Agents Programming

Jason and the golden fleece of agent-oriented programming

Rahul Arora

University Koblenz-Landau

1 Introduction

Research on Multi-Agent Systems (MAS) has led to a variety of techniques that promise to allow the development of complex distributed systems. The importance of this is that such systems would be able to work in environments that are traditionally thought to be too unpredictable for computer programs to handle.Jason is the interpreter for AgentSpeak and implements its operational semantics. It also implements the extension of the operational semantics that accounts for speech-act based communication among AgentSpeak agents. Some important features of jason are:-

- speech-act based inter-agent communication (and annotation of beliefs with information sources)
- annotations on plan labels, which can be used by elaborate selection functions
- the possibility to run a multi-agent system distributed over a network (using SACI, but other middleware can be used)
- fully customisable (in Java) selection functions, trust functions, and overall agent architecture
- straightforward extensibility by means of user-defined internal actions

Interestingly, most of the advanced features are available as optional, customisable mechanisms. Thus, because the AgentSpeak core that is interpreted by Jason is very simple and elegant, yet having all the main elements for expressing reactive planning system with BDI notions. An important strand of work related to AgentSpeak that adds to making Jason a promising platform is the work on formal verification of MAS systems implemented in AgentSpeak by means of model checking techniques.

2 Semantics and Verification

Jason implements the operational semantics of an extension of AgentSpeak. Having formal semantics also allowed us to give precise definitions for practical notions of beliefs, desires, and intentions in relation to running AgentSpeak agents.

2.1 Informal Semantics

Besides the belief base and the plan library, the AgentSpeak interpreter also manages a set of events and a set of intentions, and its functioning requires three selection functions. The event selection function (SE) selects a single event from the set of events; another selection function (SO) selects an option (i.e., an applicable plan) from a set of applicable plans; and a third selection function (SI) selects one particular intention from the set of intentions. The selection functions are supposed to be agent-specific, in the sense that they should make selections based on an agent characteristics.Therefore, here leave the selection functions undefined, hence the choices made by them are supposed to be non-deterministic.

Intentions are particular courses of actions to which an agent has committed in order to handle certain events. Each intention is a stack of partially instantiated plans. Events, which may start off the execution of plans that have relevant triggering events, can be external, when originating from perception of the agent's environment or internal, when generated from the agent own execution of a plan. In the latter case, the event is accompanied with the intention which generated it. External events create new intentions, representing separate focuses of attention for the agent acting on the environment.

2.2 Verification

One of the reasons for the growing success of agent-based technology is that it has been shown to be quite useful for the development of various types of applications, including air-traffic control, autonomous spacecraft control, health care, and industrial systems control, to name just a few. Clearly, these are application areas for which dependable systems are in demand. Consequently, formal verification techniques tailored specifically for multi-agent systems is also an area that is attracting much research attention and is likely to have a major impact in the uptake of agent technology. One of the advantages of the approach to programming multi-agent systems is precisely the fact that it is amenable to formal verification.

3 Features of Jason

3.1 Configuring MAS

The configuration of a complete multi-agent system is given by a very simple file with extension mas2j as shown below

```
MAS heathrow
infrastructure: Centralised
environment: HeathrowEnv
agents:
mds agentClass mds.MDSAgent #5;
cph agentArchClass cph.CPHAgArch agentClass cph.CPHAgent #10;
```

the environment is implemented in a class named HeathrowEnv; the system has two types of agents: five instances of MDS79, ten CPH903. MDS79 agents have a customised agent class and CPH903 have customised agent and agent architecture classes. The keyword architecture is used to specify which of the two overall agent architectures available with Jason's distribution will be used. The options currently available are either Centralised or Saci; the latter option allows agents to run on different machines over a network. It is important to note that the user's environment and customisation classes remain the same with both (system) architectures. Next an environment needs to be referenced. This is simply the name of Java class that was used for programming the environment. Note that an optional host name where the environment will run can be specified. This only works if the SACI option is used for the underlying system architecture. The keyword agents is used for defining the set of agents that will take part in the multi-agent system. An agent is specified first by its symbolic name given as an AgentSpeak term (i.e., an identifier starting with a lowercase letter); this is the name that agents will use to refer to other agents in the society

Then, an optional filename can be given where the AgentSpeak source code for that agent is given; by default Jason assumes that the AgentSpeak source code is in file (name).asl, where (name) is the agent's symbolic name. An optional number of instances of agents using that same source code can be specified by a number preceded by #; if this is present, that specified number of "clones" will be created in the multi-agent system. In case more than one instance of that agent is requested, the actual name of the agent will be the symbolic name concatenated with an index indicating the instance number (starting from 1). As for the environment keyword,an agent definition may end with the name of a host where the agent will run (preceded by at). As before, this only works if the SACI-based architecture was chosen. The following settings are available for the AgentSpeak interpreter available in Jason:

– Events: options are either discard, requeue, or retrieve; the discard option means that external events for which there are no applicable plans are discarded, whereas the requeue option is used when such events should be inserted back at the end of the list of events that the agent needs to handle. When option retrieve is selected, the user-defined select Option function is called even if the set of relevant/applicable plans is empty.
– IntBels: options are either same Focus or new Focus. When internal beliefs are added or removed explicitly within the body of a plan, the associated event is a triggering event for a plan, the intended means resulting from the applicable plan chosen for that event is pushed on top of the intention which generated the event, if the same Focus option is used. If the new Focus option is used, the event is treated as external, creating a new focus of attention.
– Verbose: a number between 0 and 6 should be specified. The higher the number, the more information about that agent is printed out in the console where the system was run. The default is in fact 1, not 0; verbose 1 prints out only the actions that agents perform in the environment and the messages exchanged between them.

Finally, user-defined overall agent architectures and other user-defined functions to be used by the AgentSpeak interpreter for each particular agent can be specified with the keywords agentArchClass and agentClass.

3.2 Creating Environments

Jason agents can be situated in real or simulated environments. This is done directly in a Java class that extends the Jason base Environment class. A very simple simulated version of the environment is given below

```
public class HeathrowEnv extends Environment {
public List getPercepts(String agName) {
if ( ... unattended luggage has been found ... ) {
// all agents will perceive the fact that there is unattendedLuggage
getPercepts().add(Term.parse(unattendedLuggage));
} if (agName.startsWith(mds))
// mds robots will also perceive their location
List customPerception = new ArrayList();
customPerception.addAll(getPercepts());
customPerception.add(agsLocation.get(agName));
return customPerception;
} else {
return getPercepts(); } }
public boolean executeAction(String ag, Term action){
if (action.hasFunctor(disarm)) {
... the code that implements the disarm action on the environment goes here
} else if (action.hasFunctor(move)) {
the code for changing the agent and updating the ags Location map goes here
}return true;
```

All percepts should be added to the list returned by getPercepts; this is a list of literals, so strong negation can be used in applications where there is open-world assumption. It is possible to send individualised perception; that is, in programming the environment the developer can determine what subset of the environment properties will be perceptible to individual agents. Within an agent's overall architecture you can further customise what beliefs.the agent will actually aquire from what it perceives. Intuitively, the environment properties available to an agent from the environment definition itself are associated to what is actually perceptible at all in the environment.The customisation at the agent overall architecture level should be used for simulating faulty perception. Customisation of agent's individual perception within the environment is done by overriding the getPercepts(agName) method; the default methods simply provide all current environment properties as percepts to all agents. In the example above, only MDS79 robots will perceive their location at the airport. Most of the code for building environments should be (referenced) in the body of the method execute Action which must be declared as described above. Whenever an agent tries to execute a basic action, the name of the agent and a Term representing the chosen action are sent as parameter to this method. So the code for this method needs to check the Term representing the action being executed, and check which is the agent attempting to execute the action, then do whatever is necessary in that particular model of an environment — normally, this means changing the percepts, i.e., what is true or false of the environment is changed according to the actions being performed. The execution of an action needs to return a boolean value, stating whether the agent attempt at performing that action on the environment was executed or not. A plan fails if any basic action attempted by the agent fails.

3.3 Internal Actions

An important construct for allowing AgentSpeak agents to remain at the right level of abstraction is that of internal actions, which allows for straightforward extensibility and use of legacy code internal actions that start with . are part of a standard library of internal actions that are distributed with Jason. Internal actions defined by users should be organised in specific libraries, which provides an interesting way of organising such code, which is normally useful for a range of different systems. In the AgentSpeak program, the action is accessed by the name of the library, followed by . followed by the name of the action. Libraries are defined as Java packages and each action in the user library should be a Java class, the name of the package and class are the names of the library and action as it will be used in the AgentSpeak programs.

4 Available Tools and Documentation

Jason is distributed with an Integrated Development Environment (IDE) which provides a GUI for editing a MAS configuration file as well as AgentSpeak code for the individual agents. Through the IDE, it is also possible to control the execution of a MAS, and to distribute agents over a network in a very simple way. There are three execution modes:

– Asynchronous: in which all agents run asynchronously. An agent goes to its next reasoning cycle as soon as it has finished its current cycle. This is the default execution mode.
– Synchronous: in which each agent performs a single reasoning cycle in every global execution step. That is, when an agent finishes a reasoning cycle, it informs Jason's execution controller, and waits for a carry on signal. The Jason controller waits until all agents have finished their current reasoning cycle and then sends the carry on signal to them.
– Debugging: this execution mode is similar to the synchronous mode; however, the Jason controller also waits until the user clicks on a Step button in the GUI before sending the carry on signal to the agents.

There is another tool provided as part of the IDE which allows the user to inspect agents internal states when the system is running in debugging mode. This is very useful for debugging MAS, as it allows inspection of agents minds across a distributed system. The tool is called Mind Inspector.

5 Interoperability and Portability

As Jason is implemented in Java, there is no issue with portability, but very little consideration has been given so far to standards compliance and interoperability. However, components of the platform can be easily changed by the user. For example, at the moment there are two system architectures available with Jason distribution: a centralised one and another which uses SACI for distribution. It should be reasonably simple to produce another system architecture which uses, e.g., JADE for FIPA-compliant distribution and management of agents in a multi-agent system.

6 Conclusion

Jason is constantly being improved and extended. The long term objective is to have a platform which makes available important technologies resulting from research in the area of Multi-Agent Systems, but doing this in a sensible way so as to avoid the language becoming cumbersome and, most importantly, having formal semantics for most, if not all, of the essential features available in Jason.

7 References

[1]. Bordini, Rafael H., Jomi F. Hübner, and Renata Vieira. Jason and the Golden Fleece of agent-oriented programming. Multi-agent programming. Springer US, 2005. 3-37.