Installation, set-up, and overview of PROMISE and its environment

1 What is already in the Virtual Machine?

- Ubuntu 16.04.5 LTS (Xenial Xerus)
  User: promise, Password: promise
- ROS Kinetic
  - Gazebo (version 7.14.0 http://wiki.ros.org/gazebo_ros_pkgs)
  - Tiago Simulation (PAL packages http://wiki.ros.org/Robots/TIAGo)
  - Internal simulation environment and robotic models (c4r_simulation, developed by Robert Bosch GmbH)
  - Local mission manager (local_mission_manager, https://github.com/SergioGarG/PROMISE_implementation)
  - LTL-based planner (ms2_kth, developed by the KTH Royal Institute of Technology))
  - Communication manager (communication_manager, https://github.com/SergioGarG/PROMISE_implementation)
- Eclipse Oxygen 3A
  - Xtext (https://www.eclipse.org/Xtext/)
  - Sirius (https://www.eclipse.org/sirius/)
  - Xtend (https://www.eclipse.org/xtend/)
- A set of sh scripts, all of them stored in the folder ∼/scripts.

2 How to

2.1 Specify missions with Eclipse

Eclipse is locked to the launch dock (at the left of the desktop). Sirius and Xtext are ready to use in the main instance of Eclipse ("new Missions diagram" and "mission.promise" respectively).

Starting the mission

To specify a mission, the user can make use of a textual and a graphical editor in combination. The textual and graphical editors will be arranged side-by-side, the textual one to the left and the graphical one to the right. While specifying the mission, it is required to save your changes before changing the context (from Xtext to Sirius, or vice-versa). To simplify the mission parameters set-up, we provide a simple wizard, which should be used at the beginning of each mission specification. To use it, in the Model Explorer (at the left) go to Promise→sle2019-artefact and right-click on `mission.promise`. Now click on "Your mission..." (Fig. 1).

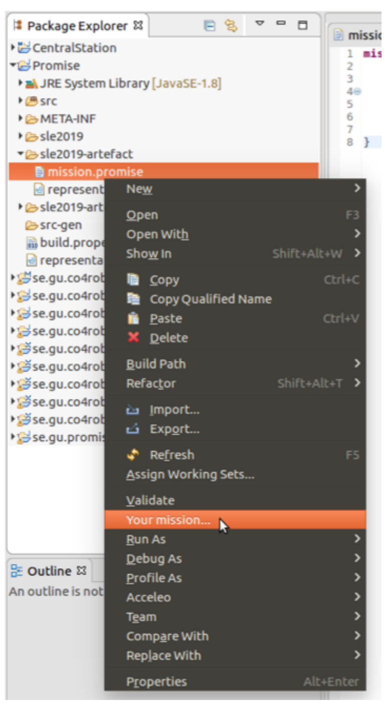


Figure 1: Start the Wizard

The first page of the wizard (Fig. 2) requests the user to fill in the boxes with the names of the robots to be used and the environment's locations (each item separated by commas). Both text boxes to continue. When ready, press Next.

In the second window (Fig. 3) the user can specify the conditions of the mission. Events are

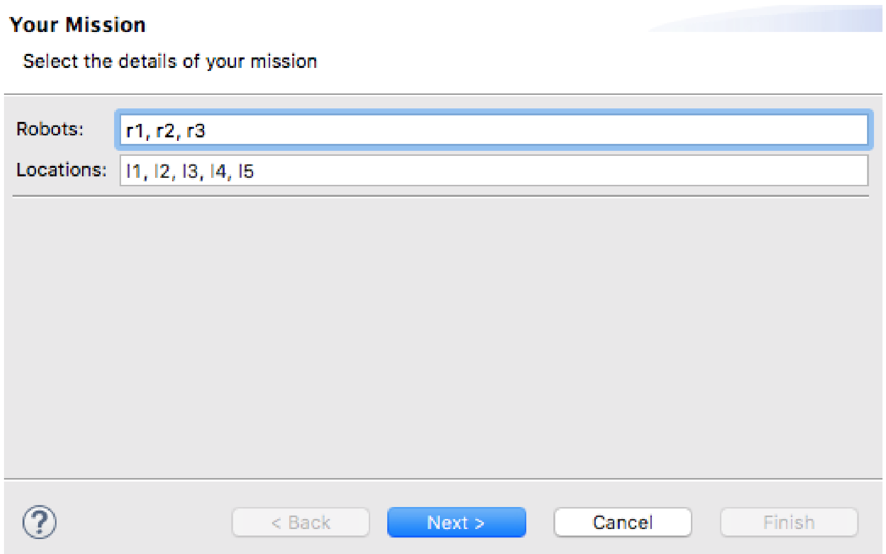


Figure 2: Wizard page 1

conditions not caused by the target robot—e.g., "a human enters the room" or "received message requesting help". Actions are conditions directly performed by the target robot—e.g., "the robot waves" or "the robot grabs a coffee". The user might leave these text boxes empty and just click on finish if no events or actions are required for the mission. **The user might need to double-click on `mission.promise` on it in the Model Explorer at the left to refresh its content.**

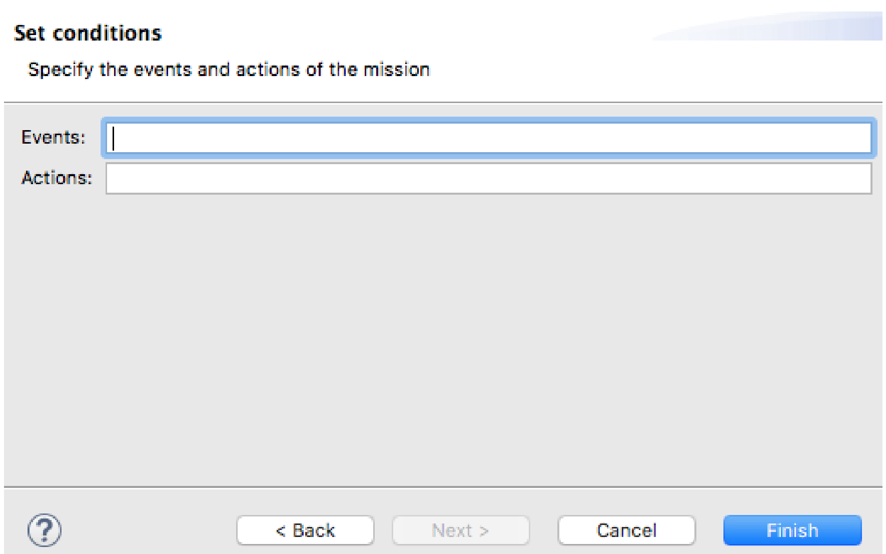


Figure 3: Wizard page 2

Adding elements to the graphical syntax

PROMISE allows defining missions using two different syntaxes, as defined in the paper. Nevertheless, this tutorial focuses on the graphical syntax since we consider it a better approach for first-time users.

Once the mission skeleton is ready the user may start working on the mission specification. Operators may be dragged and dropped from the Palette (Fig. 4) at the rightmost side of the Eclipse interface. To interconnect operators the user must use the *assignOperator* linking tool from the palette. Select the link by clicking on it, then click on the parent operator (the source) and then on the child (the target).

Tasks (e.g., visit a set of locations) are always associated with the operator *delegate*. To associate a delegate operator to a task drag and drop a task from the palette on a delegate operator.

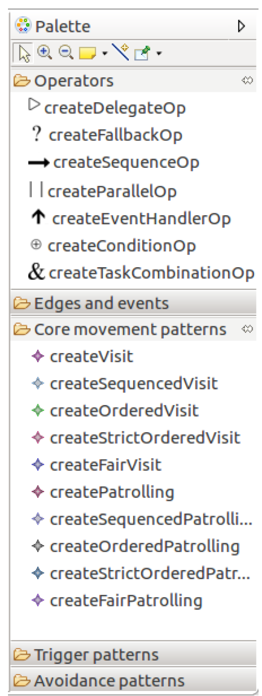


Figure 4: Palette

The graphical syntax does not implement nodes that represent robots, locations, events, or actions. Instead, such concepts are instantiating by configuring the appropriate operators, as described in the following.

Configuring your operators

Some operators need to be configured, i.e., they need the user to set some properties. The operator parallel takes as input references to a set of robots and assigns one robot to each of its branches (each child). Therefore, tasks defined in each branch are automatically delegated to a specific robot. To configure an operator, click on it in the graphical syntax and open the Properties tab (see Fig. 5). Click on the three dots button and choose the robots to be used in the mission. Robots are assigned to branches based in the order specified when configuring the operator parallel (see Fig. 5). In the case of the figure, the first branch will be assigned to *r1* and the second to *r2*. Labels in the connection edges will automatically display the associated robot (see Fig. 8).

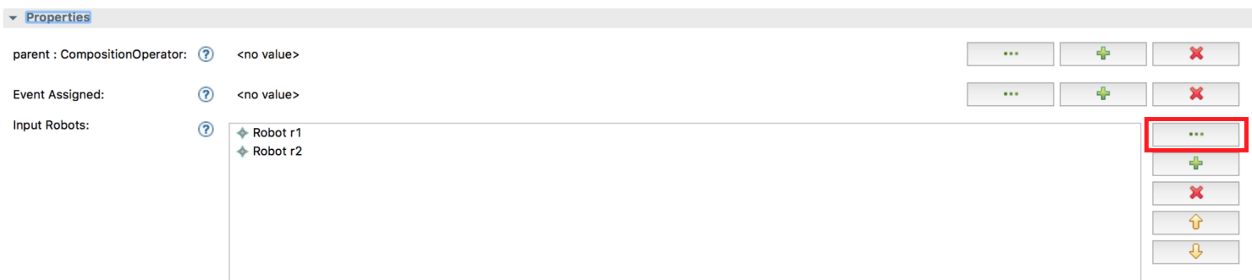


Figure 5: Configure your operators

Delegate operators require the definition of locations or actions, depending on the assigned task (e.g., a *Patrolling* task requires the specification of a set of locations). From the Properties tab, add actions or locations clicking on the three dots button next to Input Action or Input Locations, respectively.

As shown in Table 4, the operators that assign children to events (*condition* and *event handler*) use an intermediate item, which represents the assigned event (see Fig. 8). These items (named *eventAssigned*) may be dragged and dropped from the palette. The eventAssigned item must be configured by specifying the target event (similar to the parametrization explained for the operator delegate), as shown in Fig. 7. There is a special type of link (*assignEvent*) to link operators (the parent, i.e., the source) to an event. To link the event to another operator (the child, i.e., the target), the user must use the link type (*assignOperator*. Fig. 8 shows a simple mission example. This example represents how to associate an event to a specific branch of an operator. In this case, event *e1* triggers the execution of action *a2* (second branch of the event handler operator) if it occurs.

Once you are finished with the mission specification save your changes so a set of files to be sent to the simulator are generated. These files (one per robot) contain the mission defined with our Intermediate Language (please, see the definition in the paper). Saving your changes will also trigger the generation of one readme file for each robot (i.e., one for each mission), containing an explanation of the mission in natural English (it will also be printed in the Eclipse's console). **Remember that the files are automatically generated and stored in `PromisePlugin/src-gen`.**

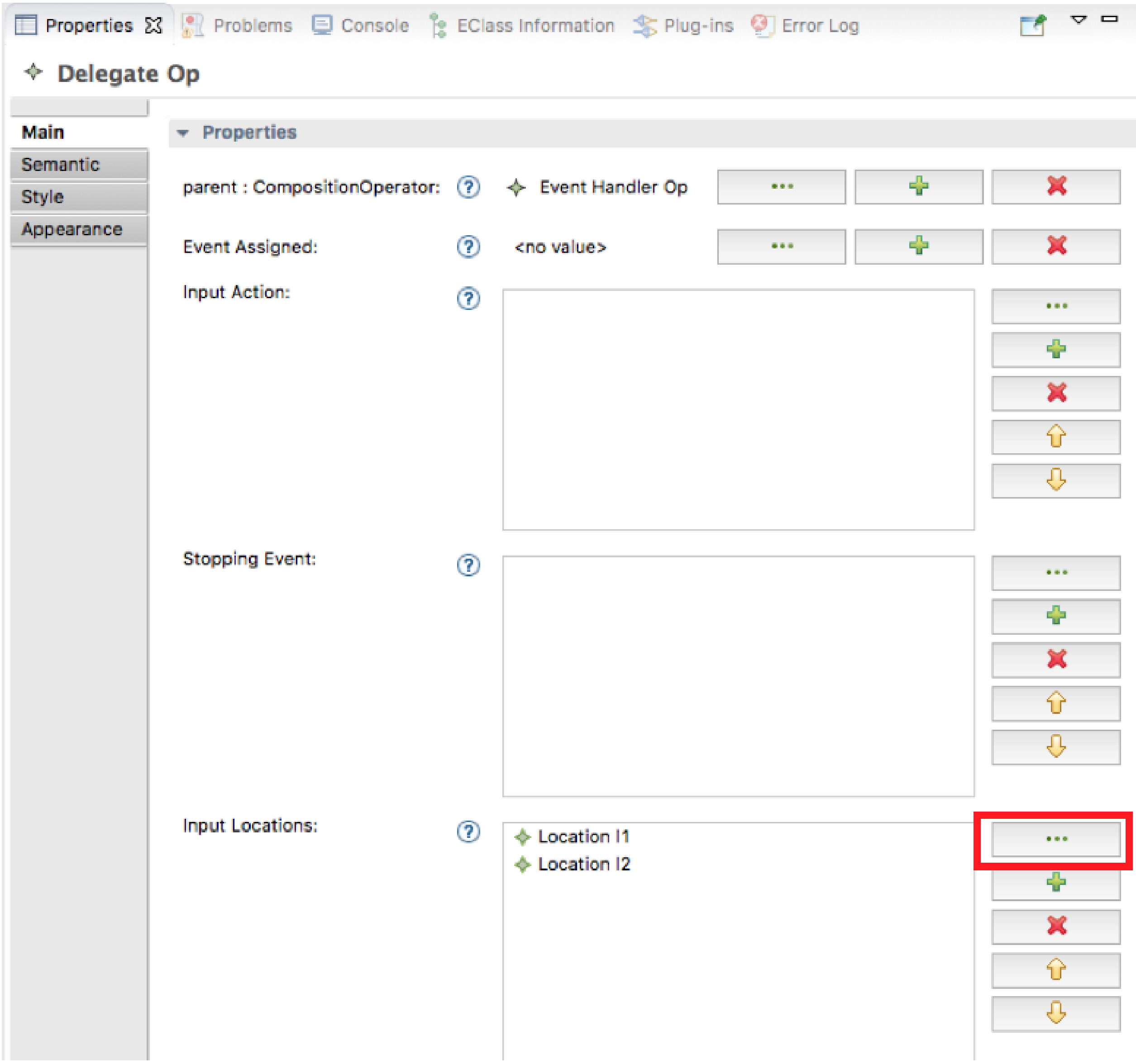


Figure 6: Configure your operators

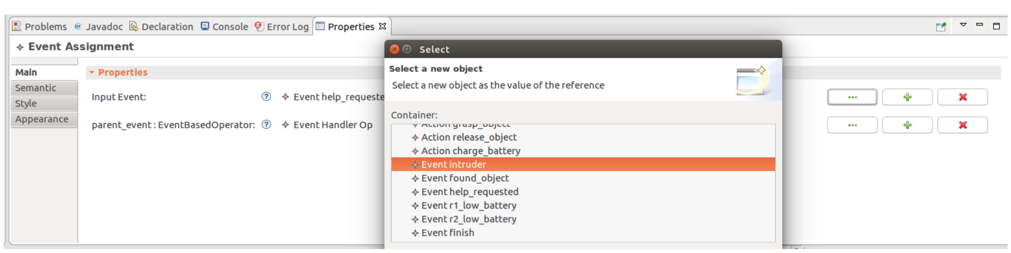


Figure 7: How to specify the target event

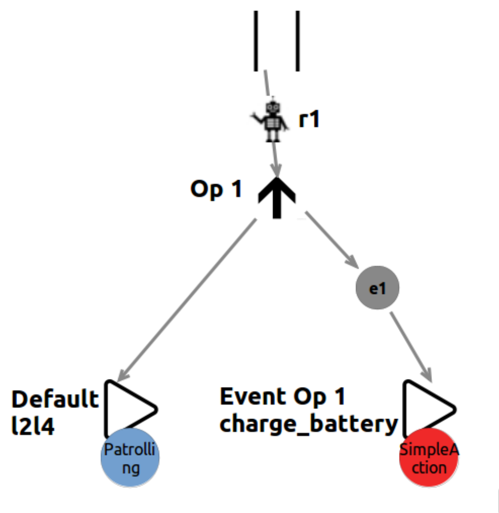


Figure 8: Mission example

2.2 Send missions

To send a mission, in Eclipse, click on Run External Tools → sendMission (see Fig. 9). This instruction sends to the simulated robotic team the last specified mission, i.e., the last mission from which intermediate language files were generated. The instance of PROMISE contained in the provided VM is configured to work in simulation and therefore the missions to the local machine's IP. In order to work with real robots connected to the same network the computer running the VM is, it is sufficient to specify the robots' IPs in a Java file: `/home/promise/catkin_ws/src/PROMISE_implementation/CentralStation/workspace/CentralStation/src/main/java/se/gu/CentralStation/ReadWithScanner.java`

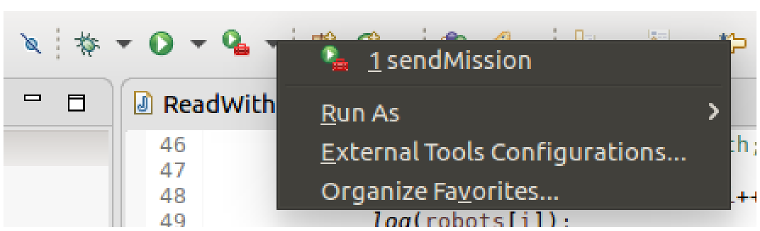


Figure 9: Send the mission.

Each time a mission is sent to a robot (either correctly or failed) you will receive an informative message. You will need to click "OK" to proceed with the next mission.

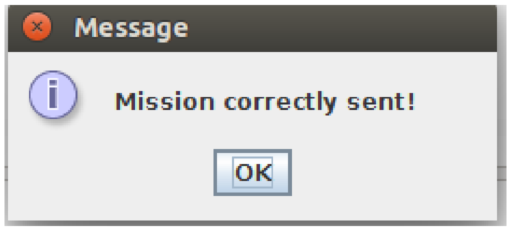


Figure 10: Mission received!

2.3 Set the simulated environment

We provide bash files to simplify the process of launching the experiment (allocated in the home folder). **Remember that all the scripts are in the ∼/scripts folder**. To launch all the required components of the underlying framework, it is enough to execute a sole script. The script requires two arguments; the first can take the values of 1 or 0. Passing 0 as an input prevents the Gazebo's GUI to be launched (Rviz will be launched) Passing 1 as the first argument makes the script launch both interfaces. The second argument specifies the number of robots to be launched (from 1 to 3, named r1, r2, and r3, respectively).

```
bash set_environment.sh GUI_ON_OFF N_robots
```

Example. Let's suppose we want to simulate two robots. Please, open a new terminal and type (from the scripts folder):

```
bash set_environment.sh 0 2
```

2.4 Send events

Events are described by string-typed messages. In this simulated scenario, events are manually sent using the terminal. We provide a script to simplify this step. The script takes as an argument a string, which name must match the specified in Eclipse.

```
bash send_event.sh 'your_event'
```

2.5 Runtime

To illustrate the artifact we provide the set of missions used in the second user study we conducted. In particular, the environment used in the simulation is shown in Fig. 11 (Rviz GUI, which is always on regardless of the parameters of the set_environment script).

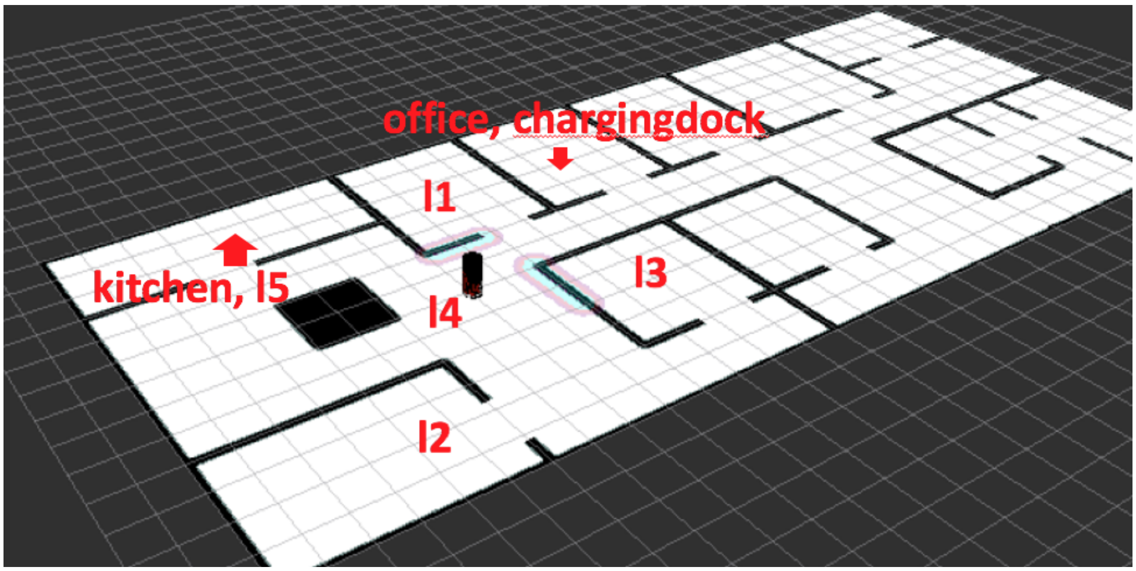


Figure 11: Locations in the environment.

In the current implementation of PROMISE, we rely on Ubuntu terminals to communicate to the user information regarding the current state of the mission. So, for instance, the two different tasks specified in the mission shown in Figure 8 will e prompt in the terminal as shown in Figure 12 and Figure 13 during their execution. Figure 14 represents the message printed for the user, which symbolizes the simulation action of charging the robot's battery.

```
Current task [] (<> (l2) && <> (l4)) ( False ) of mission ['[] (<> (l2) && <> (l4))'] (branch 1 task 0 )
```

Figure 12: Terminal message of a task of patrolling.

```
--------------------------------------------------
Current task X (charge_battery) ( True ) of mission ['X (charge_battery)'] (branch 2 task 0 )
```

Figure 13: Terminal message of a task of an action.

```
!!!-----------
Charging battery!
-----------!!!
```

Figure 14: Simulation of an action.

3 Mission specification patterns & DSL operators

This section contains useful and technical information regarding the robotic patterns and operators. Use it as a cheatsheet!

Table 1 Avoidance patterns.

	Description	**Example**
Past avoidance	A condition has been fulfilled in the past.	If the robot enters location l_1, then it should have not visited location l_2 before. The trace $l_3 \rightarrow l_4 \rightarrow l_1 \rightarrow l_2 \rightarrow l_4 \rightarrow l_3 \rightarrow (l_2 \rightarrow l_3)^\omega$ satisfies the mission requirement since location l_2 is not entered before location l_1.
Global avoidance	An avoidance condition globally holds throughout the mission.	The robot should avoid entering location l_1. Trace $l_3 \rightarrow l_4 \rightarrow l_3 \rightarrow l_2 \rightarrow l_4 \rightarrow l_3 \rightarrow (l_3 \rightarrow l_2 \rightarrow l_3)^\omega$ satisfies the mission requirement since the robot never enters l_1.
Future avoidance	After the occurrence of an event, avoidance has to be fulfilled.	If the robot enters l_1, then it should avoid entering l_2 in the future. The trace $l_3 \rightarrow l_4 \rightarrow l_3 \rightarrow l_1 \rightarrow l_4 \rightarrow l_3 \rightarrow (l_3 \rightarrow l_2 \rightarrow l_3)^\omega$ does not satisfy the mission requirement since l_2 is entered after l_1.
Upper Rest. Avoidance	A restriction on the maximum number of occurrences is desired.	A robot has to visit l_1 at most 3 times. The trace $l_1 \rightarrow l_4 \rightarrow l_1 \rightarrow l_3 \rightarrow l_1 \rightarrow l_4 \rightarrow l_1 \rightarrow (l_3)^\omega$ violates the mission requirement since l_1 is visited four times. The trace $l_4 \rightarrow l_3 \rightarrow l_1 \rightarrow l_2 \rightarrow l_4 \rightarrow (l_3)^\omega$ satisfies the mission requirement.
Lower Rest. Avoidance	A restriction on the minimum number of occurrences is desired.	A robot should enter location l_1 at least 3 times. The trace $l_4 \rightarrow l_3 \rightarrow l_2 \rightarrow l_2 \rightarrow l_4 \rightarrow (l_3)^\omega$ violates the mission requirement since location 1 is never entered. The trace $l_1 \rightarrow l_4 \rightarrow l_3 \rightarrow l_1 \rightarrow l_4 \rightarrow l_1 \rightarrow (l_3)^\omega$ satisfies the mission requirement.
Exact Rest. Avoidance	The number of occurrences desired is an exact number.	A robot must enter location l_1 exactly 3 times. The trace $l_4 \rightarrow l_3 \rightarrow l_2 \rightarrow l_2 \rightarrow l_4 \rightarrow (l_3)^\omega$ violates the mission requirement. The trace $l_1 \rightarrow l_4 \rightarrow l_3 \rightarrow l_1 \rightarrow l_4 \rightarrow l_1 \rightarrow (l_3)^\omega$ satisfies the mission requirement since location l_1 is entered exactly 3 times.

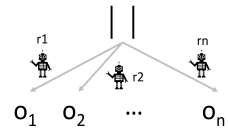
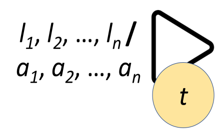
Table 2 Core movement patterns.

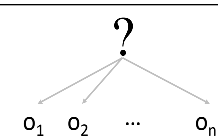
	Description	**Example**
Visit	Visit a set of locations in an unspecified order.	Locations l_1, l_2, and l_3 must be visited. $l_1 \to l_4 \to l_3 \to l_1 \to l_4 \to l_2 \to (l_{\#})^\omega$ is an example trace that satisfies the mission requirement.
Sequenced Visit	Visit a set of locations in sequence, one after the other.	Locations l_1, l_2, l_3 must be covered following this sequence. The trace $l_1 \to l_4 \to l_3 \to l_1 \to l_4 \to l_2 \to (l_{\#\setminus 3})^\omega$ violates the mission since l_3 does not follow l_2. The trace $l_1 \to l_3 \to l_1 \to l_2 \to l_4 \to l_3 \to (l_{\#})^\omega$ satisfies the mission requirement.
Ordered Visit	The sequenced visit pattern does not forbid to visit a successor location before its predecessor, but only that after the predecessor is visited the successor is also visited. Ordered visit forbids a successor to be visited before its predecessor.	Locations l_1, l_2, l_3 must be covered following this order. The trace $l_1 \to l_3 \to l_1 \to l_2 \to l_3 \to (l_{\#})^\omega$ does not satisfy the mission requirement since l_3 preceeds l_2. The trace $l_1 \to l_4 \to l_1 \to l_2 \to l_4 \to l_3 \to (l_{\#})^\omega$ satisfies the mission requirement.
Strict Ordered Visit	The ordered visit pattern does not avoid a predecessor location to be visited multiple times before its successor. Strict ordered visit forbids this behavior.	Locations l_1, l_2, l_3 must be covered following the strict order l_1, l_2, l_3. The trace $l_1 \to l_4 \to l_1 \to l_2 \to l_4 \to l_3 \to (l_{\#})^\omega$ does not satisfy the mission requirement since l_1 occurs twice before l_2. The trace $l_1 \to l_4 \to l_2 \to l_4 \to l_3 \to (l_{\#})^\omega$ satisfies the mission requirement.
Fair Visit	The difference among the number of times locations within a set are visited is at most one.	Locations l_1, l_2, l_3 must be covered in a fair way. The trace $l_1 \to l_4 \to l_1 \to l_3 \to l_1 \to l_4 \to l_2 \to (l_{\#-\{1,2,3\}})^\omega$ does not perform a fair visit since it visits l_1 three times while l_2 and l_3 are visited once. The trace $l_1 \to l_4 \to l_3 \to l_1 \to l_4 \to l_2 \to l_2 \to l_4 \to (l_{\#\setminus\{1,2,3\}})^\omega$ performs a fair visit since it visits locations l_1, l_2, and l_3 twice.
Patrolling	Keep visiting a set of locations, but not in a particular order.	Locations l_1, l_2, l_3 must be surveilled. The trace $l_1 \to l_4 \to l_3 \to l_1 \to l_4 \to l_2 \to (l_2 \to l_3 \to l_1)^\omega$ ensures that the mission requirement is satisfied. The trace $l_1 \to l_2 \to l_3 \to (l_1 \to l_3)^\omega$ represents a violation, since l_2 is not surveilled.
Sequenced Patrolling	Keep visiting a set of locations in sequence, one after the other.	Locations l_1, l_2, l_3 must be patrolled in sequence. The trace $l_1 \to l_4 \to l_3 \to l_1 \to l_4 \to l_2 \to (l_1 \to l_2 \to l_3)^\omega$ satisfies the mission requirement since globally any l_1 will be followed by l_2 and l_2 by l_3. The trace $l_1 \to l_4 \to l_3 \to l_1 \to l_4 \to l_2 \to (l_1 \to l_3)^\omega$ violates the mission requirement since after visiting l_1, the robot does not visit l_2.
Ordered Patrolling	Sequence patrolling does not forbid to visit a successor location before its predecessor. Ordered patrolling ensures that (after a successor is visited) the successor is not visited (again) before its predecessor.	Locations l_1, l_2, and l_3 must be patrolled following the order l_1, l_2, and l_3. The trace $l_1 \to l_4 \to l_3 \to l_1 \to l_4 \to l_2 \to (l_1 \to l_2 \to l_3)^\omega$ violates the mission requirement since l_3 precedes l_2. The trace $l_1 \to l_1 \to l_2 \to l_4 \to l_4 \to l_3 \to (l_1 \to l_2 \to l_3)^\omega$ satisfies the mission requirement
Strict Ordered Patrolling	The ordered patrolling pattern does not avoid a predecessor location to be visited multiple times before its successor. Strict Ordered Patrolling ensures that, after a predecessor is visited, it is not visited again before its successor.	Locations l_1, l_2, l_3 must be patrolled following the strict order l_1, l_2, and l_3. The trace $l_1 \to l_4 \to l_1 \to l_2 \to l_4 \to l_3 \to (l_1 \to l_2 \to l_3)^\omega$ violates the mission requirement since l_1 is visited twice before l_2. The trace $l_1 \to l_4 \to l_2 \to l_4 \to l_3 \to (l_1 \to l_2 \to l_3)^\omega$ satisfies the mission requirement.
Fair Patrolling	Keep visiting a set of locations and ensure that the difference among the number of times locations within a set are visited is at most one.	Locations l_1, l_2, and l_3 must be fair patrolled. The trace $l_1 \to l_4 \to l_3 \to l_1 \to l_4 \to l_2 \to (l_1 \to l_2 \to l_1 \to l_3)^\omega$ violates the mission requirements since the robot patrols l_1 more than l_2 and l_3. The trace $l_1 \to l_4 \to l_3 \to l_4 \to l_2 \to l_4 \to (l_1 \to l_2 \to l_3)^\omega$ satisfies the mission requirement since locations l_1, l_2, and l_3 are patrolled fairly.

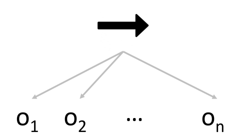
Table 3 Trigger patterns.

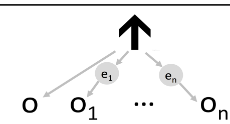
	Description	**Example**
Inst. Reaction	The occurrence of a stimulus instantaneously triggers a counteraction.	When location l_2 is reached the action a must be executed. The trace $l_1 \to l_3 \to \{l_2, a\} \to \{l_2, a\} \to l_4 \to (l_3)^\omega$ satisfies the mission requirement since when location l_2 is entered condition a is performed. The trace $l_1 \to l_3 \to l_2 \to \{l_1, a\} \to l_4 \to (l_3)^\omega$ does not satisfy the mission requirement since when l_2 is reached a is not executed.
Delayed Reaction	The occurrence of a stimulus triggers a counteraction some time later	When c occurs the robot must start moving toward location l_1, and l_1 is subsequently finally reached. The trace $l_1 \to l_3 \to \{l_2, c\} \to l_1 \to l_4 \to (l_3)^\omega$ satisfies the mission requirement, since after c occurs the robot starts moving toward location l_1, and location l_1 is finally reached. The trace $l_1 \to l_1 \to \{l_2, c\} \to l_3 \to (l_3)^\omega$ does not satisfy the mission requirement since c occurs when the robot is in l_2, and l_1 is not finally reached.
Prompt Reaction	The occurrence of a stimulus triggers a counteraction promptly, i.e. in the next time instant.	If c occurs l_1 is reached in the next time instant. The trace $l_1 \to l_3 \to \{l_2, c\} \to l_1 \to l_4 \to (l_3)^\omega$ satisfies the mission requirement, since after c occurs l_1 is reached within the next time instant. The trace $l_1 \to l_3 \to \{l_2, c\} \to l_4 \to l_1 \to (l_3)^\omega$ does not satisfy the mission requirement.
Bound Reaction	A counteraction must be performed every time and only when a specific location is entered.	Action a_1 is bound though a delay to location l_1. The trace $l_1 \to l_3 \to \{l_2, c\} \to \{l_1, a_1\} \to l_4 \to \{l_1, a_1\} \to (l_3)^\omega$ satisfies the mission requirement. The trace $l_1 \to l_3 \to \{l_2, c\} \to \{l_1, a_1\} \to \{l_4, a_1\} \to \{l_1, a_1\} \to (l_3)^\omega$ does not satisfy the mission requirement since a_1 is executed in location l_4.
Bound Delay	A counteraction must be performed, in the next time instant, every time and only when a specific location is entered.	Action a_1 is bound to location l_1. The trace $l_1 \to l_3 \to \{l_2, c\} \to \{l_1\} \to \{l_4, 1_1\} \to \{l_1\} \to \{l_4, a_1\} \to (l_3)^\omega$ satisfies the mission requirement. The trace $l_1 \to l_3 \to \{l_2, c\} \to \{l_1\} \to \{l_4, 1_1\} \to \{l_1, a_1\} \to \{l_4\} \to (l_3)^\omega$ does not satisfy the mission requirement.
Wait	Inaction is desired till a stimulus occurs.	The robot remains in location l_1 until condition c is satisfied. The trace $l_1 \to l_3 \to \{l_2, c\} \to l_1 \to l_4 \to (l_3)^\omega$ violates the mission requirement since the robot left l_1 before condition c is satisfied. The trace $l_1 \to \{l_1, c\} \to l_2 \to l_1 \to l_4 \to (l_3)^\omega$ satisfies the mission requirement.
Simple action	A counteraction is performed in the next time instant without requiring any kind of stimulus.	The robot executes the *wave* action in the next time instant.

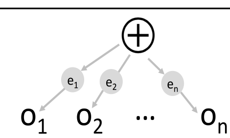
Table 4 Robotic missions specification operators

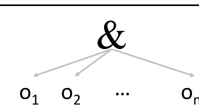
Name	Description	Semantics	Syntax
Parallel $\|\|(r_1,\dots r_n, o_1,\dots,o_n)$	Always the root of the mission. The operators $o_1, o_2, \cdots, o_n$ are executed in parallel, each by a different robot—i.e., assigns one branch to each robot. Returns success when all operators return success, failure otherwise.	$\{\text{res}_1, \text{res}_2, \cdots, \text{res}_n\}$=$\{\text{o}_1, \text{o}_2, \cdots, \text{o}_n\}$ **if** ($\text{res}_1 == \top \wedge \cdots \wedge \text{res}_n == \top$) **then** **return** $\top$ **else return** $\bot$	 parallel{r1(o_1), r2(o_2), ..., rn(o_n)}
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 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 $l_1, l_2, ..., l_n$) / delegate (t actions $a_1, a_2, ..., a_n$)

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. Failure if all operators fail.	**if** ($\{\text{o}_1,\text{o}_2,\cdots,\text{o}_n\} \neq \emptyset$) **then**
		    $res = o_1$;
		    if($res == \bot$) **then**
		        $?(\{o_2, \cdots, o_n\})$
		    else return $\top$
		else return $\bot$
			
			fallback ($o_1, o_2, \dots, o_n$)

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 ($o_1, o_2, \dots, o_n$)

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** $i = 1, ..., n$
		        $resint = o_i$;
		        if($resint == \bot$), **then**
		            return $\bot$
		        $res =$ **resume**(o);
		return res
			
			eventHandler(
			    default(o)
			    except e_1 (o_1)...
			    except e_n (o_n))

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))

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 && ... o_n **if**($res == \top$) **then** **return** $\top$ **else return** $\bot$	 combination(o_1 and o_2 and ... o_n)