Integrating Agents and Objects to Develop Distributed AI Systems

Agostino Poggi
Dipartimento di Ingegneria dell’Informazione
University of Parma, Viale delle Scienze, 43100 Parma, Italy
e-mail: poggi@CE.UniPR.IT

Abstract

This paper presents a language for the development of multi-agent systems that allows a user to write programs with a small effort thanks to a predefined agent model and a set of specialized primitives for agent management. Moreover, the user can develop agent parts at a low level taking advantage of all the potentialities of C++, Common Lisp and Java object-oriented languages, and reusing a large part of the C, C++, Lisp, Common Lisp and Java software available on the internet.

1. Introduction

The development of agent systems needs suitable programming languages. At the beginning, researchers implemented agent systems by means of conventional Artificial Intelligence programming languages (e.g., Lisp and Prolog) taking into account the ideas expressed by research on architectures and agent theories [2]. Some years ago, some researchers showed how the integration of well defined agent architectures with specialized languages can make the development and maintenance of such systems easier by embodying principles of agent theory, and introducing expressive agent communication languages [4]. However, these languages force the user to implement agents through a well established set of agent architectures that do not allow him to obtain an efficient code as he often needs. More recently, the great interest for the development of internet applications and the recognition that agent technology seems be the most suitable to do those applications drove researcher to develop agent languages offering features specialized for that domain [5]. However, their specialization for the internet domain makes them unsuitable for other types of applications. (for example, the fact that these languages are often based on interpreted code makes them unsuitable for high performance applications).

This paper presents a language for the development of multi-agent systems, ALL - Agent Level Language, that tries to overcome most of the problems presented by traditional and specialized languages in the development of such systems. In fact, this language makes easy the development of multi-agent systems through the use of a predefined agent model and a specialized set of primitives for agent programming, but it also offers the possibility of defining agent parts through three object languages, C++, Common Lisp and Java to extend ALL representation power and to reuse a large amount of code available in those three languages.

2. An Agent Level Language

ALL (Agent Level Language) is a language that offers a dedicated set of operators for agent-oriented programming. Its main features are: i) a "programmable" agent model; ii) the specialization of agents through single inheritance; iii) a limited set of programming constructs and a limited set of data types; iv) the use of C++, Common Lisp and Java objects as parts of agents; and v) three parsers translating ALL code into C++, Common Lisp and Java code.

ALL is based on a very simple agent model that derives from the agent model presented by AGENT0[4]. But, while ALL agent model can be easily specialized to implement some more complex agent architectures, AGENT0 agent model can be specialized with difficulty because AGENT0 does not offer object-oriented features.

ALL agent model is based on a state, a set of actions, an engine, and a set of rules. The state maintains agent knowledge about itself, the external environment and the other agents. Knowledge is not static, but an agent can acquire knowledge through perception and communication with other agents. Agent state is based on facts composed of slots whose value can be either an ALL elementary data (i.e., boolean, integer and string) or another fact. Facts also encapsulate external objects (i.e., C++, Common Lisp and Java objects).

The behavior of an agent is driven by a special action, called "engine". This action serves input messages, coming from other agents, through rules composed of a condition on agent state, a condition on the input message and an action.
An agent can dynamically create some other agents and some copies of itself. An agent cannot delete some other agents, but can only delete itself.

The user can develop multi-agent systems using and specializing the agent model presented above by introducing and/or modifying some rules, actions, or state components (i.e., facts or external objects). The user can easily do it by using the minimal set of ALL programming constructs composed on a subset of C++ and Common Lisp constructs. ALL joins generality to programming facility because its code can be translated into C++, Common Lisp and Java code through three parsers and these three languages seem be suitable to develop good applications in a large set of (different) domains. Moreover, the possibility to translate ALL code in the three previous languages is important because it makes possible to use two different languages during the different phases of the development of a system (Common Lisp and Java offer a stable platform for rapid prototyping and can be used during the development of the prototypes of the system, C++ allows to obtain a more efficient code than Common Lisp and Java and can be used to implement the final system).

3. An Application

We used ALL for the implementation of some applications in the fields of programming support systems and robotics. For example, we implemented an internet softbot that is an intelligent agent that uses the UNIX shell and World Wide Web (WWW) to help the user to execute UNIX commands and to interact with a wide range of internet resources [1].

To develop such an agent, we took advantage of the code developed for an our previous prototype of softbot [3]. This code is based on three Common Lisp objects whose procedures encapsulate the planning algorithm, the UNIX commands and the WWW commands. Therefore, the softbot agent definition is simply given by a set of facts and a set of rules:

```
softbot : agent
{ state: ...
    planner object("unixplanner", "cl");
    unixexec object("unixexec", "cl");
    wwwexec object("wwwexec", "cl");
    ...
    rules: ...
}
```

where "planner", "unixexec" and "wwwexec" are three facts encapsulating three Common Lisp objects that are respectively instances of the classes "unixplanner", "unixexec" and "wwwexec".

The main part of agent code is represented by rules. Let us, for example, give a brief description of the rules that manage a request to list the files:

```
rule1: ( eqslot(inmsg, msgtype, "evaluate") &&
        copyslot(inmsg, content, form) &&
        eqslot(form, action, "listfiles") &&
        copyslot(form, arg1, dir))
        eqslot(env, currentDir, dir)
        { callfobj(unixexec, ls, dir);
          getfobj(unixexec, result, listfiles);
          replymsg(inmsg, listfiles); }
```

These two rules accept an "evaluate" message whose contents indicates to list the files of a directory. The first rule is executed when the software agent is in the directory indicated by the request (this information is stored in a fact, called "env", maintaining the knowledge about the softbot UNIX state). This rule lists the files and replies the list to the user. The second rule is executed when the software agent is in another directory. This second rule calls the planner to define and to execute a plan, updates the softbot UNIX state (through the action getfobj(planer, env, env)), and gets the list of files that replies to the user.

Acknowledgement

This work has been partially supported by National Research Council (CNR) through “Sistemi Autonomi Ragionanti ed Intelligenti (SARI)” project.

References