Chapter 12
GOAL: A Multi-agent Programming Language Applied to an Exploration Game

Koen V. Hindriks and Jügen Dix

Abstract GOAL is a multi-agent programming language based on the BDI paradigm. It is a logic-based language that supports modular agent design based on established software engineering principles and interaction with environments using an environment interface standard (EIS). GOAL recently won the multi-agent programming contest (MAPC), where two teams consisting of ten agents play against each other in order to explore and defend occupied territory on a distant planet. The MAPC game is a complex and dynamic environment that supports EIS and thus facilitates easy connection of a multi-agent system (MAS) to an environment that is remotely run. We describe the design of the multi-agent solution that won the competition, the EIS interface that was used, and the MAPC scenario.

Keywords Agent programming • Environment interface • Multi-agent programming contest • Testing

1 Introduction

The aim of this chapter is not to describe *yet-another* agent programming language and claim that it is the best on the market. Developing good software for non-trivial applications using the agent paradigm is a highly complex task depending not only on the chosen programming language.

We strongly believe that documenting and discussing projects that use existing agent platforms for software development is useful for a number of reasons. Only

K.V. Hindriks
Delft University of Technology, Mekelweg 4, Delft, The Netherlands
e-mail: k.v.hindriks@tudelft.nl

J. Dix (✉)
Clausthal University of Technology, Julius-Albert-Str. 4, Clausthal-Zellerfeld, Germany
e-mail: dix@tu-clausthal.de

O. Shehory and A. Sturm (eds.), *Agent-Oriented Software Engineering*,
DOI 10.1007/978-3-642-54432-3_12,

by actually using such platforms can we learn about the effectiveness and usability of them as well as about the issues we are facing during such projects. Based upon findings related to the development process itself, comments by software developers, and facts derived from inspection of the agent software developed, insights may be gained in how agent technology is best applied and how the application of agent technology can be made more effective. We can learn new lessons from how software developers or programmers actually used the tools and technology at hand and the choices they made while doing so. We also gain more insight into the needs of agent programmers.

In this chapter, we present an example project that, given the current state of the art, represents one of the larger coding projects that used a *logic-based agent programming language* for developing multiple software agents that control non-player characters in a dynamic and real-time gaming environment. The language that was used is the agent programming language GOAL [8, 9, 16]. This agent platform supports an environment interface standard (EIS) [2]. The gaming environment that was used is the multi-agent programming contest (MAPC) made available for the 2011 contest. The MAPC game is what we call here an *exploration game* that requires multiple vehicles to explore an unknown map, and compete with opponent vehicles for resources. We discuss and analyse how the winning team of MAPC 2011 developed their code base, their approach and most important design decisions and strategies, and discuss the testing strategies that were used by the team.

The chapter is organized as follows. Section 2 introduces the GOAL agent programming language and provides the background necessary for understanding the project that we discuss. In Sect. 3, the MAPC is introduced. This section also discusses the EIS that is supplied with the MAPC software to support easy interaction between a multi-agent system (MAS) and the simulation environment. Section 4 discusses the design and strategy for the MAS implemented in GOAL. Finally, Sect. 5 presents lessons learned and concludes the chapter.

2 The Agent Programming Language GOAL

In this chapter, we present a project that has used the GOAL agent programming language for programming a MAS. It is one of the many agent programming languages that support the agent-oriented programming paradigm [3]. These languages explicitly aim for the construction of autonomous software agents. Most agent programming languages are based on the concept of a *cognitive agent*, derived from the belief–desire–intention (BDI) model of agency [7]. Such cognitive agents maintain a mental state that typically consists of one or more variants of the BDI components, including knowledge, beliefs, desires, goals, and/or intentions. These mental states are used for representing an agent's environment and for decision making or planning. In rule-based agent programming languages, rule libraries that are provided by a programmer are used by the agent to decide what to do next. Agents typically execute a *deliberation or reasoning cycle* similar to the

sense-plan-act cycle. Agent programming languages also provide support for agent interaction by means of communication at the knowledge level [11], that is, in terms of what they believe and desire to achieve.

We briefly introduce the main concepts of the agent programming language GOAL that was used by the team to program their MAS for the MAPC 2011. Some code snippets are provided in Sect. 4. We refer the reader to [8, 9, 16] for more detailed information about the language.

GOAL is a *logic-based* agent programming language for programming *cognitive agents*. GOAL agents maintain a mental state that consists of *beliefs* and *goals* and derive their choice of action from their beliefs and goals. GOAL agents also use a *knowledge base* to represent conceptual and domain knowledge. The current version of GOAL uses Prolog to represent the knowledge, beliefs, and goals of an agent.[1] Prolog is a declarative programming language. A Prolog program consists of *Horn clauses*, which are logical rules and simple facts [15]. These clauses represent *what* is the case and *what* is desired; computation in Prolog is performed by evaluating queries by means of an inferencing process. GOAL agents use Prolog for deriving new conclusions from their beliefs about the environment and the goals they want to achieve in combination with the knowledge that they have.

One of the main strengths of the language is that it facilitates the development of *high-level strategies* for agents. GOAL is a *rule-based* language. The philosophy of GOAL is that writing agent programs essentially means writing rules that determine for each situation that the agent finds itself in what it should do in that situation. Rules are ordered, which allows for imposing a priority on what needs to be done first by an agent. On top of this design philosophy GOAL mainly adds two things: a basic *reasoning cycle* and *modular programming*.

GOAL supports a basic reasoning cycle that consists of *two phases*. The purpose of the first phase is to process all *events* such as percepts and messages and make sure that the agent's mental state is up-to-date. In this phase, the GOAL agent retrieves and processes all *perceptual information* available from the environment. Percepts received can be used to update the beliefs and goals of the agent. The idea is that an agent should first make sure its mental state is up-to-date before it decides on a choice of action. The second phase of the cycle is about decision making: Agents decide what to do next. Typically, in this phase one *environment action* is selected and sent to an environment (it is also possible to perform more than one environment action in one cycle if needed). After completing the second phase, the cycle is repeated.

The concept of a *module* is a key programming construct in GOAL for structuring and writing larger agent programs. A GOAL agent *is* a set of modules. With each of the phases of the reasoning cycle corresponds a built-in module. The `event` module corresponds to the first phase and is designed to support event processing

[1] The GOAL agent programming language does not commit to Prolog or any other computational logic in particular (cf. [8]). In principle, other languages such as Answer Set Programming or ontology languages such as OWL might also be used.

whereas the `main` module corresponds to the second phase and is designed to support decision making. In addition, a special `init` module is available for initialising the mental state and other components of an agent. More importantly, however, a programmer can add and write its own set of modules for structuring and organizing code. A module provides a container for a set of rules and thus provides an abstraction mechanism: A module can be used for coding more abstract actions as well as for programming roles of agents.

GOAL is a multi-agent programming language and supports *communication* between agents. Both communication from agent-to-agent as well as broadcasting information to all other or a selected set of agents is available. GOAL also supports the distributed running of agents in a MAS on multiple machines.

The GOAL language is distributed with an Integrated Development Environment for coding, testing, and debugging. It provides the usual program editing tools as well as tools to analyse the code (e.g., creating an overview of predicates used in a program). It also provides extensive debugging tools including introspectors for inspecting agent states, stepping functionality, (conditional) breakpoints, runtime querying and modification of agent states, tracing and logging functionality at different levels of granularity, and basic performance measurements of Prolog queries.

The GOAL platform, moreover, fully supports the EIS [2]. EIS provides an elegant interface for interacting with environments. It facilitates the exchange of actions from agents to an environment and the exchange of percepts from an environment to agents. As we will discuss below in more detail, this allowed the team to focus completely on the strategic aspects of the MAPC scenario and no time needed to be spent on low-level details related to, for example, communicating with the simulation server.

3 The MAPC

The MAPC has been annually organized by the CIG-group from Clausthal University of Technology since 2005 [1]. The contest has been initiated with the aim of putting agent programming frameworks to the test, gaining new insights and detecting problems with these platforms that may stimulate research in the area of MAS development and programming [18]. The focus of the contest has shifted more and more toward *coordinated action*, which is perceived as a key issue associated with MAS design and, therefore, should be an essential ingredient in any scenario for evaluating multi-agent programming languages, platforms, and tools. More pragmatically, the contest is also expected to be useful for debugging existing agent platforms and tools and for identifying the strengths and weaknesses of various platforms.

Since 2005 various scenarios have been used in the contest, including *food gathering* (2005), *gold mining* (2006–2007), *cows and cowboys* (2008–2010), and a *Mars* scenario (2011). Scenarios have been changed to focus the contest more and more on coordinated action. All of the scenarios, however, have required agents to

explore an *unknown map*. The maps used have been *grids with obstacles* except for the Mars scenario which uses a *graph* as a map. In essence, therefore, all of the contest environments can be classified as *exploration games*. In addition, all scenarios are *competitive* and require two agent systems to compete for scarce resources.

The performance of a system developed by a participating team is determined in a series of matches where the systems contributed by various teams compete against all other agent systems. A single match between two competing agent systems consists of several simulations. Winning a simulation yields three points for a team, a draw is worth one point and a loss zero points. The winner of the whole contest is evaluated on the basis of the overall number of collected points in all the matches. HactarV2, the MAS discussed in the next section, scored the highest possible score of 72 points whereas the runner-up scored 60 points.

Technically, the contest is realized by means of a test-bed environment specifically designed for the MAPC called the *MASSim* (multi-agent systems simulation) platform that provides the server infrastructure for running the contest. The contest scenario is realized as a plug-in for the MASSim platform. Participating agent systems connect via TCP/IP to and exchange plain XML messages with the simulation server. In other words, agents receive percepts encoded as XML messages from the server and can act in the gaming environment by encoding their actions as XML messages and transmitting them to the server. The MASSim test-bed supports round-based game simulations where all agents are allowed to perform one action in each round. Agents need to act in real time because the window for transmitting a valid action to the server for each agent is fixed. In the 2011 scenario this time window has been reduced from the 4 s it used to be to 2 s. Taking into account that participating teams are located all over the world and connect via the Internet, which introduces latency, this means that agents need to act well under 2 s to ensure they submit an action to the server in time. After a finite number of steps the simulation server stops and the agents that participated in a simulation receive a notification about the end of that simulation.

3.1 The 2011 Mars Scenario

The Mars scenario used in the 2011 contest concerns an exploration game on the planet Mars [19]. The game requires a set of vehicles to explore, locate and occupy valuable zones on the planet Mars. At the start of a game, vehicles are placed randomly on an unknown map. At first vehicles therefore need to individually explore the map and exchange information. Vehicles need to coordinate their actions to occupy a zone of the planet that is as large as possible.

Story The story of the scenario is that water wells have been discovered on planet Mars. The objective of a team of vehicles is to identify locations with large water wells and to occupy those places. Because multiple companies want to profit from

this discovery, a team will have to compete for the possession of water wells. A graph is used to represent Mars, where nodes denote locations and have a value indicating the amount of water that is present in a well. The graph is mirrored to provide a fair symmetric map on which ten vehicles from each team can move around.

Roles Vehicles are each assigned one out of five different roles: *explorer*, *sentinel*, *inspector*, *saboteur*, and *repairer*. Given that ten vehicles are available, each role is evenly distributed and assigned exactly twice. Explorers can determine the amount of water at nodes. Sentinels have a better vision to provide more information about what happens on the planet. Inspectors can determine the roles and status of opponent vehicles. Saboteurs have the ability to attack and disable opponent vehicles. Repairers are able to restore disabled vehicles back to a working state.

Scoring Scheme Two teams play a match over three games each with a duration of 750 steps. The final score of a game is the total of all the step scores in that game. Each team starts the game with ten achievement points which can be spent on upgrades. A team can collect more points by gaining achievement points for actions like attacking enemies and exploring the map. A zone score is determined each step by the nodes that are controlled/guarded by the agents of a team and is computed as the sum of the values of all the nodes in the controlled area. This means that a zone with higher valued nodes will provide a better score. The step score then is determined by adding the number of unspent achievement points to the zone score of that step. This scoring mechanism thus requires a team to weigh and balance scoring achievement points by performing particular actions such as exploring a node or maximizing the value of the occupied zone on the map. Typically, at the start of a game vehicles are not "connected" yet and therefore do not occupy a zone.

3.2 Support for the Environment Interface Standard

The MAPC software provides an implementation of the EIS interface [2] to facilitate easy connection to the MAPC server. This interface automatically establishes and maintains connections to the MASSim-server. It provides support for configuring some parameters of the simulation, registering agents, associating agents with the vehicles in the game, starting a simulation, perceiving the simulation environment, and acting in it. Because the GOAL platform fully supports EIS, a programmer does not need to concern himself with low-level details of connecting to an environment and the functionality that MAPC provides is made available without requiring any effort from a programmer. In addition, the support for the EIS interface by GOAL also ensures that a programmer does not need to concern himself with the low-level details of the XML-format for percepts and actions that is used by the MASSim-server. Instead, a programmer can concentrate completely on how to handle these percepts. Similarly, a programmer can focus on coding a strategy for selecting actions without any need to consider how the environment is able to process actions.

Actions and Percepts We briefly describe some of the more important actions and percepts out of the ten actions that can be performed and out of the 33 percepts that may be received from the simulation environment. For more details, refer to [19].

Actions have a name and some of them have a parameter which identifies a MAPC entity by its name. A saboteur can perform an attack on any vehicle that is in the same location by `attack(<Identifier>)` and the vehicle can use the action `parry` to defend against an attack. Upgrades can be bought by performing `buy(<Identifier>)`. A vehicle moves to a neighbour vertex by performing `goto(<Identifier>)`, which has an energy cost equal to the weight associated with the traversed edge. The actions `probe` and `survey` yield, respectively, the amount of water present on the current vertex and the weights of visible edges.

An action may cost energy, health, and achievement points (money). These costs vary depending on the success or failure of the action, and on whether the agent is in a normal or disabled state. Actions may fail at random with a certain probability and may yield achievement points for six different types of achievements that can be realized. Achievement points can be scored by probing a specific number of nodes, surveying a specific number of edges, inspecting a specific number of opponent vehicles, performing a specific number of successful attacks, performing a specific number of successful parries, and by obtaining points for a zone that is occupied.

Just like an action a percept consists of a name followed by a (possibly empty) list of parameters. Besides names represented by `<Identifier>` a percept may also provide numerical information represented by `<Numeral>`. Percepts differ per individual vehicle and depend on the location and range of sight of the vehicle. Percepts are omitted with a certain probability by the server. The Mars simulation environment provides a large number of different percepts to inform agents about what is going on during the game. Agents are informed about their role, the actual and maximal amounts of energy, health, and strength they can have, their visibility range, whether an action was performed successfully or not, the amount of money (achievement points) available to the team, the total number of vertices and edges present in a simulation, the current round number, and the current (zone) score. The percept `achievement(<Identifier>)` indicates an achievement that has been realized. `position(<Identifier>)` provides the name of the vertex the vehicle is on. `probedVertex(<Identifier>,<Numeral>)` and `surveyedEdge(<Identifier>,<Identifier>,<Numeral>)` yield, respectively, the result of a probing and survey action. Several percepts such as `simStart`, which indicates the start of a simulation, are available that inform agents about the current state of the simulation and the server.

4 Developing a Multi-agent Program for MAPC

This section provides a detailed overview of the code development process of the MAS *HactarV2*. HactarV2 performed exceptionally well during the contest and won every single one of the 24 simulation games against eight other teams. The MAS

has been programmed completely in the agent programming languageGOAL. One of the strengths of GOAL is that it facilitates the development of high-level strategies for agents by providing a declarative way to represent and reason about an agent's beliefs and goals.

We provide some information and statistics about the project to indicate the project's size and effort that went into developing the MAS. The agent system has been developed by a team of six students at the Delft University of Technology (henceforth referred to as *the team*). All team members were familiar with GOAL because it is being taught as a first year bachelor course in the Computer Science curriculum at Delft University of Technology. The agile software development approach *Scrum* [14], supported by the open-source platform *iceScrum*[17], has been used to manage the project. The team decided not to use an agent-based development methodology such as Prometheus[13] because of a lack of experience with these methodologies. In total, the team spent roughly 500 man hours on the project. About 60 % of the time was spent on implementing and debugging the multi-agent strategy and the remaining 40 % was spent on system performance and other problems. The final code base consists of 1,758 lines of code spread over 18 files.

The MAS has been run on a single high-end desktop computer consisting of an Intel core i7-870 quad-core CPU running at 3.53 GHz, and 8 GB of DDR3 RAM running at 1,600 MHz. The option of distributing the MAS on multiple machines was considered as a possibility, mainly for performance reasons, but because the MAS turned out to be efficient enough to run on a single machine this option was not investigated any further. The team considered the development on a single machine to be easier. This poses a challenge because the MAS needs to control ten nonplayer characters that each individually need to act within a two second time frame. As explained, because communication with the server over the Internet takes time as well, in fact this means that each agent needs to decide on an action within about a 100 ms (given that agents take turns on a single machine).

In the remainder of this section, we discuss the design of the MAS (Sect. 4.1), the overall flow of control (Sect. 4.2), the ontology that was developed (Sect. 4.3), the testing strategies of the team (Sect. 4.4), and briefly assess the code base against a set of proposed design guidelines (Sect. 4.5).

4.1 Design of the HactarV2 MAS

The design of the HactarV2 MAS has been based on several observations related to the game. Most importantly, two phases may be distinguished within the game: a first phase in which agents do not yet act as a team (initially agents are randomly placed on the map) and a second phase in which agents act as a team in order to occupy valuable zones on the map. The 2011 MAPC map generator produces maps that have a single cluster of higher valued nodes more or less at the center of the map. Because of this, the two phases can be clearly distinguished from each other

based upon the fact whether or not this zone has been identified. The main strategy therefore consists of finding these nodes of highest value by means of explorer vehicles and, when such a node (called the *optimum* by the team) is found, informing all other agents about this node so they will start moving toward this node as well. The second phase is called the *swarming* phase because agents are perceived as being part of a swarm that aims at occupying a zone of valuable nodes that is as large as possible.

Because the MAS identifies a single node as the "center" of the *optimum zone* at most one swarm will be created. Agents that are part of this swarm identify the highest valued node directly outside the zone occupied by the swarm and move toward that node. By using this tactic the swarm will always expand in the direction of the highest valued nodes that are not yet owned by the MAS.

Finally, in games where it is difficult to occupy a large, valuable zone, the points that are obtained by achievements can determine the difference between winning and losing. Based on the observation that attack and parry actions yield the most achievement points it was decided to focus on these achievements. Of course, inspections of opponent vehicles and probing nodes, for example, also need to be performed to do well in the game.

It is clear that this general design of the MAS strategy completely depends on a proper *understanding of the MAPC simulation environment*. Such an understanding comes about only after running a MAS in the environment. This suggests that initial experimentation with and testing of a MAS in an environment is a very important aspect of designing a MAS.

Decentralized Coordination and Communication Strategy One of the main challenges of the Agent Contest is to design a *decentralized* MAS that is able to strategically compete with other agent teams. This excludes, for example, the design of a MAS with a central manager that has access to all information available in the MAS and sends instructions to individual agents what to do. The team decided to address this challenge by designing a strategy of HactarV2 that is based on *implicit coordination* between agents.

Another reason for choosing a decentralized design over a centralized design that uses a managing agent is that a decentralized design may reduce the need for communication if properly designed. A managing agent that coordinates the activities of all other agents creates overhead because all information needs to be made available to this manager agent and instructions need to be send back to these agents, which can significantly impact performance.

In order to minimize the communication between agents, agents were designed to base their decisions mainly on the information that is perceived by the agent itself. The main exception concerns the information that is obtained by different agents about the map. Map information is shared by communication between agents because more knowledge about the map can be used to optimize the exploration process and allows agents to prevent doing probe and survey actions twice. Sharing this information may require each agent to process up to 90 messages that are received from the other nine agents per round. Because all agents have to process

received map information, special attention has been paid to optimizing the updating of an agent's beliefs with this information. Although in a centralized design only the central manager would maintain a map and need to perform such updates, this single agent would still have to process all information received from all other agents.

In addition to messages about the map, messages with requests for repairs are exchanged between disabled agents and repairer agents and messages with information about the location of opponent agents are exchanged between non-saboteur agents and saboteur agents. Communication has been optimized by making sure that an agent will only send a message if it knows that the receiving agent does not perceive this information itself (which can be deduced from local information and previous messages).

One of the key issues that needs to be addressed in the design of a decentralized MAS concerns the question *how to avoid that agents perform the same action*. Decisions of agents on the action it will perform need to be coordinated to avoid doing the same thing twice. To this end, agents in the HactarV2 MAS have been equipped with the capability to *predict* what other agents will do. Using a simple *agent ranking principle* each agent then can decide by itself which action to perform and rule out conflicts. For example, this principle is used to decide which out of multiple agents on the same edge of the occupied zone will perform a move to another node to expand the zone. The basic idea is simple: Agents that are located on the same node are ranked and assigned a unique number called the *agent's rank*.[2] This rank is used to arbitrate between multiple agents that are about to perform the same action. This mechanism allows agents to divide tasks without having to communicate and ensures that each agent performs a unique action whenever possible.

The design choice to develop a decentralized MAS for the MAPC environment has raised some interesting issues that need to be taken into account. Two issues stand out: The design needs to explicitly deal with *minimizing communication overhead* and the *prevention of the duplication of effort* by agents. One mechanism for dealing with the latter issue used in the HactarV2 team is *prediction* of what other agents may do. An interesting topic for future research is the question whether, and if so, which, alternative mechanisms may be employed to the same end.

Agent Roles and Strategies Apart from the overall MAS strategy discussed above, various goals and strategies were designed and identified at the agent level including strategies for specific *agent roles*, for *defence*, and for *buying* upgrades.

The main goal of *explorer* agents at the start of the game is to locate the highest valued node on the map, called the *optimum*. Once this node has been found, it is the task of the explorer agent to communicate the name of this node to the other agents and start forming a swarm that occupies the zone around this node. The strategy for finding the optimum consists of performing `probe`, `survey` and `goto` actions according to a set of specific rules: Always probe a node if it has not been probed

[2]This can be done, for example, by using the fact that GOAL attaches numbers to names in order to create unique names for each agent.

yet and survey any edges that have not been surveyed yet. The agent then will go to a node that has not been probed yet only if (a) this node is connected to the current and last visited node and (b) the current node has a lower value then the last visited one. If there is a neighboring node that has a higher value than the current one, the agent will go there. The agent will also try to go to a neighbouring node that is not close to a (potentially) dangerous opponent but the agent will take a chance in case there is no such node. Otherwise, the agent will go back to the last visited node, if unexplored options are available at that node. The agent will conclude that the optimum has been found if no move can be made any more. This conclusion may not always be right but turned out to work well in practice. Once the "optimum" has been found, an explorer agent will team up with the other agents and start swarming around this node. It will continue to probe nodes as doing so allows for finding even higher valued nodes than the currently believed optimum.

The defensive strategy of an explorer is to move away from nodes it considers unsafe. A node is considered to be unsafe if an opponent agent is located on that node that is either a saboteur or its role is unknown.

A *sentinel* agent basically uses the same exploration strategy as explorer agents. The defensive strategy of a sentinel is to parry opponent saboteurs. If successful, parry achievements are gained. If the opponent's role is unknown, a repairer will also initially parry. However, if no attack was performed, with a 50 % chance, a sentinel agent will ignore opponents with unknown roles on the same node.

An *inspector* agent also uses the same exploration strategy as explorer agents. The difference is that an inspector agent gives priority to inspecting opponents in order to identify saboteurs, to keeping track of the status of these agents (by repeating inspection of these agents every 50 rounds), and to sharing this information with all other agents. The defensive strategy of an inspector is to move away only from known opponent saboteurs.

The main goal of *repairer* agents is to repair friendly disabled agents. Priority is given to repairing a disabled repairer agent and repairs of other agents are interrupted when a repairer is itself disabled or upon receiving a request from another repairer agent. Disabled agents request a repairer agent for help and will start moving toward the closest repairer. They send a path to the repairer they are moving to which prevents the repairer from having to calculate the same path. Repairers use the same defensive strategy as sentinels.

The main goal of *saboteur* agents is to disable opponent agents. These agents move toward a nearest and last known location of an opponent agent to attack that agent. Tests showed that this strategy reduced the effectiveness of opponent teams. Saboteurs do not have a defensive strategy but are designed to be superior to any opponent agent by means of HactarV2's buying strategy to which we now turn.

Buying is an important aspect of the game but the team considered achievement points (money) more important and they decided to try to spend less money than the opponent does. The reason is that the amount of money available each round has a high impact on the score for that round. Although the team experimented with sentinels that buy sensors to increase visibility range, this performance gain was considered insufficient compared to the costs and the team decided to only upgrade

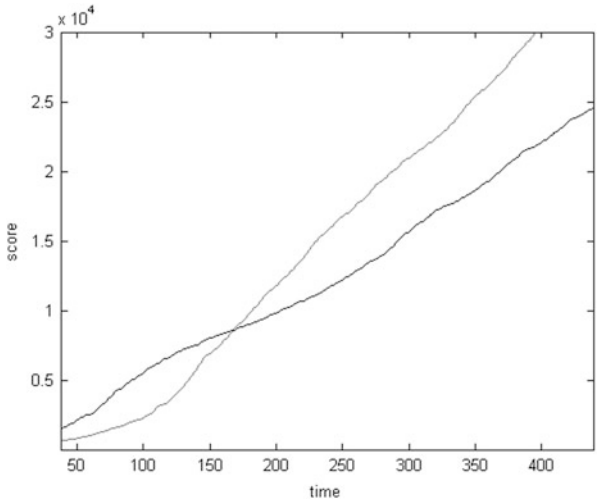

Fig. 12.1 Scores HactarV2 (*gray*) vs TUB (*black*)

saboteur agents. Upgrades are bought right at the start of the game and throughout when it is discovered that upgrades are needed to match opponent health or strength. Upgrading is aimed at two things: (a) saboteurs have one health point more then the maximal strength of opponent saboteurs and (b) the strength of saboteurs is at least equal to the maximal health of opponent saboteurs. If both these goals are realized, saboteur agents will survive opponent attacks while disabling opponents by a single attack. The initial investment at the start of the game means that the score often is lower than that of opponents in the first 100 or so steps but it starts to pay off in the remainder of the match. See Fig. 12.1 for an example game illustrating this.

The fact that the exploration strategy is among the most complex strategies matches the fact that the MAPC environment is what we have called an *exploration game*. It is obvious that in a game of competition that has entities with different roles agent specific role and defensive strategies need to be designed. More interestingly, however, is the fact that the design of the buying strategy is derived from the results of extensive testing, which highlights again the importance of this activity.

4.2 Control Flow of the MAS

GOAL agents execute an Observe–Orient–Decide–Act (OODA) loop [5].[3] At the start of a reasoning cycle of an agent, events including percepts and messages are collected (Observe) and processed by means of so-called *event rules* (Orient). This

[3] In many areas of competitive activity, the theory is that if you can cycle through the OODA loop faster than your opponent, you have the advantage.

ensures that an agent can make a decision based on the most up-to-date information available. A decision on what to do next (Decide) is made using so-called *action rules*. Upon making a choice, the action selected is sent to the environment (Act).

The control flow of this cycle matches the general structure of a GOAL agent program. More specifically, the `event` module of a GOAL agent corresponds to the Observe–Orient part of the loop and its `main` module with the Decide–Act part of the loop. A programmer can add additional structure to the agent's cycle by adding as many user-defined modules as needed. For example, code related to percept handling, communication, navigation, and roles can be placed in separate modules.

A more detailed overview of the structure and flow of control of the event module is provided by the diagrams in Fig. 12.2a, b. Horizontal rectangular boxes in the figures refer to particular modules and submodules, whereas vertical rectangular boxes indicate the flow of control. The notation **LA** in the latter boxes indicates that the order of rule evaluation in the corresponding module is **L**inear and that **A**ll applicable rules need to be applied (in order). This linear-all style of rule evaluation is the default mode for the **init** and **event** modules. In all other modules, the default mode is a linear style mode of evaluation where *only the first* applicable rule is applied. Using the **order** option the rule evaluation style of a module can be changed. This explains the fact that the submodules such as `selectPercepts`, etc. are also indicated in Fig. 12.2 to use linear-all style evaluation.

An agent starts a new cycle upon receiving information from the simulation server that a new round has started. The `commonPercepts` module handles the percepts that every agent uses. The `surveyVertices` module processes vertex-related percepts and broadcasts this information to the other agents if a successful `survey` action just was performed. Next role specific modules handle any role specific percepts. The `selectReceive` module then processes messages, which in a similar fashion uses various sub-modules. For example, a disabled agent uses module `disabledReceiveMail` to handle messages specific to disabled agents. The `clearMailbox` module finally cleans the mailbox of an agent by deleting all received and sent messages.

After all events are processed and the mental state of the agent is made up-to-date again, the agent decides what to do next in the **main** module. Instead of providing a flow diagram for this module, we list the code in Fig. 12.3. As explained above, the rule evaluation in this and user-defined modules is linear style. This means that the action rules in these modules are evaluated one by one from top to bottom and only the first applicable rule is actually applied. If no decision has been made yet (**not**`(doneAction)`), first it is checked whether the agent is disabled and the `disabled` module is entered in that case to ensure the agent gets itself fixed as soon as possible. A special case where the MAS is in control of the entire map (`allMapAreBelongToUs`[4]) because all opponent agents are disabled is checked next which is handled by the `superioritySelect` module. Only if none of

[4]See http://nl.wikipedia.org/wiki/All_your_base_are_belong_to_us.

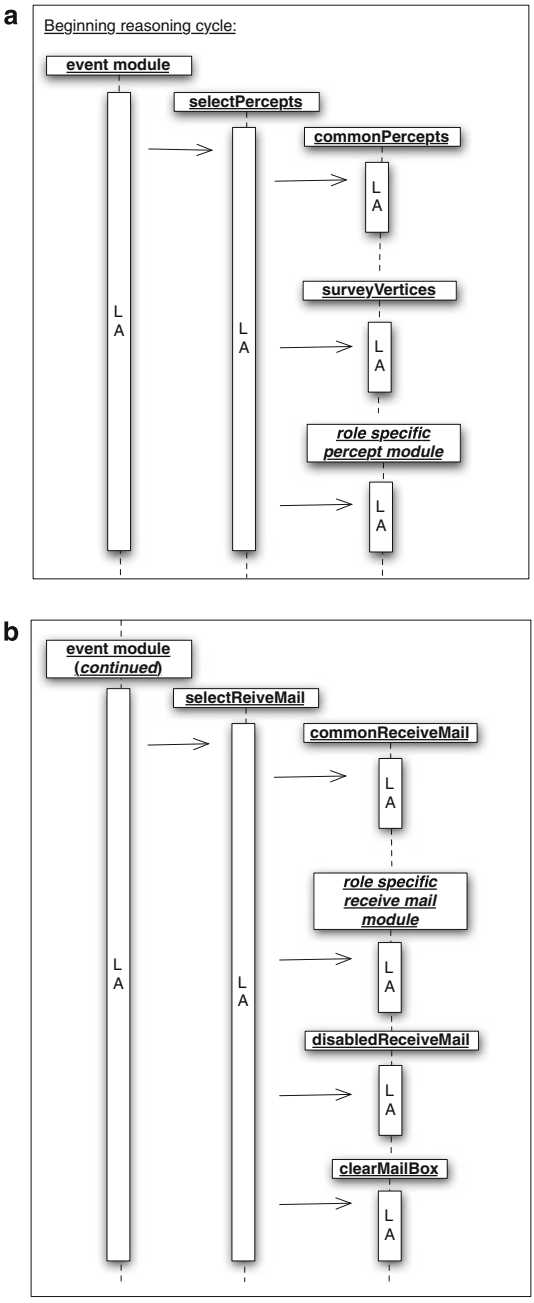

Fig. 12.2 Control flow of the event module. (**a**) First part; (**b**) second part

```
main module{
  program{
    if bel(not(doneAction)) then {
      if bel(disabled, not(role('Repairer'))) then disabled.

      if bel(allMapAreBelongToUs) then superioritySelect.

      if bel(role('Repairer')) then repairerAction.
      if bel(role('Inspector')) then inspectorAction.
      if bel(role('Explorer')) then explorerAction.
      if bel(role('Saboteur')) then saboteurAction.
      if bel(role('Sentinel')) then sentinelAction.

      if bel(true) then explore.
    }
  }
}
```

Fig. 12.3 Main module code

these cases apply enter agents their role specific modules. Finally, if there are no role specific tasks that need to be performed these modules are exited and an agent will try to swarm, or, if that is not an option explore the map.
The use of modules has several benefits. It facilitates programmers that are part of a team to each focus on a specific part of code while at the same time maintaining a clear structural overview of the MAS. It also reduces the chance of code duplication. And last but not least, it facilitates structuring code of roles by means of a pattern similar to the Strategy design pattern [6]. The agent program of every agent in our MAS uses the same structure while still being able to handle agent specific roles due to code that allows an agent to adapt to the particular role associated with a vehicle.

4.3 Ontology

Besides the 33 percept predicates that agents may receive from the environment, in the HactarV2 MAS an additional 60+ Prolog predicates were defined that are used throughout the agent program. In a team of programmers where each programmer codes part of the MAS it is important to have easy access to such large numbers of predicates and their intuitive meaning. Code in a sub-module of the **main** module, for example, may depend on predicates in the belief base that are updated in a sub-module of the **event** module. One lesson learned from a first year bachelor project where student teams have to program a MAS for controlling bots in the real-time, first-person shooter game UNREAL TOURNAMENT 2004 [10] is that the teams that did a better job at maintaining an *ontology* outperformed other teams and obtained better results in the final competition. For this reason, the team also maintained an ontology for the HactarV2 MAS.

Ontology Structure An ontology for a GOAL MAS documents all predicates that are used in the MAS code base. As Prolog is used, the ontology documents in the

usual Prolog format `name/nr` a predicate named `name` that has `nr` of arguments. For example, `enabledEnemy/2` means a predicate `enabledEnemy` with two arguments is used. The ontology maintained by the HactarV2 team in the form of a table also indicates the *type* of a predicate label, that is, whether it is used for representing a belief, goal, percept, or knowledge of an agent. It also briefly explains the intuitive meaning of each predicate, how its parameters should be instantiated, and the code base location where the predicate is defined (i.e., the file where it is used).

Example Predicate Definitions In the remainder of this section, we briefly discuss and illustrate two of the predicates used and their definitions. Important other predicates that were defined were used, for example, for implementing the agent ranking principle discussed above (Sect. 4.1), path planning,[5] and for keeping track of which vehicles that are part of the team can be relied upon.

The concept of an agent being connected to others is used in the swarming phase, that is, the second phase of the game. It is an important concept for establishing that nodes are owned by a group of agents. The nodes that connected agents are located on are also called *swarm positions*. Informally, an agent is said to be *connected* if that agent has links with at least two other agents it can depend on. A link between two agents is said to exist if there are at most two edges that connect the nodes on which these agents are located and these nodes are owned by the agent team. The concept is implemented by the predicate `connectedAgent/2`; see Fig. 12.4). Figure 12.4 also lists the most important predicates related to swarming.
Code explanation: the predicate `connectedAgent/2` indicates whether the second agent is connected to the first agent. This means the second agent must be one or two edges apart from the first agent, and must not be considered an independent agent (see below for the concept of independency); `connectedPos/2` does the same as `connectedAgent` but instead of reasoning from the position of the first agent it reasons from any node position; `edgeDest/1`: finds a list of probed nodes (and their corresponding values) that are not in the optimum zone but have a direct edge to a node in the optimum zone; `swarmPos/1`: a vertex that is a swarm position is a vertex that makes sure the agent is still connected to two other agents; `expandPos/1`: checks if a node is neutral (has no vertex owner) and is a swarming position; `expandDest/2`: finds all expanding destination (using expandPos) from the agents current position; `bestExpandDest/3`: finds the highest value expanding position to expand the swarm to from a certain node.

Recall that the map generator produces maps that have one cluster of higher valued nodes at the center of the map. It is the goal of explorers to locate these high valued nodes and identify the *optimum* node. Occupying a zone around this node is very important during the game. Such a zone is called the *optimum zone*. In order

[5] It is often argued that path planning is better delegated to another software component that is not programmed using a logic-based agent programming language. The HactarV2 agents, however, use Prolog for path planning and implement variants of Dijkstra's shortest path algorithm.

```
connectedAgent(Agent1, Agent2) :- team(Team),
  visibleEntity(Agent1, Pos1, Team, normal),
  visibleEntity(Agent2, Pos2, Team, normal),
  visibleEdge(Pos1, Pos2), not(independableAgent(Agent2)),
  vertexOwner(Pos1, Team), vertexOwner(Pos2, Team).
connectedAgent(Agent1, Agent2) :- team(Team),
  visibleEntity(Agent1, Pos1, Team, normal),
  visibleEntity(Agent2, Pos2, Team, normal),
  visibleEdge(Pos1, Pos3), visibleEdge(Pos3, Pos2),
  not(Pos1 == Pos2), not(independableAgent(Agent2)),
  vertexOwner(Pos1, Team), vertexOwner(Pos2, Team),
  vertexOwner(Pos3, Team).

connectedPos(X, Agent) :- currentPos(Agent, Y),
  not(independableAgent(Agent)), visibleEdge(X, Y).
connectedPos(X, Agent) :- currentPos(Agent, Z),
  not(independableAgent(Agent)), not(X == Z), visibleEdge(Z, Y),
  team(Team), vertexOwner(Y, Team), visibleEdge(Y, X).

edgeDest(List3) :- neighboursOfOptimumZone(F), !,
  findall([Value, Vertex], (member(Vertex, F), vertexValue(Vertex, Value),
     not(Value == unknown)), List),
  not(List == []), sort(List, List2), reverse(List2, List3).

swarmPos(X) :- connectedPos(X, Agent1), connectedPos(X, Agent2),
  not(Agent1 == Agent2), !.

expandPos(ID) :- vertexOwner(ID, none), swarmPos(ID).

expandDest(List3, Pos):-
  findall([Value,Neighbour],(neighbour(Pos,Neighbour), expandPos(Neighbour),
    vertexValue(Neighbour,Value), not(Value==unknown)),List),
  not(List == []), sort(List, List2), reverse(List2, List3).

bestExpandDest(ID, Value, Pos):- expandDest(List,Pos), List=[[Value,ID]|_].
```

Fig. 12.4 Related predicates and predicate definition for `connectedAgent/2`

```
allInformationOptimumZone([], [], []) :- not(optimum(_)), !.
allInformationOptimumZone(Agents, Nodes, Neighbours) :-
  optimum(Opt), team(Team),
  allInformationOptimumZone([Opt], [], Nodes, Agents, Neighbours, Team),!.

allInformationOptimumZone([], _, [], [], [], _).
allInformationOptimumZone
 ([First|ToConsider], Visited, [First|Nodes], Agents, Neighbours, Team) :-
  vertexOwner(First,Team),
  findall([Agent, First], visibleEntity(Agent, First, Team, normal), Agts),
  findall(Node,(e4(First, Node, _), not(member(Node, Visited))), TempNodes),
  list_to_set(TempNodes, FoundNodes),
    union(FoundNodes, ToConsider, NewToConsider),
  allInformationOptimumZone(NewToConsider,[First|Visited], Nodes,
        NewAgents, Neighbours, Team),
  union(NewAgents, Agts, Agents).
allInformationOptimumZone([First|ToConsider], Visited, Nodes, Agents,
 [First|Neighbours], Team) :- not(vertexOwner(First, Team)),
  allInformationOptimumZone(ToConsider, [First|Visited], Nodes,
  Agents, Neighbours, Team).

inOptimumZone :- me(Id), agentsInOptimumZone(A), member([Id,_], A).
```

Fig. 12.5 Related predicates and predicate definition for `allInformationOptimumZone/3`

to be able to reason about this important zone, various predicates related to this concept have been defined; see Fig. 12.5.

Code explanation: `allInformationOptimumZone/3`: finds all nodes and agents that are currently in the optimum zone. The definition uses the helper

predicate `allInformationOptimumZone/6`. Using a breadth first search this latter predicate finds all nodes owned by the team that have a path to the optimum node, using only nodes that are owned by the team. It also finds all the agents that are currently on these nodes as well as all neutral and enemy owned nodes that have an edge to these nodes; `inOptimumZone/0` checks if the agent is currently in the zone that contains the optimum.

Maintaining an ontology facilitates keeping track of what programmers that are part of a team are doing. The HactarV2 team has reported that using an ontology has saved them a lot of time. They found that it is important to pay special attention to the predicates that are used for representing the environment. An ontology also provides support for understanding the program code and communication between team members. The GOAL platform provides some functionality for automatically identifying the predicates that are used and warns if redundant predicates are present in a MAS. Given the usefulness of an ontology it is worth while to consider extending this functionality and provide more automated support for maintaining an ontology.

4.4 Testing

The team has put a lot of effort into testing and analysing the results while developing the MAPC MAS and reported that extensive testing was very important for becoming familiar with the gaming environment. We briefly discuss the various testing strategies that were used by the team.

The Use of Dummy Agents It is important to test whether the MAS has bugs without other agents disturbing the environment. In order to do so, dummy agents that do nothing were used as opponents. Problems such as agents getting stuck at a certain point, or performing no operations at all are more easily detected and solved this way. For example, if a vehicle controlled by an agent does not perform an action, by stepping through the code of the agent in debugging mode it is often relatively easy to determine what goes wrong in a set-up with dummy agents (other options such as that an agent has been disabled cannot occur in this case).

Strategy Testing Testing is not only suitable for detecting and solving errors but also needed for measuring the performance of a MAS. In a competitive setting, an adequate and readily available way of measuring performance is by testing a current version of the MAS against older versions. This yields insight into whether recent code changes have improved performance. During these tests the team observed suboptimal behavior that they believe could only have been found because the strategy of the opposing MAS of an older version is still quite similar (assuming testing is regularly performed). For the same reason why it is a good idea to test against earlier versions it is also necessary to test against MAS written by other teams, whenever the possibility is available. Only by doing so are issues detected that occur only against MAS that have a very different strategy.

```
module debug[exit=noaction]{
  program{
    if bel(debug(attack(X))) then attack(X).
    if bel(debug(survey)) then survey.
    if bel(debug(probe)) then probe.
    ...
    bel(debug(moving(X)), currentPos(X))
       then delete(debug(moving(X))) + explore.
  }
}
```

Fig. 12.6 Example debug module for the MAPC

Debug Modules In order to properly test agent behavior, it is necessary to create particular situations in an environment to be able to observe the behavior that a program produces in those situations. In order to create such situations, it can be useful to "manually" assign each agent a new task at runtime. For example, it is often quite useful in the MAPC environment to direct an agent to go to a particular location and stay there.

It is useful to have support for setting up particular situations. In GOAL, the team came up with the idea of using a combination of a what they called a *debug module* and so-called *debug facts*. An example of such a module is provided in Fig. 12.6. A debug module is a module like any other module with the name `debug`. The module includes a set of simple action rules that are executed when a corresponding `debug` fact is part of the agent's belief base. A debug module is used in combination with a feature in GOAL that allows to insert new beliefs in the belief base of a particular agent while the MAS is running. Once an agent believes a "debug fact," it will deviate from its normal behavior and will immediately give full priority to the rules in the corresponding debug module. The team reported that this proved to be a very useful debugging tool. All agents can, for example, be instructed to line up in a particular way to make it easy to test a strategy or situation in a controlled manner.

Real-Time Debugging An important problem with testing a MAS is that the environment upon which the system acts is highly dynamic. Many different agents perform actions in real time and continuously affect the state of the environment.

A more specific testing tactic that was used while debugging the MAPC MAS involved the use of an edited XML configuration file for a simulation which granted the agent team two million seconds for sending actions. According to official game settings, all agents have only 2 s to submit their actions. While stepping agents in debugging mode, however, such a time limit is too strict and action would not be submitted in time. In the MAPC environment, this would mean that agents that are being debugged perform skip actions, while the opponent MAS is sending valid actions. By raising the time limit for submitting an action, the server would "pause" during that time and it is possible to complete debugging a simulation step. As a result, bugs were found more easily.

Testing was performed at all levels distinguished in [12], including unit, agent, integration, system, and acceptance testing. Acceptance testing in this context meant testing the system in the environment provided by the MAPC organizers. This required some creativity, as discussed above, from the team. Testing of MASs

may be differentiated from other types of software systems and is particularly challenging due to the many interactions that need to be taken into account. Agents run concurrently and interact with other agents, both by means of communication as well as by interacting in a shared environment, and need to take into account how to coordinate their actions or compete with other agents for resources. The metrics that needed to be considered for the MAPC competition in particular were related to real-time performance and the performance of the MAS in terms of the scoring scheme of the competition. Other metrics related to code quality are discussed in the next section.

Generally speaking, the lesson learned from this project is that the more a MAS is being tested the better it is. As noted above, testing is very important to gain a proper understanding of the environment a MAS needs to be programmed for. Interestingly, some techniques were used by the team that can be reused in other cases. The idea of putting an agent in "debug mode" by means of debug modules in order to create specific testing conditions provides only one example. An issue that often arises while debugging MAS for complex environments concerns real-time. Whereas for the MAPC environment the real-time pace of the game could be controlled, parameters for doing so are not available for all environments. It is therefore clear that more effort is needed to improve the tooling for effective debugging and for developing effective testing approaches for multi-agent programs [1].

4.5 A Look at the HactarV2 Code Base

Due to space limitations, we only discuss and illustrate a small but important part of the code base related to the swarming behavior of agents that occupy a zone.

The swarming module shown in Fig. 12.7 is a key module during the second phase of a game, when the objective of the MAS is to occupy a zone that is as large as possible. To be more precise, the objective is to obtain a higher zone score than the opponent team. In order to do so, the swarming agents sometimes will even reduce the occupied zone in order to maintain a steady flow of score instead of aiming for occupied territory that is easily disrupted by the opponent. Agents will only enter this module when they do not have any more important role specific tasks to perform, such as repairing a broken agent or destroying an enemy saboteur that is disrupting one of the repairers.

The module heavily depends on several defined predicates, such as the `swarmPos` predicate defined in Fig. 12.4. The module also uses the agent rank system to efficiently distribute possible moves between agents. Two main cases are distinguished in the module: the agent is (a) inside the occupied zone (dealt with by the first rule) and (b) on a boundary node and has options available for expanding the zone. Using the `expandDest/1` predicate these options are retrieved, the agent's rank is determined, and using the `expandDest/2` predicate options of connected agents are retrieved. In case the agent is allowed to expand (it has more options than other agents), it does so using the `moveSplit/2` module.

```
program{
  if bel(insideZone, edgeDest(List), agentRankHere(Rank))
    then moveSplit(Rank, List).
  if bel(expandDest(List),List=[[Value,Vertex]|_],me(Id),agentRankHere(Rank))
    then {
      if bel(not((connectedAgent(Id, Agent), currentPos(Agent, Pos),
             bestExpandDest(_, Value2, Pos), Value2 >= Value)))
        then gotoSplit(Rank, List).
      if bel(not(kingOfTheHill), Rank2 is Rank-1)
        then gotoSplit(Rank2, List).
      if bel(currentPos(Pos), not(swarmPos(Pos)), optimum(Opt),
                path(Pos, Opt, [Here,Next|Path], _))
        then advancedGoto(Next).
    }

  if true then recharge.
}
```

Fig. 12.7 Program section of the swarming module

The code fragments discussed provide some indication of the quality of the code produced but do not provide an overall perspective. More generally, we can assess the code quality produced by the HactarV2 team by means of a set of design guidelines that have been proposed for GOAL agent programs [16]. Part of these guidelines also concern the earlier discussed topics of ontology and testing. Table 12.1 provides an overview of these guidelines and indicates to what extent they were followed.

It turns out that the HactarV2 team followed most of the design guidelines for producing quality code but not all. Overall, rules were grouped according to purpose (e.g., communication rules were grouped together) and a declarative style of programming has been used. Percepts are not all handled in the event module, however, and the deletion of facts was not always handled by the **delete** action. The team explained that they did not follow these guidelines for reasons of performance and preferred less expensive queries here instead of relatively expensive update actions. Other items that stand out concern the high level of testing and the fact that a project management tool was used at the start of the project but not used actively any more later on.

5 Conclusion

The design and development of a MAS for an exploration game such as the MAPC involves all challenges that will typically be encountered when developing a MAS for controlling a complex and dynamic environment. We think that the Mars contest scenario poses some interesting challenges with respect to coordinating agents. We discussed the programming project and results of a team of six bachelor students that coded a MAS they called *HactarV2* that won the 2011 contest. The team had to *design a winning strategy* for ten agents in a competitive environment facing ten opponent agents, *design a coordination strategy* for coordinating the activities of

Table 12.1 Which design guidelines and best practices were followed?

Code quality and style	
Predicate labels are declarative	✓
Beliefs represent current state	✓
Program does not contain redundant predicates	✓
Knowledge represents conceptual and domain logic	✓
Agent program uses goals	✓
Goals are declarative	✓
Goals are concrete	✓
Only action specifications for environment actions are present	✓
All environment actions are declared in the `init` module	✓
Specified action preconditions match environment constraints	✓
`insert` is used to add and `delete` is used to remove beliefs	✗
Action rules are only used in the `main` module or linked modules	✓
Percepts are only used in the event module	✗
Percepts are handled by `forall` rules	✗
Communication rules are located in the event module after percept handling code	✓
Rules for goal management are located at the end of the event module	✗
Unrelated modules are placed in separate files	✓
Comments (documentation)	
% predicates in knowledge base that are explained in comments	100 %
% action specifications that are explained in comments	50 %
% of modules the use of which are explained in comments	90 %
% of program rule groups that are explained in comments	100 %
Ontology (documentation)	
The ontology was kept up to date throughout the project	✓
Items in the ontology are properly explained	✓
The ontology was used by team members during the project	✓
Testing	
Level of testing during project	High
Team performed module tests	✓
Team performed full MAS system tests	✓
Team performed systematic tests on domain configurations	✓
Project management	
A project management tool was used during the project	✓
The project management tool was kept up to date throughout the project	✗

these agents, *develop a relatively large code base* as a team, and *perform extensive tests* to validate the performance and strategy of the MAS that was developed.

The code for the MAS that was developed has been completely written in the logic-based agent programming language GOAL. This made the coding project a useful object for our study to learn more about the actual use of such a language in a relatively larger project. We discussed key aspects of the project including program

and strategy design, the use of modules in a team programming effort, the ontology used for reasoning and representing the gaming environment, strategies for testing a MAS, and we briefly discussed whether code followed proposed design guidelines.

According to the team, the concept of a *module* for structuring code turned out to be of great value. Modules were used to write code for specific roles that were used by only some agents, as well as for shared functions such as navigating the map, for communication, etc. The team reported that being able to structure code by means of modules facilitated the division of coding tasks among team members. Furthermore, writing code in a logic-based agent programming language as a team requires that all team members are aware of the logical predicates that are used throughout the code. We have called this the *ontology* used by the MAS. The team reported that documenting and updating the ontology while developing code facilitated team coordination and saved time. The team followed most but deviated also from some of the proposed guidelines for quality code mainly for reasons of efficiency.

In conclusion, we have found that developing a MAS is far from trivial. In particular, testing a MAS remains one of the key challenges that seems to set development of such a system apart from other software systems. The key differences are the potentially large number of agents that may have different roles and the fact that the MAS is developed for controlling entities in and is connected to an external environment that cannot be fully controlled. A development team needs to become familiar with the external environment at the start of a project. This means that different testing strategies may be useful at the beginning than toward the end of a project. The team used some interesting techniques for debugging some of which can be applied more generally to the development of other MASs as well. In particular, in a *gaming* environment where bots *compete* for resources, at the start of a project it may be more effective to use dummy opponents that pose little or no challenge while testing initial versions of a MAS. A testing approach to evaluate whether subsequent versions of the MAS improve the system's performance is *self-play*, that is, have a newer version play an older version.

In this chapter, we have made an attempt to gain insights from a coding project that uses a logic-based agent programming language. We believe that we have been able to identify and illustrate some useful strategies for making such a project a success. Much, however, remains to be done and we believe it would be useful to draw more lessons learned from other agent-oriented software engineering projects. Analysis of such projects is useful for identifying coding (design) patterns, best practices, improving agent-based development tools, and developing automated testing tools, as well as evolve agent programming languages in a way that enhances their use in real-world applications [20].

Acknowledgments We would like to recognize the effort the students put into developing the HactarV2 MAS and their help in explaining their code while writing this chapter. The chapter is partly based on the MAPC paper for the HactarV2 MAS [4].

References

1. Behrens T, Dastani M, Dix J, Köster M, Novák P (2010) The multi-agent programming contest from 2005–2010. Ann Math Artif Intell 59(3):277–311
2. Behrens TM, Hindriks KV, Dix J (2011) Towards an environment interface standard for agent platforms. Ann Math Artif Intell 61(4):261–295
3. Bordini R, Braubach L, Dastani M, Seghrouchni AEF, Gomez-Sanz J, Leite J, O'Hare G, Pokahr A, Ricci A (2006) A survey of programming languages and platforms for multi-agent systems. Informatica 30(1):33–44
4. Dekker M, Hameete P, Hegemans M, Leysen S, van den Oever J, Smits J, Hindriks KV (2012) Hactarv2: an agent team strategy based on implicit coordination. In: Dennis L, Boissier O, Bordini RH (eds) 9th International Workshop, ProMAS 2011, Taipei, Taiwan, 3 May 2011, Revised Selected Papers. LNAI, vol 7217, pp 173–184
5. Eaton J, Redmayne J, Thordsen M (2007) Joint analysis handbook, 3rd edn. Joint Analysis and Lessons Learned Centre, Lisbon. www.jallc.nato.int
6. Freeman E, Freeman E, Sierra K, Bates B (2004) Head first design patterns, 1st edn. O'Reilly Media, Inc., Sebastopol
7. Georgeff MP, Pell B, Pollack ME, Tambe M, Wooldridge M (1999) The belief-desire-intention model of agency. In: Proceedings of the 5th international workshop on intelligent agents, vol V. Agent theories, architectures, and languages (ATAL '98). Springer, Berlin, pp 1–10
8. Hindriks K (2009) Programming rational agents in goal. In: Multi-agent programming: languages, tools and applications. Springer, Heidelberg, pp 119–157
9. Hindriks K, de Boer FS, van der Hoek W, Meyer J (2001) Agent programming with declarative goals. In: Intelligent agents VII agent theories architectures and languages. Springer, Berlin, pp 248–257
10. Hindriks K, van Riemsdijk B, Behrens T, Korstanje R, Kraayenbrink N, Pasman W, de Rijk L (2011) UNREAL GOAL bots. In: Dignum F (ed) Agents for games and simulations, vol II. Lecture notes in computer science, vol 6525. Springer, Berlin, pp 1–18. http://dx.doi.org/10.1007/978-3-642-18181-8_1
11. Newell A (1981) The knowledge level. AI Mag 2(2):1–20
12. Nguyen C, Perini A, Bernon C, Pavn J, Thangarajah J (2011) Testing in multi-agent systems. In: Gleizes MP, Gomez-Sanz J (eds) Agent-oriented software engineering, vol X. Lecture notes in computer science, vol 6038. Springer, Berlin, pp 180–190
13. Padgham L, Winikoff M (2003) Prometheus: a methodology for developing intelligent agents. In: Proceedings of the 3rd international conference on agent-oriented software engineering, vol III (AOSE'02). Springer, Berlin, pp 174–185
14. Schwaber K (1995) Scrum development process. In: Proceedings of the 10th annual ACM conference on object oriented programming systems, languages, and applications (OOPSLA), pp 117–134
15. Shapiro L, Sterling E (1994) The art of prolog: advanced programming techniques. MIT Press, Cambridge
16. The GOAL website (2012). http://ii.tudelft.nl/trac/goal
17. The iceScrum website (2012). http://www.icescrum.org/en/
18. The Multi-Agent Programming Contest website (2012). http://www.multi-agentcontest.org/
19. The Multi-Agent Programming Contest 2011 website (2012). http://www.multi-agentcontest.org/2011
20. van Riemsdijk MB, Hindriks KV, Jonker CM (2012) An empirical study of cognitive agent programs. Multiagent Grid Syst 8(2):187–222