ISBN 978-83-60810-14-9
ISSN 1896-7094

Agent architecture for building Robocode players with SWI-Prolog

Vasile Alaiba, Armand Rotaru
Al.I.Cuza University,
Department of Computer Science
General Berthelot 16
Iaşi, România
Email: {alaiba, armand.rotaru}@info.uaic.ro

***Abstract*—Effective study of autonomous agents is a challenging activity, for researchers and programmers alike. In this paper we introduce a new agent environment built on top of the tank-simulation game Robocode. We consider this game a good choice for testing agent programming strategies and an excellent learning opportunity. We integrated SWI-Prolog into Robocode and built an architecture for writing agents with Prolog. Game generated events are translated by our framework into predicate invocations, allowing the agent to perceive the environment. An API to issue commands to the tank from Prolog acts as agent's effectors. We tested our architecture by implementing a simple autonomous agent that uses a static strategy.**

I. Motivation

AGENT oriented programming is hailed to be the next paradigm shift in software engineering. Four key notions were identified that distinguish agents from arbitrary programs: reaction to the environment, autonomy, goal-orientation and persistence [4]. A lot of research goes in the areas connected to the field. Almost unquestionable goes the statement that the most important feature is autonomy, without which an agent can not be. A system is autonomous to the extent that its behavior is determined by its own experience [5].

The agent related ideas were around for some time and under different names. If we are to consider programming games like Robocode [3], we will see that these robots not only fulfill most if not all of the requirements to be named "agents", but the environment itself is well defined and complex enough to provide insight into harder real-life problems. We will describe with greater detail later on the characteristics of the game and the challenges it poses.

Logic programming, both as a field of science and as a practical tool, is old and new at the same time. Starting with the famous statement made by Kowalski, "algorithm = logic + control", made back in 1979 [6], the field knew ups and downs for the last 30 years. Although this may seem relatively old for the computer science field, there is still a lot of room for improvement. Software agents, especially, can benefit from a declarative logic programming language like Prolog and its dialects.

Last but not least, we strongly believe that research should be done as close to a practical field as possible, in order to benefit from a quick feedback loop. We chose the tank-simulation game Robocode to be our practical software virtual environment and Prolog our agent programming language. Thus we provide a new architecture for developing and experimenting agent programming techniques with ease.

II. Robocode: the agent's environment

Robocode is a robotics battle simulator that runs across all platforms supporting any recent Java runtime (version 2 or more). It is an easy-to-use framework that was originally created to help teach object-oriented programming in Java [1]. The project was started by M. Nelson in late 2000 and was available through IBM's *alphaWorks* in July 2001. Now there is a strong open-source community around it with competitions runing all over the world [3].

The basic idea behind Robocode is letting autonomous software agents (robots) compete in a virtual closed environment against each other. Each robot is programmed in Java and has to implement a given base class. This code controls a virtual robot in arena by moving its body, steering, targetting and shooting other robots. The same base class provides handlers that the robot can implement to respond to external events. The whole system follows an event-driven architecture. Moreover, each robot runs in a security sandbox, allowing for easy and safe distribution of robot code throughout the community, protecting against harmful effects.

A. Robocode simulation engine architecture

The Robocode game consists of several robots engaging in combat in a closed environment, a rectangular arena. The size of the arena and some other options can be configured just before a battle is started. As seen in figure 1, each robot runs on its own thread with its own event queue, responding to events generated by the game engine independently of the battle manager and the other robots [1]. In addition to this, each robot has a `run()` method where the main processing loop occurs. This is usually where the general strategy is implemented, i.e. what the robot does when no events occur.

Each robot is made up from three parts: the vehicle, the radar and the gun (figure 2). They can be moved both together and independent of each other. The vehicle can be moved ahead and back, in order to reposition the robot's body. Any of the parts can be turned left and right. Turning the vehicle changes the direction of further moves, turning the scanner

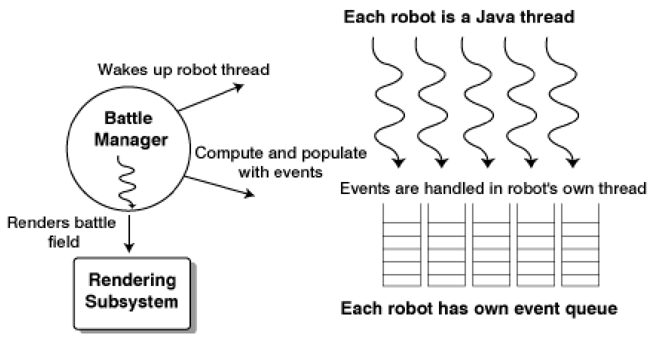
Each robot is a Java thread
Wakes up robot thread
Battle Manager
Compute and populate with events
Events are handled in robot's own thread
Renders battle field
Rendering Subsystem
Each robot has own event queue

Fig. 1. Robocode simulation engine architecture (see [1])

changes the area that the robot "senses" and turning the gun sets the direction where the bullet will be fired on.

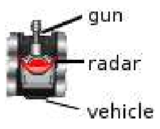

Fig. 2. Anatomy of a Robocode robot

B. Robocode physics

Robocode *time* is measured in "ticks", which are equivalent to frames displayed on the screen. Each robot gets one turn per tick. *Distance* measurement is done in pixels, with double precision, so a robot can actually move a fraction of a pixel. Robots *accelerate* at the rate of 1 pixel/tick and *decelerate* at the rate of 2 pixels/tick. Acceleration is automatically determined based on the distance the robot is trying to move. *Velocity* is calculated multiplying acceleration with time, and can never exceed 8. The direction is always the heading of the robot's body.

Some limits are imposed for the rotation of a robot's parts. Maximum rate of vehicle rotation is set such as it is limiting the ability to turn with speed. Maximum rate of gun rotation is 20 degrees per tick and is added to the current rate of rotation of the robot. Maximum rate of radar rotation is 45 degrees per tick and is added to the current rate of rotation of the gun.

Collisions cause damage, decreasing the energy level of the robot(s) involved. For example, hitting another robot damages both of them with 0.6, while a robot hitting a wall will take damage according to its speed.

III. Logic programming agent architecture

The motivations behind our choice to implement a new logic programming agent architecture were already explained in section I. Still, we want to emphasize the practical need that drove our research effort. A strong community and momentum exists around Robocode both as an educational game and a tool for robot programming hobbists [3]. Agent oriented programming, on the other hand, is a field holding much promise not necessarily from a technological point of view, but mostly as a new design method. The shift in focus from designing parts (objects) that interact with each other in order to build a system that solves a problem to a more autonomous way of designing software poses new challanges. Artificial intelligence, and especially logics, play an important role in finding solutions to these problems. We believe that robot programming with Prolog is a field that was not explored enough. Our agent architecture is directed at making the entry into this field much easier for researchers and software engineers alike.

The architecture we propose is built around the natural event-driven nature of Robocode. We provide the means for a Prolog program to receive events from the game and to issue commands to it, thus implementing the basic requirements of an agent: "[it is] perceiving its environment through sensors and acting upon that environment through effectors" [5].

A. Overall design

Any robot implementation in Robocode has to inherit from `robocode.Robot` class (or any subclass of it), and our design can not be an exception. We wrote a generic Java class `PrologRobotShared` that acts as an adaptor between the game environment and the concrete Prolog implementation of a robot. Its main responsability is to pass along events from the game engine, and to start the main strategy loop.

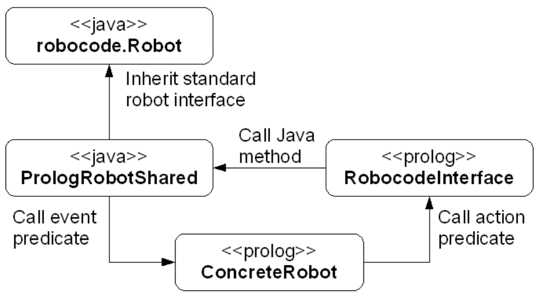

Fig. 3. Interaction model of a concrete Java-Prolog robot

By convention, the predicate that is being invoked when the game starts is `startStrategy`. Just as its Java counterpart, the `run` method, this predicate must not finish. Instead, this is where the overall strategy is specified. Acting upon the environment is possible using a collection of predicates that wrap the action calls from `robocode.Robot`. Calling any of these from the `startStrategy` predicate will freeze the robot thread while the game engine actually executes the command.

In order to improve the design and reusability of code between robot implementations, we separated any generic Prolog predicates to a different component (file), called `RobocodeInterface`. We also made sure that there is no Java code that needs to be changed if the robot's logic is changed.

B. Perceiving the environment

There are two ways of getting information regarding the environment. In the Robocode API there is a group of methods named `get*`. Most important ones are listed in table I. For all these wrapper predicates were implemented in `RobocodeInterface`, with exactly the same name. We can call this way of perceiving the environment *passive*, as these should be called from within the `startStrategy` predicate or any of the event handlers (see next section). The information they gather is always there, as it is the state of the game or of the robot.

The alternative way of perceiving the environment is by responding to events. A list of the most important events available is listed in table II. For all these a Java handler was written that calls the handler predicate with the same name from the concrete Prolog robot implementation, if it exists. We call this event-driven way *active*, as the environment interrupts the robot to inform it of a change in the environment or a sensory activity.

Maybe the most important of these methods is `onScannedRobot`. It is called when a robot's radar scan detects the presence of another robot. Responding to this event is usually a good strategy or at least a potentially strategy altering event.

C. Acting upon the environment

A couple of "hardware" behavior properties can be altered using the `set*` methods listed in table III. These don't really count as effectors, but more like altering future behavior of effector methods. A couple more strictly cosmetic methods exist to set the appearance of the robot icon on the board. We did not implement these as Prolog predicates. If the user wants to change the default colors, they should do it directly in `PrologRobotShared`.

In table IV are listed the real effectors available to use for the robot's implementor. All these were implemented and are available in Prolog. Any call to these predicates will freeze the robot thread execution until the required action is completed, or an event occurs.

IV. Reference implementation of an agent

Developing a framework can not be done without testing. We did a reference implementation of a robot in order to validate our approach, and provide an example of how to get started. We deliberately chose a simple robot, already present in the Robocode distribution as a sample. What we did was to start from the Java implementation and rewrite it using the framework. The end result is a functional Java-Prolog program implementing the same functional requirements as the original one.

A. Corners: a simple robot

We chose a robot named "Corners" for our implementation. The strategy it applies is very simple: it chooses a corner, based on previous experiences, then proceeds to secure it. After it took the corner, it aims and shoots at the other robots using a fairly simple targetting algorithm.

B. Implementing the strategy in Prolog

The strategy of a Robocode robot is made up, in general, of two parts: the main loop, triggered by the method `run()`, and the event handlers (see table II). Our implementation of the main loop initialises and customizes the robot, then calls the predicate `startStrategy` (listing 1). The first argument (R) of this predicate is received from Java and is a reference to the `PrologRobotShared` instance. It is necessary in order to issue commands back to the framework (actions, see table IV).

Algorithm 1 Implementation of $startStrategy$ in Prolog

```
startStrategy(R) :-
        getOthers(R, Num),
        retractall(others),
        assert(others(Num)),
        goCorner(R),
        spinGunLoop(R, 3).

spinGunLoop(R, X) :-
        spinGun(R, X, 30),
        NewX is -X,
        spinGunLoop(R, NewX).
% ...
```

Algorithm 2 Implementation of $onScannedRobot$ in Java and Prolog

```
public void
onScannedRobot(ScannedRobotEvent e) {
 executeQuery("onScannedRobot",
                new JRef(e));
}

onScannedRobot(R, E) :-
        stopWhenSeeRobot(Result),
        onScannedRobot(R, E, Result).

onScannedRobot(R, E, true) :-
        stop(R),
        getDistance(E, Distance),
        smartFire(R, Distance),
        scan(R),
        resume(R).

onScannedRobot(R, E, false) :-
        getDistance(E, Distance),
        smartFire(R, Distance).
%...
```

The event handlers are implemented in Java as delegators to Prolog, to predicates with similar names (listing 2). A ref-

TABLE I
ROBOCODE API: GATHERING INFORMATION ABOUT THE ENVIRONMENT AND SELF

getBattleFieldHeight	the height of the current battlefield measured in pixels	getBattleFieldWidth	the width of the current battlefield measured in pixels
getGunCoolingRate	the rate at which the gun will cool down	getWidth	the width of the robot measured in pixels
getHeight	the height of the robot measured in pixels	getName	the robot's name
getNumRounds	the number of rounds in the current battle	getRoundNum	the current round number (0 to `getNumRounds` - 1) of the battle
getOthers	how many opponents are left in the current round	getTime	the game time of the current round
getGunHeading	the direction that the robot's gun is facing, in degrees	getGunHeat	the current heat of the gun
getEnergy	the robot's current energy	getHeading	the direction that the robot's body is facing, in degrees
getRadarHeading	the direction that the robot's radar is facing, in degrees	getVelocity	the velocity of the robot measured in pixels/turn
getX	the X position of the robot	getY	the Y position of the robot

TABLE II
ROBOCODE API: EVENTS A ROBOT CAN RESPOND TO

onScannedRobot	the robot sees another robot (i.e. the robot's radar scan "hits" another robot)	onStatus	called every turn in a battle round in order to provide the status to the robot
onBulletHit	called when one of robot's bullets hits another robot	onBulletHitBullet	called when one of robot's bullets hits another bullet
onBulletMissed	called when one of robot's bullets misses (i.e. hits a wall)	onHitByBullet	called when the robot is hit by a bullet
onHitRobot	called when the robot collides with another robot	onHitWall	called when the robot collides with a wall
onBattleEnded	called after end of the battle, even when the battle is aborted	onDeath	called if the robot dies
onRobotDeath	called when another robot dies	onWin	called if the robot wins a battle

TABLE III
ROBOCODE API: BEHAVIORAL PROPERTIES THAT CAN BE SET

setAdjustGunForRobotTurn	turn the gun independent from the robot's turn
setAdjustRadarForGunTurn	turn the radar independent from the gun's turn
setAdjustRadarForRobotTurn	turn the radar independent from the robot's turn.

TABLE IV
ROBOCODE API: ACTIONS A ROBOT CAN TAKE

ahead	immediately moves the robot forward	back	immediately moves the robot backward
doNothing	do nothing this turn (i.e. skip this turn)	stop	immediately stops all movement, and saves it for a call to `resume()`
resume	immediately resumes the movement you stopped by `stop()`, if any	turnLeft	immediately turns the robot's body to the left
turnRight	immediately turns the robot's body to the right	scan	scans for other robots
turnRadarLeft	immediately turns the robot's radar to the left	turnRadarRight	immediately turns the robot's radar to the right
fire/fireBullet	immediately fires a bullet	turnGunLeft	immediately turns the robot's gun to the left
turnGunRight	immediately turns the robot's gun to the right		

erence to the `PrologRobotShared` instance is passed to each predicate, to be able to call the action predicates.

V. RELATED WORK

Research on Agent architectures and frameworks is in progress for some time [9]. Even though, the field is still not developed enough and did not achieve the full embrace of industry yet. Logic programming is a good choice for implementing agents, as shown in several studies [10], [11].

The game Robocode was used in several studies before, including the creation of a team of robots [12] or a single robot's strategy [13]. Genetic programming was applied with some success, leading to generation of Robocode tank fighters [7], [8].

There is no study that we are aware of trying to implement Robocode fighters in Prolog.

VI. CONCLUSION AND FUTURE WORK

A new agent environment and framework for the creation of autonomous agents in Prolog was introduced. It is particularly interesting as it makes available the well known Robocode environment for programming in conjunction with the logic programming system SWI-Prolog. It also enables competitions with other agents built with different technologies, both created by humans and evolved [8]. It provides a good testing ground to validate different theories related to programming autonomous agents.

The framework can also be used to learn and experiment with agent-oriented programming, both in class and as a self-study tool. We plan to use it as learning environment for an undergraduate Logic programming course next year. The source code is released under an open-source GPL license and is available at http://www.info.uaic.ro/ alaiba/robocodepl/.

Several future directions can be followed. Our first priority is to extend the framework to support the full API of the latest version of Robocode (at the time of this writing 1.6.0). Tests will be done to determine the performance penalty, if it exists, that the framework adds on top of existing Java implementations of some common robots.

Programming teams of robots as multi-agent systems is another interesting area of research. Robocode supports the notion of teams and allows for collaborative combat [2]. Support in the framework should be added to enable inter-agent communicaton, both as point-to-point and as broadcast.

In the long term, we plan to add higher abstractions to the framework that will model concepts such as beliefs and goals, and support for different kinds of non-standard logics (such as temporal logic). Ideally this will lead to the development of design patterns for agent programming and their validation against a well defined, real environment.

REFERENCES

[1] S. Li, “Rock ’em, sock ’em Robocode!, Learning Java programming is more fun than ever with this advanced robot battle simulation engine,” *IBM developerWorks,* 2002, http://www.ibm.com/developerworks/java/library/j-robocode/.

[2] S. Li, “Rock ’em, sock ’em Robocode: Round 2, Go beyond the basics with advanced robot building and team play,” *IBM developerWorks,* 2002, http://www.ibm.com/developerworks/java/library/j-robocode2/.

[3] *Robocode homepage,* http://robocode.sourceforge.net/.

[4] S. Franklin and A. C. Graesser, “Is it an agent, or just a program? A taxonomy for autonomous agents,” *in Intelligent agents,* iii, Springer Verlag, Berlin, 1997, pp. 21–35.

[5] J. S. Russell, P. Norvig, *Artificial intelligence: A modern approach,* Prentice Hall, Englewood Cliffs, NJ, 1995.

[6] R. A. Kowalski, “Algorithm = Logic + Control,” *in Comm. ACM,* 22(7), 1979, pp. 424–436.

[7] J. Eisenstein, “Evolving Robocode Tank Fighters,” *in CSAIL Technical Reports,* Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2003.

[8] Y. Shichel, E. Ziserman, “GP-Robocode: Using genetic programming to evolve robocode players,” *in Proceedings of 8th European Conference on Genetic Programming,* 2005.

[9] R. A. Brooks, “A robust layered control system for a mobile robot”, *in IEEE Journal of Robotics and Automation*, 2(1), 1986, pp. 14-23.

[10] J. A. Leite, J. J. Alferez, L. M. Pereira, “MINERVA - A Dynamic Logic Programming Agent Architecture”, *in Lecture Notes In Computer Science, Revised Papers from the 8th International Workshop on Intelligent Agents VIII*, 2333, Springer, 2001, pp. 141–157.

[11] S. Costantini, A. Tocchio, “The DALI Logic Programming Agent-Oriented Language”, *in Proceedings of 9th European Conference on Logics in Artificial Intelligence JELIA, Lecture Notes in Computer Science*, 3229, Springer, 2004, pp. 685-688.

[12] J. Frokjaer, P. B. Hansen, M. L. Kristiansen, I. V. S. Larsen, D. Malthesen, T. Oddershede, R. Suurland, “Robocode—Development of a Robocode team”, Technical note, Department of Computer Science, Aalborg University, 2004.

[13] K. Kobayashi, Y. Uchida, K. Watanabe, “A study of battle strategy for the Robocode”, *in Proceedings of SICE Annual Conference*, Fukui University, Japan, 2003.