these are specific scenarios, often simplified. Simulations do not reflect the overall operational requirements of a robotic system precisely; this is not the goal of simulations. On the other hand, it is feasible to record assumptions about the environment [7]. RoboWorld supports this practice; the formal semantics of a RoboWorld document can be regarded as a precise model of the environment, but it is highly nondeterministic. Such a non-determinism implicitly describes all valid environments of a robotic system as all those that satisfy the assumptions described in the RoboWorld document.

Natural language processing techniques can be statistical or symbolic [41]. Statistical approaches assume that a large dataset of (raw) text is available, from which techniques, such as, machine learning, extract processing rules by creating models. Differently, symbolic approaches rely on grammars to define rules for analysing and producing valid text; these rules define a 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 the huge variety of possible writing styles. The control imposed by symbolic approaches can make this inference process easier, since we restrict ourselves to a controlled subset of styles. The challenge is to achieve a compromise between naturalness, expressiveness, and control.

RoboWorld is devised as a CNL 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 toautomatically provide a formal semantics for such requirements. Nevertheless, RoboWorld is a natural, expressive and extensible language, even if 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 [24, 27] and Isabelle/UTP [19].

In terms of the semantics, we define an intermediate representation that ensures changes to the concrete syntax do not directly affect 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.

In the next section, we give a detailed account of related work, covering CNLs used in robotics or with a formal semantics. With that we emphasise the unique features of RoboWorld. Background material is presented in Section 3, where we cover RoboStar, *CyPhyCircus*, and the **Grammatical Framework** (**GF**) used to define the concrete syntax of RoboWorld. Section 4 specifies the structure of RoboWorld documents: their abstract syntax via a metamodel, with associated well-formedness conditions. The concrete grammar is defined in Section 5. In Section 6, we describe our intermediate representation for RoboWorld documents. Section 7 formalises the semantics. RoboTool support for RoboWorld is the object of Section 8. Applications to testing and proof are the topic of Section 9. We conclude and discuss future work in Section 10.

2 RELATED WORK

Here, we first position our work with respect to the literature on approaches to consider the environment when developing software systems in general, including control software for robotic systems (Section 2.1). Afterwards, we discuss the adoption of CNLs, especially when used in robotics or with a formal semantics (Section 2.2).

[2]robostar.cs.york.ac.uk/robotool/

2.1 Software System Environments

Taking a view of the environment as the context in which a software operates, we can say that a software system always affects and is affected by the environment. Considering properties of that deployment environment is, therefore, needed during specification, design, development, and verification. For example, in [48], the authors describe an approach for generating environments of Java program fragments from formally specified assumptions and abstractions. Here, the environment is a group of classes. When specifying concurrent systems using a process algebra, such as CSP for example, as mentioned in the very beginning (Chapter 1) of [43], we set up and reason about processes that interact with their environments. Those environments are mechanisms that can be (potentially) represented by other processes. Here, since our focus is on robotics, we consider the literature related to the physical environments of cyber-physical systems.

In [22], the authors argue that the modelling activity in the development of software systems should formalise as much as possible of the environment, since analysing the correctness of the system relies on an accurate model of the environment, of the software, and of their interaction. RT-Tester is a tool for automatic test generation, execution, and real-time evaluation [39] that follows this point of view. The expected software behaviour is modelled as state machines, which are also used to describe the system environment. When generating test cases, for instance, the models of the software and the environment are considered together to focus on valid scenarios, where realistic operational requirements are assumed.

In [28], a timed input/output conformance relation (s rtioco$_e$ t) is proposed to relate correct implementations s of a specification t, under the environmental constraints expressed by e. The models of s and t, and even of the environment assumptions e, are all given as timed automata.

Closer to our work is that in [44] by Santos, Carvalho, and Sampaio, where environment restrictions are specified according to a CNL. That work is for cyber-physical systems in general, and the language is inspired by concepts of **linear temporal logic** (**LTL**). The semantics is given in CSP and used as part of a test-generation technique.

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

When modelling robotic systems, some works consider the environment to avoid unrealistic designs. For instance, in [16, 40], implicit assumptions of the environment are to some extent captured by 3D and 2D grid maps. They describe a specific scenario where the designed robots are assumed to work, as opposed to general assumptions that might identify a collection of maps.

In [3], a UML profile is used for designing human-robot collaborative systems. This profile has specific stereotypes to model entities from a scenario that interact with the robot in class and component diagrams. The RoboWorld notion of arena corresponds to that of a layout in [3], but layouts are discrete spaces divided in sections that can be obstructed. In a component diagram, each section is a component, with connections representing adjacency. The component diagram is, therefore, a sort of map. Mathematical models for verification automatically generated use a temporal logic with a notion of discrete time. Differently, RoboWorld has a hybrid semantics which accounts for the continuous nature of space and movement, for example.

In [33], the MontiArcAutomaton language [42] is used for modelling components of robotic systems. In this approach, environment assumptions are specified as LTL properties, using AspectLTL [34], a language whose syntax is similar to that of the SMV model checker. In RoboWorld, at the user discretion, properties of the environment are described in a more natural way, considering a CNL, or referring to diagrams. Therefore, RoboWorld distinguishes itself by its flexibility on specifying general assumptions of the environments where a robot can work.

In [29], the authors acknowledge the importance of modelling the environment of robotic applications; this is done using **Stochastic Hybrid Automata** (**SHA**). Formal analyses are performed using **statistical model checking** (**SMC**) and the UPPAAL model checker [6] to assess how likely the missions of the robotic system are to end in success.

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

2.2 Controlled Natural Languages

As discussed in Section 1, techniques designed to process natural language specifications are commonly based on statistical approaches (that is, model-driven artificial intelligence techniques), where it is assumed that a large dataset of raw text is available to extract processing rules. This is not the reality of the development of robotic systems, where operational requirements are frequently left implicit. Employing statistical approaches trained with text not related to the robotics field would impose challenges to the automatic generation of a formal semantics. Therefore, we devise RoboWorld as a natural language, controlled, yet flexible, whose underlying structure favours automation. In the following paragraphs, we comment on other CNLs, mostly with formal semantics.

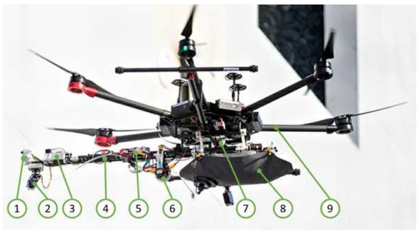
In [38], use cases are used as a source for the generation of a CSP specification. The devised CNL is tailored for mobile applications. In [47], PENG is proposed as a restricted computer-processable CNL for writing unambiguous and precise requirements. The specifications written in PENG can be translated into first-order predicate logic. In [18], requirements are written in a limited standardised format according to a strict if-then sentence template. This enables the translation of requirements into the **Formal Requirement Language** (**FRL**) proposed by the authors. In [46], assuming that the system specification is manually represented conforming to a set of templates, developed for automotive systems, a **Temporal Qualified Expression** (**TQE**) is derived.

More recently, in [32], the authors propose RailCNL: a CNL for the railway domain that was designed as a middle ground between informal regulations and Datalog code. As RoboWorld, this CNL is also implemented using GF. In [23], the authors use a structured natural language (FRETISH) that incorporates previous knowledge from NASA applications and has a **Real-Time Graphical Interval Logic** (**RTGIL**) semantics. The proposed CNL was used to capture and analyse requirements for a Lockheed Martin Cyber–Physical System challenge.

CNLs are also employed by the **Behaviour Driven Development** (**BDDl**) approach that allows the tester or business analyst to create test cases in a simple text language (English). This language helps even non-technical team members to understand what is going on in the software project. For instance, Cucumber[3] is a tool for BDD whose language (Gherkin) uses a set of special keywords to give structure and meaning to executable specifications. For a comprehensive survey of English-based CNLs, we refer to [26], where 100 languages, covering the literature since 1930, are described.

In the robotics domain, previous works have also investigated the use of (controlled) natural languages. In [30], the authors present a form of natural language called system-English (sEnglish) tailored for programming complex robotic systems. In [17], besides presenting a survey of domain-specific languages for robot mission specification, the authors propose PROMISE, a textual and graphical DSL for describing complex multi-robot missions. To the best of our knowledge, there are no CNLs tailored to the description of assumptions about the environment of a robotic application. So, the effort to specify such assumptions using an existing CNL would involve defining

[3]https://cucumber.io/



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

Fig. 1. Firefighting UAV.

from scratch the semantics of concepts specific to the robotics domain, which are pre-defined in RoboWorld. They are identified in the metamodel of RoboWorld, and given a formal semantics. They are also at the core of the intermediate representation we define for RoboWorld documents.

RoboWorld is realised using GF, which has been previously used to define CNLs with formal semantics. In addition to [32], a language for deontic-based specifications for normative systems is presented in [9]. In that work, a semi-automatic way is provided to extract a description using the CNL from free-form texts. The language has a timed-automata semantics suitable for use of UPPAAL for verification [8]. Like we do here, the semantics is defined using a translation into an intermediate XML-like language. Like in our work, syntactic queries to check simple validity constraints can use the intermediate language, while more complex semantic queries use UPPAAL.

3 PRELIMINARIES

In this section, we give an overview of the background material for our work: the RoboStar framework (Section 3.1), *CyPhyCircus* (Section 3.2), and GF (Section 3.3). Regarding language-design approaches and techniques, we refer to [1, 21], which cover background concepts used here such as abstract and concrete syntax, grammars, and metamodels.

3.1 RoboStar Framework

At the design level, a RoboWorld document complements (platform-independent) models of control software. In the RoboStar framework, these are written using RoboChart, a timed state-machine based notation with a specialised component model. Platform independence is achieved by writing models in terms of the services of the robotic platform described by events, operations, and variables. 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 by a challenge for an international robotics competition.[4] We use this later on as a running example to illustrate the structure of RoboWorld documents, the intermediate representation of such documents, and their formal semantics. Figure 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

[4]www.mbzirc.com/ - Challenge 3 in 2020.

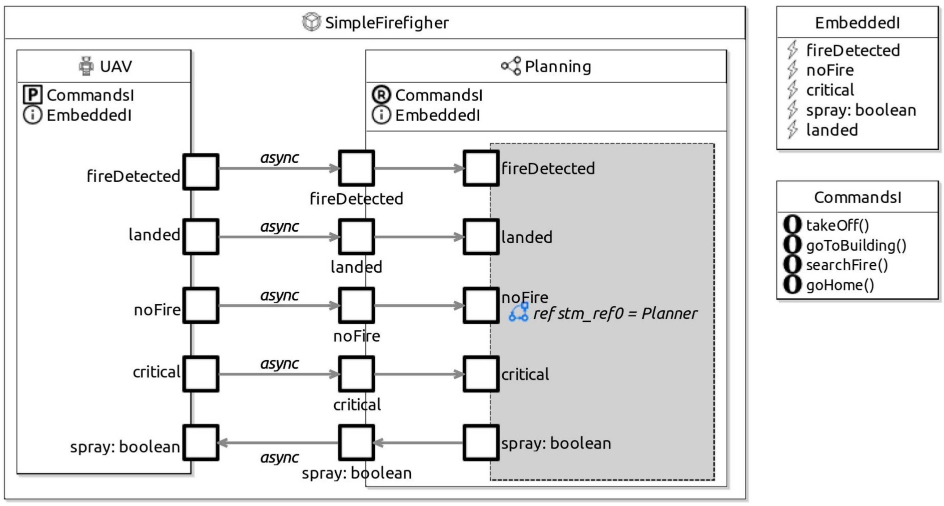


Fig. 2. RoboChart module for a simplified firefighting UAV application.

deadlines. A key element of a RoboChart model is the block that specifies the services of a robotic platform. In Figure 2, this is the block named UAV inside the block SimpleFireFighter.

The model for the real firefighter drone defines 21 services.[5] In our simpler version, we have five events and four operations declared in interface blocks called EmbeddedI and CommandsI on the right in Figure 2. The UAV block declares these interfaces, making their events and operations available to the software. Here, the software behaviour is defined by a controller block Planning. The block SimpleFirefighter is an example of a RoboChart module, used to define a model for the control software of a robotic system, using a robotic-platform, 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 conciseness, we omit its definition. 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.

Like RoboWorld, RoboChart has a process-algebraic semantics based on CSP [43]. It is compatible with the semantics we provide here for RoboWorld, using *CyPhyCircus* [37], described next.

3.2 *CyPhyCircus*

CyPhyCircus is a hybrid process algebra that extends *Circus* [15], which itself combines CSP with Z for modelling abstract data types and operations. Didactic accounts of Z and CSP are available in the literature [45, 49]. The *Circus* syntax is defined in [15]; it is, with just small changes, the

[5] robostar.cs.york.ac.uk/case_studies/firefighting-UAV/index.html

$$\begin{array}{l}\textbf{channel}\ \mathit{fireDetectedTriggered} : \mathbb{B}\\ \textbf{channel}\ \mathit{sprayHappened}\\ \textbf{channel}\ \mathit{getRobotPosition} : \mathit{Position}\\ \textbf{channel}\ \mathit{setRobotTank\_of\_water} : \mathit{Tank\_of\_waterType}\end{array}$$

Fig. 3. Some channels declared in the semantics of the firefighter RoboWorld document.

$$\left|\begin{array}{l}\mathit{arena} : \mathit{ArenaProperty}\\ \mathit{building} : \mathit{BuildingProperty}\end{array}\right.$$

Fig. 4. Some global constants and constraints in the semantics of the firefighter.

syntax of *CyPhyCircus* as well. The only significant difference, from a syntactic point of view, is the availability in *CyPhyCircus* of a Z-like schema that can be used to specify evolution of continuous variables. In terms of semantics, the UTP theories used to define *CyPhyCircus* are presented in [20, 37]. Here we give a brief overview of *CyPhyCircus*.

Like in CSP, *CyPhyCircus* models define mechanisms via processes communicating with each other and their environment via atomic and instantaneous events. Like in *Circus*, however, *CyPhyCircus* processes include a state. 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 via events, but, like in *Circus*, can also define data updates. The new *CyPhyCircus* schemas that define evolution of continuous variables via (differential) equations are an additional mechanism for data update.

We explain the constructs of *CyPhyCircus* as we use them. Our overview of *CyPhyCircus* here is based on excerpts from the semantics of our running RoboWorld example, the firefighter UAV. Here, however, we focus on the structure and meaning of *CyPhyCircus* constructs, and require no knowledge of RoboWorld or its semantics, which are presented in subsequent sections.

Like for Z, a *CyPhyCircus* specification is given by a sequence of paragraphs (that is, definitions). *CyPhyCircus* paragraphs can declare channels, types, global constants, and processes. The type system is that of Z, but we note that we have support for real numbers ($\mathbb{R}$).

The declaration of channels in *CyPhyCircus* is global to processes. Figure 3 shows the declaration of a few of the channels used in the RoboWorld semantics. Channels with a type, such as *fireDetectedTriggered*, define events representing communications of values of that type, such as *fireDetectedTriggered.true*. Typeless channels such as *sprayHappened* define synchronisation points.

Global constants along with constraints on them are declared using the Z notation for axiomatic definitions, indicated by a vertical line on the left. Examples are shown in Figure 4, where we declare two constants *arena* and *building*, giving just their types, omitted here.

Processes can be combined via several operators. In particular, since CSP and *CyPhyCircus* have a common semantic framework, namely, the UTP, we can combine CSP and *CyPhyCircus* processes. For example, we sketch below the process *UAV* that gives semantics to the system whose control software is defined by SimpleFireFighter (see Figure 2) and whose operational requirements are given by a *CyPhyCircus* process *RWDocument*, the semantics of some RoboWorld document.

$$\begin{array}{l}\mathit{UAV} = (\mathit{SimpleFireFighter} \llbracket \{\!|\, \mathit{fireDetected}, \mathit{noFire}, \ldots, \mathit{tock} \,|\!\} \rrbracket \mathit{RWDocument})\\ \qquad\qquad \setminus \{\!|\, \mathit{fireDetected}, \mathit{noFire}, \ldots \mathit{tock} \,|\!\}\end{array}$$

UAV is defined by the parallel composition (CSP operator $\llbracket \ldots \rrbracket$) of the processes *SimpleFireFighter*, which captures the semantics of the RoboChart module, and *RWDocument*.

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

EnvironmentState == [**visible** *robot* : *RobotProperty*; **visible** *fires* : seq *FireProperty*;
time : $\mathbb{R}$; *stepTimer* : $\mathbb{R}$; *tockTimer* : $\mathbb{R}$]

state *EnvironmentState*

RobotMovementAction $\widehat{=}$ $\cdots$

● (*EnvironmentLoop* $\llbracket$ *triggerChannels* $\rrbracket$ *EventBuffers*) $\setminus$ *triggerChannels*

end

Fig. 5. Example of a basic *CyPhyCircus* process called *Environment*.

These processes synchronise on the events in the set $\{\!|$ *fireDetected*, *noFire*, ..., *tock* $|\!\}$, which includes CSP events to represent the services of the platform, as defined in the module, and a special CSP event *tock*. These events, however, are hidden (operator $\setminus$). So, the visible behaviour of *UAV* is described using the events and continuous variables of *RWDocument* representing the elements of the environment of the system (as declared in the RoboWorld document). 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 between the brackets.

The special event *tock* is used in a version of CSP, namely, tock-CSP, to capture discrete time: an occurrence of *tock* represents the passage of one time unit: an abstract notion of time. This is the version of CSP that we use to give semantics to RoboChart. To ensure that there is a single global clock, in the parallelism between a RoboChart process and a RoboWorld process, as in the definition of *UAV* above, we require these processes to synchronise on *tock*. These events are, however, hidden. So, the time model of the composition is that of the *CyPhyCircus* process: a continuous time model. In the *CyPhyCircus* process, based on the definition of the length of a time unit, the *tock* events become available for synchronisation with the tock-CSP process.

A process like *UAV* is defined in terms of other processes. A basic process is defined by a sequence of paragraphs that specify a **state**, a main action, defining its behaviour at the end after a spot (●), and a series of other actions that may be used to define the main action. Differently from a process, an action is local to a process and has access to the state of that process. The main action defines the behaviour of the process in terms of its interactions via events, and by the value of the visible variables of its state. In Figure 5, we present an example of a basic process called *Environment*.

The state of the *Environment* process is given by a Z schema *EnvironmentState*. Some of the state components it declares, namely, *robot* and *fires* 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. *EnvironmentState* also contains encapsulated components: *time* and *stepTimer*. The main action of *Environment* is a parallel composition of actions *EnvironmentLoop* and *EventBuffers*, synchronising on the events in a channel set *triggerChannels*. Communications on these channels are hidden, so that they are internal to *Environment*.

Additional action operators are explained as needed in Section 7.

3.3 Grammatical Framework

The concrete syntax of RoboWorld is defined using GF, which provides support for inflection paradigms (for example, singular and plural forms), as well as agreement between elements of a sentence (for instance, the subject-verb number agreement), for more than 35 languages. There is a notion of module, which may describe an abstract or concrete grammar, but also helper functions. In GF, grammars can be characterised by functions to cater for context-sensitive languages. 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.

We illustrate the main features of GF using a toy version of RoboWorld (called `ToyRoboWorld`), in which we can write clauses about `robots` and `wheels`, using exclusively the verb `to have`.

Example 1. 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.

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

Listing 1. Abstract grammar of ToyRoboWorld.

A concrete grammar of ToyRoboWorld, called `ToyRoboWorldEng` and sketched 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 Line 8, we provide linearisation for a `Determiner` as a record 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.

Next, we define the linearisation of the lexicon of `ToyRoboWorldEng`. This is where we provide English spellings, taking into account inflections. For instance, we provide the singular and plural forms of nouns (Lines 13 and 14). Line 16 illustrates the linearisation of a function. When creating a noun phrase, its number information is inherited from the associated determiner (`n = det.n`). 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

```
concrete ToyRoboWorldEng of ToyRoboWorld = {
------------------------------------------------------------------------------
  -- parameters
  param Number = Sg | Pl ;
  param VerbForm = VPresent Number ;
------------------------------------------------------------------------------
  lincat -- categories
    Determiner = {s : Str ; n : Number} ;
    Noun = {s : Number => Str} ; ...
------------------------------------------------------------------------------
  lin -- lexicon
    robot_Noun = {s = table {Sg => "robot" ; Pl => "robots"}} ;
    wheel_Noun = {s = table {Sg => "wheel" ; Pl => "wheels"}} ; ...
------------------------------------------------------------------------------
  lin -- functions
    mkNounPhrase det noun = {s = det.s ++ (noun.s ! det.n) ; n = det.n} ; ...
}
```

Listing 2. Concrete grammar of ToyRoboWorld.

the inflection form of the noun whose number information is given by `det.n`. We note that `noun.s` is a table, and `!` denotes table (function) application in GF.

GF has a **Resource Grammar Library** (**RGL**), covering 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`), and so on. When a category has a number appended to its name, 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. 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 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 4.1). Next, in Section 4.2, we present a metamodel for RoboWorld. Finally, Section 4.3 lists well-formedness conditions that must be satisfied by a valid RoboWorld document. The structure and, more generally, the metamodel of RoboWorld documents identify the domain-specific concepts that need to be considered in the definition of operational requirements. The well-formedness conditions complement the metamodel. The need to conform to the metamodel and well-formedness conditions provides guidance to designers.

4.1 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 6 and 7. As illustrated, a RoboWorld document includes assumptions and mappings. Assumptions declare and restrict elements of the environment: they are described in Section 4.1.1. The mappings define the services of an associated (RoboChart) design model using the elements defined in the assumptions. We give more details in Section 4.1.2.

4.1.1 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 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.
```

Fig. 6. Firefighter UAV RoboWorld assumptions.

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.

The assumptions in Figure 6 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 6, or if the gradient of the ground is defined. So, in Figure 6, the declaration of the `floor` can be removed.

```
## 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 z-position of the robot is 0.0, the event landed 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.

## 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.
```

Fig. 7. Firefighter UAV RoboWorld mappings.

Two types of entities are declared in Figure 6: 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 an obstacle. For instance, the arena assumptions for a foraging robot may declare obstacles as shown below. Entities are assumed to block movement.

Example 2. 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 6 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,[6] 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 3. 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 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 [14, 35]. 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 article, 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.

4.1.2 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. If there is no associated RoboChart model, or any other model describing the control software in terms of the capabilities of the robot, these sections are not needed. In the presence of such a model, however, they allow us to reason about the software behaviour with the environment in the loop.

In Figure 7, we have mappings for four `INPUT EVENTS`: fireDetected, noFire, landed, and critical. 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, in defining fireDetected and noFire, for instance, we refer to a property distance related to the robot and fires. To define landed we refer to the position of the robot. The event critical is characterised by time conditions, in relation to occurrences, at a previous point in time, 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` declared that the robot has an odometer, and we had an output event resetDist, then the mapping for this event could be as follows. As mentioned, odometer is one of the sensors regarded as a primitive concept in RoboWorld.

[6] robostar.cs.york.ac.uk

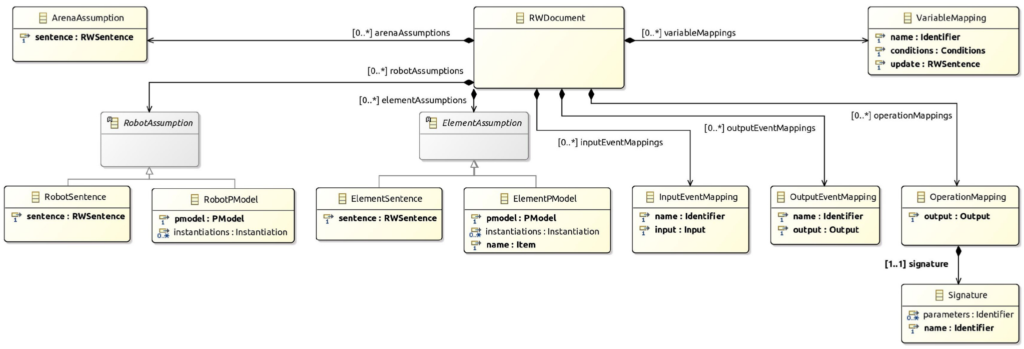


Fig. 8. RoboWorld metamodel: top classes.

Example 4. When the event resetDist occurs, the odometer is reset.

For a drone, we may have an output 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 5. When the event land occurs, the velocity of the robot is set to 1.0 m/s downward.

The mappings in Figure 7 for the operations takeOff, goToBuilding, and goHome are similar.

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 (omitted here but available in [4]).

As for the p-model block diagrams, state machines are available in RoboWorld 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 machine to define it may be simpler than giving an English description. The mapping for the operation searchFire() is specified by a diagram (omitted here but available in [4]). 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. We can also require variables (and constants). As illustrated in Figure 7, if using a diagram to define a mapping, we need to state that and define the value of a time unit, which in RoboChart is left unspecified. In our example, the time unit is `1.0 s`.

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.

4.2 Metamodel

Figure 8 presents a diagram including the top-level classes of the RoboWorld metamodel. A RoboWorld document is an element of the class RWDocument. It is formed by a sequence of zero or more objects of the classes for each of the assumption and mapping groups.

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

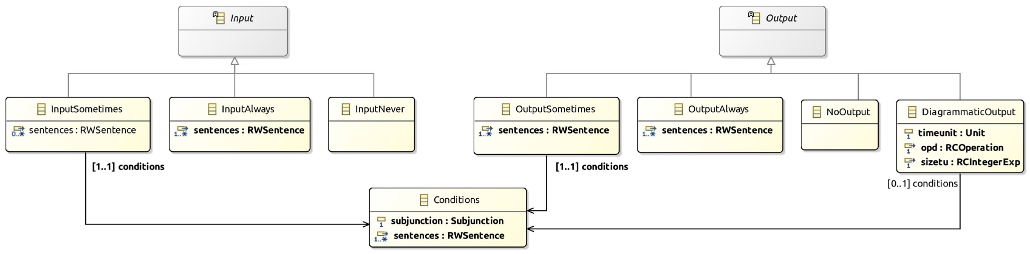


Fig. 9. RoboWorld metamodel: inputs and outputs.

An ArenaAssumption is defined by a sentence. As said, a RobotAssumption can be defined by a sentence, as represented by the subclass RobotSentence, or by a p-model, represented by the class RobotPModel. The attribute pmodel of RobotPModel has type PModel. This is a class in the RoboSim metamodel[7] that represents a specialised form of block 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, the metamodel indicates that the name of the element is an Item as defined in Figure 10: 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 9, we define the class Input as an abstract class with three concrete subclasses: InputSometimes, InputAlways, and InputNever.

In Figure 7, the input mappings for fireDetected, noFire, and landed all represent an element of InputSometimes, with an attribute condition. 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 because the input events in question do not communicate any values. 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, the input mapping for an event with name *angularSpeed* uses an instance of InputAlways as 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 5).

Example 6. **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

[7] robostar.cs.york.ac.uk/publications/techreports/reports/physmod-reference.pdf

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 predefined 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 7. **The event** *transferred* **never happens**.

The InputNever instances are useful for abstraction. An example of where the mapping in Example 7 is useful is provided by one of our case studies[8]: 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 9). 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 4, 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 8. When **the operation** *Transfer()* **is called, nothing happens**.

The use case here is the same as that for the Example 7. 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 DiagrammaticOutput. We refer to Figure 7, 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 is an instance of the class RCOperation from the RoboChart metamodel that defines the value of opd in the instance of DiagrammaticOutput. The value of the time unit 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 set of sentences. Section 5 gives the details; the specification of RWSentence is not domain specific and is not further discussed in this section. As indicated in Figure 10, however, the definition of RWSentence depends on that of an ItemPhrase, which we present in Figure 10 and describe in what follows.

An ItemPhrase identifies an element of the environment; it is a restricted form of noun phrase, which is a general, rather than domain-specific, concept typical of natural languages. ItemPhrase has five direct subclasses. An ItemPhrase can be just a pronoun, represented by an instance of the class PronounIP. Its attribute pronoun is of a type Pronoun. We do not further define classes that correspond directly to general grammatical categories, such as Pronoun, Adverb, and so on. Another 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 ItemPhrases are constructed using a

[8]robostar.cs.york.ac.uk/

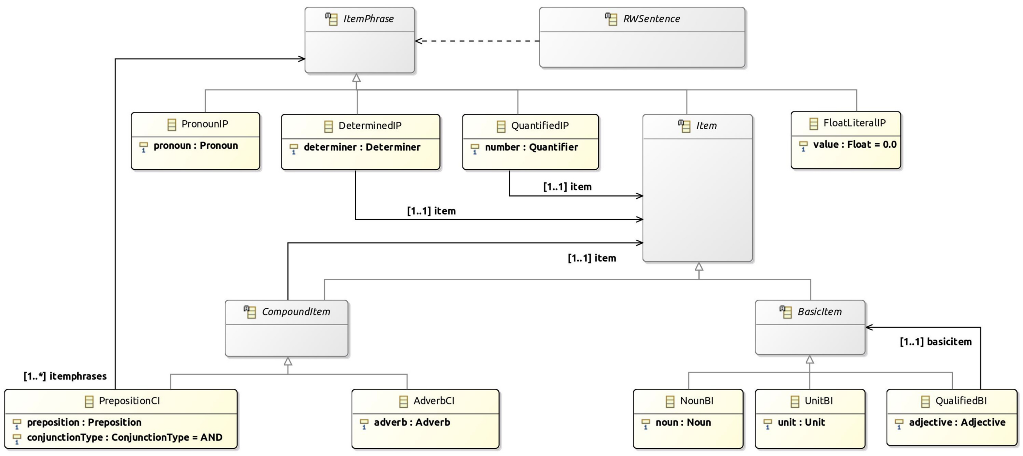


Fig. 10. RoboWorld metamodel: sentences and item phrases.

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

Example 9. A possible pronoun is "`it`". In "`the angular velocity`", we have a determined ItemPhrase created from the determiner "`the`" and the BasicItem "angular velocity". Finally, in "1.0 rad/s upward", we have a quantified ItemPhrase created from number 1 and CompoundItem "`rad/s upward`".

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

Example 10. Examples of BasicItems are "`velocity`", "`angular velocity`", and "`m/s`".

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

Example 11. 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 5, we describe a grammar that justifies the use of English sentences to describe instances of our metamodel. Not every instance represents a valid RoboWorld document, though. So, we now present the well-formedness conditions that must be satisfied.

4.3 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 metres or its prefixes. Time must be

Table 1. Well-formedness Conditions of RoboWorld

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

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 6.

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, if applicable (since RoboWorld can be used in conjunction with other design notations or even on its own). The conditions are presented in Table 1. In the next sections, we present additional well-formedness conditions. In Section 5, we present restrictions related to the vocabulary used in RWSentences. In Section 6, 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 [35, 36].

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 6). 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, if any. The mappings in the RoboWorld document must be for exactly the platform services defined in the RoboChart model. It is those services that define how the robot can perceive and affect the environment. If there is no RoboWorld model in context, there can be no mappings. Similar restrictions can be imposed if other notations are used to specify control software.

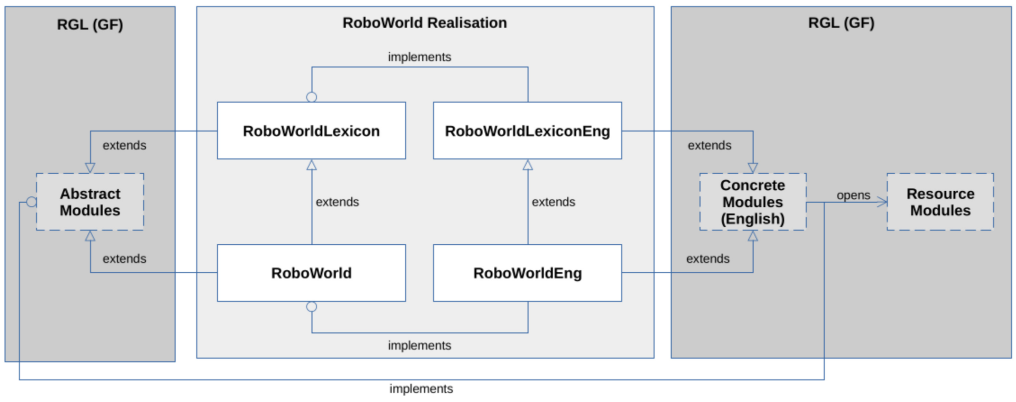


Fig. 11. Architecture of RoboWorld realisation in GF.

The name of a p-model used in a RoboWorld document, if any, 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 ensures 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 ensures that the name used in the RoboWorld document is that in the diagram, but we note that the diagram for an event n, such as spray, is supposed to be nmapping (spraymapping, in our example), to avoid conflict with the name of the event. RW11 ensures that, for operation mappings, the whole signature, not only the name, matches.

5 REALISATION IN THE GRAMMATICAL FRAMEWORK

In the following sections, we detail how the RoboWorld metamodel is realised by grammars in GF. In Section 5.1, we present an overview of our approach, before getting into details in Section 5.2.

5.1 Overview

Figure 11 shows the structure of our realisation of the RoboWorld metamodel in GF. Our approach, which defines an abstract and a concrete syntax, along with the support provided by RGL, means that we have a general mechanisation of RoboWorld that is language independent. Using this structure, we can provide concrete implementations for RoboWorld considering other languages, such as Portuguese, French, and others, without having to implement the metamodel again. RGL takes into account more than 35 languages. Here, we restrict ourselves to English.

In Figure 11, a module is represented as a box, and a collection of RGL modules as a dashed box. The RoboWorld metamodel is realised by the abstract grammar RoboWorld. The concrete grammar RoboWorldEng describes how sentences in English correspond to elements of the metamodel.

The collections of RGL modules used in our realisation of RoboWorld are shown on the left and on the right in Figure 11. RGL is concerned with morphology and syntax rules of languages. The RGL abstract grammars that we use, shown on the left in Figure 11, cover terms such as noun phrases and clauses, for instance, which are common to many languages. On the right, Figure 11 shows RGL modules that implement the abstract modules in the English language.

Table 2. Well-formedness Conditions of the Dictionary

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

In the middle box in Figure 11, 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, its vocabulary. All these grammars use RGL grammars to cater for general concepts. They are all publicly available in [4].

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

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

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

5.2 Lexicon, Item Phrases, Clauses, and Sentences

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 8).

When enriching the dictionary, the well-formedness conditions in Table 2 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.

The realisation of sentences is very general and flexible. As mentioned before, this concept is not domain specific. So, this part of our work, which provides a very extensible mechanism to deal with a wide variety of grammatical structures, can be used in the realisation of other languages.

Generally speaking, sentences in RoboWorld relate ItemPhrases by means of verbs. The realisation of ItemPhrase closely mimics the metamodel in Figure 10. Concretely, BasicItems, CompoundItems, and Items are defined as common nouns, and ItemPhrases as noun phrases. So, the functions in our grammar reflect the metamodel and identify the expected forms of common nouns and noun phrases. For instance, in Listing 3, 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.

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

Listing 3. Excerpts of the RoboWorld grammar: `BasicItem` and `CompoundItem`.

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 10). In the GF-realisation, we expect both adverbs (`Adv`) and adjectives (`A`)—see Listing 3, Lines 6 and 7. In the second case, we use an RGL function to create an adverb from a given adjective. The realisation of `ItemPhrases` 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`.

RoboWorld clauses (defined by the category `RWClause`) are used to define RWSentences; they are instances of RGL clauses, and define the writing structures supported in RoboWorld. There are 12 forms of `RWClause`, each defined by a `mK` function. An `RWClause` can be written in the active voice or in the passive voice. In the active voice, a `mK` function creates `RWClauses` using transitive verbs. There is also support for modal and progressive verbs in the active voice. Additional functions give a special treatment to clauses written using the verb "to be". In the passive voice, we can use intransitive and transitive verbs. The latter expects a preposition followed by an `ItemPhrase`.

The linearisation of the aforementioned functions uses RGL functions to ensure agreement between elements. In Listing 4, we give an example linearisation, along with an example `RWClause` of the form considered. First, a verb phrase (`VP`) named `progressive` is declared. The function `mkVP` creates a verb phrase from the text embedded in the provided verb (`lin V2 v2`) and the second `ItemPhrase` (`itemPhrase2`). A type annotation (`< ... : V2>`) is applied to `lin V2 v2` to ensure the text is cast to the type `V2` (since verbs can have several types). Afterwards, the RGL function `progressiveVP` transforms this verb phrase, taking into account the progressive 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.

```
-- the robot is carrying an object
mkRWClause_ActiveVoice_Progressive_TransitiveVerb_ItemPhrase
itemPhrase1 v2 itemPhrase2 =
  let progressive : VP =
    progressiveVP (mkVP <(lin V2 v2) : V2> itemPhrase2) ;
  in mkCl itemPhrase1 progressive ;
```

Listing 4:== Linearisation of `mkRWClause_ActiveVoice_Progressive_TransitiveVerb_ItemPhrase`.

Example 12. 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`".

RoboWorld sentences are instances of RGL sentences. Here, we deal with verb tenses (present and past) and polarity (positive and negative sentences). Since these possibilities apply to arbitrary `RWClauses`, the 12 different writing structures for clauses discussed above 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, for example, just a single new structure for writing `RWClauses` is added to the language, the number of different types of sentences automatically increases by 8.

The GF realisation of RoboWorld assumptions and mapping definitions closely follows their metamodel in Figures 8 and 9; it is almost a one-to-one relation, with one function in GF for each type of assumption or mapping definition. `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`". Similarly, for the mappings, restrictions enforce the structure to ensure, for example, the presence of the name of the event, operation, or variable mapped.

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.

6 INTERMEDIATE REPRESENTATION

We define the semantics of a RoboWorld document in terms of an **intermediate representation** (**IR**) of that document. With this representation, we insulate the semantics specification presented in the next section from some evolutions of RoboWorld. For example, further case studies are likely to suggest different phrasings for the same meanings, which we may be able to support by extension of the dictionary or of the concrete grammar. With the IR, such extensions, which are important to make the language more flexible, do not affect the semantics definition.

In the IR, information about the arena, the robot, and the other elements is grouped, and structured using notions of expressions and actions, although the original sentences are still recorded. The main new features are classes Constraint and Statement, which record, besides the sentences in the RoboWorld document, additional attributes that 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. The values of these annotation attributes are determined by the rules to generate the IR. Two sets of rules formalise how an IR can be automatically generated for a given RoboWorld document.

In Section 6.1, we present the IR, via the definition of its metamodel and well-formedness conditions. In Section 6.2, 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 1 and 2, it must also be the case that the application of the rules discussed in Section 6.2 to that document generates a valid IR according to the conditions discussed in Section 6.1. Some of the well-formedness conditions are guaranteed by the rules, and some need to be checked.

6.1 Metamodel and Well-formedness Conditions

Figure 12 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 8), 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.

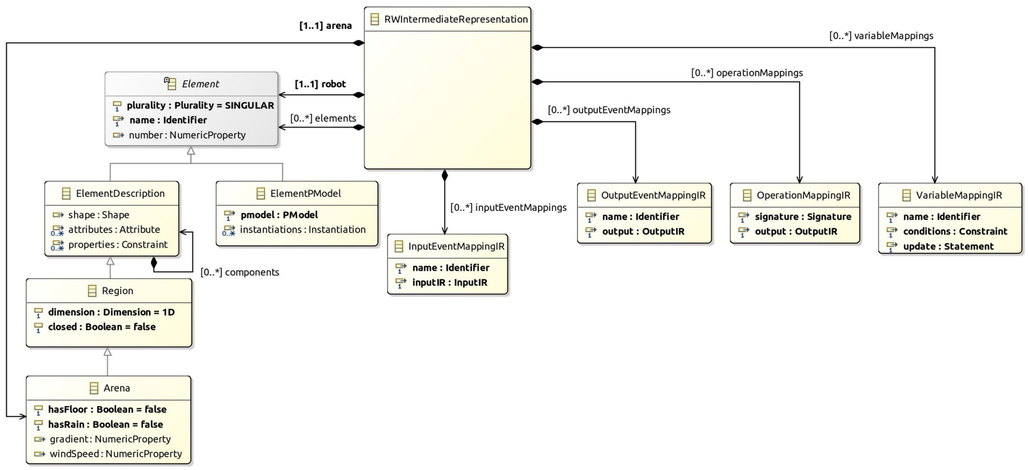


Fig. 12. RoboWorld IR: top classes.

Table 3. 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	An ElementReference to an element whose plurality is SINGULAR must be an instance of UniqueElement.
IR7	An ElementReference to an element whose plurality is PLURAL must not be an instance of UniqueElement.
IR8	An ElementReference to an element whose plurality is UNCOUNTABLE must be an instance of UniqueElement or PotentialElement.
IR9	In an Assign, if the expressions of the assignto and of value are not null, then their types are equal.

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 CNL, 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 3 presents this well-formedness condition (IR1) and others for the IR. Figure 13 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

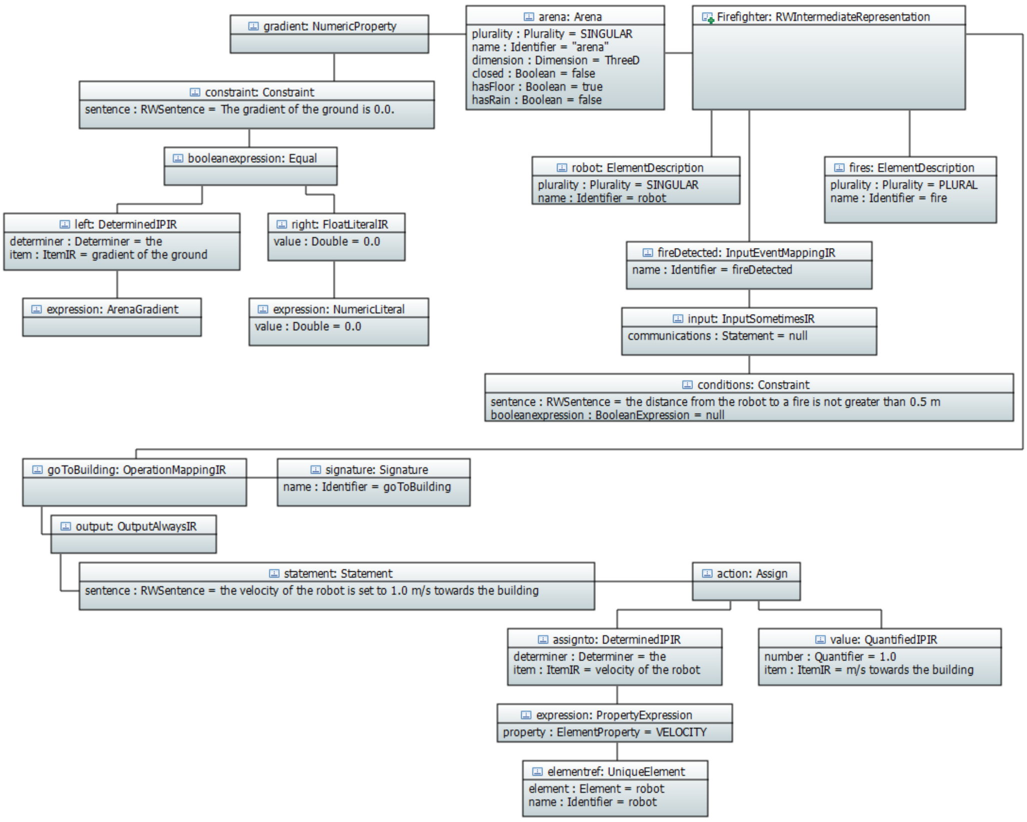

gradient: NumericProperty
arena: Arena
plurality : Plurality = SINGULAR
name : Identifier = "arena"
dimension : Dimension = ThreeD
closed : Boolean = false
hasFloor : Boolean = true
hasRain : Boolean = false
Firefighter: RWIntermediateRepresentation
constraint: Constraint
sentence : RWSentence = The gradient of the ground is 0.0.
booleanexpression: Equal
robot: ElementDescription
plurality : Plurality = SINGULAR
name : Identifier = robot
fires: ElementDescription
plurality : Plurality = PLURAL
name : Identifier = fire
left: DeterminedIPIR
determiner : Determiner = the
item : ItemIR = gradient of the ground
right: FloatLiteralIR
value : Double = 0.0
fireDetected: InputEventMappingIR
name : Identifier = fireDetected
expression: ArenaGradient
expression: NumericLiteral
value : Double = 0.0
input: InputSometimesIR
communications : Statement = null
conditions: Constraint
sentence : RWSentence = the distance from the robot to a fire is not greater than 0.5 m
booleanexpression : BooleanExpression = null
goToBuilding: OperationMappingIR
signature: Signature
name : Identifier = goToBuilding
output: OutputAlwaysIR
statement: Statement
sentence : RWSentence = the velocity of the robot is set to 1.0 m/s towards the building
action: Assign
assignto: DeterminedIPIR
determiner : Determiner = the
item : ItemIR = velocity of the robot
value: QuantifiedIPIR
number : Quantifier = 1.0
item : ItemIR = m/s towards the building
expression: PropertyExpression
property : ElementProperty = VELOCITY
elementref: UniqueElement
element : Element = robot
name : Identifier = robot

Fig. 13. Partial sketch IR for RoboWorld document in Figures 6 and 7.

the IR (see Section 6.2), 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 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 are defined in [4]. 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 [49] for convenience of support for proof.

ElementPModel is similar to the homonymous class in the metamodel (see Figure 8).

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 13 shows the gradient attribute, a NumericProperty characterised by a Constraint. The class NumericProperty has a single attribute property containing one or more Constraints, a class whose definition is shown in Figure 14.

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.

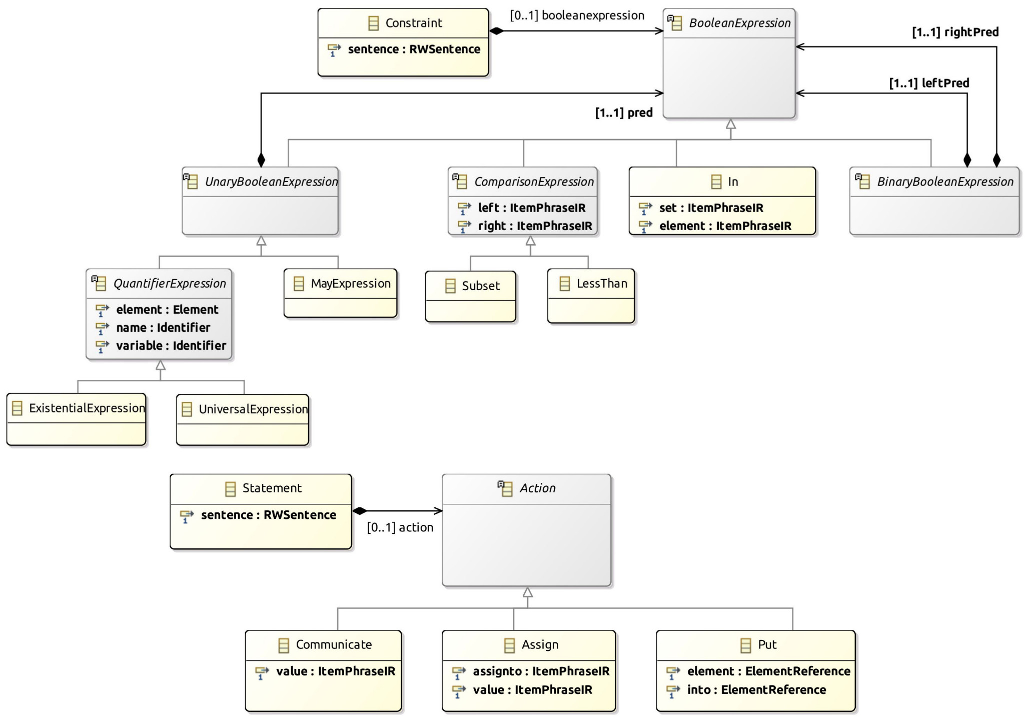


Fig. 14. RoboWorld IR: constraints and statements.

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 8), 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 14, 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 13, we show the IR objects related to the input event fireDetected (see block labelled fireDetected: InputEventMappingIR above the middle right and those connected to it). Similarly, OutputIR subclasses have an attribute statements instead of sentences because they define updates. In Figure 13, we show the IR objects related to the call to the operation goToBuilding, which is recorded as an output (see block labelled goToBuilding: OperationMappingIR below the middle left).

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 13.

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 13, 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 13, the instance of Equal for the booleanexpression 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 14 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 13. It is an Assign, whose assignto attribute is a DeterminerIPIR whose expression is a reference to a property of an element (see Figure 12), 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 15. 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 13 (see block robot: ElementDescription just above that for fireDetected). 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.

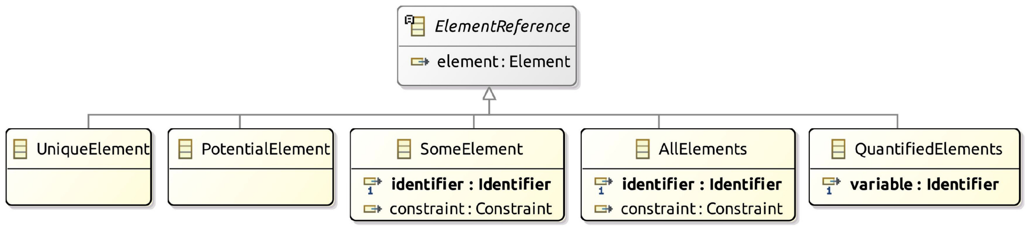


Fig. 15. RoboWorld IR: element references.

Example 13. 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.

6.2 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.

6.2.1 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* $C\{\ldots\}$, where we list, between curly brackets, the value of each attribute. 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 8), produces an instance of RWIntermediateRepresentation (see Figure 12). 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 12) 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.

The map functions are defined by additional rules, in turn defined by using yet more functions. It is the functions concerned with domain-specific concepts that define the structure required in the IR. The function updateArena, for example, is defined by Rule 2; we show 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 refers to the arena dimensionality. In Rule 2, we use these find functions to determine whether the second argument arena of updateArena can be enriched with information from assumption.

Rule 1 (Map RWDocument).

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

Rule 2 (Update Arena).

```
updateArena(assumption : ArenaAssumption, arena : Arena) : Arena =
    if findArenaDimensionInfo(assumption) then arena.dimension = getArenaDimensionInfo(assumption)
    elseif findArenaClosedInfo(assumption) then arena.closed = getArenaClosedInfo(assumption)
    elseif findArenaFloorInfo(assumption) then arena.hasFloor = getArenaFloorInfo(assumption)
    elseif findArenaRainInfo(assumption) then arena.hasRain = getArenaRainInfo(assumption)
    . . .
```

Rule 3 (Map InputSometimes).

```
mapInput(input : InputSometimes) : InputIR =
    new InputSometimesIR {
        conditions = conditions
        communications = communications
    }
where
    conditions = map (λ x ⟶ new Constraint{sentence = x}) input.conditions.sentences
    communications = map (λ x ⟶ new Statement{sentence = x}) input.sentences
```

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. The definitions, however, use general concepts such as “refers to arena”, “dimension adjectives”, and so on. These are domain-specific concepts and the function definitions can be easily extended if new forms of references or adjectives are added to the grammar. This is part of the overall approach to support easy extension of RoboWorld.

Information about mappings is also extracted from the sentences. Rule 3 presents the definition of mapInput for instances input of InputSometimes. An InputSometimes has conditions and sentences, which are recorded as constraints and statements. To provide a concise definition, we rely on the standard map function from functional programming to apply anonymous functions that create *Constraint*s and *Statement*s from the respective sentences.

Rule 4 (Annotate Constraint)**.**

```
annotateConstraint(constraint : Constraint) : Constraint =
    if positiveSentence(constraint.sentence)
      ∧ constraint.sentence.clause instanceof mkRWClause_ActiveVoce_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)
            }
    elseif ...
```

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

6.2.2 Annotating the Intermediate Representation. 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 to enrich the IR via the expression and action attributes of Constraints and Statements. 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.

For illustration, we present the next part of Rule 4, which defines a function annotateConstraint. Specifically, we focus on the fragment that deals with positive sentences that have clauses created with the function mkRWClause_ActiveVoce_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 13. The mk function has two parameters of type ItemPhrase. For example, the first ItemPhrase is "`the gradient of the ground`" and the second is "`0.0`".

Rule 4 annotates the *constraint* by setting its *booleanexpression* to an instance of an *Equal* expression with *left* and *right* attributes for the ItemPhraseIRs 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; to ensure it is created using mkRWClause_ActiveVoce_ToBe_ItemPhrase we use a cast. 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 13, we get a DeterminerIPIR and a FloatLiteralIR for the item phrases in the constraint of the arena.

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

7 SEMANTICS

In this section, we give an overview of the formal semantics of RoboWorld documents (Section 7.1), and present semantic functions that apply to the IR (Section 7.2). Together with

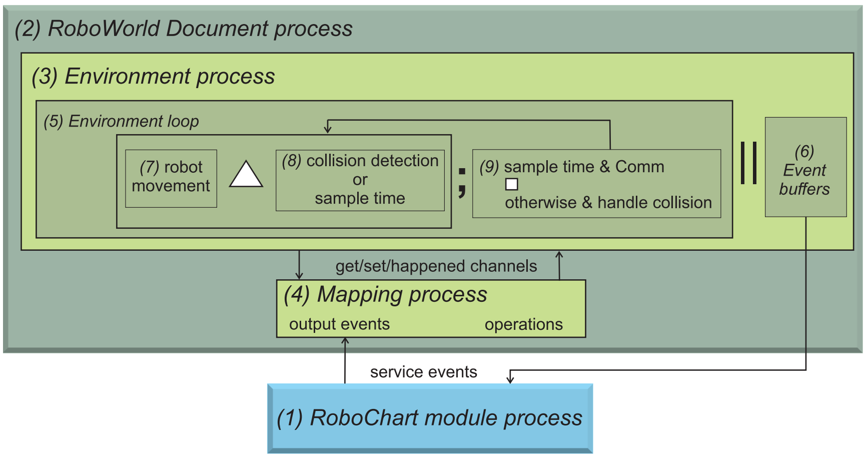

(2) RoboWorld Document process
(3) Environment process
(5) Environment loop
(7) robot movement
△
(8) collision detection or sample time
;
(9) sample time & Comm □ otherwise & handle collision
||
(6) Event buffers
get/set/happened channels
(4) Mapping process
output events
operations
service events
(1) RoboChart module process

Fig. 16. The structure of the RoboWorld semantics.

the rules presented in Section 6.2, they can be used to generate the semantics of a RoboWorld document automatically.

7.1 Formal Semantics: Overview

The overall structure of the RoboWorld semantics and how it connects with the semantics of RoboChart is indicated in Figure 16. 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 need to compose the RoboWorld process (represented by the box labelled (2) in Figure 16) with the process that defines the semantics of the RoboChart module ((1) in Figure 16). This is indicated at the bottom of Figure 16 and illustrated for the firefighter example in Section 3.2. The RoboWorld process communicates with the RoboChart process using *CyPhyCircus* (and CSP) events representing the services of the robotic platform (service events in Figure 16).

The semantics of a RoboWorld document is a *CyPhyCircus* process comprised of two further processes composed in parallel: an environment process ((3) in Figure 16), which represents the objects in the environment and handles triggering of events, and a mapping process ((4) in Figure 16), which contains the semantics for the output-event and operation mappings.

The environment process is defined by the parallelism (represented by parallel bars in box (3) of Figure 16) of two actions. The structure of its definition is shown in Figure 5 in *CyPhyCircus*. The first action (*EnvironmentLoop* in Figure 5 and box (5) in Figure 16) 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 (box (7) in Figure 16) 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 16) 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 ((8) in Figure 16). After the interruption ((9) in Figure 16), 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 ((6) in Figure 16) and then communicates with the mapping process ((4) in Figure 16) to allow it to get and set the values of state variables, before starting again. If the interruption of robot movement is due to a collision, the collision is handled by an abstract action that just ensures that there is no overlapping between the robot body and any physical element in the environment, and then *EnvironmentLoop* loops back.

$$\textbf{channelset}\ \mathit{getSetChannels} == \{\!|\ \mathit{getRobotPosition}, \mathit{getRobotVelocity}, \ldots\ |\!\}$$
$$\textbf{channelset}\ \mathit{eventHappenedChannels} == \{\!|\ \mathit{sprayHappened}, \mathit{takeOffHappened}, \ldots\ |\!\}$$

$$\begin{array}{l}\textbf{process}\ \mathit{RWDocument} \mathrel{\widehat=} (\mathit{Environment} \llbracket \mathit{getSetChannels} \cup \mathit{eventHappenedChannels} \cup \{\mathit{proceed}\} \rrbracket\ \mathit{Mapping}) \\ \qquad\qquad \setminus \mathit{getSetChannels} \cup \mathit{eventHappenedChannels} \cup \{\mathit{proceed}\}\end{array}$$

Fig. 17. The *RWDocument* process for the firefighter example.

$$\begin{array}{|l} \hline \mathit{ArenaProperty} \\ \mathit{xwidth}, \mathit{ywidth}, \mathit{zwidth} : \mathbb{R} \\ \mathit{windSpeed} : \mathbb{R} \\ \mathit{locations} : \mathbb{P}\ \mathit{Position} \\ \mathit{ground} : \mathit{GroundProperty} \\ \mathit{home} : \mathit{HomeProperty} \\ \hline \mathit{locations} = \{x : 0.0 \mathrel{..} \mathit{xwidth};\ y : 0.0 \mathrel{..} \mathit{ywidth};\ z : 0.0 \mathrel{..} \mathit{zwidth}\} \\ \mathit{ground}.\mathit{xwidth} = \mathit{xwidth} \land \mathit{ground}.\mathit{ywidth} = \mathit{ywidth} \\ \hline \end{array}$$

Fig. 18. Example of a type declared in the semantics of the firefighter RoboWorld document.

The action *EventBuffers* ((6) in Figure 16) 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 ((1) in Figure 16). A buffer for an output event records the time in which it last happened. It takes that information from the mapping process ((4) in Figure 16) via a happened channel. Buffering the inputs and outputs allows the evolution of the environment in robot movement ((7) in Figure 16) to proceed independently from their communication to the RoboChart process ((1) in Figure 16), directly in the case of input events, or indirectly via the mapping process, in the case of output events.

The mapping process ((4) in Figure 16) is defined by the interleaving of processes that accept output events and operation calls from the RoboChart process ((1) in Figure 16), and pass on the relevant information to the environment process. These mapping processes capture each of the mapping definitions in the RoboWorld document.

Figures 17–20 sketch the semantics for the firefighter document presented in Figures 6 and 7. Figure 17 shows the definition of the overall *RWDocument* process that captures the semantics of the whole document. As already said, it is defined by a parallel composition ($\llbracket \ldots \rrbracket$) of processes *Environment* (see Figures 5 and 19) and *Mapping* (see Figure 20). The union of the sets *getSetChannels*, *eventHappenedChannels* and $\{\mathit{proceed}\}$, indicated between the $\llbracket$ and $\rrbracket$ symbols, contains the events that require synchronisation between *Environment* and *Mapping*. The same set is indicated after $\setminus$ to define that the events happen instantaneously and are not visible by the RoboChart process.

Figure 17 also sketches the definitions of the sets *getSetChannels* and *eventHappenedChannels*. As their names indicate, these are events for communication with the *Mapping* process (*getSetChannels*) and with the buffers (*eventHappenedChannels*) as sketched in Figure 16. The channel *proceed* is just a signal, that is, it does not communicate any values. It is used by the process *Mapping* to indicate to the process *Environment* that it can proceed with the loop (see Figure 16) 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 18 shows the type *ArenaProperty* used to record properties of the arena. The complete model is available in [4].

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 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 *ground* recording its properties as a record of type *GroundProperty* (omitted here). We always record the *windspeed*, but use closed and hasRain from arena to define the action that models the movement of the robot ((7) in Figure 16).

The component *locations* is a set (specified in Z by $\mathbb{P}$) of *Position*s, 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.

Global constants along with constraints on them capture environment assumptions; examples are shown in Figure 4. We declare global constants for the properties of the static elements of the environment; their types are *Property* records. The *arena*, for example, is unique and static, so its constant captures the values for its properties as a record of the type *ArenaProperty* defined above. Similarly, the type *BuildingProperty* of the *building* constant capture properties arising from the assumptions on the building. 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 5, and some of its actions, not included in Figure 5, are sketched in Figure 19.

We recall that the *Environment* main action, defining its behaviour, is the parallel composition of actions *EnvironmentLoop* and *EventBuffers* ((3) in Figure 16). *EnvironmentLoop*, shown in Figure 19 first initialises the state using another action *EnvironmentStateInit*, and then enters a loop, defined by a recursion that introduces a local name X ($\mu\, X$). *EnvironmentStateInit*, omitted here, is a data operation that specifies the initial value of the encapsulated state components using the global constants. In the body of the recursion, *EnvironmentLoop* performs *RobotMovementAction*. Afterwards, *EnvironmentLoop* makes a choice ($\Box$) based on whether $stepTimer < timeStep$ or not, that is, on the reason for interrupting *RobotMovementAction* ((5) in Figure 16), and then recurses (X).

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. As shown in Figure 19, the *robot*'s *position* evolves with a derivative equal to its *velocity*; other equations are omitted. Timers (*time*, *stepTimer*, and *tockTimer*) evolve with a derivative of 1, so that they keep track of the time in the environment. Every component in *EnvironmentState* not mentioned in the equations of *RobotMovement*, including the discrete components, remains the same throughout the evolution.

In *RobotMovementAction*, *RobotMovement* is interrupted ($\triangle$) by the detection of a collision or the *stepTimer* reaching the *timeStep*. The interruption condition is a disjunction covering four cases, two of which are shown in Figure 19. The first three cases are about the robot colliding with the ground, the building, or a fire. In each case, a collision is detected if the *robot*'s *position*

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RobotMovement}} \\
\Lambda \textit{EnvironmentState} \\
\hline
\frac{\mathrm{d}\textit{robot.position}}{\mathrm{d}t} = \textit{robot.velocity} \ldots \\
\frac{\mathrm{d}\textit{time}}{\mathrm{d}t} = 1 \land \frac{\mathrm{d}\textit{stepTimer}}{\mathrm{d}t} = 1 \land \frac{\mathrm{d}\textit{tockTimer}}{\mathrm{d}t} = 1 \\
\hline
\end{array}$$

$$\begin{array}{l}
\textit{RobotMovementAction} \mathrel{\widehat=} (\textit{RobotMovement}) \triangle \\
\quad \left(\begin{array}{l} (\textit{robot.position} \in \textit{arena.ground.locations}) \\ \lor \ldots \lor (\textit{stepTimer} \geq \textit{timeStep}) \end{array} \right)
\end{array}$$

$$\textit{InputTriggers} \mathrel{\widehat=} \textit{fireDetected\_InputEventMapping} \mathbin{|||} \textit{noFire\_InputEventMapping} \mathbin{|||} \ldots$$

$$\begin{array}{l}
\textit{fireDetected\_InputEventMapping} \mathrel{\widehat=} \\
\quad \mathbf{if}\,(\exists\, \textit{fire}1 : \operatorname{ran} \textit{fires} \bullet \neg\ (\textit{distance}(\textit{fire}1.\textit{position}, \textit{robot.position}) > 0.5)) \longrightarrow \\
\qquad \textit{fireDetectedTriggered}!\mathbf{True} \longrightarrow \textit{fireDetectedOccurred}, \textit{fireDetectedTimer} := \mathbf{True}, \textit{time} \\
\quad [\!]\ \neg\ (\exists\, \textit{fire}1 : \operatorname{ran} \textit{fires} \bullet \neg\ (\textit{distance}(\textit{fire}1.\textit{position} - \textit{robot.position}) > 0.5)) \longrightarrow \\
\qquad \textit{fireDetectedTriggered}!\mathbf{False} \longrightarrow \mathbf{Skip} \\
\quad \mathbf{fi}
\end{array}$$

$$\textit{Communication} \mathrel{\widehat=} ((\textit{GetRobotPosition} \mathbin{\square} \textit{GetRobotVelocity} \mathbin{\square} \cdots)\ ;\ \textit{Communication}) \mathbin{\square} \textit{proceed} \longrightarrow \mathbf{Skip}$$

$$\textit{CheckTock} \mathrel{\widehat=} \mathbf{if}\ \textit{tockTimer} \geq \textit{tockLength} \longrightarrow \textit{tock} \longrightarrow \textit{tockTimer} := 0.0\ [\!]\ \textit{tockTimer} < \textit{tockLength} \longrightarrow \mathbf{Skip}\ \mathbf{fi}$$

$\ldots$

$$\textit{InputEventBuffers} \mathrel{\widehat=} \textit{fireDetected\_Buffer} \mathbin{|||} \textit{noFire\_Buffer} \mathbin{|||} \textit{critical\_Buffer} \mathbin{|||} \textit{landed\_Buffer}$$

$$\begin{array}{l}
\textit{fireDetected\_Buffer} \mathrel{\widehat=} \mathbf{var}\ \textit{fireDetectedTrig} : \mathbb{B} \bullet \textit{fireDetectedTrig} := \mathbf{False}; \\
\quad \mu X \bullet \left(\begin{array}{l} \textit{fireDetectedTriggered}?b \longrightarrow \textit{fireDetectedTrig} := b \\ \square \\ (\textit{fireDetectedTrig} = \mathbf{True})\ \&\ \textit{fireDetected.in} \longrightarrow \mathbf{Skip} \end{array} \right);\ X
\end{array}$$

$$\textit{OutputEventBuffers} \mathrel{\widehat=} \textit{spray\_Buffer} \mathbin{|||} \textit{takeOff\_Buffer} \mathbin{|||} \textit{goToBuilding\_Buffer} \mathbin{|||} \textit{goHome\_Buffer} \mathbin{|||} \textit{searchFire\_Buffer}$$

$$\textit{takeOff\_Buffer} \mathrel{\widehat=} \textit{takeOffHappened} \longrightarrow \textit{takeOffOccurred}, \textit{takeOffTime} := \mathbf{True}, \textit{time}\ ;\ \textit{takeOff\_Buffer}$$

$$\textit{EventBuffers} \mathrel{\widehat=} \textit{InputEventBuffers} \mathbin{|||} \textit{OutputEventBuffers}$$

$$\begin{array}{l}
\textit{EnvironmentLoop} \mathrel{\widehat=} (\textit{EnvironmentStateInit})\ ;\ \mu X \bullet \\
\quad \textit{RobotMovementAction}; \\
\quad \left(\begin{array}{l} (\textit{stepTimer} < \textit{timeStep})\ \&\ \textit{HandleCollision} \\ \square \\ (\textit{stepTimer} \geq \textit{timeStep})\ \&\ \textit{InputTriggers}\ ;\ \textit{Communication}\ ;\ \textit{CheckTock}\ ;\ \textit{stepTimer} := 0.0 \end{array} \right);\ X
\end{array}$$

Fig. 19. Local actions of the *Environment* process for the firefighter example.

is within the element with which it is colliding. In Figure 19, we consider collision with the ground: *robot.position* belonging to the *arena.ground.locations*. The fourth disjunct of the interruption condition shown in Figure 19 is about the *stepTimer* reaching the *timeStep* ($\textit{stepTimer} \geq \textit{timeStep}$).

In *EnvironmentLoop*, a choice checks if the *stepTimer* has reached *timeStep*. If not, a *HandleCollision* specifies that the robot is to be taken to a position where there is no collision, leaving open how this is to be achieved. If the *timeStep* is reached, trigger conditions for input events are checked in interleaving ($|||$), that is, independently, as defined by *InputTriggers* in Figure 19. The conditions for `fireDetected` are checked by the action in *fireDetected_InputEventMapping* shown in Figure 19.

In *fireDetected_InputEventMapping*, 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**

$$\begin{aligned}
&\mathbf{process}\ \mathit{Mapping} \mathrel{\widehat=} \mathit{spray\_OutputEventMapping} \ [\![\ \{\!|\ \mathit{proceed}\ |\!\} \ ]\!] \ \mathit{takeOff\_OperationMapping} \\
&\qquad [\![\ \{\!|\ \mathit{proceed}\ |\!\} \ ]\!] \ \mathit{goToBuilding\_OperationMapping} \ [\![\ \{\!|\ \mathit{proceed}\ |\!\} \ ]\!] \ \mathit{goHome\_OperationMapping} \\
&\qquad [\![\ \{\!|\ \mathit{proceed}\ |\!\} \ ]\!] \ \mathit{searchFire\_OperationMapping}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{process}\ \mathit{goToBuilding\_OperationMapping} \mathrel{\widehat=} \mathbf{begin} \\[1ex]
&\mathit{goToBuilding\_Semantics} \mathrel{\widehat=} \mathit{goToBuildingCall} \\
&\qquad \longrightarrow \mathit{getRobotPosition?robotPos} \longrightarrow \mathit{getBuildingPosition?buildingPos} \\
&\qquad \longrightarrow (\mathit{setRobotVelocity}!(1.0 * ((\mathit{buildingPos} - \mathit{robotPos}) / \mathit{norm}\,(\mathit{buildingPos} - \mathit{robotPos})))) \longrightarrow \mathbf{Skip}; \\
&\qquad \mathit{proceed} \longrightarrow \mathit{goToBuilding\_Semantics} \\[1ex]
&\mathit{goToBuilding\_Monitor} \mathrel{\widehat=} \mathit{goToBuildingCall} \longrightarrow \mathit{goToBuildingHappened} \longrightarrow \mathit{goToBuilding\_Monitor} \\[1ex]
&\bullet\ \mathit{goToBuilding\_Semantics} \ [\![\ \{\!|\ \mathit{goToBuildingCall}\ |\!\} \ ]\!] \ \mathit{goToBuilding\_Monitor} \\[1ex]
&\mathbf{end}
\end{aligned}$$

Fig. 20. The *Mapping* and *goToBuilding_OperationMapping* processes for the firefighter UAV example.

is signalled through the *fireDetectedTriggered* channel to communicate to *EventBuffers* the occurrence of `fireDetected` (as stated in the RoboWorld mapping—see Figure 7). 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 actions (omitted in Figure 19) to 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 ((4) in Figure 16). When *Mapping* finishes, for the current loop, it signals that via *proceed*. An action *CheckTock* then defines whether to signal to the software process, via a *tock* event, passage of time. This is based on whether a *tockTimer* has reached *tockLength*, the value of the time unit used to specify the software. Finally, 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 19.

Regarding *fireDetected_Buffer*, it initialises the boolean state component for the event, here *fireDetectedTrig*, to **False**, then enters a 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—(1) in Figure 16) 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 19 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 20. For illustration, we show the process for the *goToBuilding* operation, called *goToBuilding_OperationMapping*, also presented in Figure 20.

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

As shown in Figure 20, the *goToBuilding_Semantics* action captures the semantics corresponding to the mapping definition "`when the operation goToBuilding is called, the velocity of the robot is set to 1.0 m/s towards the building`". After *goToBuildingCall*, it obtains from *Environment* the position of the robot (*robotPos*) and of the building (*buildingPos*) via *get* channels. It then sets, via a *set* channel, the velocity of the robot to 1, multiplied by a normalised vector from the *robotPos* to *buildingPos*, representing 1.0 m/s towards the building. When finished setting values, a *_Semantics* action signals the *Environment* to *proceed* and recurses. Since all *Mapping* actions synchronise on *proceed*, the *Environment* proceeds only when all *_Semantics* actions are done.

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

The semantics of a RoboWorld document can be generated automatically. Next, we discuss the formalisation of the semantics, via generative rules that define semantic functions.

7.2 Semantics Generation: Transformation Rules

In this section we give an overview of the rules for generating the semantics of a RoboWorld document from its IR presented in Section 6. The top-level Rule 5 defines the overall semantics as a *CyPhyCircus* section (that is, sequence of definitions). As in Section 6.2, 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 5 defines the semantic function $[\![\_]\!]_{\mathscr{RW}}$ that characterises the *CyPhyCircus* section that includes all definitions needed to specify the top process *RWDocument* (see Figure 17) that captures the behaviours of the robot and environment elements allowed by the assumptions and mappings in a well-formed instance rw of the IR class RWIntermediateRepresentation given as argument.

The definition of Rule 5 uses functions defined by other rules to specify groups of definitions. Their definitions are omitted here, but are available in [4]. The first function, 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 getSetChannelDefintions. Finally, *proceed* is declared. Each function takes as argument the attributes of rw that contain the relevant information.

The constraints on elements are defined by an application of entityGlobalAssumptions. The declaration of *timeStep* is in the body of Rule 5 directly. It is followed by the definitions of the process *Environment*, of *Mapping* processes and of *Mapping* itself, and finally *RWDocument*. Each of these processes is characterised using further functions.

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

Rule 5 (Semantics of RoboWorld Documents)**.**

```
[[rw : RWIntermediateRepresentation]]_RW =
    typeDefinitions(rw.arena, rw.robot, rw.elements)
    eventTriggeredChannelDefinitions(rw.inputEventMappings, rw.variableMappings)
    eventHappenedChannelDefinitions(rw.outputEventMappings, rw.operationMappings)
    getSetChannelDefinitions(rw.robot, rw.elements)
    channel proceed
    entityGlobalAssumptions(rw.arena, rw.robot, rw.elements)
    | timeStep : ℝ
    process Environment ≙ environmentProcess(rw)
    for output in rw.outputEventMappings do
        outputMappingDefinition(output)
    endfor
    for operation in rw.operationMappings do
        operationMappingDefinition(operation)
    endfor
    process Mapping ≙ mappingProcess(rw.outputEventMappings, rw.operationMappings)
    process RWDocument ≙ (Environment [| communicationEvents |] Mapping) \ communicationEvents
where
    communicationEvents =
        getSetEvents(rw.robot, rw.elements)
        ∪ eventHappenedSignals(rw.outputEventMappings, rw.operationMappings)
        ∪ {proceed}
```

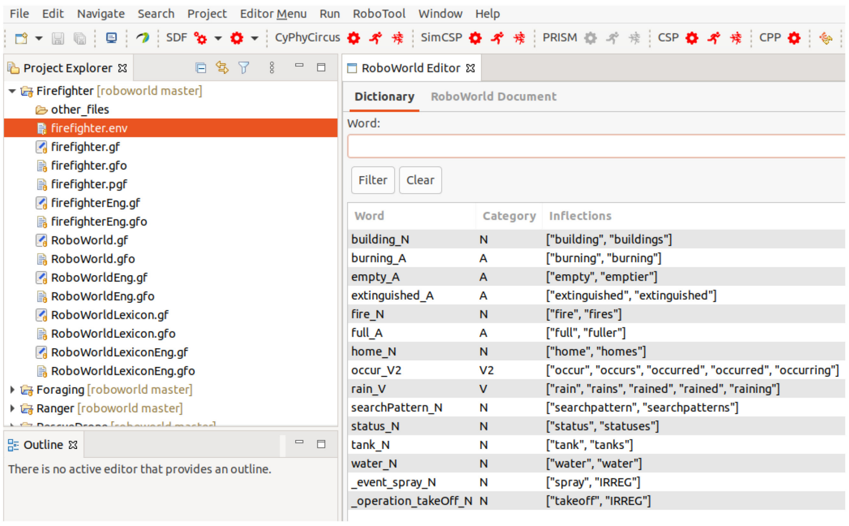
Finally, the *RWDocument* process is defined as the parallel composition of *Environment* and *Mapping*. The synchronisation set, communicationEvents, is defined in the where clause as the union of three sets: the get and set channels, defined by getSetEvents, the signals that output events have happened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

We recall that definitions omitted here are in [4]. The rules are not complex and basically define a *CyPhyCircus* semantics using the approach explained in the previous section. They, however, make RoboWorld a formal notation, and enable automated reasoning, as illustrated in what follows. Mechanisation of the rules and their use to calculate the semantics or a range of examples provides evidence of the appropriateness of the rule set. Use of the semantics improves confidence.

8 ROBOTOOL

Tool support for authoring RoboWorld documents is provided by a specific plug-in for RoboTool. In this way, the support for RoboWorld is integrated with the other RoboTool plug-ins, providing, for example, support for modelling using RoboChart, for generating tests from RoboChart models, and much more. 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 21, 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 21, 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.


File Edit Navigate Search Project Editor Menu Run RoboTool Window Help
SDF CyPhyCircus SimCSP PRISM CSP CPP
Project Explorer
RoboWorld Editor
Dictionary
RoboWorld Document
Word:
Filter
Clear
Outline
There is no active editor that provides an outline.

Firefighter [roboworld master]
- other_files
- firefighter.env
- firefighter.gf
- firefighter.gfo
- firefighter.pgf
- firefighterEng.gf
- firefighterEng.gfo
- RoboWorld.gf
- RoboWorld.gfo
- RoboWorldEng.gf
- RoboWorldEng.gfo
- RoboWorldLexicon.gf
- RoboWorldLexicon.gfo
- RoboWorldLexiconEng.gf
- RoboWorldLexiconEng.gfo

Foraging [roboworld master]
Ranger [roboworld master]

Word	Category	Inflections
building_N	N	["building", "buildings"]
burning_A	A	["burning", "burning"]
empty_A	A	["empty", "emptier"]
extinguished_A	A	["extinguished", "extinguished"]
fire_N	N	["fire", "fires"]
full_A	A	["full", "fuller"]
home_N	N	["home", "homes"]
occur_V2	V2	["occur", "occurs", "occurred", "occurred", "occurring"]
rain_V	V	["rain", "rains", "rained", "rained", "raining"]
searchPattern_N	N	["searchpattern", "searchpatterns"]
status_N	N	["status", "statuses"]
tank_N	N	["tank", "tanks"]
water_N	N	["water", "water"]
_event_spray_N	N	["spray", "IRREG"]
_operation_takeOff_N	N	["takeoff", "IRREG"]

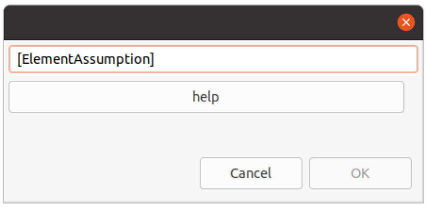
Fig. 21. RoboWorld plug-in in RoboTool: dictionary editor.

Whenever a new word is added to the dictionary, the plug-in automatically extends the RoboWorld lexicon for this project and recompiles all related grammars. The user does not need to understand the underlying details or GF at all. Nevertheless, as we can see in the left-side of Figure 21, the underlying grammars (that is, `.gf` files) are listed so that advanced users can still inspect them.

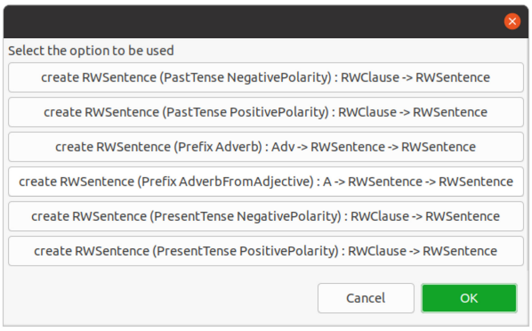
According to [31], there are two predominant paradigms when writing sentences to adhere to a CNL: structural and surface editing. In structural editing, the user mostly follows a structural approach (for instance, clicking on predefined possibilities) that prevents the writing of invalid sentences according to the grammar of the CNL. In surface editing, the user inputs texts with varying degrees of guidance from the editor. In such an approach, it is possible to write sentences that are invalid. Therefore, the validity of the sentences needs to be checked afterwards.

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. On the other side, we can write sentences by selecting the desired structure among those supported (see Figure 22). 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 22, we illustrate our combination of the editing paradigms. Figure 22(a) 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`. Then, a new dialog is 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 22(b) is shown, listing the different ways of creating RWSentences. 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.



(a) New ElementAssumption


Select the option to be used
create RWSentence (PastTense NegativePolarity) : RWClause -> RWSentence
create RWSentence (PastTense PositivePolarity) : RWClause -> RWSentence
create RWSentence (Prefix Adverb) : Adv -> RWSentence -> RWSentence
create RWSentence (Prefix AdverbFromAdjective) : A -> RWSentence -> RWSentence
create RWSentence (PresentTense NegativePolarity) : RWClause -> RWSentence
create RWSentence (PresentTense PositivePolarity) : RWClause -> RWSentence
Cancel
OK

(b) Types of Sentences

Fig. 22. Combination of structural and surface editing.

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.

9 APPLICATIONS

The RoboWorld *CyPhyCircus* semantics allows the assumptions on the environment to be taken into account when performing analysis of the control software. In this section, we give examples of what is possible when we have a RoboChart model describing that software. In this case, there is automation for theorem proving over the composition of the RoboWorld and RoboChart semantics using Isabelle/UTP [19], which has support for *CyPhyCircus*. The assumptions about the environment provided by RoboWorld are also useful in simulation. They form an abstract specification for a simulation scenario, against which a concrete scenario can be checked.

The application we have explored in detail so far, however, is in testing. RoboWorld's design has been primarily motivated by the fact that tests automatically generated from a control software model can define invalid scenarios. We can use the information about the environment in a RoboWorld document to automate the elimination of such tests.

In previous work, using an approach to mutation of RoboChart models, we obtain traces of incorrect behaviour [12], and create tests to guard against these faults based on the test theory in [5]. To illustrate our extension of this approach to take environment assumptions into account, we consider the RoboChart model in Figure 23. It defines control software for a simple rescue drone. We omit the definition of the simple module and controller, but present the state machine (FinderM).

The software is defined in terms of input events found and origin, output events takeoff and land, and operations turnBack() and move(LV). Control begins in the state Off, capturing the initial state when the robot is deactivated. When the robot is switched on, the software proceeds with sending the takeoff event to the robotic platform, causing the robot to take off, and then waits for TOP

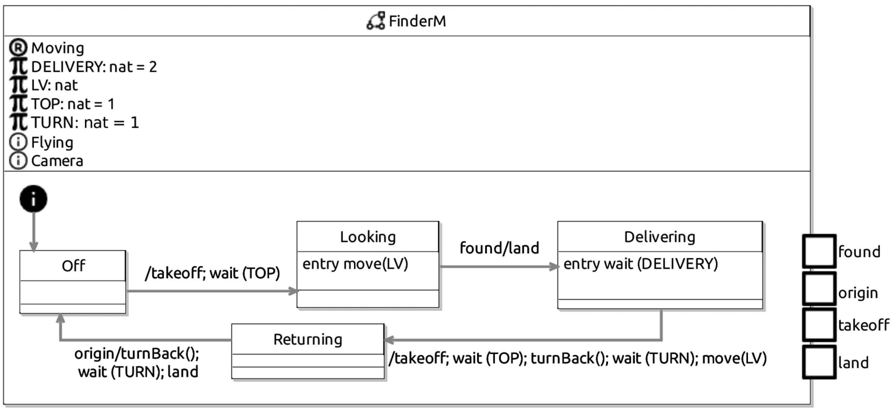


Fig. 23. RoboChart model for a rescue drone.

time units while the robot rises into the air. After that, the call move(LV) causes the robot to start moving towards the target, until the robot's camera signals the detection of the target with the found event, and the robot is signalled to land with the land event. After waiting for DELIVERY time units while the robot lands and delivers supplies to a person in need of rescue, the robot then takes off again (takeoff; wait(TOP)), turns back towards its starting point for TURN time units (turnBack(); wait(TURN)), and starts moving (move(LV)). When the starting point is found, signalled by the origin event, the robot turns back round and is signalled to land (turnBack(); wait(TURN); land).

Our mutation testing approach generated 21 unique traces of incorrect behaviour for this model, but some of them are not interesting because they rely on environments that we are not interested in. For example, the following trace of tock-CSP events is obtained using RoboTool:

$$takeoff.out, tock, moveCall.1, found.in, tock, tock, takeoff.out$$

In this trace, the passage of time is indicated by the *tock* event. Events of the robotic platform are postfixed with *in* or *out* indicating whether they are used as inputs or outputs of the software. The *takeoff* event at the end is the incorrect behaviour to test against. The *found* event occurs here immediately after the *moveCall* event, representing a call to the move(LV) operation. For that, the target must be in the same place as the starting point of the robot; this does not make sense, so a test generated from this trace is not useful. Using RoboWorld, we can state assumptions about the environment that can be used to identify such useless tests.

For our example, a relevant RoboWorld document is shown in Figure 24. The assumptions state that the arena has an `origin` and a `target` region. The robot is stated to begin at the `origin`, and the distance between the `origin` and `target` is required to be at least 1 m. With this, the event found cannot happen as soon as the robot starts moving, since found is linked to the position of the robot in relation to the `target` region by the RoboWorld mapping definitions (omitted here).

We have checked the infeasible trace shown above against a discretised tock-CSP version of the RoboWorld semantics using the model checker FDR [24]. This shows that the trace is indeed infeasible, since it is not a possible trace in any environment that meets the RoboWorld assumptions. This kind of check allows us to automatically exclude such traces from our test generation approach. In total, 13 of the 21 traces generated are excluded by this approach, all for containing a *found* event when the robot has not yet moved off the `origin` and so cannot have found the `target`.

```
## ARENA ASSUMPTIONS ##
The arena is three-dimensional.
The arena has an origin region.
The arena has a target region.
The arena has a floor.
The gradient of the ground under the origin is 0.0.
The gradient of the ground under the target is 0.0.
## ROBOT ASSUMPTIONS ##
Initially the robot is at the origin.
Initially the orientation of the robot is towards the target.
The robot is a point mass.
## ELEMENT ASSUMPTIONS ##
The origin is on the ground.
The origin has an x-width of 1.0 m and a y-width of 1.0 m.
The target is on the ground.
The target has an x-width of 1.0 m and a y-width of 1.0 m.
The distance from the target to the origin is greater than 1.0 m.
```

Fig. 24. RoboWorld assumptions for the rescue drone.

ALGORITHM 1: Test generation approach incorporating RoboWorld

```
1:  procedure TestGeneration(robochart, mutants, roboworld)
2:      feasibleTraces, infeasibleTraces ← {}, {}
3:      while more tests needed do
4:          testTraces ← {}
5:          robochartPlusTraces ← robochart ⊓ ($\bigsqcap$ tr : infeasibleTraces • tr ; RUN(Σ))
6:          for mutant ← mutants do i
7:              refines, counterexample ← CheckRefinement(robochartPlusTraces, mutant)
8:              if (¬ refines) then testTraces ← testTraces ∪ {counterexample} end if
9:          end for
10:         for trace ← testTraces do
11:             isTrace, counterexamplePrefix ← CheckTraceOf(trace, roboworld)
12:             if isTrace then
13:                 feasibleTraces ← feasibleTraces ∪ {trace}
14:             else
15:                 infeasibleTraces ← infeasibleTraces ∪ {counterexamplePrefix}
16:             end if
17:         end for
18:     end while
19:     return feasibleTraces
20: end procedure
```

Since this elimination of infeasible traces can remove a lot of the generated traces, we need to ensure that useful tests can be generated instead. This can be done by following the procedure shown in Algorithm 1. It takes as input (line 1) the RoboChart software model for the robot, *robochart*, for which tests are generated, and a set of *mutants*, which are modified versions of the RoboChart model with errors introduced. The RoboWorld semantics, *roboworld*, is also taken as an input and used to eliminate infeasible tests. Algorithm 1 constructs two sets, which are both initialised to the empty set on line 2: *feasibleTraces*, the feasible test traces output by Algorithm 1, and *infeasibleTraces*, infeasible test traces to be eliminated from the test generation.

The algorithm proceeds in a loop, starting on line 3, with tests generated and infeasible tests eliminated in each iteration. The loop is repeated as many times as needed to generate a sufficient number of feasible tests. In Algorithm 1, we use a condition "more tests needed", which can be simply defined by an application-dependent number of tests to be included in *feasibleTraces*.

Inside the loop, test traces are first generated in a set, *testTraces*, which is initialised to the empty set on line 4. This set is built by comparing the *mutants* to the *robochart* model, similar to what we do in our existing approach to generating traces. In Algorithm 1, however, instead of using the original *robochart* process as a specification for the *mutants*, we use an enriched process *robochartPlusTraces* that composes *robochart* with another process that adds the *infeasibleTraces* we wish to eliminate to the specification (see line 5). With the extended specification, the infeasible traces are no longer counted as incorrect behaviour of the software when generating traces.

Each of the infeasible traces is added to the specification using internal choice ($\sqcap$), so that the composition does not affect the behaviour of *robochart*. In Algorithm 1, *robochartPlusTraces* is defined by an internal choice where one of the choices is *robochart* itself, and the others are determined by the traces *tr* in *infeasibleTraces*. We use an iterated internal choice ($\bigsqcap$) over traces *tr* in *infeasibleTraces* in the description of Algorithm 1, but that is used here as metanotation to specify a process defined using simple $\sqcap$. If *infeasibleTraces* is empty, which is always the case in the first iteration of the loop, then the result is just *robochart* itself. We also use here *tr* as metanotation for the simple process that offers each of the events of *tr* in sequence, before terminating.

In *robochartPlusTraces*, each trace process *tr* is followed with a $RUN(\Sigma)$ process, which repeatedly offers all events in scope. (Σ is the set of all events.) This prevents trivial continuations of the infeasible trace from being generated. For example, regenerating tests with the trace shown above eliminated, but without eliminating continuations of it, yields the following trace:

$$takeoff.out, tock, moveCall.1, found.in, tock, tock, takeoff.out, tock$$

Here, the infeasible trace has been avoided by adding a *tock* event at the end, but the new trace is still infeasible for the same reason as the original infeasible trace is. The erroneous *takeoff* event is also no longer at the end of the trace, although that is the desired form for defining tests.

Each *mutant* in *mutants* is compared to *robochartPlusTraces* in a loop beginning on line 6. The comparison is performed on line 7, by checking if *mutant* refines *robochartPlusTraces*, meaning all the traces of behaviour *mutant* can perform are also traces *robochartPlusTraces* can perform. The refinement check outputs a boolean, *refines*, indicating the result of the check and, if the result is false, a *counterexample* giving a trace of behaviour that *mutant* can perform but *robochartPlusTraces* cannot. If *refines* is false, the resulting *counterexample* is added to *testTraces*, on line 8.

After each of the *mutants* has been checked, in a for loop on line 10, each *trace* in *testTraces* is checked for feasibility. This is done by checking if the *trace* is a trace of behaviour that can be performed by *roboworld*. As with the refinement check, the output of this check is a boolean, *isTrace*, that records the result and, if *isTrace* is false, a *counterexamplePrefix* recording the shortest prefix of *trace* that is not a trace of *roboworld*. If *isTrace* is true then *trace* is feasible and is added to the *feasibleTraces* set, on line 13. If *isTrace* is false, then *trace* is infeasible and *counterexamplePrefix* is added to *infeasibleTraces*. Adding *counterexamplePrefix* rather than *trace* ensures variations of *trace* after the point the infeasibility are also excluded from the generation of test traces. For example, generating traces with the infeasible trace shown earlier eliminated, as well as any continuations of it, as discussed above, produces the following trace:

$$takeoff.out, tock, moveCall.1, found.in, tock, tock, tock, takeoff.out$$

This trace is similar to the infeasible trace, but has *tock* inserted before the end, so that it is not a continuation of the infeasible trace. The infeasible behaviour *found.in* immediately after *moveCall.*1 is, however, still present. To eliminate similar infeasible traces, *counterexamplePrefix* with the events up to and including *found.in* must be eliminated (along with its continuations).

Once *testTraces* have been sorted into *feasibleTraces* and *infeasibleTraces*, the while loop continues. Algorithm 1 returns the *feasibleTraces* set on line 19. It terminates if "more tests needed", based on the increasing size of *feasibleTests* and an upper bound on the number of iterations, for example. It is also linear on the number of mutants. The bottleneck is the refinement and trace checks.

The mutants for the rescue drone example generate 13 infeasible traces, leaving just eight feasible traces. This forms the output from the first iteration of the loop. In the checking on the second iteration eight new feasible traces are generated, in addition to the eight existing feasible traces, but 12 infeasible traces are also generated. Among the new feasible traces generated in the second iteration is a feasible variant of the trace considered above:

$$takeoff.out, tock, moveCall.1, tock, found.in, tock, tock, takeoff.out$$

This is feasible, since it allows time for the drone to move towards the target area after *moveCall*.1, before it finds the target area with *found.in*. Of the 12 infeasible traces generated in the second iteration, nine are formed by placing *tock* at the start of an infeasible trace. The remaining three traces are infeasible due to detecting the origin while still over the target region, the dual case of the infeasibility observed earlier. Running a third iteration of the loop generates a further three feasible traces and four new infeasible traces. This brings the total number of feasible traces generated to 19, so that we have feasible traces for almost all of the original 21 traces generated.

As said, the work above is supported by a discretisation of our semantics. This discretisation scales poorly for larger environments, since it involves a lot of computation to represent the evolution of variables, which can cause a state explosion. An alternative is to use hybrid model checkers, where the continuous evolution of state is given special handling. Many hybrid model checkers are limited in the kinds of models they can support, with some having limited support for non-linear equations or parallel composition of components. We are currently working to implement the RoboWorld semantics in the hybrid model checker CORA [2].

In future work, we will improve automation by integrating model checking with theorem proving so that results from automated model checking can be used to provide facts that form part of proofs. Given the restricted nature of the checks required by Algorithm 1, for example, based on automatically generated processes, we will work to devise proof tactics to achieve automation and scalability in a setting that combines theorem proving and model checking.

10 CONCLUSIONS

We have presented RoboWorld, a CNL for documenting operational requirements of robotic systems. We have described the overall structure of a RoboWorld document using a metamodel, defined using elements of the English grammar, such as Sentence, Noun, and so on. A concrete grammar, defined using the GF, specifies 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. Two sets of transformation rules are used to characterise an intermediate representation for a RoboWorld document, from which other rules define a precise hybrid process-algebraic semantics written in *CyPhyCircus* for that document.

The concrete grammar is very powerful, allowing and enforcing correct use of inflections, for example. The 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. Afterwards, 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 following transformation, from the intermediate representation to a *CyPhyCircus* model. The intermediate representation can also be used as a gateway to consider semantics in several notations.

The design of RoboWorld captures an extensible collection of concepts relevant to the definition of operational requirements for a robotic application. With RoboWorld, roboticists can use English to capture such requirements, an approach in line with current practice. Moreover, an English description of assumptions is accessible to a wide range of stakeholders.

It is possible, however, to use, for example, a diagrammatic notation based on the intermediate representation. It is a purely technological matter to extend RoboTool to support writing of models based on the metamodel for the intermediate representation. Moreover, due to our use of GF, realisation of RoboWorld in different natural languages is just a matter of dealing with the different linearisations of standard concepts, such as noun and verb phrases, in the new language.

Robotics is a very wide domain, and the open vocabulary, the intermediate representation, and the compositional sets of transformation rules are important assets to support future evolution and additional specialisation of RoboWorld. For example, we can cover, via a richer dictionary, classes of robotic applications, such as terrestrial or aerial robots, or even specific application areas, like healthcare or agriculture. On the other hand, we can envisage use of RoboWorld for cyber-physical systems in general, where a physical platform and an environment are also key components. Moreover, the part of the RoboWorld metamodel that is presented in Figure 10, defining ItemPhrases and sentences, can be reused in other CNLs.

In future work, we will consider additional case studies. RoboWorld can cater for 96 different structures for writing sentences, and the GF 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 [31]. 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, from 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 take advantage of Isabelle/UTP, unique in that it builds on a widely used theorem prover and the UTP to support very rich hybrid, reactive, and concurrent models. Automation can benefit from integrated use of theorem proving and model checking.

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.

ACKNOWLEDGMENTS

We are grateful to the members of the RoboStar centre and anonymous referees for valuable suggestions and discussions.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2007. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[2] M. Althoff. 2015. An introduction to CORA 2015. In *1st and 2nd International Workshop on Applied Verification for Continuous and Hybrid Systems.* G. Frehse and M. Althoff (Eds.), EPiC Series in Computing, Vol. 34. EasyChair, 120–151.

[3] M. Askarpour, L. Lestingi, S. Longoni, N. Iannacci, M. Rossi, and F. Vicentini. 2021. Formally-based model-driven development of collaborative robotic applications. *Journal of Intelligent and Robotic Systems* 102, 3 (2021), 59.

[4] J. Baxter, A. L. C. Cavalcanti, G. Carvalho, and F. Rodrigues Jr. 2022. *RoboWorld Reference Manual.* Technical Report. RoboStar Centre on Software Engineering for Robotics. Retrieved from robostar.cs.york.ac.uk/publications/techreports/reports/roboworld-reference.pdf

[5] J. Baxter, A. L. C. Cavalcanti, M. Gazda, and R. Hierons. 2022. *Testing using CSP Models: Time, Inputs, and Outputs – Extended Version.* Technical Report. RoboStar Centre on Software Engineering for Robotics. Retrieved from robostar.cs.york.ac.uk/publications/reports/BCGH22.pdf

[6] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. 2006. UPPAAL 4.0. In *3rd International Conference on the Quantitative Evaluation of Systems*. IEEE Computer Society, 125–126.

[7] A. Burns, I. J. Hayes, and C. B. Jones. 2020. Deriving specifications of control programs for cyber physical systems. *The Computer Journal* 63, 5 (2020), 774–790.

[8] J. J. Camilleri, M. R. Haghshenas, and G. Schneider. 2018. A web-based tool for analysing normative documents in english. In *33rd Annual ACM Symposium on Applied Computing*. Association for Computing Machinery, 1865–1872.

[9] J. J. Camilleri, G. Paganelli, and G. Schneider. 2014. A CNL for contract-oriented diagrams. In *Controlled Natural Language*. B. Davis, K. Kaljurand, and T. Kuhn (Eds.), Springer International Publishing, 135–146.

[10] A. L. C. Cavalcanti, W. Barnett, J. Baxter, G. Carvalho, M. C. Filho, A. Miyazawa, P. Ribeiro, and A. C. A. Sampaio. 2021. *RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing*. Springer International Publishing, 249–293. DOI: https://doi.org/10.1007/978-3-030-66494-7_9

[11] A. L. C. Cavalcanti, J. Baxter, and G. Carvalho. 2021. RoboWorld: Where can my robot work?. In *Software Engineering and Formal Methods*. R. Calinescu and C. S. Păsăreanu (Eds.), Lecture Notes in Computer Science, Springer, 3–22. DOI: https://doi.org/10.1007/978-3-030-92124-8_1

[12] A. L. C. Cavalcanti, J. Baxter, R. M. Hierons, and R. Lefticaru. 2019. Testing robots using CSP. In *Tests and Proofs*, D. Beyer and C. Keller (Eds.), Springer, 21–38. DOI: https://doi.org/10.1007/978-3-030-31157-5_2

[13] A. L. C. Cavalcanti, B. Dongol, R. Hierons, J. Timmis, and J. C. P. Woodcock (Eds.). 2021. *Software Engineering for Robotics*. Springer International Publishing. DOI: https://doi.org/10.1007/978-3-030-66494-7

[14] A. L. C. Cavalcanti, A. C. A. Sampaio, A. Miyazawa, P. Ribeiro, M. Conserva Filho, A. Didier, W. Li, and J. Timmis. 2019. Verified simulation for robotics. *Science of Computer Programming* 174 (2019), 1–37. DOI: https://doi.org/10.1016/j.scico.2019.01.004

[15] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. 2003. A refinement strategy for *circus*. *Formal Aspects of Computing* 15, 2–3 (2003), 146–181. DOI: https://doi.org/10.1007/s00165-003-0006-5

[16] A. Desai, I. Saha, J. Yang, S. Qadeer, and S. Seshia. 2017. DRONA: A framework for safe distributed mobile robotics. In *8th International Conference on Cyber-Physical Systems*. IEEE, 239–248.

[17] Swaib Dragule, Sergio García Gonzalo, Thorsten Berger, and Patrizio Pelliccione. 2021. *Languages for Specifying Missions of Robotic Applications*. Springer International Publishing. DOI: https://doi.org/10.1007/978-3-030-66494-7_12

[18] M. Esser and P. Struss. 2007. Obtaining models for test generation from natural-language like functional specifications. In *International Workshop on Principles of Diagnosis*. 75–82.

[19] S. Foster, J. Baxter, A. L. C. Cavalcanti, A. Miyazawa, and J. C. P. Woodcock. 2018. Automating verification of state machines with reactive designs and isabelle/UTP. In *Formal Aspects of Component Software*. K. Bae and P. C. Ölveczky (Eds.). Springer, Cham, 137–155. DOI: https://doi.org/10.1007/978-3-030-02146-7_7

[20] S. Foster, A. L. C. Cavalcanti, S. Canham, J. C. P. Woodcock, and F. Zeyda. 2020. Unifying theories of reactive design contracts. *Theoretical Computer Science* 802 (2020), 105–140. DOI: https://doi.org/10.1016/j.tcs.2019.09.017

[21] Martin Fowler. 2010. *Domain Specific Languages* (1st. ed.). Addison-Wesley Professional.

[22] Carlo A. Furia, Matteo Rossi, and Dino Mandrioli. 2007. Modeling the environment in software-intensive systems. In *International Workshop on Modeling in Software Engineering (MISE'07: ICSE Workshop 2007)*. 11–11. DOI: https://doi.org/10.1109/MISE.2007.11

[23] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. 2021. Automated formalization of structured natural language requirements. *Information and Software Technology* 137 (2021), 106590. DOI: https://doi.org/10.1016/j.infsof.2021.106590

[24] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. 2014. FDR3 - a modern refinement checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*. 187–201.

[25] C. A. R. Hoare and He Jifeng. 1998. *Unifying Theories of Programming*. Prentice-Hall.

[26] Tobias Kuhn. 2014. A survey and classification of controlled natural languages. *Computational Linguistics* 40, 1 (2014), 121–170. DOI: https://doi.org/10.1162/COLI_a_00168

[27] M. Kwiatkowska, G. Norman, and D. Parker. 2004. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer* 6, 2 (2004), 128–142.

[28] K. Larsen, M. Mikucionis, and B. Nielsen. 2005. Online testing of real-time systems using UPPAAL. In *Formal Approaches to Software Testing*. J. Grabowski and B. Nielsen (Eds.), Springer, Berlin, 79–94.

[29] Livia Lestingi, Davide Zerla, Marcello M. Bersani, and Matteo Rossi. 2023. Specification, stochastic modeling and analysis of interactive service robotic applications. *Robotics and Autonomous Systems* 163 (2023), 104387. DOI: https://doi.org/10.1016/j.robot.2023.104387

[30] N. Lincoln and S. M. Veres. 2013. Natural language programming of complex robotic BDI agents. *Journal Intelligent Robotics Systems* 71, 2 (2013), 211–230.

[31] B. Luteberget. 2019. *Automated Reasoning for Planning Railway Infrastructure*. Ph.D. Dissertation.

[32] B. Luteberget, J. J. Camilleri, C. Johansen, and G. Schneider. 2017. Participatory verification of railway infrastructure by representing regulations in RailCNL. In *Software Engineering and Formal Methods.* A. Cimatti and M. Sirjani (Eds.), Springer International Publishing, 87–103.

[33] S. Maoz and J. Ringert. 2015. Synthesizing a lego forklift controller in GR(1): A case study. In *4th Workshop on Synthesis (EPTCS).* P. Cerný, V. Kuncak, and P. Madhusudan (Eds.), Vol. 202, 58–72.

[34] S. Maoz and Y. Saar. 2011. AspectLTL: An aspect language for LTL specifications. In *10th International Conference on Aspect-Oriented Software Development.* Association for Computing Machinery, 19–30.

[35] A. Miyazawa, A. L. C. Cavalcanti, S. Ahmadi, M. Post, and J. Timmis. 2020. *RoboSim Physical Modelling: Diagrammatic Physical Robot Models.* Technical Report. University of York, Department of Computer Science, York, UK. Retrieved from robostar.cs.york.ac.uk/notations/

[36] A. Miyazawa, P. Ribeiro, K. Ye, A. L. C. Cavalcanti, W. Li, J. Timmis, and J. C. P. Woodcock. 2020. *RoboChart: Modelling, Verification and Simulation for Robotics.* Technical Report. University of York, Department of Computer Science, York, UK. Retrieved from www.cs.york.ac.uk/robostar/notations/

[37] J. H. Y. Munive, G. Struth, and S. Foster. 2020. Differential hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In *18th International Conference on Relational and Algebraic Methods in Computer Science.* Vol. 12062, Lecture Notes in Computer Science, Springer, 169–186.

[38] S. Nogueira, A. C. A. Sampaio, and A. C. Mota. 2014. Test generation from state based use case models. *Formal Aspects of Computing* 26, 3 (2014), 441–490.

[39] J. Peleska, E. Vorobev, F. Lapschies, and C. Zahlten. 2011. *Automated Model-Based Testing with RT-Tester.* Technical Report. Universität Bremen.

[40] M. Quottrup, T. Bak, and R. Izadi-Zamanabadi. 2004. Multi-robot planning: A timed automata approach. In *IEEE International Conference on Robotics and Automation.* 4417–4422.

[41] A. Ranta. 2011. *Grammatical Framework: Programming with Multilingual Grammars.* CSLI Publications.

[42] Andreas Wortmann, Jan Oliver Ringert, and Bernhard Rumpe. 2014. Architecture and behavior modeling of cyber-physical systems with MontiArcAutomaton. Shaker Verlag, Aachen.

[43] A. W. Roscoe. 2011. *Understanding Concurrent Systems.* Springer.

[44] T. Santos, G. Carvalho, and A. Sampaio. 2018. Formal modelling of environment restrictions from natural-language requirements. In *Formal Methods: Foundations and Applications.* T. Massoni and M. Mousavi (Eds.), Springer International Publishing, 252–270.

[45] S. Schneider. 2000. *Concurrent and Real-time Systems: The CSP Approach.* Wiley.

[46] Matthias Schnelte. 2009. Generating test cases for timed systems from controlled natural language specifications. In *International Conference on System Integration and Reliability Improvements.* 348–353.

[47] R. Schwitter. 2002. English as a formal specification language. In *International Workshop on Database and Expert Systems Applications.* France.

[48] O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu. 2003. Automated environment generation for software model checking. In *18th IEEE International Conference on Automated Software Engineering.* 116–127. DOI: https://doi.org/10.1109/ASE.2003.1240300

[49] J. C. P. Woodcock and J. Davies. 1996. *Using Z - Specification, Refinement, and Proof.* Prentice-Hall.

Received 27 January 2023; revised 19 September 2023; accepted 22 September 2023