BW4T3
Agent-Environment Interface Specification
June 26, 2014

This document describes the *agent-environment interface* for the Blocks World for Teams 3 (BW4T3) environment that allows *players* to control *robots* that are part of this simulated environment.

The environment consists of *rooms* and a so-called *drop zone* which are connected through *corridors*. And *charge zones* and *blockade*s, to recharge robots equipped with batteries, or block the path. See Figure 1 and 2 for an example of a basic map; Figure 1 shows the underlying graph for the map depicted in Figure 2. Coloured *blocks* are placed inside rooms. The colors at the bottom of the graphical interface for the BW4T environment (see Figure 2) specify an *ordered sequence of colors* from left to right. It is the main goal of the robot(s) to collect and pick up blocks from the rooms and deliver and put down blocks with colors matching that order in the drop zone. In the example Figure 2 this means that first a blue block, then a white block, etc. should be delivered to the drop zone.

A robot can only see blocks in a room when it is inside that room and then only can see those blocks. Robots cannot see each other. Access to a room or the drop zone is limited to one robot at a time. That is, once a robot has entered a room or the drop zone, no other robots can enter that space any more. A robot has four basic capabilities: *going to a place* (the labels in Figure 1 denote places), *going to a block* in a room (from any other place), *picking up a block* if the robot stands next to that block, and *putting down a block* at arbitrary places. Blocks disappear from the environment when put down in the hall or in the drop zone. Blocks put down in a room do not disappear.
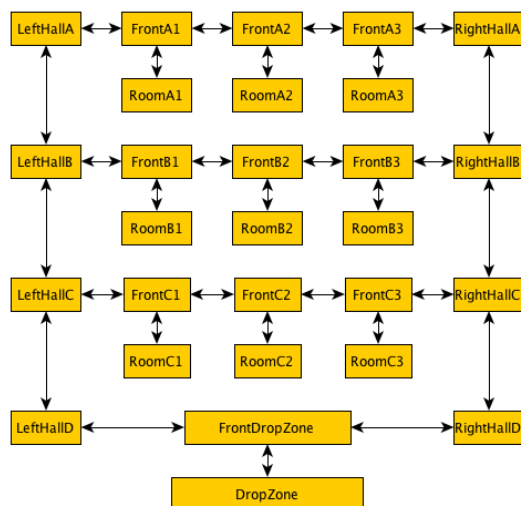

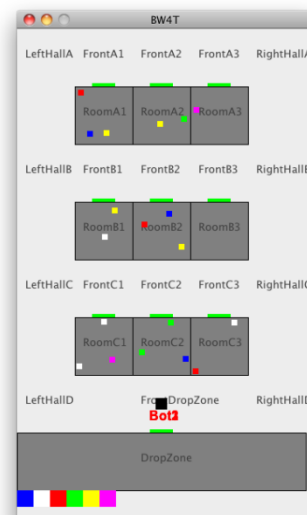**Figure 1. Rooms and connections (default Map1)**

Figure 2. Overview window of the Environment as shown by Repast.

Figure 3 shows the layout of a new map, containing a charge zone and two blockades. The four black squares represent robot spawning points.
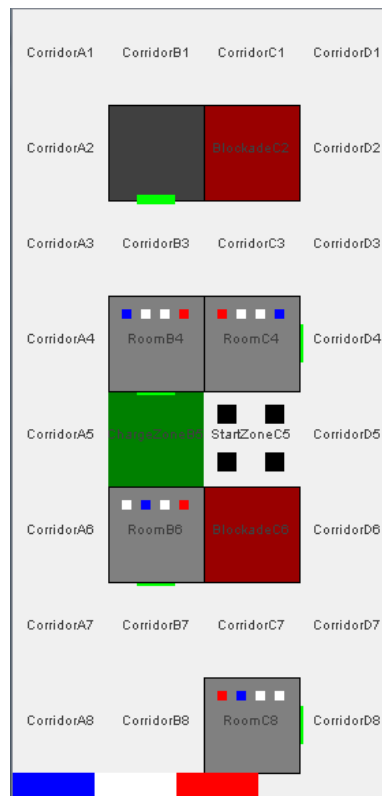

Figure 3. Overview window of a new Environment

Robots can be controlled by *agents* or *humans* (both are referred to as *players*), thereby providing the possibility to investigate *human-agent robot teamwork*. Players can send *messages* to each other. To ensure that agents can understand the messages, a fixed set of messages has been defined. Humans can select these through a GUI, and agents can send them using a predefined action (see below for more on this).

Alongside with robots, e-Partners are also available for use, and can be picked up by *humans*. E-Partners are controlled by GOAL agents, and bring support to their holder. They are represented as yellow triangles on the map, but change their color to green when they are picked up.

Domain Labels

In order to be able to reason about and to represent the BW4T environment, an *ontology* has been designed. An ontology consists among others of *labels* to refer to items in the environment. For the BW4T environment, labels are provided for players, blocks, places, rooms, colors, states, and IDs. These labels will be used as *parameters* for actions, percepts, and messages. They are part of a *language* that allows players to talk about the environment. Below, this language and its labels are introduced and explained. We also introduce notation using brackets (<…>) to refer generically to the *type* of parameters that actions, etc. can have.

Players

- Description: Players, i.e., the humans or agents that steer robots in the BW4T environment, get names (IDs) assigned.
- Parameter: `<PlayerID>`
- Values: Alphanumeric strings starting with a letter.
- Additional explanation: Each player is connected to one simulated robot in the environment (there can also be un-connected simulated robots). Robots can drive around and pick up and drop blocks.

E-Partners

- Description: Entities which help players achieve their goal, and support them throughout the environment, using various
- Parameter: `<EPartnerID>`
- Values: Alphanumeric strings starting with a letter.
- Additional explanation: E-Partners can have different functionalities. The currently available ones are: GPS, which helps the player's navigation, and Forget-me-not, which ensures the player does not forget his or her e-Partner.

Blocks

- Description: Blocks are the things that can be picked up and put down by a robot. If dropped in the right order in the drop zone, the overall goal of delivering a sequence of colored blocks is achieved.
- Parameter: `<BlockID>`
- Values: Any natural number.
- Additional explanation: Block labels enable agents to distinguish between blocks that have the same color.

Places

- Description: Rooms, hallways and the drop zone are places in the BW4T environment.
- Parameter: `<PlaceID>`
- Values: Strings that denote names of places. For example, `FrontDropZone'` is an example of a place name for the area just in front of the `DropZone'`.
- Additional explanation: Figure 1 provides names for different places on that example map.

Rooms

- Description: Rooms are a subclass of places with special properties, i.e., a room can be occupied and rooms only have one way to enter or exit a room.
- Parameter: `<RoomID>`
- Values: Examples, see also Figure 1 and 2, are `RoomA1'`, `RoomA2'`, …, `RoomC3'`, `DropZone'`.
- Additional explanation: Rooms and the drop zone allow only one agent to be in there at any time.

Colors

- Description: Blocks have colors. Blocks should be delivered to the drop zone in an order specified by color, e.g., first deliver a blue block, then a red block, etc.
- Parameter: `<ColorID>`
- Values: `Blue'`, `Cyan'`, `Magenta'`, `Orange'`, `Red'`, `White'`, `Green'`, `Yellow'`, `Pink'`.

State

- Description: A controlled robot can be in one of three different states, *traveling* if moving from one to another location, *collided* if moving failed, and *arrived* if the robot arrived at its target location. Target locations can be specified by means of the `goTo` and `goToBlock` actions described below.
- Parameter: `<State>`
- Values: `collided, arrived, travelling`

ID

- Description: Each object, whether it is a block, robot or room, is assigned a unique numeric id.
- Parameter: `<ObjectID>`
- Values: Any number.
- Additional explanation: Id's are provided, e.g., for locations.

Doors are not provided with names, i.e., they cannot be referred to directly by players. Doors are associated with a *unique* room. Doors have special behavior and act as guards: all doors to a room "close" simultaneously when someone enters the room (from the outside perspective), but from the inside perspective in a sense there are no doors (effectively, you can always leave a room without having to open doors).

Percepts

The predicates introduced are used by agents for representing features of the BW4T environment. Instantiations of these predicates are provided as *percepts* to an agent, if that agent is connected to a robot in the environment. Percepts allow an agent to keep track of some of the things that have changed. Most percepts provide only local information, e.g., when a robot is in a room it can see colored blocks but otherwise it cannot see these blocks.

Four different types of percepts are distinguished, so-called *send once*, *send on change*, *send on change with negation*, and *send always* percepts. We briefly describe each type of percept:

1. **Send once**: these percepts are only sent once, at the start of a simulation when an agent is connected to a robot. For example, map information is only sent once.
2. **Send on change**: this type of percept is only sent initially and in case something changes. For example, if a robot just moved to a particular place it receives the `at(<PlaceID>)` percept and it will receive a new percept if it moves to another place.
3. **Send on change with negation:** These percepts are similar to the "send on change" type of percepts but a negation of an earlier received percept is sent when a change occurs. For example, after entering a room a robot will receive the `in(<RoomID>)` percept and when leaving it will receive the `not(in(<RoomID>))` percept.
4. **Send always**: This type of percept is continuously sent. Sometimes certain conditions must hold as well. For example, the `color(<BlockID>,<ColorID>)` percept is send continuously for each block present in a room while a robot is in that room.

The environment keeps running while percepts are being collected. Therefore, the percepts that the agent receives may actually be outdated at the time he receives them. And more subtly, the percepts can in some cases be inconsistent, for example the occupied percept may not match the own robots location because the robot was not yet in a room when the location percept was collected, while the robot had just entered the room when the occupied percept was collected.

We now proceed with describing the various percepts that are made available to an agent that controls a robot in the BW4T environment. A brief description is given for each predicate used as percept, and an informal "translation", the type of the percept, some additional explanation and an example are also provided. We also indicate when a predicate is used especially for the human GUI that is available for controlling a robot by adding "HumanBot" as type and have been highlighted in yellow. *NB: these "HumanBot" percepts are irrelevant when developing your own agent that controls a robot in BW4T.*

Predicate: `place(<PlaceID>)`

- Description: Information about which places there are in the environment.
- Translation: "`<PlaceID>` is a place."
- Type: Send once.
- Additional explanation: The interface does not provide information as to which places are rooms.
- Example: `place('RoomA1')`.

Predicate: `in(<RoomID>)`

- Description: This player is in `<RoomID>`.
- Translation: "This player is is in `<RoomID>`."
- Type: Send on change with negation.
- Additional explanation: Percept is local, i.e., is sent only to the player for which it holds.
- Example: `in('RoomC1')`

Predicate: `at(<PlaceID>)`

- Description: This player is at `<PlaceID>`.
- Translation: "This player is located at/near `<PlaceID>`."
- Type: Send on change.
- Additional explanation: Percept is local, i.e., is sent only to the player for which it holds.
- Example: `at('RightHallB')`

Predicate: `atBlock(<BlockID>)`

- Description: This player is at the location of block <BlockID>.
- Translation: "This player is located at block <BlockID>."
- Type: Send on change with negation.
- Additional explanation: Percept is local, i.e., is sent only to the player for which it holds. When robots are located at a block, it means they are close enough to pick it up.
- Example: `atBlock(3)`

Predicate: `occupied(<RoomId>)`

- Description: Holds if a robot is in a room with name <RoomId>.
- Translation: "Room `<RoomId>` is occupied."
- Type: Send on change with negation.
- Additional explanation: Percept is global, i.e., if a robot enters a room, the percept is sent to *all* players. Whenever a room is occupied it cannot be entered by anyone, i.e., from the perspective outside the room all doors of that room are closed.
- Example: `occupied('RoomA1')` means that you cannot enter room A1.

Predicate: `color(<BlockID>, <ColorID>)`

- Description: Information about the color of blocks.
- Translation: "Block <BlockID> has color <ColorID>."
- Type: Send always.
- Additional explanation: Percept is local, i.e., is sent only to the player who is in the same room as the blocks.
- Example: `color(1, red)` means that box 1 is red.

Predicate: `holding(<BlockID>)`

- Description: Information on the block this player is holding.
- Translation: "This player is holding block <BlockId>."
- Type: Send on change with negation.
- Additional explanation: Percept is local. It is sent only to the agent for which it holds. Robot can hold at most one block at a time.
- Example: `holding(1).`

Predicate: `player(<PlayerID>)`

- Description: <PlayerID> is an active player.
- Translation: "<PlayerID> is a player."
- Type: Send on change with negation.
- Additional explanation: Only names of other players are sent. The player's own name is sent with the `ownName(<PlayerID>)` percept. We talk about players because we prefer referring to the controlling entity instead of to the controlled entity, i.e. the robot. Each player is assigned exactly one robot to control. The names are assigned by the BW4T environment and not by an agent platform such as GOAL.
- Example: `player('Bob').`

Predicate: `ownName(<PlayerID>)`

- Description: <PlayerID> is this player's name.
- Translation: "<PlayerID> is this player's name"
- Type: Send once.
- Additional explanation: We talk about players because we prefer referring to the controlling entity instead of to the controlled entity, i.e. the robot. Each player is assigned exactly one robot to control.
- Example: `ownName('Bob').`

Predicate: `sequence([<ColorID>])`

- Description: Sequence of colors; this predicate is used to communicate the goal, i.e., the color order in which blocks have to be dropped at the drop zone.
- Translation: "Blocks should be dropped in the following order: `[<ColorID>]`."
- Type: Send once.
- Additional explanation: At the start of the game, the entire sequence is sent. The percept is global, i.e., sent to all players.
- Example: `sequence([Blue,Red,Blue])`

Predicate: `sequenceIndex(<Integer>)`

- Description: Expresses the index of the currently needed color in the sequence. The first index of the sequence is 0.
- Translation: "Goal <Integer>-1 is met, on to goal <Integer>"
- Type: Send on change.
- Additional explanation: The percept is global, i.e., sent to all players. The sequence index is also increased and sent when the last block of the goal sequence has been delivered. Players should thus infer themselves that the team goal has been achieved.
- Example: `sequenceIndex(3)`

Predicate: `state(<State>)`

- Description: The state that the controlled robot is in.
- Translation: "Robot is in state <State>"
- Type: Send on change.
- Additional explanation: Percept is local. It is sent only to the agent for which it holds. Possible states are `collided`, `arrived` and `traveling`.
- Example: `state(collided)`

Predicate: `message(<PlayerID>, <Content>)`

- Semantics: <Content> is a message received from <PlayerID>.
- Translation: "<Content> is a message received from <PlayerID>."
- Type: Send on change, HumanBot.
- Additional explanation: Each message is perceived only one time when the message comes in. Old messages are removed after they have been given as percept. See below for the list of messages that can be sent by players. This percept should only be used by Java agents. To transform <Content> to a BW4TMessage object use the MessageTranslator.translateMessage method. See the Javadoc for more details. An agent that you write in an agent platform like GOAL has no need for these percepts and does not need to process this type of percept.
- Example: `message(Bob,'I am at RoomC1')`.

Predicate: `position(<ObjectID>,X,Y)`

- Description: Information about where each object is in the environment. Only visible blocks are reported.
- Translation: "Object with ID <ObjectID> is located at X,Y"
- Type: Send on change.
- Additional explanation: Unless specifically wanting to use the `goTo(X,Y)` action you should not need these percepts. X and Y represent the center of an object.
- Example: `position(1, 15.0, 15.0)`.

Predicate: `robot(<ObjectID>)`

- <u>Description</u>: Information about what the ID of the controlled robot is in Repast.
- <u>Translation</u>: "Robot has ID RobotID"
- <u>Type:</u> Send once.
- <u>Additional explanation</u>: Agents should not need this percept.
- <u>Example</u>: `robot(15).`

Predicate: `location(X,Y)`

- <u>Description</u>: Information about where the controlled robot is.
- <u>Translation</u>: "Robot is located at X,Y"
- <u>Type:</u> Send on change.
- <u>Additional explanation</u>: Unless specifically wanting to use the `goTo(X,Y)` action you should not need these percepts.
- <u>Example</u>: `location(45.0,21.0).`

Predicate: `zone(zoneID, zoneName, X, Y, Neighbours)`

- <u>Description</u>: Information about each zone in the environment.
- <u>Translation</u>: "zoneName with ID zoneID is located at X,Y and has neighbours Neighbours"
- <u>Type:</u> Send once.
- <u>Additional explanation</u>: zones are interconnected and for path planning. See the action goTo(Id).

Predicate: `sendMessage('all', Message)`

- <u>Description</u>: This percept is reserved for "HumanBots" running in GOAL. There, this percept is generated when the user sends a message. The GOAL agent coupled to the HumanBot then should take care of sending the message using the GOAL send action.
- <u>Translation</u>: "user asked to send Message to all"
- <u>Type:</u> Sent every time when user selects the action, HumanBot.
- <u>Additional explanation</u>: An agent that you write yourself has no need for these percepts and does not need to process this type of percept.

Predicate: `goToBlock(Id)`

- <u>Description</u>: This percept is reserved for HumanBots running in GOAL. There, this percept is generated when the user requests to go to a block. The GOAL agent coupled to the HumanBot then should take the appropriate action.
- <u>Translation</u>: "user asked to go to a block"
- <u>Type:</u> Sent every time when user selects the action, HumanBot.
- <u>Additional explanation</u>: An agent that you write yourself has no need for these percepts and does not need to process this type of percept.

Predicate: `goTo(Id)`

- Description: This percept is reserved for HumanBots running in GOAL. There, this percept is generated when the user requests to go to a place. The GOAL agent coupled to the HumanBot then should take the appropriate action.
- Translation: "user asked to go to Id"
- Type: Sent every time when user selects the action, HumanBot.
- Additional explanation: An agent that you write yourself has no need for these percepts and does not need to process this type of percept.

Predicate: `goTo(X,Y)`

- Description: This percept is reserved for HumanBots running in GOAL. There, this percept is generated when the user requests to go to a place. The GOAL agent coupled to the HumanBot then should take the appropriate action.
- Translation: "user asked to go to position (X,Y)"
- Type: Sent every time when user selects the action, HumanBot.
- Additional explanation: An agent that you write yourself has no need for these percepts and does not need to process this type of percept.

Predicate: `pickUp()`

- Description: This percept is reserved for HumanBots running in GOAL. There, this percept is generated when the user requests to pick up a block. The GOAL agent coupled to the HumanBot then should take the appropriate action.
- Translation: "user asked to pick up a block"
- Type: Sent every time when user selects the action.
- Additional explanation: An agent that you write yourself has no need for these percepts and does not need to process this type of percept.

Predicate: `putDown()`

- Description: This percept is reserved for HumanBots running in GOAL. There, this percept is generated when the user requests put down a block. The GOAL agent coupled to the HumanBot then should take the appropriate action.
- Translation: "user asked to put down a block"
- Type: Sent every time when user selects the action.
- Additional explanation: An agent that you write yourself has no need for these percepts and does not need to process this type of percept.

Predicate: `getSizes()`

- Description: This percept is generated when getting all the sizes of bots in the environment is required.
- Type: Sent on change.
- Additional explanation: This percept is used by the client at the creation of the bots so that it can give them the correct sizes.

Predicate: `getBatteryPercentage()`

- Description: This percept is generated when the bot requests its battery percentage. The GOAL agent coupled to the HumanBot then should take the appropriate action, depending on the battery level.
- Type: Sent on change.
- Additional explanation: The agent should be able to decide to go to the nearest charge zone, once the result of this percept is too low.

Predicate: `getEPartners()`

- Description: This percept is reserved for HumanBots running in GOAL. There, this percept is generated when the user wants to know the e-Partners that are present in the zone.
- Type: Sent always.

Predicate: `getOldTargetUnreachable()`

- Description: This percept is generated when the bot has a target that is unreachable. The GOAL agent coupled to the bot should then take the appropriate action.
- Type: Sent on change.

Predicate: `bumped()`

- Description: This percept is generated when the bot collides with something blocking its path. The GOAL agent coupled to this bot should take the appropriate action.
- Type: Sent on change negation.
- Additional explanation: The agent should be able to decide to take another path around the obstacle. If the obstacle is a robot, the name of that robot is also returned.

Basic Actions

This section describes the basic actions that are available to an agent to control a robot in the BW4T environment.

Action: `goTo(X,Y)`

- Precondition: None.
- Postcondition: You are located at the specified X and Y coordinates if no obstacles to going to that location are present.

Additional explanation: The robot first moves to the center of the zone that it is in, or to the nearest zone center. From there it uses a path planner to find the shortest path to the zone that contains (X,Y) or the zone center that is nearest to (X,Y). From there it then moves to (X,Y).

- The goTo action is successfully terminated when location (X,Y) is reached, the percept `state(arrived)` is sent.
- If doors are closed, a robot may not be able to go to a location, and in that case the goTo action fails and the robot will be close to the door where the action failed. A percept `state(collided)` is sent when the goTo action fails this way.
- if the map has been customized to use oneBotPerCorridorZone, a robot may not be able to enter a zone on the route to the target. Similarly to closed doors, A percept `state(collided)` is sent when the goTo action fails this way. Also notice that zones in this case may overlap slightly (or much if designed improperly), and in such cases a bot may be in multiple zones at once.

Action: `goTo(<PlaceID>)`

- Precondition: `<PlaceID>` must be a place.
- Postcondition: You are located at the specified <PlaceID> if no obstacles to going to that location are present.

Additional explanation: see goTo(X,Y) for details.

Action: `goToBlock(<BlockID>)`

- Precondition: Robot must be in same room as block `<BlockID>`.
- Postcondition: You are located at the specified <BlockID> if no obstacles to going to that location are present.

Additional explanation:

- The `goToBlock` action is successfully terminated when location <BlockID> is reached, the percept `state(arrived)` is sent.
- If doors are closed, a robot may not be able to go to a location, and in that case the goTo action fails and the robot will be close to the door where the action failed. A percept `state(collided)` is sent when the `goTo` action fails this way.

Action: `pickUp`

- Precondition: Robot is close to a block and does not hold a block yet.
- Postcondition: Robot is holding the block, and the block is not located anywhere (i.e., there is no "at" percept for the block) until it is dropped.

Additional explanation: A robot always is closest to at most one block, and blocks are not stacked on top of each other. A percept `holding(<BlockID>)` will be provided to the agent if the action is successful.

Action: `putDown`

- Precondition: Robot is holding a block at location <PlaceID>.
- Postcondition: (i) If robot is in a room, the block is dropped within a tolerance range of the current position of the robot. (ii) If robot is not in a room and not at the drop zone, then the block leaves the environment and (iii) if robot is at the drop zone, the block is also removed from the environment; in case it matches the current color needed according to the sequence predicate (representing the goal of the game) then this sequence (goal) is updated as well correspondingly.

Additional explanation: Action is simply ignored by environment when robot is not holding a block.

Action: `sendmessage(<PlayerID>, <Content>)`

- Precondition: None.
- Postcondition: The message is delivered as a `message` percept (see percept section) to the player with the specified <PlayerID >. This can also be set to 'all' in which case it will be sent to all other players.

Additional explanation: See for list of supported messages below. It is advised not to use this action when developing a GOAL agent; instead use the built-in `send` command of GOAL and use the `allother` instead of `all` label for broadcasting.

Action: `pickUpEpartner`

- Precondition: The HumanBot can pick up an e-Partner.
- Postcondition: The HumanBot has picked up the e-Partner.

Additional explanation: if a non-human bot attempts to pick the e-Partner, nothing will happen.

Action: `dropEPartner`

- Precondition: The HumanBot is holding a e-Partner.
- Postcondition: The e-Partner is dropped.

Action: `navigateObstacles`

- <u>Precondition</u>: None.
- <u>Postcondition</u>: The bot navigates around the obstacle it is facing.

The following list of percepts and actions is exclusive to e-Partners. The initialization and messaging percepts/actions are the same as the robot.

E-Partner Percepts

Predicate: `heldBy()`

- <u>Description</u>: This percept is generated when the e-Partner's agent wishes to know by whom it is currently held.
- <u>Type:</u> Sent on change negation.
- <u>Additional explanation</u>: This percept is only available for Forget-me-not e-Partners.

Predicate: `getFunctionalities()`

- <u>Description</u>: This percept is generated when the e-Partner inquires on which functionality it has.
- <u>Type:</u> Sent once.

Predicate: `isTaken()`

- <u>Description</u>: This percept is generated when the e-Partner is picked up by a HumanBot.
- <u>Type:</u> Sent on change negation.
- <u>Additional explanation</u>: This percept is only available for Forget-me-not e-Partners.

Predicate: `leftBehind()`

- <u>Description</u>: This percept is generated when the e-Partner is dropped.
- <u>Type:</u> Sent on change.
- <u>Additional explanation</u>: This percept is only available for Forget-me-not e-Partners.

Predicate: `getAt()`

- <u>Description</u>: This percept is generated when the e-Partner requests its location, returning the zone the e-Partner is in.
- <u>Type:</u> Sent on change.
- <u>Additional explanation</u>: This percept is only available for GPS e-Partners.

Predicate: `getLocation()`

- <u>Description</u>: This percept is generated when the e-Partner requests its location, returning the point the e-Partner is at.
- <u>Type:</u> Sent on change.

Predicate: `getRooms()`

- <u>Description</u>: This percept returns all the rooms on the map.
- <u>Type:</u> Sent once.
- <u>Additional explanation</u>: This percept is only available for GPS e-Partners.

Predicate: `getRoom()`

- <u>Description</u>: This percept is generated when the e-Partner requests to know in which room it is.
- <u>Type:</u> Sent once.

Messages

Below you can find a list of all supported messages in the BW4T2 Environment. GOAL agents can directly send the message as prescribed, however Java agents should use a BW4TMessage object to define a message, please refer to the javadoc. The BW4TMessage defines 5 parameters:

- MessageType type
- String room
- String color
- String playerID
- int number

For each message described below we will also add how to create the equivalent in Java. The MessageTranslator class provides translation capabilities from natural language to BW4TMessage and vice versa for Java agents.

Imperative Messages (me)

Message: `imp(in(<Me>,<PlaceID>))`

- Natural language: I am going to <PlaceID>
- Java: new BW4TMessage(MessageType.goingToRoom,<PlaceID>,null,null)

Message: `imp(found(<Me>,<ColorID>)`

- Natural language: I am looking for a <ColorID> block
- Java: new BW4TMessage(MessageType.lookingFor, null, <ColorID>, null)

Message: `imp(pickedUpFrom(<Me>, <ColorID>, <RoomID>))`

- Natural language: I am going to room <RoomID> to get a <ColorID> block/I am getting a <ColorID> block from room <RoomID>
- Java: new BW4TMessage(MessageType.amGettingColor,<RoomID>, <ColorID>, null)

Message: `imp(holding(<Me>,<ColorID>))`

- Natural language: I will get a <ColorID> block
- Java: new BW4TMessage(MessageType.willGetColor,<RoomID>,<ColorID>,null)

Message: `imp(putDown(<Me>))`

- Natural language: I am going to put down a block
- Java: new BW4TMessage(MessageType.aboutToDropOffBlock)

<u>Imperative Messages (other players), i.e., requests</u>

Message: `imp(in(<PlayerID>, <RoomID>))`

- Natural language: <PlayerID> go to <RoomID>
- Java: new BW4TMessage(MessageType.goToRoom,<RoomID>,null,<PlayerID>)

Message: `imp(found(<PlayerID>,<ColorID>))`

- Natural language: <PlayerID> find a <ColorID> block
- Java: new BW4TMessage(MessageType.findColor,null,<ColorID>,<PlayerID>)

Message: `imp(pickedUpFrom(<PlayerID>,<ColorID>, <RoomID>))`

- Natural language: <PlayerID> get the <ColorID> block from room <RoomID>
- Java: new
  BW4TMessage(MessageType.getColorFromRoom,<RoomID>,<ColorID>,<PlayerID
  >)

Message: `imp(putDown(<PlayerID>))`

- Natural language: <PlayerID> put down the block you're holding
- Java: new BW4TMessage(MessageType.putDown,null,null,<PlayerID>)

<u>Declarative messages concerning state of environment</u>

Message: `at(<ColorID>,<RoomID>)`

- Natural language: room <RoomID> contains a <ColorID> block
- Java: new BW4TMessage(MessageType.roomContains,<RoomID>,<ColorID>,null)

Message: `at(<Integer>, <ColorID>,<RoomID>)`

- Natural language: room <RoomID> contains <Integer> <ColorID> blocks
- Java: new
  BW4TMessage(MessageType.roomContainsAmount,<RoomID>,<ColorID>,null,<Int
  eger>)

Message: `empty(<RoomID>)`

- <u>Natural language:</u> room <RoomID> is empty
- <u>Java:</u> new BW4TMessage(MessageType.roomIsEmpty,<RoomID>,null,null)

<u>Declarative messages concerning the task and the robot</u>

Message: `in(<Me>,<RoomID>)`

- <u>Natural language:</u> I am in room <RoomID>
- <u>Java:</u> new BW4TMessage(MessageType.inRoom,<RoomID>,null,null)

Message: `at(<BlockID>)`

- <u>Natural language:</u> I am at a <BlockID> box
- <u>Java:</u> new BW4TMessage(MessageType.atBox,null,<ColorID>,null)

Message: `holding(<Me>,<ColorID>)`

- <u>Natural language:</u> I have a <ColorID> block.
- <u>Java:</u> new BW4TMessage(MessageType.hasColor,null,<ColorID>,null)

Message: `pickedUpFrom(<Me>, <ColorID>, <RoomID>)`

- <u>Natural language:</u> I have a <ColorID> block from room <RoomID>.
- <u>Java:</u> new BW4TMessage(MessageType.hasColor,<RoomID>,<ColorID>,null)

Message: `putDown(<Me>)`

- <u>Natural language:</u> I just dropped off a block
- <u>Java:</u> new BW4TMessage(MessageType.droppedOffBlock)

Message: `putDown(<Me>,<ColorID>)`

- <u>Natural language:</u> I just dropped off a <ColorID> block
- <u>Java:</u> new BW4TMessage(MessageType.droppedOffBlock,null,<ColorID>,null)

Message: `waitingOutside(<Me>,<RoomID>)`

- Natural language: I am waiting outside room <RoomID>
- Java: new
  BW4TMessage(MessageType.amWaitingOutsideRoom,<RoomID>,null,null)

Message: `need(<ColorID>)`

- Natural language: We need a <ColorID> block.
- Java: new BW4TMessage(MessageType.weNeed,null,<ColorID>,null)

Message: `checked(<RoomID>)`

- Natural language: room <RoomID> has been checked.
- Java: new BW4TMessage(MessageType.checked,<RoomID>,null,null)

Message: `checked(<PlayerID>,<RoomID>)`

- Natural language: room <RoomID> has been checked by <PlayerID>
- Java: new BW4TMessage(MessageType.checked,<RoomID>,null,<PlayerID>)

Ask messages concerning state of environment

Message: `int(at(_,<RoomID>))`

- Natural language: What is in room <RoomID>?
- Java: new BW4TMessage(MessageType.whatIsInRoom,<RoomID>,null,null)
- Additional explanation: The message is interpreted as asking which color is in a
  certain room. Thus the first parameter of "at" stands for a <ColorID>.

Message: `int(at(<ColorID>,_))`

- Natural language: Where is a <ColorID> block?
- Java: new BW4TMessage(MessageType.whereIsColor,null,<ColorID>,null)
- Additional explanation: The message is interpreted as asking in which room a certain
  color is. The second parameter of "at" thus stands for a <RoomID>.

Ask messages concerning the task and the robot

Message: `int(in(_,<RoomID>))`

- Natural language: Who is in room <RoomID>?
- Java: new BW4TMessage(MessageType.whoIsInRoom,<RoomID>,null,null)
- Additional explanation: The first parameter of "in" stands for a <PlayerID>.

Message: `int(imp(in(<Me>,_)))`

- Natural language: Where should I go?
- Java: new BW4TMessage(MessageType.whereShouldIGo)
- Additional explanation: The second parameter of "in" stands for a <PlaceID>.

Message: `int(imp(holding(<Me>,_)))`

- Natural language: What color should I get?
- Java: new BW4TMessage(MessageType.whatColorShouldIGet)
- Additional explanation: The second parameter of holding stands for a <ColorID>, not a block ID.

Message: `int(imp(in(_,<RoomID>)))`

- Natural language: Is anybody going to room <RoomID>?
- Java: new BW4TMessage(MessageType.isAnybodyGoingToRoom,<RoomID>,null,null)
- Additional explanation: The first parameter of "in" stands for a <PlayerID>.

Message: `int(willBeLong(<PlayerID>))`

- Natural language: <PlayerID>, will you be long?
- Java: new BW4TMessage(MessageType.willYouBeLong,null,null,<PlayerID>)

Message: `int(checked(_, <RoomID>))`

- Natural language: Has anybody checked room <RoomID>?
- Java: new BW4TMessage(MessageType.hasAnybodyCheckedRoom,<RoomID>,null,null)
- Additional explanation: The first parameter of "checked" stands for a <PlayerID>.

Message: `int(areClose(<PlayerID>))`

- Natural language: <PlayerID>, are you close?
- Java: new BW4TMessage(MessageType.areYouClose,null,null,<PlayerID>)

Message: `int(holding(_, <ColorID>))`

- Natural language: Who has a <ColorID> block?
- Java: new BW4TMessage(MessageType.whoHasABlock,null,<ColorID>,null)
- Additional explanation: The first parameter of "holding" stands for a <PlayerID>.

Answer messages

Message: `yes/no/dontknow/wait/ok/idont/ido/`
`ontheway/faraway/delayed/almostthere/couldnot`

- Natural language: yes/no/I don't know/wait/OK/I don't/I do/I am on the way/I am far away/I am delayed/I am almost there
- Java: new BW4TMessage(MessageType.(yes/no/iDoNotKnow/wait/ok/iDoNot/iDo/onTheWay/farAway/delayed/almostThere/couldNot))

Message: `ok(<RoomID>)`

- Natural language: OK, room <RoomID>
- Java: new BW4TMessage(MessageType.ok,<RoomID>,null,null)