High-Level Mission Specification for Multiple Robots

Sergio García
Chalmers | University of Gothenburg
Gothenburg, Sweden
sergio.garcia@gu.se

Patrizio Pelliccione
Chalmers | University of Gothenburg
Gothenburg, Sweden
University of L'Aquila
L'Aquila, Italy
patrizio.pelliccione@gu.se

Claudio Menghi
University of Luxembourg
Luxembourg, Luxembourg
claudio.menghi@uni.lu

Thorsten Berger
Chalmers | University of Gothenburg
Gothenburg, Sweden
thorsten.berger@gu.se

Tomas Bures
Charles University
Prague, Czech Republic
bures@d3s.mff.cuni.cze

Abstract

Mobile robots are increasingly used in our everyday life to autonomously realize missions. A variety of languages has been proposed to support roboticists in the systematic development of robotic applications, ranging from logical languages with well-defined semantics to domain-specific languages with user-friendly syntax. The characteristics of both of them have distinct advantages, however, developing a language that combines those advantages remains an elusive task. We present PROMISE, a novel language that enables domain experts to specify missions on a high level of abstraction for teams of autonomous robots in a user-friendly way, while having well-defined semantics. Our ambition is to permit users to specify high-level goals instead of a series of specific actions the robots should perform. The language contains a set of atomic tasks that can be executed by robots and a set of operators that allow the composition of these tasks in complex missions. The language is supported by a standalone tool that permits mission specification through a textual and a graphical interface and that can be integrated within a variety of frameworks. We integrated PROMISE with a software platform providing functionalities such as motion control and planning. We conducted experiments to evaluate the correctness of the specification and execution of complex robotic missions with both simulators and real robots. We also conducted two user studies to assess the simplicity of PROMISE. The results show that PROMISE effectively supports users to specify missions for robots in a user-friendly manner.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '19, October 20–22, 2019, Athens, Greece
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6981-7/19/10...$15.00
https://doi.org/10.1145/3357766.3359535

CCS Concepts • **Computer systems organization → Robotics**; • **Software and its engineering → Domain specific languages**.

Keywords Multi-robot, domain-specific language, mission specification

ACM Reference Format:
Sergio García, Patrizio Pelliccione, Claudio Menghi, Thorsten Berger, and Tomas Bures. 2019. High-Level Mission Specification for Multiple Robots. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357766.3359535

1 Introduction

Future robotic applications will include general-purpose mobile robots that may be configured by end-users to perform missions of everyday life, as analyzed by the H2020 Robotics Multi-Annual Roadmap [48]. For example, a user may want to assign the following mission to a robotic application: "patrol a set of locations l_1, l_2, l_3, and l_4 for security purposes" and "raise an alarm whenever an unknown person is found during night hours." This can be performed either programmatically in a General-Purpose Language (GPL), via Domain-Specific Languages (DSLs), or by using logical languages that allow to precisely describe the mission the robotic application should achieve [10, 19, 30, 46]. A declarative specification can be defined in many languages, including formal temporal logics, as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). Those logics are increasingly used in the research community and are becoming standard tools for specifying robotic missions as they can be automatically processed by planners [15, 16, 20, 21, 29, 32, 35, 37, 54, 60]. A planner is a software component that receives as input a model of the mission specification and derives the sequences of actions robots must execute. However, writing correct formulae in temporal logic requires knowledge of their syntax and semantics, which makes mission specification a cumbersome and error-prone task, even for experts [3, 24].

Existing solutions for mission specification do not combine in a convincing way *expressiveness*, *simplicity*, and *rigorousness*. Mission specification must be expressive enough to model a variety of behaviors in the robot's mission to support, for instance, recovering after errors: "A robot *r* grasps an object *o*, and if *o* falls during the action, *r* shall look for it and try to grasp *o* again." Programmatic approaches with GPLs as Python provide enough expressive power to the user, but they often require specific knowledge from them. In this light, we see the benefits of an intuitive language tailored to the domain that aids users to easily define their concerns. Rigorous languages ensure that mission specification precisely and unambiguously represents the mission to be executed. Among them, those that can be integrated with existing planners (e.g., temporal logics) often require mathematical knowledge or are constrained to specific robotic platforms.

Rigorous and expressive solutions such as Petri Nets [56, 61] and Statecharts [4, 27, 52] have been proposed for mission specification, but often require a step-by-step description of a robot's mission instead of a high-level description, as we intend to provide. Robotics companies have also worked on providing support for mission specification, and almost every robot model is released with an IDE or framework to support its software development [31, 38, 40]. However, those instruments are often platform-dependent, their usage is constrained to a limited number of robotic models and therefore, they provide a limited set of non-customizable features.

This paper presents PROMISE (simPle RObot MIssion SpEcification), a DSL that 1) offers a middle-ground solution, supporting the user with a user-friendly syntax while having well-defined (translational) semantics; 2) enables a rigorous and precise specification required for the use of planners, analysis tools, simulators or other modules; 3) allows the specification of complex missions by providing executable and combinable tasks and operators; and 4) is platform-independent and highly customizable.

PROMISE builds upon a recently proposed catalog of mission specification LTL-based patterns [37], ensuring the correctness of the semantics while at the same time raising the level of abstraction so as to support roboticists without knowledge on temporal logics. PROMISE is developed as a standalone application, but could be integrated with various tools and platforms. The rigorousness of PROMISE is granted by the use of LTL as the underlying language used by the robot's planner. We evaluate PROMISE in terms of expressiveness and simplicity. To validate its expressiveness, we check how PROMISE effectively supports the specification of complex missions; specifically, we conducted experiments with PROMISE in simulation and real-world scenarios. To validate our DSL's simplicity and users' satisfaction level, we conducted two user studies. We target as (end) users roboticists who may not have broad programming expertise nor expertise in formal methods and temporal logics.

Organization. Sec. 2 discusses the background and the related work. Sec. 3 introduces the research methodology. Sec. 4 presents PROMISE. Sec. 5 describes the DSL implementation and Sec. 6 its evaluation. Sec. 7 concludes with final remarks.

2 Background and Related Work

Behavioral modeling. A wide range of languages and formalisms for modeling and reasoning about behaviors has been developed. Many of these have also been used for describing the behavior of robots. Specifically, Statecharts [22] have been successfully applied to describe robotic missions [4, 27, 52]. Statecharts are an extension of state machines and state diagrams, with a graphical syntax and formal semantics. This type of diagrams requires a detailed definition of the steps a robot (or a team of robots) must perform in order to achieve a mission, which may become complicated in practical cases.

Another formalism is Petri Nets [39], which in the past have been successfully applied to develop robotic applications [56, 61]. In particular, Petri Net Plans (PNPs) [61] were developed to allow developers to describe plans for teams of robots. Despite their expressive power, PNPs require a precise definition of every action a robot is requested to perform.

Temporal Logic Languages. Temporal logics, and in particular LTL and CTL, are being increasingly used by robotics experts for mission specification [15, 16, 20, 21, 29, 32, 35, 37, 54, 60]. LTL is also widely used due to the variety of existing planners which take LTL-based models as input [15, 21, 35]. Furthermore, a temporal logic-based specification has other benefits as allowing users to analyze mission satisfaction through the use of model checkers. However, manually writing LTL specifications is complex and error-prone [3, 24]. To aid users in this process, mission specification patterns [37] have been recently proposed to automatically generate mission specification in temporal logics from recurrent mission specification problems. In our language PROMISE, we refer to those patterns as *tasks*, since they describe simple recurrent mission specifications that can be composed to obtain complex missions. In order to compose such complex missions, we propose a set of *operators*. The operators are inspired by Behaviour Trees [26], a mathematical model of plan execution that allows composing tasks in a modular fashion through a set of nodes representing tasks and connections among them.

In summary, both contemporary behavioral modeling and temporal logic languages have the advantage of coming with well-defined semantics but, on the other hand, they are relatively low-level and mainly imperative. Still, LTL specifications are declarative descriptions of what a robot should do, while Statecharts and PNPs specify the required behavior of the robot to achieve a certain mission. Consequently, LTL can be seen as an abstraction over Statecharts and PNPs and suitable to be implemented as a layer between our DSL and the low-level actions to be achieved by a robot.

Domain-Specific Languages. Model-Driven Engineering (MDE) has been identified [48] as a core technology to support developers when designing robotic systems. Special emphasis is given to DSLs, which are required to achieve a separation of roles in the robotics domain while also improving composability and system integration, and addressing non-functional properties [48]. The research community has already worked in the last years on the use of languages for the development of robotic software systems [6, 44, 45, 50].

Contemporary DSLs have the advantage of being more user-friendly and being tailored to the robotics domain. On the other hand, none of the existing DSLs sufficiently combine expressiveness, simplicity, and rigorousness. For instance, rigorousness is required for certification of robotic systems, which in turn requires well-defined semantics. We discuss the combination of expressiveness, simplicity, and rigorousness in existing DSLs in the remainder of the section.

Mission Specification for Robots. Various DSLs [45] have been proposed for modeling robotic systems and their mission. They have also been used to reason on the robot's behaviors through automated reasoning techniques such as simulation, model checking, and theorem proving [38].

Specification of robotic missions using mission specification patterns is supported by the PsALM tool [36]. This tool allows composing mission specification patterns with "and" and "or" logical operators. PROMISE and its tool support drastically extend the expressiveness and support provided by this tool, with complex compositional operators that pave the usage of the proposed patterns in practical scenarios.

Götz et al. [19] propose NaoText, a rule-based Java-like language for specifying collaborative robot applications. Campusano et al. [7] present Live Robot Programming (LRP), which enables "live programming", i.e., maximizing the feedback provided to the programmer of the program behavior. With LRP programmers benefit of a much closer and immediate connection to the program they are writing, minimizing the time and effort required. Despite simplifying the specification of collaborative tasks with respect to GPLs, the purpose of both NaoText and LRP is to support programmers, while we strive to support roboticists (who may not have the same programming expertise) by promoting our DSL's simplicity.

The DSL presented by Schwartz et al. [46] represents a robotic mission as a set of routing elements for moving the robot between locations; actions can be performed when the robot reaches waypoints. Conditional branches are used to add a certain level of flexibility to the modeled missions. However, the proposed DSL lacks reactive responses to external events between waypoints and does not support other options apart from if-branches—i.e., it lacks expressiveness.

FLYAQ provides a platform and DSLs [5, 10], which aim at enabling non-technical operators to define and execute missions for a team of multicopters. FLYAQ has been extended [13] by providing support for automatically generating property specification languages. An attempt to generalize FLYAQ to generic robots has been conducted [8]. These works present a user-friendly language, however, as discussed for behavioral modeling, the mission specification becomes more complicated, since the operator is required to state precisely how the robots should behave (in a higher-level). Contrarily, our DSL is declarative and the specifier needs to state only what the goal of the mission is instead of the steps needed to perform to satisfy the mission.

The MissionLab framework [51], which allows the specification and execution of robotic missions, has been extended by researchers [11, 30]. In the work of MacKenzie et al. [30], the only task that does not require specific knowledge from the user is the "assemblage" of robotic skills, whose semantics must be previously defined programmatically by users with knowledge of C++.

Doherty et al. [11] define a language that allows the generation of Temporal Action Logic (TAL), which is a high-level language that supports the specification of actions as preconditions and effects. Differently, in our approach, we used LTL, which enhances the mission specification expressiveness and rigorousness, since it supports the specification of missions based on the order in which actions occur. This is different from TAL, in which the specification is reactive, it uses pre- and post-conditions.

LTLvis [49, 57] uses a graphical specification environment for LTL specifications. However, even though graphical, the language is based on LTL operators augmented with locations that must and must not be visited by a robot. Therefore, the specifier is requested to know the semantics of LTL. Differently, our DSL simplifies the mission specification task by working at a higher level of abstraction to support users without this knowledge.

Silva et al. [47] propose an XML dialect that allows the description of missions to be performed by a team of autonomous vehicles. The missions are defined as high-level concepts that can be triggered based on conditions of the environment. The authors define a set of "triggers" to execute such high-level concepts of the mission. However, the mission expressiveness is reduced because trigger types are limited (e.g., altitude, speed), as opposed to PROMISE, where events are defined by the user.

Doherty et al. [12] define a task specification language based on an abstract data structure called "task specification tree". This language allows the mission specification based on a set of nodes that represent sequential and concurrent action execution. Nevertheless, the language is not designed for direct creation of LTL specification that can be used as input by planners. The ambition of our work is to define a DSL and to automatically generate specifications in LTL.

3 Research Methodology

We followed the Design Science research approach [23] with the aim of iteratively designing, validating, and improving the semantics of our language and its implementation. An overview of our methodology is represented in Fig. 1.

Definition of the DSL. PROMISE stems from the necessity of allowing roboticists to specify missions for a team of (possibly) heterogeneous robots. It has been defined in the context of a project [42] in collaboration with two companies: PAL Robotics (PAL) [1] and the Bosch Center of AI (BCAI) [2], both working in the robotics domain. In this context, we elicited requirements for a mission specification tool from practitioners and researchers—e.g., user-friendliness. We then investigated the current state-of-the-art to assess the best approach to develop such a tool. This first stage of our methodology corresponds to the "Understand the environment" phase of the framework conceived by Hevner et al. [23]. Below, we describe how we explored the problem space by retrieving *applicable knowledge* and *business needs*.

DSL and Tool Design. We conceived a prototype of PROMISE and performed demonstrations for: 1) PAL, where two product managers, and a robotics engineer (13, seven, and two years of experience, respectively) were involved; and 2) BCAI, to which a senior manager (10 years of experience) and a Ph.D. student participated. We collected their observations (e.g., enhance the feedback loop with the user by generating a natural English file with their specification of the mission) and applied according changes after experimentation to reach a company-validated version of the DSL.

We then conducted two user studies to continue the iterative validation and improvement of PROMISE—i.e., the DSL was updated after each study based on the participants' feedback. The design of both studies is similar: the participants received a set of tasks to be achieved within a given time frame and were requested to use a simulator to check their solutions. After the tasks' completion, the participants were asked to submit their results and to fill in a questionnaire. Participants chosen from the University of Gothenburg (UGOT) are experts in Software Engineering (SE) and the ones from the University of L'Aquila (UNIVAQ) in SE and Algorithms. There is no overlapping between the two user groups. Participants had no expertise in robotics.

Iterative Experimentation. We first conducted an exploratory user study to assess whether our language and its implementation are perceived as simple, user-friendly, and not error-prone. The participants were nine Ph.D. students from

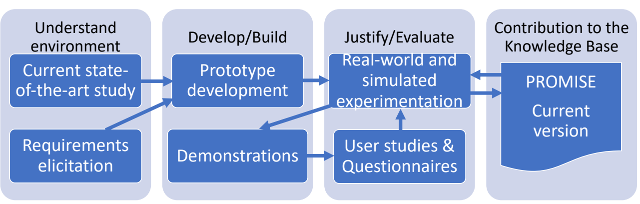

Figure 1. Followed research methodology.

UNIVAQ. For the first study, we distributed an image of a virtual machine (VM) containing Ubuntu 16.04, Eclipse Neon 3.1, a set of plugins for installing PROMISE, ROS Lunar [41], the framework implementation of PROMISE, Gazebo [28], and a set of scripts to ease the user experience. VM's setup and usage guidelines were also distributed. Then, we provided a training session consisting of a lecture of two hours, where the semantics of LTL, robotic mission patterns, and PROMISE were explained. The participants were tested right after the training and we randomly divided the group of nine Ph.D. students in two groups. Each group was requested to specify one mission using both LTL and PROMISE, having one hour for each task (the tasks' order changed for each group). The received feedback helped us to design the second user study.

Six participants (five Ph.D. students and one Post-doc) from UGOT contributed to the second user study, to which we applied some changes based on our previous experience—e.g., in the first study, the VM installation was too problematic for many participants, so we decided to conduct the tests using our computer. This study was focused on the usability of PROMISE, on how clear was the semantics of each operator, and on the participants' experience. We also strove to understand the differences between the perception of PROMISE as a language and of its implementation. We provided a 45 minutes seminar for the whole group of participants. During this seminar, we elaborated on the semantics of each operator with a set of examples, showed the optimal workflow, and explained technical details such as setting the operators' parameters. The material provided for this study was a laptop with the (updated) software used for the first study, an updated version of the DSL's guidelines, and a notebook to sketch during the test. Each test was conducted individually and was divided into three blocks, having a limit of 30 minutes for each.

The first block consisted of the last stage of training, where participants were requested to specify four missions with the help of a trainer to get familiar with the DSL, its framework, and the simulator. The two other blocks had to be achieved by the participants without help and consisted of: **B1**, a set of 4 simple missions, each one focusing on the use of an operator (i.e., *sequence*, *fallback*, *eventHandler*, and *TaskCombination*); and **B2**, a complex mission that required the combination of several operators and robots, based on the running example introduced in Sec. 4. During the study, participants had the freedom of using either editor (textual or graphical) or both of them. However, we encouraged the participants to use the graphical editor for being considered easier for beginners.

The questionnaires of both studies contain multiple Likert-scale and open-ended questions about different features of PROMISE and its implementation. The evaluation of both studies is detailed in Sec. 6.

4 PROMISE

This paper's contribution is PROMISE, a DSL that supports roboticists in mission specification for teams of robots.

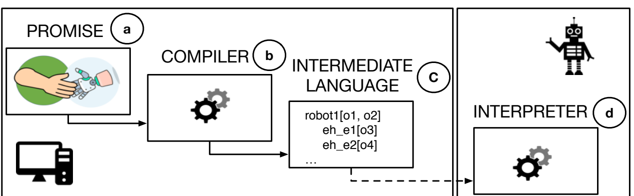

Figure 2. PROMISE and the software framework

PROMISE is integrated into a software framework that allows executing the specified missions on simulators and in actual robots. Figure 2 represents the software framework and its components, which are identified with letters. PROMISE (ⓐ) is composed of tasks and operators. Tasks allow the specification of high-level actions that can be performed by a single robot and operators allow the composition of these tasks into complex multi-robots missions. PROMISE provides both a graphical and a textual syntax, each of which is supported by a dedicated editor. Graphical and textual syntaxes are kept synchronized to enable end-users to switch from one editor to the other while specifying a mission. The graphical syntax maps mission specification concerns to graphical elements that can be understood by roboticists. The textual syntax allows the specification of a mission on a textual basis.

Running example. We illustrate our work with an example that involves two robots operating in a dynamic environment, (possibly) populated by humans where they have to react to events. The example describes a mission that is specified using PROMISE's graphical syntax in Fig. 3 and using the textual syntax in Fig. 4. Figure 4 was graphically modified to improve its readability by showing events, actions, and locations. Both figures are annotated with circled numbers depicting the nodes we refer to in the remaining of the paper.

A robot $r1$ should patrol locations l_1, l_2, and l_3 (in this specific order) within a building for security purposes. If $r1$ finds an unknown person then it will raise an alarm. During the patrolling, if $r1$ finds a recognizable object o it must request help from robot $r2$. $r2$ waits in l_4 until it receives a request of help from $r1$. Then, $r2$ proceeds to l_2, grasps o, and tries to go to location office_1. If $r2$ cannot reach this location (e.g., there is an unavoidable obstacle in its path), it tries to reach office_2, and then releases o. Moreover, both robots should recharge if their batteries are running low.

PROMISE relies on a compiler and an interpreter. The compiler ⓑ transforms the mission specified through the DSL into an intermediate language ⓒ. The intermediate language is platform-agnostic and, therefore, users only need to write a new interpreter to support a new robot platform. Then, the interpreter ⓓ takes as input a mission specified in the intermediate language ⓒ to execute the mission on the actual robots or on simulators. Summarizing, the framework in Fig. 2, and more specifically the DSL and the intermediate language are explicitly designed to enable: 1) the mission decomposition among the robots, 2) the mission execution by an appropriate interpreter, and 3) the mission execution by teams of heterogeneous robots. The synchronization among robots is currently performed by using events, as depicted in Fig. 3. The adoption of a more flexible approach for synchronization remains as future work.

4.1 Domain-Specific Language

Listing 1 represents an excerpt of the grammar of our language (we do not include the syntax of each task). PROMISE provides two concrete syntaxes, one graphical and one textual, as shown in Table 2. The semantics of our DSL is provided in pseudo-code and it is shown in the same table. In this context, each operator is executed when its result is retrieved, i.e., the operator's result is assigned to a variable (e.g., `res=`o_1). The semantics of some operators require the execution of several operators (i.e., their children) in one instruction (e.g., the semantics of the operators *Parallel* and *TaskCombination* in Table 2).

The DSL is based on *tasks*, which are executed through one *basic operator*, called delegate in the remainder, and on a set of *composition operators*, which allow composing tasks into complex missions. Tasks represent simple activities whose execution can be delegated to the robots. For example, a task may require a robot to visit a set of locations.

Tasks. Tasks are the basic entities of the proposed DSL. They represent elementary operations that can be performed by robots. Tasks are the mission specification patterns proposed

Table 1. Tasks catalog

	Name	Description
Core movement tasks	*Visit*	Visit a set of locations. The order may be defined in a sequenced, in a ordered, or in a strict ordered manner. Areas can be also requested to be visited a fair amount of times.
	Patrolling	Keep visiting a set of locations. The same rules for *Visit* apply here.
Avoidance tasks	*Past avoidance*	Requires a condition to not occur until another condition is satisfied.
	Global avoidance	An avoidance condition globally holds throughout the mission.
	Future avoidance	After the occurrence of an event, avoidance has to be fulfilled.
	Restricted avoidance	A restriction on the number of occurrences is desired. It might apply to the maximum, the minimum, or an exact number of times.
Trigger tasks	*Reaction*	The occurrence of a stimulus triggers a counteraction. The triggered counteraction might be executed instantaneously or some time later.
	Wait	Inaction is desired until a stimulus occurs.
	Simple action	A counteraction is performed in the next time instant without requiring any kind of stimulus.

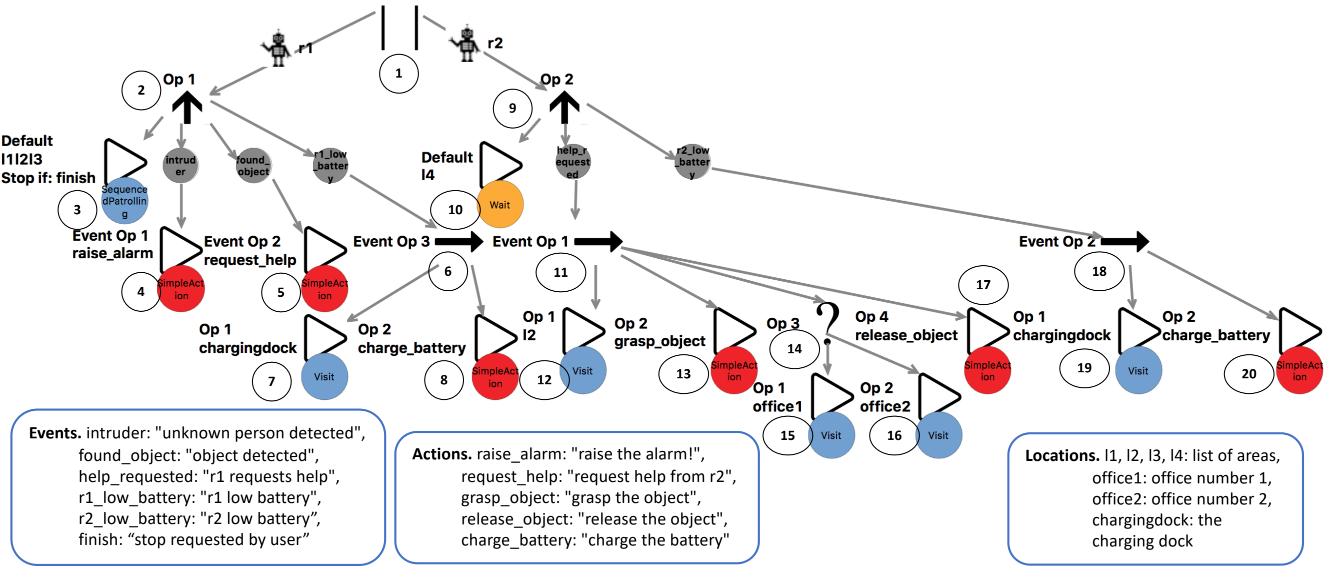

Figure 3. Running example specified with the graphical syntax of the DSL.

```
operators{ parallel{
  r1(eventHandler(
    default(delegate ( SequencedPatrolling locations l1, l2, l3
      stoppingEvents finish))
    except intruder (delegate (SimpleAction actions raise_alarm))
    except found_object (delegate (SimpleAction actions request_help))
    except r1_low_battery (sequence(
      delegate(Visit locations chargingdock),
      delegate(SimpleAction actions charge_battery))))),
  r2(eventHandler(
    default(delegate(Wait locations l4))
    except help_requested( sequence(
      delegate(Visit locations l2),
      delegate(SimpleAction actions grasp_object),
      fallback (
        delegate(Visit locations office1),
        delegate(Visit locations office2)),
      delegate ( SimpleAction actions release_object ) ) )
    except r2_low_battery(sequence(
      delegate(Visit locations chargingdock),
      delegate (SimpleAction actions charge_battery)))))
  }
}
```

Figure 4. Running example specified with the textual syntax of the DSL.

to specify missions of mobile robots [37]. In Table 1, we show an abstract description of the proposed catalog, splitting the tasks into three groups: *core movement tasks*, *avoidance tasks*, and *trigger tasks*. Each task maps a recurrent robotic specification problem identified in the literature to well-known solutions with proved effectiveness expressed in temporal logics (specifically, LTL and CTL). For example, visit has five different variants. In particular, the task *ordered visit* forces a robot to visit a set of locations following an ordering and it forbids a successor to be visited before its predecessor. The LTL formulation for the task is:

$$\mathcal{F}(l_1 \wedge \mathcal{F}(l_2 \wedge \ldots \mathcal{F}(l_n))) \bigwedge_{i=1}^{n-1} (\neg l_{i+1})\, \mathcal{U}\, l_i$$

where the initial part of the formula specifies that the visit of the different locations $l_1, \ldots, l_n$ should be done in order, through the nested use of the eventually temporal operator ($\mathcal{F}$). The second part of the formula forbids the visit of l_{i+1}

```
Mission:
  'mission' '{'
  ('conditions''{'
  ('events'events+=Event (','events+=Event)*)?
  ('actions'actions+=Action ( ','actions+=Action)*)?'}')?
  'robots' robots+=Robot ( ',' robots+=Robot)*
  ('locations'locations+=Location (','locations+=Location)*)?
  'operators''{'operator+=Operator (','operator+=Operator)*'}''}';
Operator:
  //List of operators from Table 2
Tasks:
  //List of tasks from the provided catalog
Robot:
  name=String;
Location:
  name=String;
Event:
  name=ID ':' description=String;
Action:
  name=ID ':' description=String;
FallBackOp:
  'fallback' '(' inputOperators+=Operator (',' inputOperators+=Operator)* ')';
SequenceOp:
  'sequence' '(' inputOperators+=Operator (',' inputOperators+=Operator)*')';
ParallelOp:
  'parallel' '{'(inputRobots+=[Robot|String] '(' inputOperators+=Operator')'
   (',' inputRobots+=[Robot|String] '('inputOperators+=Operator')')*)?'}';
EventHandlerOp:
  'eventHandler' '('
  'default''('inputOperators+=Operator')'
  ('except' inputEvents+=EventAssignment)+')';
ConditionOp:
  'condition' '(' ('if' inputEvents+=EventAssignment )+')';
TaskCombinationOp:
  'combination' '(' inputOperators+=Operator
  (('&' | 'AND' | 'and') inputOperators+=Operator)+ ')';
DelegateOp:
  'delegate' '(' task=Tasks
  ('locations' inputLocations+=[Location|String]
  (',' inputLocations+=[Location|String])*)?
  ('actions' inputAction+=[Action|String] ( ',' inputAction+=[Action|String])*)?
  ('stoppingEvents' stoppingEvent+=[Event|String]
  (',' stoppingEvent+=[Event|EString])*)?')';
EventAssignment:
  inputEvent=[Event|String] '(' inputOperators=Operator')';
```

Listing 1. PROMISE's grammar.

before visiting the location l_i for every $1 \leq i \leq n-1$, through the use of temporal operator until ($\mathcal{U}$). The logic formulation

is hidden to the end-user that is instead asked to write the task name instantiated with a set of parameters that are task-dependent. For instance, the task *ordered visit* should be instantiated with the specific locations that need to be visited.

Delegate operator. Let us consider a finite set of events $\mathcal{E}$. The basic operator *delegate*, represented as $\rhd(t, \mathcal{E})$, allows the task execution. The operator delegate receives a task t and a set of events $\mathcal{E}$. It specifies that a task t, instantiated through a mission specification task, must be executed and the task execution must be suspended if an event $e \in \mathcal{E}$ occurs. For example, in Fig. 3 the task $\rhd(SequencePatrolling(l1, l2, l3), \{finish\})$ delegates the task $SequencePatrolling(l1, l2, l3)$ to the robot $r1$ and suspends the task execution if the event $finish$ is received (③). Notice that the $finish$ event is needed since a patrolling task is *non-terminating*. More precisely, tasks can be classified into two categories depending on the mission specification pattern used within the task: *terminating* and *non-terminating* tasks. The execution of terminating tasks can be satisfied by performing finite plans, i.e., finite execution of actions that allow the achievement of the desired mission. Thus, the terminating task's execution can 1) succeed if the corresponding plan is performed correctly, 2) fail if an exception occurs during the plan execution, or 3) suspend if an event $e \in \mathcal{E}$ occurs.

For example, if a robot has to visit a set of locations, a plan corresponds to the trajectory that must be followed by the robot to visit all the locations. If the robot is able to follow the trajectory and visits all the locations, the task execution succeeds. If an obstacle is detected and an alternative trajectory cannot be computed, then the task execution fails. If an event $e \in \mathcal{E}$ occurs while the task is executed, then the plan is suspended. Non-terminating tasks are associated with infinite plans. For example, if a robot has to patrol a set of locations, it has to enter these locations an infinite number of times. A non-terminating task's execution can 1) fail if an exception occurs during the plan execution, 2) suspend if the event e occurs, or 3) never terminate if neither an exception nor the event e occurs.

Composition operators. PROMISE allows the generation of complex missions by combining basic operators through a set of composition operators. Composition operators combine operators and manage events that can occur within the environment. The composition operators are presented in Table 2, where $\{e_1, e_2, \cdots, e_n\}$ and $\{o_1, o_2, \cdots, o_n\}$ indicate ordered sets of events and operators, respectively. As shown in Fig. 3, a mission is graphically represented as a graph in which nodes represent operators, and edges specify how operators are nested within each other. Syntactically, we constrain operators to be composed in a way that the final generated graph is a tree, called *mission tree*. The leaves of the tree always contain a delegate operator, represented by the symbol $\rhd$. Then, each operator delegate is associated with a task, represented by a colored circle (each type of tasks shown in Table 1 is associated to a color) and labeled with the task's name. The other nodes of the tree represent composition operators and are represented by symbols that are associated with the different operators.

We conceptually decompose missions into *global*—one general mission to be achieved by the whole team, e.g., the mission in Fig. 3 and Fig. 4—and *local*—specific missions for each robot, e.g., the mission assigned to $r1$ in Fig. 3 and Fig. 4—missions. To enable global mission decomposition into local ones, the operator *parallel* can only be used as a root operator. This operator is always the root of global missions, even for single-robot missions. The operator *parallel* aggregates different operators, each assigned to one robot. The operators *delegate* that are descendants of an operator assigned (i.e., delegated) to a robot, can only assign a task to that robot—see the syntaxes of the operator *parallel* in Table 2.

Task combination operator. This operator has been added after the first user study we conducted, since, thanks to the feedback from the experiment, we realized that the DSL was providing no way of combining or "merging" different tasks, like performing an action in the locations visited during a visit pattern. The *TaskCombination* operator takes a set of delegate operators as input (see Table 2 and Listing 1, Lines 34–36) and merges their associated tasks into one LTL formula by adding the logical operator &&. The combination possibilities are restricted to a core movement task combined with a set of avoidance tasks and/or a set of trigger tasks. We created this constraint to avoid semantically correct but meaningless combined missions—e.g., $r1$ must wait in l_1 and avoid entering location l_2. An example of usage of the operator *TaskCombination* is: "Robot *r1* must patrol locations l_1, l_2, and l_4 while avoiding location l_3, and once in location l_1, if it finds an unknown person, it should raise an alarm".

Running example: mission defined using PROMISE. The root of the mission specification, i.e., the operator *parallel*, is identified by the node ① and specifies that $r1$ and $r2$ must perform their missions in parallel. A robot is assigned to each branch associated with this operator, as indicated with labels in the edges between ① and ② and between ① and ⑨ in Fig. 3, and with the name of the assigned robot in Fig. 4. The operator marked with the symbol ↑ (②) is the *eventHandler*. It has a default behavior; in our example, it forces the robot to sequentially patrol locations l_1, l_2, and l_3 (③). This behavior is paused when one of the events that are assigned to the *eventHandler* is detected. Each event is assigned to a child of the *eventHandler* (as represented in Fig. 3) as gray circles and invoked in Fig. 4 by the keyword `except`. If the event "intruder" is detected, the first delegate operator instantiated with the *simple action* pattern is executed (④). In this case, the robot must raise an alarm. If the event "found_object" is detected, ⑤ is triggered, performing the action "request_help". Otherwise, the event "r1_low_battery" triggers the operator *sequence* ⑥, which makes the robot

go to its charging dock (⑦) and then perform the action "charge_battery" (⑧). The default robot's behavior (③) is resumed whenever any of the behaviors triggered by an event are finished (either succeeding or failing).

Meanwhile, r_2 waits in l_4 (⑨ and ⑩). The detection of "help_requested" triggers a sequence of executions (⑪), starting from the visiting of l_2 (⑫), followed by the action "grasp_object" (⑬). The operator *fallback* ⑭ encodes that r_2 must try to reach *office₁* (⑮), and if it fails (e.g., the office's door is closed) it tries to reach *office₂* (⑯). r_2 then releases the object in the reached office (⑰). The child of ⑨ triggered by "r2_low_battery" (⑱) is a replica of ⑥.

4.2 The Intermediate Language

The intermediate language is an intermediate representation of the global mission the robots should achieve. It allows decoupling the mission specification from the robotic platform and the development of interpreter tools. In this way, PROMISE becomes robot-agnostic since only the interpreter has to be adapted when using a new robot platform. Specifically, the interfaces of the interpreter with the underlying components of the robot must be specified (if they are not compliant with the ones already provided). Interpreters are responsible for executing missions by sending commands to the software controllers of the robots. We provide a grammar for the intermediate language, whose vocabulary contains a set of terminals denoted in orange in Listing 2.

The grammar's production rules are presented in Listing 2, being the starting symbol "Mission" (Line 1). A mission consists of a set of indented lines, each line containing a header, which specifies the name of the line, and a body (see Listing 3 for an example of a generated mission). The mission root is always an operator *parallel*, so the header of the first line of a mission is always the name of a robot, followed by its body. For other operators, the line's name is formed by two parts: 1) a reference to the operator that executes the line's behavior; and 2) either a reference to the condition that triggers the line's behavior (in the cases of operators *condition* and *eventHandler*, in Lines 18, 21, 24), or a counter (in the case of operator *fallback* in Line 15). The body is represented within square brackets and contains a set of indexed tasks to be executed sequentially, separated by commas (Line 3).

The grammar specifies that operators of the DSL presented in Section 4.1 are mapped on statements in the intermediate language as tasks, as specified in Table 2 (Line 4 of Listing 2). The operator *delegate* is substituted by an LTL formula corresponding to its associated task (Line 5), and the *taskCombination* is substituted by a combination of several tasks (Line 6) by means of the && logical operator. The operator *sequence* is translated into a set of indexed tasks (Line 7). The remaining operators are referenced by the prefixes "fb", "cond", and "eh" for *fallback*, *condition*, and *eventHandler*, respectively (Lines 9, 11, 13 of Listing 2).

```
1  Mission: (Robot '[' Body ']')+;
2  Robot=String;
3  Body: Task (',' Task)*;
4  Task: Delegate | TaskComb | Sequence | Fallback |
       Condition | EventHandler;
5  Delegate: pattern=LTL_formula;
6  TaskComb: pattern=LTL_formula('&&'pattern=LTL_formula)*;
7  Sequence: Task (',' Task)*;
8  Fallback:
9    'fb' Fb_line;
10 Condition:
11   'cond' Cond_line;
12 EventHandler:
13   'eh' Eh_default;
14 Fb_line:
15   'fb_' N '[' Body ']'
16   Eh_default | Cond_line | Fb_line;
17 Cond_line:
18   'cond_' Event '[' Body ']'
19   Eh_default | Cond_line | Fb_line;
20 Eh_default:
21   'eh_default' '[' Body ']'
22   Eh_default | Eh_event | Cond_line | Fb_line;
23 Eh_event:
24   'eh_' Event '[' Body ']'
25   Eh_default | Eh_event | Cond_line | Fb_line;
26 Event=String;
27 N=Integer;
```

Listing 2. Grammar of the intermediate language.

Running example: intermediate language. An instance of the intermediate language compiled from the mission described in Fig. 3 and Fig. 4 is presented in Listing 3, where ◇, □, $\mathcal{W}$, and $\mathcal{X}$ are the classical "Finally", "Globally", "Weak Until", and "Next" LTL operators. Indentation is used to denote the relation of parent-child among operators—i.e., a child operator has a greater indentation than its parent (e.g., Lines 1 and 2 and Lines 8 and 9). The first line of each robot's mission (Lines 1, 6) encode the operator *parallel* (①), specifying the required robot in the line's header ($r1$ and $r2$, respectively). The body of Line 1 contains a reference to the first child ②, an *eventHandler*. Line 2 encodes the default mission behavior of the *eventHandler* (②), i.e., the operator *delegate* marked with ③. Lines 3, 4, and 5 encode the behaviors triggered by ② if "intruder" (④), or "found_object" (⑤), or "r1_low_battery" (⑥) occur, respectively. Line 5 encodes a *sequence* of tasks executed by ⑥ (⑦ and ⑧).

The body of Line 6 references to the operator parallel's second child (⑨). The default mission (Line 7) of this child encodes a variation of the pattern *wait* (⑩). Line 8 encodes the behavior triggered by ⑨ if "help_requested" occurs: a sequence of tasks (⑪), being the third item a reference to ⑭, an operator fallback. Lines 9 and 10 encode the possible behaviors of this operator. Line 11 is a replica of Line 5.

```
1  r1[eh]
2    eh_default[□(◇((l1) && ◇((l2) && ◇((l3))))))]
3    eh_intruder[(X raise_alarm)]
4    eh_found_object[(X request_help)]
5    eh_r1_low_battery[◇(chargingdock),(X charge_battery)]
6  r2[eh]
7    eh_default[((l4) 𝒲 (FALSE)]
8    eh_help_requested[◇(l2), (X grasp_object), fb, (X
       release_object)]
9      fb_1[◇(office1)]
10     fb_2[◇(office2)]
11   eh_r2_low_battery[◇(chargingdock),(X charge_battery)]
```

Listing 3. Intermediate languages for $r1$, $r2$ (Fig 3 mission).

Table 2. Robotic missions specification operators

Name	Description	Semantics	Syntax	Intermediate language

Parallel
$\|\|(r_1,\dots r_n,o_1,\dots o_n)$
	Always the root of the mission. The operators $o_1, o_2, \cdots, o_n$ are executed in parallel, each by a different robot—i.e., assigns one branch to each robot. Returns success when all operators return success, failure otherwise.	$\{res_1, res_2, \cdots, res_n\}=\{o_1, o_2, \cdots, o_n\}$
		if ($res_1 == \top \wedge \cdots \wedge res_n == \top$) **then**
		return $\top$
		else return $\bot$
			parallel{r1(o_1), . . . , rn(o_n)}	`r1[o1]`
				`r2[o2]`
				`...`
				`rn[on]`

Delegate $\triangleright(\mathcal{E}, t)$	Delegates execution of a task t to a specific robot (specified by the Parallel operator). Tasks are specified using patterns for robotic missions that take as input parameters as locations (indicated as $l_1, l_2, \dots, l_n$) and actions (indicated as $a_1, a_2, \dots, a_n$).	**execute**($\mathcal{E}$, t)	delegate(t locations $l_1, \dots, l_n$) delegate(t actions $a_1, \dots, a_n$)	LTL formula of the pattern specified by the task t.

Fallback
$?(\{o_1, o_2, \cdots, o_n\})$
	Executes the first operator; if it is executed successfully, ends with success. If the execution of the first operator fails, tries to execute the second operator. This procedure is repeated for all the other operators. Returns failure if all operators fail.	**if** ($\{o_1,o_2,\cdots,o_n\} \neq \emptyset$) **then**
		$res = o_1$;
		if($res == \bot$) **then**
		$?(\{o_2, \cdots, o_n\})$
		else return $\top$
		else return $\bot$
			fallback ($o_1, o_2, \dots, o_n$)	`parent[fb]`
				`fb_1[o1]`
				`fb_2[o2]`
				`...`
				`fb_n[on]`

Sequence
$\rightarrow(\{o_1, o_2, \cdots, o_n\})$
	Executes all the operators from the first to the last. If an operator returns success executes the subsequent operator. If an operator returns a failure returns failure. Returns success if and only if all the operators return success.	**if**($\{o_1, o_2, \cdots, o_n\} \neq \emptyset$) **then**
		$res = o_1$;
		if($res == \top$) **then**
		$\rightarrow(\{o_2, \cdots, o_n\})$
		else return $\bot$
		else return $\bot$
			sequence ($o_1, o_2, \dots, o_n$)	`[o1,o2,...,on]`

EventHandler
$\Uparrow(e_1, \dots, e_n, o, o_1, \dots, o_n)$
	Executes a by default operator o. Once an event e_i occurs, executes operator o_i in response. Once the execution of o_i is finished, resumes the operator o. Returns success if the operator o succeeds and all the events that occurred during the execution of o are correctly handled.	$res = \bot$;
		while($res \neq \top$)
		$res = o$;
		if($res == \top$) **then**
		return $\top$
		if($e_i == \top$) **then**
		$resint = o_i$;
		if($resint == \bot$) **then**
		return $\bot$
		res = **resume**(o);
		return res
			eventHandler(
			default(o)
			except e_1 (o_1)
			except e_2 (o_2)...
			except e_n (o_n))
				`parent[eh]`
				`eh_default[o]`
				`eh_e1[o1]`
				`eh_e2[o2]`
				`...`
				`eh_en[on]`

Condition
$\oplus(\{e_1, \cdots, e_n, o_1, \cdots, o_n\})$
	Evaluates the conditions from the first to the last. If the evaluation of one or more conditions is true, executes the corresponding operators. Returns $\bot$ if an operation is not successful, i.e., either it fails or an event occurs. Returns $\top$ when all the executed operations return $\top$.	**if**($e_1 == \top$) **then**
		$res = o_1$
		if($res == \bot$) **then**
		return $\bot$
		...
		if($e_n == \top$) **then**
		$res = o_n$
		if($res == \bot$) **then**
		return $\bot$
		return $\top$
			condition(
			if e_1 then (o_1)
			if e_2 then (o_2)...
			if e_n then (o_n))
				`parent[cond]`
				`cond_e1[o1]`
				`cond_e2[o2]`
				`...`
				`cond_en[on]`

TaskComb. $\&(\{o_1, o_2\})$	Allows the composition of a *core movement* task with one or more *avoidance* tasks and with one or more *trigger* tasks. The composition is performed by means of the *and* logical operator.	$res = o_1$ && o_2 && $\cdots$ o_n **if**($res == \top$) **then** **return** $\top$ **else return** $\bot$	combination(o_1 and o_2 and ... o_n)	`[o1 && o2 && ... on]`

5 Implementation

Our DSL is developed as a stand-alone application that can be integrated and used in various contexts [18]. Robotic mission specification patterns [37] have been used together with model checking tools, such as NuSMV [9], simulation tools, such as Simbad [25], and design tools for robotic applications, such as Spectra [33]. Since the DSL relies on the mission specification patterns for mobile robots, all such tools can be

easily integrated with PROMISE. In the following, we explain the components of the implementation by referring to Fig. 2.

Domain-specific language ⓐ. PROMISE is developed as an Eclipse plugin implemented in Xtext [14], a framework for the development of programming and domain-specific languages; and Sirius [55] (a tool for graphical model manipulation and management).

Compiler ⓑ. The compiler is implemented in Xtend [53], a dialect of Java that we use for code generation. It 1) takes as input the textual representation of the mission specification, 2) decomposes the global mission, and 3) automatically generates the encoding of missions of each robot in the intermediate language (one file for each robot). It also generates one *readme* file for each local mission, containing the translation of the mission into natural English. The goal is to express the same content presented in the introduction in terms of the conditions and locations defined with PROMISE. The file is used to improve the users' experience and help them during mission definition by enhancing the feedback loop. An example of this translation is:

> *Robot* $r1$ *does by default patrol in sequence location(s)* l_1, l_2, *and* l_3, *and if event* $intruder$ *occurs, it will perform action* $raise_alarm$, *and if event* $found_object$ *occurs, it will perform action* $request_help$, *and if event* $r1_low_battery$ *occurs, it will visit (without any specific order) location(s)* $chargingdock$ *and perform action* $charge_battery$.

Both ⓐ and ⓑ are allocated in a central station (e.g., a laptop), which communicates to each robot when sending the mission. The state of such mission (e.g., what operator is being executed, whether the mission was successful) is communicated to the user by the interpreter (ⓓ), which in the current implementation of PROMISE relies on ROS. To enhance the feedback loop to the user, we plan to embed a display with runtime information about the mission to ⓐ.

Intermediate language ⓒ. It contains a rewriting of the specified mission in terms of LTL formulae, one for each delegate operator, which should be properly orchestrated according to the used composition operators, as explained in Sec. 4.2. The contents of each generated intermediate language file are sent to the correspondent robots when requested.

Interpreter ⓓ. There is one instantiation of the interpreter deployed in each robot. It parses the specification of the mission to be accomplished by the robot specified in ⓒ and transforms it into concrete movements and actions. It reads the instructions and executes them by sending messages to the low-level parts of the robot's architecture depending on the respective semantics of the operators in the specified mission.

6 Validation

To evaluate PROMISE with simulators and real robots, we developed an interpreter for the DSL, which is integrated into the SERA platform [17]. This platform provides several robotic functionalities, including motion control, self-localization, and planning. The experiments we performed included the specification of missions, the generation of the specification in the intermediate language, and the plugging into the interpreter of the robots shown in Fig. 5. From left to right, the robots' models are: 1) a TIAGo robot [43] in simulation [28] (top); 2) two intelligent transport assistants (ITA) in simulation (bottom); 3) a TurtleBot2 [58] in UGOT's facilities; and 4) a TIAGo robot in PAL's facilities. To conduct the validation, we formulated two research questions:

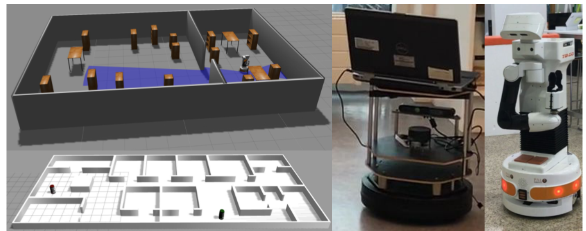

Figure 5. Robots and scenarios used in our experiments

- **RQ1**: does PROMISE *effectively* support the specification of complex missions?
- **RQ2**: how *simple* is it to specify missions with PROMISE?

6.1 Expressiveness of PROMISE (RQ1)

In order to answer this question, we defined complex missions that describe possible real-world scenarios. In particular, we specified three of the missions proposed for the 2018's edition of RoboCup@Home (Stage II)—rules available at [34]: 1) *dishwasher challenge*, a robot has to remove all dishes from a table (presumably after dinner) and place them into the dishwasher; 2) *tour guide*, a robot guides spectators to the audience area and answers certain predefined questions; 3) *restaurant scenario*, where robots shall serve food and beverages to customers. The graphical and textual syntaxes and the output intermediate language of each specified mission are available in the provided repository [18]. During the experimentation, complex actions (e.g., grasping an object) were simulated. All the RoboCup missions were validated through simulation in Gazebo using a TIAGo robot. Furthermore, the restaurant scenario was evaluated using different platforms: 1) a TIAGo in simulation, 2) a Turtlebot2 in UGOT's facilities, 3) a TIAGo robot in PAL's facilities, and 4) two ITA robots in simulation. We also formulated a homegrown mission consisting of two robots performing parallel and collaborative tasks, introduced as a running example in Sec. 4. This scenario was validated through simulation with two ITA robots.

Discussion. We answer to RQ1 through experimentation: with PROMISE a user is able to specify complex missions (as the ones we selected from a well-known competition as RoboCup or the one we conceived), which in turn can be executed by using the provided framework in different robotic platforms. We also make use of simulators and real

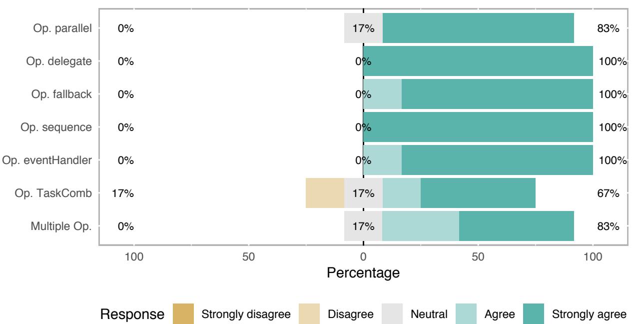

Figure 6. Second study's questionnaire: operators' semantics and multiple operator mission specification.

robots to ensure that the robots effectively perform a mission that is consistent with the semantics of the specification. In all the cases, PROMISE was able to describe the intended missions showing that it is able to effectively support the specification of complex missions.

6.2 Usability of PROMISE (RQ2)

To fulfill the goals of PROMISE, we evaluate the simplicity of our DSL by conducting two different user studies, as explained in Sec. 3. The first study was a preliminary evaluation that triggered important refinement of the language and the tool, so we only focus on the qualitative data provided as feedback by the participants after the study. The obtained results from the second study and the collected feedback from both studies are discussed in the remainder of this section.

To improve PROMISE and its implementation, we requested feedback from the participants in the form of open-ended questions in the first study. For instance, we: 1) reduced the number of required Eclipse instances, 2) embedded framework functionalities in scripts, 3) created a better grouping in the drag and drop palette 4) extended the information related to each operator in their labels, and 5) developed a wizard to guide the user in the first steps of the mission specification. We also learned from this study that the participants were not confident in the correctness of their missions.

For the second study, we strove to understand what elements of PROMISE could be perceived as error-prone or make the participants less confident in their solutions. With this in mind, we designed the study as explained in Sec. 3 and asked the participants to fill in a questionnaire. The responses to such questionnaire are represented in Fig. 6 and Fig. 7.

Figure 6 shows the answers of the participants to the statements "During the experiment, after the clarifications from the instructors, the semantics of the operator *X* was clear to me"—only the operators that were part of the experiment are present in the list—and to "The mission specification using multiple operators was simple" (last item).

Discussion. From the answers, we conclude that after a thorough training the semantics of all the operators were clear. Also, although the mission of **B2** was perceived as complex, its specification resulted simple for the participants—from an open-ended question of the questionnaire we learned that the main challenge for the participants was to remember the semantics of each operator and pattern.

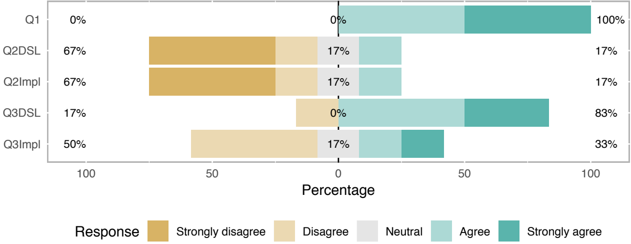

Figure 7. Second study's questionnaire: participants' satisfaction. Q1: I am confident that my solutions are correct. Q2: Writing mission specifications with the [DSL/current tool implementation] is error-prone. Q3: The user-friendliness of the [DSL/current tool implementation] is satisfactory.

Figure 7 shows the answers regarding the participants' satisfaction with PROMISE after the experiment. All the participants agreed and strongly agreed with Q1. To understand the difference on the participant's perception between the DSL as a language and its current tool support we split Q2 and Q3 into two questions each. Q2 measures the users' perception of how error-prone the DSL and the tool support were. Among all the participants, 17% found the language and its implementation error-prone. With Q3 we assess whether the participants perceived PROMISE and its implementation as user-friendly. The user-friendliness of the DSL was considered satisfactory by 83% of the participants while 33% considered the user-friendliness of the implementation satisfactory.

Discussion. As said before, the users were able to validate their solutions in two ways, which may explain the general agreement for Q1. Most likely, due to the changes introduced after the first study, most of the participants now perceived both the DSL and its implementation not error-prone (Q2). Q3 shows a discrepancy on the participants' satisfaction between PROMISE and its implementation: while the DSL was in general considered user-friendly, its implementation was not considered as equally satisfactory.

Four open-ended questions were included to collect qualitative data from the participants and be able to address the problem denoted by Q3.

- "How difficult was the mission specification using multiple operators? Please, elaborate."
- "What was your strategy for defining missions with PROMISE?"
- "What were your main challenges?"
- "Suggestions for improvement."

Discussion. According to the open-ended questionnaire's responses, the solutions for **B1** were rather straightforward with the use of the provided wizard. As expected, the struggles began with **B2** (e.g., analyzing the provided text, trying to identify which operator to use). The most common strategy among the participants was to identify all the elements

from the given text (i.e., locations, events, actions) and then sketch a preliminary mission tree using either Eclipse or the provided notebook. It is important to remark that for both cases they used the PROMISE's syntax. The last step was to create the operator *delegate*'s instantiations and set their properties up. Most of the participants used the generated file that expresses the mission specification in natural English as a preliminary validation of their mission before simulation. The questionnaire's responses indicate the following steps to improve the DSL's current implementation, as for example:

- "I found it a bit cumbersome to define a Delegate Operator for each Action, and assign it to the action."
- "Context-dependent forms would make it easier to select/specify the details [...] Only present the options necessary for a specific operator."
- "The biggest challenge was the lack of feedback on easy-to-neglect errors in the models"

Since the comments did not affect the language but only the implementation, the two first problems remain as future work. The rest of the problems have been already addressed.

All the participants of the second study were able to completely solve the three proposed blocks and validate them using the generated natural language file within the expected time frame (30 minutes for each block). They were also able to validate at least the first two blocks through simulation within the time frame. Furthermore, half of the participants could validate their solutions for **B2** in simulation within the time frame. The time the participants expended for specifying and validating **B1** ranges from 22 to 28 minutes, that is, between 5,5 and 7 minutes for each mission as an average. The time for **B2** ranges from 22 to 30 minutes.

Discussion. To the best of our knowledge, the time invested for mission specification by the user is not usually discussed in scientific papers. However, we consider it a good metric to evaluate the simplicity and user-friendliness of our language and supporting tool. We consider that the time invested in mission specification *and validation* are remarkably low for users without robotics expertise. The results resemble the positive perception of the participants to the second user study and corroborate our claim for RQ2.

6.3 Threats to Validity

We use the standard categorization by Wohlin et al. [59] to discuss the threats to validity to our work.

Internal Validity. Selection bias may be a potential threat since participants of the conducted user studies have or are working on a Ph.D in computer science (as described in Sec. 3), while we claim that our DSL should be usable by users that might lack knowledge on formal methods and temporal logic or programming languages. Yet, we worked with two groups of participants from two different universities, who were not experts in robotics nor had previous knowledge of how to use PROMISE. Continuing with the Design Science research approach by performing additional user studies, including people from different backgrounds, is a valuable future work. Experimenter bias is a threat to our study, and to mitigate it each study in each university was conducted by a different trainer.

Construct Validity. A threat to construct validity is that some specified missions are only executed in simulation. However, part of the experimentation has been also performed with real robots. Performing experiments with further real robots is subject to our future work.

External Validity. A threat to external validity is that our experiments include only groups of one or two robots, which is not a sizable number for multi-robot coordination. Moreover, in the validated multi-robot missions we use the same robotic model and we aim to support a heterogeneous group of robots. On the other hand, the three different robotic models we use for validation differ substantially in their functionalities. Another possible threat to external validity is replication. Conducting additional user studies, including people from different backgrounds, is valuable future work.

Conclusion Validity. The number of participants of both user studies might not be enough to be expressive. We mitigate this low statistical power by conducting two studies, also conducting more studies is valuable future work.

7 Conclusion

In this paper, we introduced PROMISE, a DSL conceived to support roboticists for the effective and user-friendly specification of multi-robot missions. PROMISE is integrated into a software framework that supports the specification, compilation, and interpretation of missions. We developed our DSL striving to maximize its simplicity while keeping its expressiveness and enabling a rigorous and precise specification. We validated our research by performing experiments in simulation and in real-world scenarios (including missions existing in literature) and by conducting two user studies.

As future work, we plan: 1) refinements in the current implementation of the DSL to solve some of the problems stated by the participants of the conducted user studies; 2) refinements of the DSL in order to mitigate its current limitation: create support for run-time changes to a mission specification; 3) to conduct further user studies with participants with different expertise to asses the simplicity of PROMISE and to collect users' feedback; and 4) to investigate the optimal ways for supporting the synchronization among robots. We will also experiment and test our language with other robotic models. As a starting point, we will conduct experiments with a robotics manufacturer (PAL) and a multinational company (BCAI) doing research in the robotics domain.

Acknowledgements

Research partly supported by the EU H2020 Research and Innovation Prog. under GA No. 731869 (Co4Robots).

References

[1] 2004. PAL Robotics. http://pal-robotics.com/.

[2] 2017. Bosch Center of AI. https://www.bosch-ai.com/.

[3] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. 2015. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *Transactions on Software Engineering* (2015).

[4] Jonathan Bohren and Steve Cousins. 2010. The SMACH high-level executive [ROS news]. *IEEE Robotics & Automation Magazine* 17, 4 (2010), 18–20.

[5] Darko Bozhinoski, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Massimo Tivoli. 2015. FLYAQ: Enabling Non-expert Users to Specify and Generate Missions of Autonomous Multicopters. In *International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society.

[6] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. 2013. The BRICS Component Model: A Model-based Development Paradigm for Complex Robotics Software Systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*. ACM, New York, NY, USA, 1758–1764. https://doi.org/10.1145/2480362.2480693

[7] Miguel Campusano and Johan Fabry. 2017. Live robot programming: The language, its implementation, and robot API independence. *Science of Computer Programming* 133 (2017), 1–19.

[8] Federico Ciccozzi, Davide Di Ruscio, Ivano Malavolta, and Patrizio Pelliccione. 2016. Adopting MDE for Specifying and Executing Civilian Missions of Mobile Multi-Robot Systems. *Journal of IEEE Access* (2016).

[9] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. 1999. NuSMV: A new symbolic model verifier. In *International conference on Computer Aided Verification (CAV)*. Springer, 495–499.

[10] Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Massimo Tivoli. 2016. Automatic Generation of Detailed Flight Plans from High-level Mission Descriptions. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. ACM.

[11] Patrick Doherty, Fredrik Heintz, and Jonas Kvarnström. 2013. High-level mission specification and planning for collaborative unmanned aircraft systems using delegation. *Unmanned Systems* 1, 01 (2013), 75–119.

[12] Patrick Doherty, Fredrik Heintz, and David Landén. 2012. A Distributed Task Specification Language for Mixed-Initiative Delegation. In *Principles and Practice of Multi-Agent Systems*, Nirmit Desai, Alan Liu, and Michael Winikoff (Eds.). Springer Berlin Heidelberg.

[13] Swaib Dragule, Bart Meyers, and Patrizio Pelliccione. 2017. A Generated Property Specification Language for Resilient Multirobot Missions. In *Software Engineering for Resilient Systems*, Alexander Romanovsky and Elena A. Troubitsyna (Eds.). Springer International Publishing, Cham, 45–61.

[14] Eclipse. 2006. Xtext. https://www.eclipse.org/Xtext/.

[15] Georgios E Fainekos, Antoine Girard, Hadas Kress-Gazit, and George J Pappas. 2009. Temporal logic motion planning for dynamic robots. *Automatica* 45, 2 (2009), 343–352.

[16] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. 2010. LTLMoP: Experimenting with language, temporal logic and robot control. In *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 1988–1993.

[17] Sergio García, Claudio Menghi, Patrizio Pelliccione, Thorsten Berger, and Rebekka Wohlrab. 2018. An Architecture for Decentralized, Collaborative, and Autonomous Robots. In *International Conference on Software Architecture (ICSA)*.

[18] Sergio García, Patrizio Pelliccione, Claudio Menghi, Thorsten Berger, and Tomas Bures. 2019. PROMISE Implementation. https://github.com/SergioGarG/PROMISE_implementation.

[19] Sebastian Götz, Max Leuthäuser, Jan Reimann, Julia Schroeter, Christian Wende, Claas Wilke, and Uwe Aßmann. 2012. A Role-Based Language for Collaborative Robot Applications. In *Leveraging Applications of Formal Methods, Verification, and Validation*. Springer Berlin Heidelberg.

[20] Meng Guo and Dimos V. Dimarogonas. 2015. Multi-agent plan reconfiguration under local LTL specifications. *The International Journal of Robotics Research* 34, 2 (2015), 218–235.

[21] Meng Guo, Karl H Johansson, and Dimos Dimarogonas. 2013. Revising motion planning under linear temporal logic specifications in partially known workspaces. In *International Conference on Robotics and Automation*.

[22] David Harel and Michal Politi. 1998. *Modeling reactive systems with statecharts: the STATEMATE approach*. McGraw-Hill, Inc.

[23] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. 2004. Design Science in Information Systems Research. *MIS Q.* (2004).

[24] Gerard J. Holzmann. 2002. The Logic of Bugs. In *Symposium on Foundations of Software Engineering (SIGSOFT '02/FSE-10)*.

[25] Louis Hugues and Nicolas Bredeche. 2006. Simbad: an autonomous robot simulation package for education and research. In *International Conference on Simulation of Adaptive Behavior*. Springer, 831–842.

[26] D. Isla. 2005. Handling Complexity in the Halo 2 AI. In *In Game Developers Conference*.

[27] Markus Klotzbücher and Herman Bruyninckx. 2012. Coordinating robotic tasks and systems with rFSM statecharts. (2012).

[28] Nathan Koenig and Andrew Howard. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, Vol. 3. IEEE, 2149–2154.

[29] Hadas Kress-Gazit. 2011. Robot challenges: Toward development of verication and synthesis techniques. *IEEE Robotics & Automation Magazine* 18, 4 (2011), 108–109.

[30] Douglas C MacKenzie, Jonathan M Cameron, and Ronald C Arkin. 1995. Specification and execution of multiagent missions. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[31] Stéphane Magnenat, Philippe Rétornaz, Michael Bonani, Valentin Longchamp, and Francesco Mondada. 2011. ASEBA: A Modular Architecture for Event-Based Control of Complex Robots. *IEEE/ASME Transactions on Mechatronics* 16, 2 (April 2011), 321–329. https://doi.org/10.1109/TMECH.2010.2042722

[32] Shahar Maoz and Jan Oliver Ringert. 2015. GR(1) Synthesis for LTL Specification Patterns. In *Foundations of Software Engineering (FSE)*. ACM, 96–106.

[33] Shahar Maoz and Jan Oliver Ringert. 2019. Spectra: A Specification Language for Reactive Systems. *arXiv preprint arXiv:1904.06668* (2019).

[34] Mauricio Matamoros, Caleb Rascon, Justin Hart, Dirk Holz, and Loy van Beek. 2018. RoboCup@Home 2018: Rules and Regulations. http://www.robocupathome.org/rules/2018_rulebook.pdf.

[35] Claudio Menghi, Sergio García, Patrizio Pelliccione, and Jana Tumova. 2018. Multi-Robot LTL Planning Under Uncertainty. In *International Symposium on Formal Methods (FM)*.

[36] Claudio Menghi, Christos Tsigkanos, Thorsten Berger, and Patrizio Pelliccione. 2019. PsALM: Specification of Dependable Robotic Missions. In *International Conference on Software Engineering (ICSE): Companion Proceeedings*.

[37] Claudio Menghi, Christos Tsigkanos, Thorsten Berger, Patrizio Pelliccione, and Carlo Ghezzi. 2018. Property specification patterns for robotic missions. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 434–435.

[38] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. 2014. A survey on domain-specific languages in robotics. In *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 195–206.

[39] James L. Peterson. 1977. Petri Nets. *ACM Comput. Surv.* 9, 3 (Sept. 1977), 223–252. https://doi.org/10.1145/356698.356702

[40] Emmanuel Pot, Jérôme Monceaux, Rodolphe Gelin, and Bruno Maisonnier. 2009. Choregraphe: a graphical tool for humanoid robot programming. In *RO-MAN 2009-The 18th IEEE International Symposium on Robot and Human Interactive Communication*. IEEE, 46–51.

[41] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. *ICRA workshop on open source software* 3, 3.2 (2009), 5.

[42] European Union's Horizon 2020 research and innovation programme. 2017. Co4Robots. http://www.co4robots.eu/.

[43] PAL Robotics. 2015. TIAGo. http://tiago.pal-robotics.com/.

[44] C. Schlegel, T. Hassler, A. Lotz, and A. Steck. 2009. Robotic software systems: From code-driven to model-driven designs. In *Advanced Robotics, 2009. ICAR 2009. International Conference on*.

[45] Douglas C. Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. *Computer* 39, 2 (Feb. 2006), 25–31. https://doi.org/10.1109/MC.2006.58

[46] Benjamin Schwartz, Ludwig Nägele, Andreas Angerer, and Bruce A. MacDonald. 2014. Towards a graphical language for quadrotor missions. *CoRR* (2014).

[47] Daniel Castro Silva, Pedro Henriques Abreu, Luís Paulo Reis, and Eugénio Oliveira. 2014. Development of a Flexible Language for Mission Description for Multi-robot Missions. *Inf. Sci.* (2014).

[48] SPARC. 2016. Robotics 2020 Multi-Annual Roadmap. https://eu-robotics.net/sparc/upload/about/files/H2020-Robotics-Multi-Annual-Roadmap-ICT-2016.pdf.

[49] S. Srinivas, R. Kermani, K. Kim, Y. Kobayashi, and G. Fainekos. 2013. A graphical language for LTL motion and mission planning. In *2013 IEEE International Conference on Robotics and Biomimetics*. https://doi.org/10.1109/ROBIO.2013.6739543

[50] Andreas Steck, Alex Lotz, and Christian Schlegel. 2011. Model-driven Engineering and Run-time Model-usage in Service Robotics. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE '11)*. 73–82. https://doi.org/10.1145/2047862.2047875

[51] Georgia Tech. 2006. MissionLab. https://www.cc.gatech.edu/ai/robot-lab/research/MissionLab/.

[52] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. 2013. A new skill based robot programming language using UML/P Statecharts. In *2013 IEEE International Conference on Robotics and Automation*. IEEE, 461–466.

[53] TypeFox. 2011. Xtend. https://www.eclipse.org/xtend/.

[54] Alphan Ulusoy, Stephen L Smith, Xu Chu Ding, Calin Belta, and Daniela Rus. 2011. Optimal multi-robot path planning with temporal logic constraints. In *International Conference on Intelligent Robots and Systems*.

[55] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE, 233–238.

[56] Fei-Yue Wang, Konstantinos J Kyriakopoulos, Athanasios Tsolkas, and George N Saridis. 1991. A Petri-net coordination model for an intelligent mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics* 21, 4 (1991), 777–789.

[57] W. Wei, K. Kim, and G. Fainekos. 2016. Extended LTLvis motion planning interface. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 004194–004199. https://doi.org/10.1109/SMC.2016.7844890

[58] Melonee Wise and Tully Foote. 2010. Turtlebot 2. https://www.turtlebot.com/turtlebot2/.

[59] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[60] Eric M Wolff, Ufuk Topcu, and Richard M Murray. 2013. Automaton-guided controller synthesis for nonlinear systems with temporal logic. In *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 4332–4339.

[61] Vittorio A Ziparo, Luca Iocchi, Daniele Nardi, Pier Francesco Palamara, and Hugo Costelha. 2008. Petri net plans: a formal model for representation and execution of multi-robot plans. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 79–86.