ROS-AIL Integration

Louise Dennis

November 5, 2014

This technical report outlines the integration of the Robot Operating System (ROS) [4] with the Agent Interface Layer (AIL) toolkit [3]. It assumes some familiarity with the AIL, ROS and ROSJava.

1 The EASS variant of Gwendolen

The EASS variant of the Gwendolen programming language was developed as part of the EPSRC project Engineering Autonomous Space Software and is included in the MCAPL distribution which includes the AIL. A key aspect of this language variant is the existence of an *abstraction engine* that communicates with the agent via a set of *shared beliefs*. The use of the abstraction engine is important in mediating between sensors and agent percepts and so this language variant is recommended for developing agents that form a component of an a autonomous system. The integration of the AIL and ROS was therefore done as an integration of the EASS Gwendolen variant. The key aspects of this variant can be found in [1] while Gwendolen is described in [2]. It may be possible to make the integration more generic in future.

As originally developed, the EASS language variant was intended to interface to a MatLab simulation. Adapting this set-up to handle communication with the Robot Operating System has involved some restructuring of the way EASS agents interact with environments. In particular, there is now a more generic class structure which allows a unified support not just for ROS and MatLab but also for interfacing to NXT Mindstorms Robots running leJOS and, potentially, to other platforms.

2 EASS Environments

An EASS environment is a subclass of the more general AIL environment supported by the toolkit. These environments are specified as Java interfaces. AIL environments are required to supply methods for executing actions, adding and removing percepts from the environment and returning a set of percepts to an agent. EASS environments extend this with additional methods supported the addition of abstraction engines and managing shared beliefs. EASS environments also implement the `MCAPLJobber` interface from the larger MCAPL

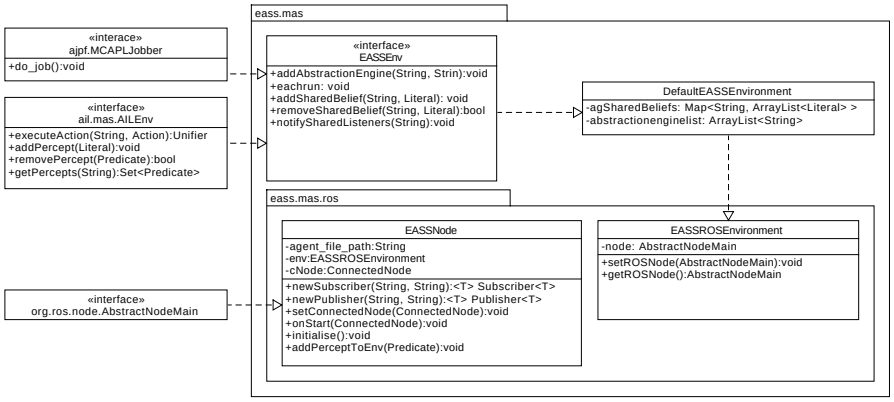

Figure 1: Class Diagram for ROS Support

framework. This requires that environments implement a `do_job` method which is called by the scheduler. This embodies the implicit assumption that autonomous system environments have behaviours which are outside the explicit control of the agents.

3 The ROS classes

The integration with the Robot Operating System is handled as a sub-package, `eass.mas.ros`, of `eass.mas`. Two classes are provided. `eass.mas.ros.EASSNode` is an extension of the ROSJava class `AbstractNodeMain` and embodies an idea of a node which is part of an EASS environment. In particular it provides a method `onStart` that handles the creation of an EASS multi-agent system from a program file when the node is created from within ROS. It also provides a method `addPerceptToEnv` which supports adding new percepts to the environment. The assumption is that `EASSNode` will be subclassed for specific applications which will receive ROS messages and convert them into predicates which can then be added to the environment.

`eass.mas.rosEASSROSEnvironment` is similarly provided as a starting point for building environment for ROS based agents. It extends `DefaultEASSEnvironment` and adds methods for attaching a ROS Node (e.g., an `EASSNode`) to the environment.

The class structure for the ROS classes is shown in figure 1. In this figure dotted arrows indicate either the implementation of an interface or a sub-class relationship (as appropriate) while the package structure is show by included classes and interfaces in packages.

4 Installing the MCAPL ROS integration

At present the integration with ROS is not part of the main distribution of the MCAPL project. This is largely because ROS and JavaPathfinder (the model-checker that underpins MCAPL) expect different code structures and build environments and so the some of the integration is not entirely seamless.

The MCAPL project code is stored in a git repository on sourceforge as detailed at `mcapl.sourceforge.net`. The ROS integration can be found on a branch called `rasl`. We recommend installing the master branch first and then switching to the `rasl` branch once the master branch is successfully installed.

5 An Example

In the MCAPL source code there is a directory `src/examples/eass/ros/pubsub`. This provides an implementation of a simple publisher and subscriber which show how `EASSNode` and `EASSROSEnvironment` can be extended to provide specific nodes for some system. Simple EASS agents either publish or echo messages. The results of agent actions are encoded in the standard AIL fashion in `executeAction` for `PubSubEnvironment`. This code gets the associated ROS node and publishes the message to a topic. Similarly `ExampleSubscriber` uses the `initialise` method to set up a listener on a topic and convert messages to percepts which are then added to the environment.

5.1 Installing and Running EASS agents as ROS Nodes

To run the example you will need an installation of ROS hydro (including the rosjava packages) and a build of the `rasl` branch of the MCAPL project.[1]

Make sure you have set the environment variable `AJPF_HOME` to point to the `mcapl` directory.

You will need to create a rosjava package in your ROS workspace e.g.,

```
mkdir -p rosjava/src
cd rosjava/src
catkin_create_rosjava_pkg eass
cd eass
```

[1]The MCAPL distribution comes with Eclipse project files. In order to view the ROS integration classes in Eclipse you will need to edit the project build path to to include some rosjava jar files. The version numbers on these files depend upon your ROS version and change regularly so this has to be customised to your build. The files are:

- control_msgs in org/ros/rosjava_messages
- geometry_msgs in org/ros/rosjava_messages
- std_msgs in org/ros/rosjava_messages
- message_generation in org/ros/rosjava_bootstrap
- rosjava in /org/ros/rosjava_core

Once these are linked the MCAPL project should display without too much red!.

To run the publisher subscriber example you need to create a ROS project for it in a ROS workspace e.g.,

```
catkin_create_rosjava_project eass_pubsub
```

Then you need to copy `build.gradle` from the MCAPL pubsub directory into `eass_pubsub`

```
cp $AJPF_HOME/src/examples/eass/ros/pubsub/build.gradle .
```

Create a location for the example's java files and copy them there.

```
mkdir -p src/main/java/eass/ros/pubsub
cp $AJPF_HOME/src/examples/eass/ros/pubsub/*.java src/main/java/eass/ros/pubsub/.
```

And lastly build the application using gradle.

```
../gradlew installApp
```

To run the application you need to start a ROS core, `roscore`.

Then run the publisher and subscriber (you will need a separate terminal window for each one)

```
./build/install/eass_pubsub/bin/eass_pubsub eass.ros.pubsub.ExamplePublisher
./build/install/eass_pubsub/bin/eass_pubsub eass.ros.pubsub.ExampleSubscriber
```

You should then be able to view the publisher publishing the message `hello` and the subscriber echoing the message when it is received.

References

[1] L. A. Dennis, M. Fisher, N. Lincoln, A. Lisitsa, and S. M. Veres. Declarative Abstractions for Agent Based Hybrid Control Systems. In *Proc. 8th Int. Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 6619 of *LNCS*, pages 96–111. Springer, 2010.

[2] Louise A. Dennis and Berndt Farwer. Gwendolen: A BDI Language for Verifiable Agents. In *Workshop on Logic and the Simulation of Interaction and Reasoning*. AISB, 2008.

[3] Louise A. Dennis, Michael Fisher, Matthew P. Webster, and Rafael H. Bordini. Model Checking Agent Programming Languages. *Automated Software Engineering*, 19(1):5–63, 2012.

[4] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an Open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.