Chapter 11

Example: A Society of Small Agents

Much research concerning the design of multi-agent systems (at a conceptual level) addresses complex agents that exhibit complex interaction patterns. Due to this complexity, it is difficult to perform rigorous experimentation. On the other hand, systematic experimental work regarding behaviour of societies of more simple agents, while reporting valuable results, often lacks conceptual specification of the system under consideration. In this chapter, the DESIRE modelling framework is not only used to develop a conceptual specification of the simple agents discussed in (Cesta, Micelli & Rizo, 1996a), but also to simulate the behaviour in a dynamical environment. The prototype automatically generated implementation of the conceptual specification of the simple agents has been used to replicate, and extend, one of the experiments reported in (Cesta *et al.*, 1996a).

This chapter is outlined as follows. Section 11.1 introduces the problem approached in this chapter. Section 11.2 provides a description of the multi-agent system examined in (Cesta et al., 1996a), for which a conceptual specification using DESIRE is introduced in Section 11.3. In Section 11.4, the conceptual specification is further developed, showing a level of detail suitable for automatic prototype generation. Section 11.5 presents results obtained by experimenting with the generated prototype. Section 11.6 discusses the results obtained as well as the relation with the semantic structure developed in this thesis. Section 11.7 provides a complete listing of all knowledge bases used in the primitive components. Preliminary versions of this chapter appeared at the International Conference on Computer Simulations and Social Sciences, ICCS&SS '97, (Brazier, Eck and Treur, 1997a) and in *Applied Intelligence* (Brazier, Eck and Treur, 2001a).

11.1 Introduction

Although much research within the multi-agent community has focused on the design of individual agents and their interaction, other research has addressed emergent behaviour within societies of agents (see for instance the three papers in

the chapter on emergence in (Velde *et al.*, 1996)). The behaviour of an individual agent can often be conceptually specified, as can the interaction between individual agents. The result of interaction between larger numbers of agents in a dynamic environment is often not easy to predict (Axelrod, 1997). Experimental research, in which interaction between agents is studied in a simulated dynamic environment, provides a means to actually test and compare results of interacting agents (Hanks, Pollack & Cohen, 1993). As stated in (Axelrod, 1997), the KISS principle is of utmost importance: the elements of a model need to be fully understood to be able to interpret results of experimentation.

In this chapter, the DESIRE modelling framework is used to conceptually specify individual agents and to examine the behaviour of relatively simple agents within a large group of agents. This method is supported by tools with which detailed specifications (with which the behaviour of individual agents and their interactions is defined) are automatically translated into prototype implementations. To examine such behaviour, experiments reported by (Cesta *et al.*, 1996a) that test social theories by simulating interaction between different types of simple agents (i.e., agents with limited knowledge and capabilities), have been repeated and extended.

Due to the nature of the environment in which the experimentation of (Cesta *et al.*, 1996a) was originally performed (using the MICE testbed (Montgomery & Durfee, 1990)), most information about agent characteristics and behaviour is implicitly defined by the implementation and simulation environment. On the basis of the informal, textual descriptions provided by (Cesta *et al.*, 1996a), a generic model of a simple agent is defined and refined for each of the four types of agents (Cesta et al., 1996a) distinguished: social, solitary, selfish and parasite. One of the aims of this chapter is to show how this approach leads to a flexible, conceptual-level specification, from which prototypes can be generated automatically for experimentation.

The conceptual models of the different agents are fully specified within DESIRE, including knowledge about how each individual agent interacts with its environment, and with other agents. One of the advantages of a conceptual description of an agent and its behaviour is that not only does a conceptual specification define the behaviour of an individual agent explicitly, it can also be easily adapted at a conceptual level (without having to rewrite low-level code for each agent). In the experiments described in (Cesta *et al.*, 1996a), agents could move in four different directions. In this chapter, not only are these experiments repeated with agents automatically implemented from the conceptual DESIRE specifications, additional experimentation is performed to examine the influence of an increase in the number of directions (8 instead of 4) in which agents can move.

11.2 The Original Experiment

(Cesta *et al.*, 1996a) examined the behaviour of different types of agents in interaction. Four types of agents are distinguished on the basis of their social

characteristics: social agents, parasite agents, solitary agents and selfish agents. The effect of an agent's social characteristic on interaction with other agents is measured by simulating agent behaviour in a situation in which 30 agents try to survive on a 15 * 15 grid in which 60 pieces of food are continually available in random positions. An agent's welfare is measured on the basis of its energy level. The end result of a simulation is the number of agents that survive in a given society of agents, given the energetic value of the food available. Agents do not communicate explicitly but implicitly: a hungry agent changes colour, and this can be seen by other agents. Agents' social characteristics are assumed to be static. An agent does not change from being, for example, selfish to social. The implications of agents' social characteristics for its behaviour are shown below in Table 11.1, taken from (Cesta *et al.*, 1996a), p. 131.

TYPE OF AGENT	INTERNAL STATE	GOAL
Solitary	*any*	Find Food
Parasite	*any*	Look for Help
Selfish	Danger	Look for Help
	Hunger, Normal	Find Food
Social	Danger	Look for Help
	Hunger	Find Food
	Normal	(if help-seekers are seen:) Give Help (if no help-seekers are seen:) Find Food

Table 11.1: Relationships among Types of Agent, Internal States, and Goals (from (Cesta et al., 1996a), p. 131).

The effects of interaction between societies in which 30 agents with varying configurations of social characteristics (for example, 15 social agents and 15 parasite agents in one world), and varying energetic food values have been examined in a number of experiments described in (Cesta *et al.*, 1996a). The MICE testbed, a discrete event simulator, was used to perform the experiments

mentioned above. The MICE testbed does not support a specific agent architecture: agents are LISP functions, activated pseudo-concurrently.

11.3 Conceptual Model of Simple Agents

At the highest level of abstraction, the conceptual model of the society of simple agents consists of 31 components: 30 agents and the external world. The agent components are named agent00 to agent29. The external world is connected to each agent by two links, e.g. agent00_to_world, world_to_agent00, etc. The highest abstraction level of the society of simple agents is depicted in Figure 11.1.

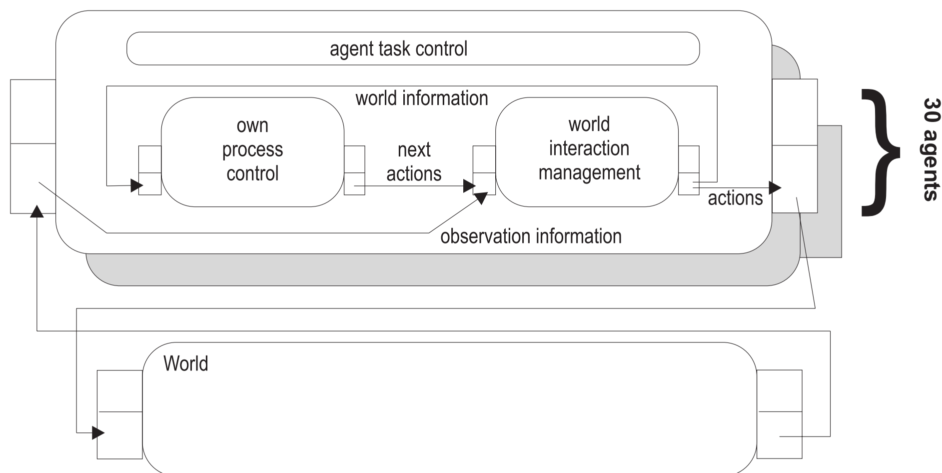

Figure 11.1: Model of the society.

The generic agent model presented in Chapter 3 includes more functionality than required for the small agents described in (Cesta *et al.*, 1996a). These small agents are not capable of communication with other agents and reasoning about other agents' knowledge, nor are they capable of reasoning about communication. Their only task is to stay alive in a dynamic environment. In fact, the only components within the generic agent model applicable to these small agents are the components own_process_control and world_interaction_management. Figure 11.2 depicts not only the remaining composition of a small agent's tasks at the highest level of abstraction, it also shows the information links between the components.

The only information a small agent receives is the information it observes in the external world. This information is forwarded directly to the component world_interaction_management. The component world_interaction_management interprets this information. The result, information about the agent's position, about available food, and, if applicable, information about other needy agents is transferred to the component own_process_control. The component own_process_control determines which

actions should be taken next, depending on the small agent's social characteristics and the agent's direct environment. This information is transferred to the component world_interaction_managment, which derives the information required to actually perform the action in the external world. This information is the only output a small agent provides to the external world. The external world maintains a representation of the grid in which the agents live and is responsible for actually performing the agent's actions by changing the state of the grid with respect to agent's positions and appearance. This updated state may be observed by other agents. (As in (Cesta *et al.*, 1996a), agents have a visibility range of three cells, that is, they observe a rectangular piece of the grid of size 7*7 cells, with the observing agent in the middle of this rectangle.) Other tasks of the external world are to place new food at random locations if a piece of food is eaten and maintaining statistics with respect to the number of alive agents. The internal structure of the component own_process_control is described below in Section 11.3.1. The internal structure of the world_interaction_managment is described in Section 11.3.2. The external world, fully specified in a C program, is not further discussed.

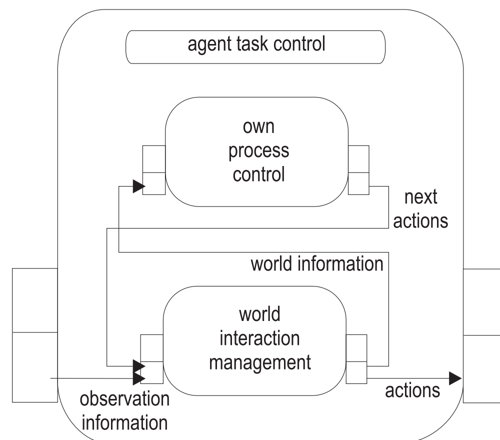

Figure 11.2: Generic structure of a small agent.

11.3.1 The Internal Structure of Component Own Process Control

The component own_process_control is composed of four components: own_resource_management, own_characteristics, goal_determination and plan_determination. The component own_resource_management receives information about its current energy level and the resources it has consumed. This component uses this information to determine its new energy level. On the basis of information the component goal_determination receives about its own social characteristics and its own energy level, it determines the goals the agent is to pursue: for example to find food, or to look for help. The component own_characteristics receives information on the agent's

energy level from the component own_resource_management. This information is used to determine the agent's next state (e.g., hungry, normal or in danger). The component plan_determination receives information (1) from the component own_characteristics, namely the agent's current state, (2) from the component goal_determination, namely which goals are to be pursued and (3) from outside the component, namely the current state of the world. With this information the component plan_determination determines which actions to take in the external world. Figure 11.3 depicts the composition structure of own_process_control.

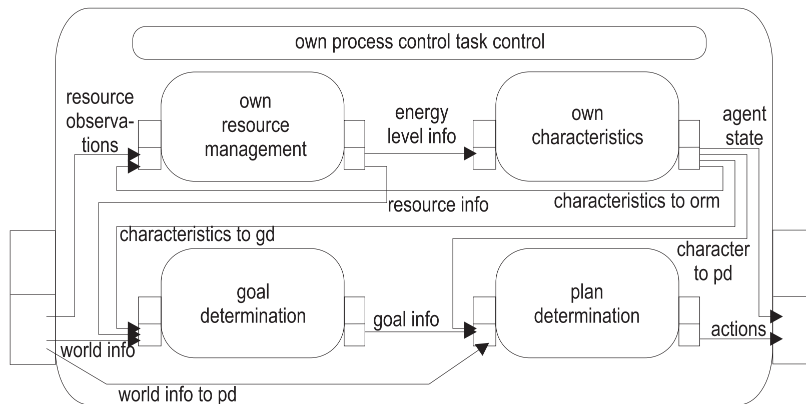

Figure 11.3: Component own_process_control.

11.3.2 Internal Structure of the Component World Interaction Management

The component world_interaction_management interprets information received from the external world, and transforms information about actions to be taken in the external world into specifications for actions to be executed in the external world. Two components are defined to perform these tasks: the component observation_information_interpretation and the component action_execution_preparation.

The component observation_information_interpretation receives information from the external world, for example, information on which pieces of food and which agents have been observed within a given range. This information, termed sensory information in (Cesta *et al.*, 1996a), is translated into information which can be used by the component own_process_control to reason about new goals and plans.

As stated above, the component action_execution_preparation receives information about actions to be taken in the external world from the component own_process_control and translates these actions into specifications to be executed in the external world. These specifications are also the output of effectors in the terminology used in (Cesta *et al.*, 1996a): elementary actions to be performed in the external world. Figure 11.4 depicts the composition structure of world_interaction_management.

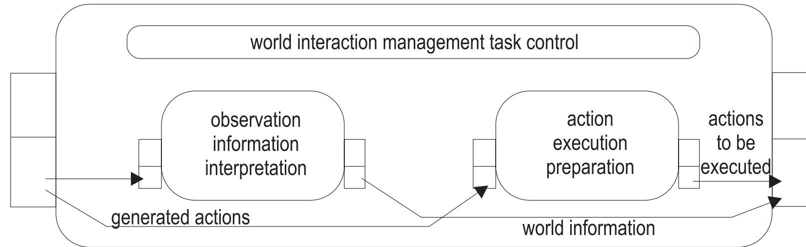

Figure 11.4: Component world_interaction_management.

11.4 Detailed Design

The conceptual generic model of a small agent presented above, has been specified in detail. In this section, relevant examples of the specification are presented to illustrate the level of abstraction with which knowledge is represented. (See Section 11.7 for a complete overview of the knowledge bases used in the primitive components of the agents.) The specification as a whole contains sufficient detail to allow for automatic prototype implementation and simulation. The knowledge structures used by the two main components of a generic small agent, the component world_interaction_management and the component own_process_control are discussed below in more detail, together with the information exchange.

11.4.1 World Interaction Management

The composed component world_interaction_management receives, as input, observation information (obtained from the external world), expressed by the (unary) relation observed, and information from the component own_process_control, expressed by the relation next_action. As output, world information is provided (to be used by own_process_control), and the actions that are to be executed, expressed by the relation to_perform (to be placed in the output interface of the agent).

11.4.1.1 Observation Information Interpretation

The primitive component observation_information_interpretation receives observation information as input and draws conclusions from this information. As an example, the following rules interpret the last action performed by the agent:

if observed(prev_performance(pick_up_food))
then food_in_possession;

if observed(prev_performance(feed_help_seeker))
then **not** food_in_possession;

if observed(prev_performance(feed_self))

```
then    not food_in_possession;
```

These rules state that food is in possession if an agent has previously picked up a piece of food. Food is no longer in possession if an agent has either eaten the food itself, or has given it to a help seeking agent. See Section 11.7.1.1 for the complete knowledge base.

11.4.1.2 Action Execution Preparation

In the primitive component action_execution_preparation the executability of actions that have been selected within the component own_process_control is verified. If an action is dependent on a number of preconditions, these preconditions are tested. If the preconditions are satisfied, the conclusion is drawn that the action should be performed. For example, to be able to take food, that food must be available at the same position as the agent. This is formalised using the following rule:

```
if      next_action( pick_up_food )
        and observed( food_at( cell( 0, 0 )))
then    to_perform( pick_up_food );
```

Cell locations in observations are relative to the position of the agent. Therefore, to be able to pick up food, it must be observed at location *(0,0)*. The component action_execution_preparation also prepares the execution of plans to move to a specific cell. The following rules, marked with a bar in the margin for reference purposes later in this chapter, show the preparation of movement to the north or the south:

```
| if      next_action( move_to( cell( X: ints, Y: ints )))
|         and  Y: ints > 0
| then    to_perform( go_north );

| if      next_action( move_to( cell( X: ints, Y: ints )))
|         and Y: ints < 0
| then    to_perform( go_south );
```

See Section 11.7.1.2 for the complete knowledge base.

11.4.2 Own Process Control

The component own_process_control receives world information from the component world_interaction_management and determines the next actions to be performed, expressed by the relation next_action.

11.4.2.1 Own Characteristics

Within the primitive component own_characteristics an agent's own characteristics are specified. Because the example society contains agents of different types the knowledge differs between agents; it is expressed using the following information types. The first information type expresses the characteristics that in the current chapter are assumed to be static (they are pre-specified), such as the (meta-)fact

that an agent is selfish; the second information type expresses dynamic characteristics, such as being in danger.

```
information type agent_character_it
        sorts   Agent_character
        objects solitary, parasite, selfish, social: Agent_character;
        relations agent_character: Agent_character;
end information type

information type agent_state_it
        sorts Agent_state
        objects dead, in_danger, hungry, normal: Agent_state;
        relations current_state: Agent_state;
end information type
```

In this component, the state of an agent is determined based on its energy level, as shown by the following rule:

```
if      not energy_level < 20
        and energy_level < 60
then    current_state( hungry );
```

See Section 11.7.2.1 for the complete knowledge base.

11.4.2.2 Own resource management

In the primitive component own_resource_management the energy level of an agent is determined. An agent's energy level changes as a result of actions an agent has performed: for example after eating food with food value 40 the result is to_increment_energy_level_with(40). An information link links these atoms to atoms of the form delta_energy_level = 40, after which the new energy level can be determined. The following rule, marked with a bar in the margin for reference purposes later in this chapter, shows that in the case that an agent does nothing, its energy level decreases by one:

```
if      not observed( prev_performance( eat_food ))
        and not observed( prev_performance( go_north ))
        and not observed( prev_performance( go_south ))
        and not observed( prev_performance( go_east ))
        and not observed( prev_performance( go_west ))
        and not observed( prev_performance( change_appearance ))
        and not observed( prev_performance( pick_up_food ))
        and not observed( prev_performance( feed_help_seeker ))
        and not observed( prev_performance( feed_self ))
then    to_increment_energy_level_with( -1 );
```

See Section 11.7.2.2 for the complete knowledge base.

11.4.2.3 Goal Determination

In the primitive component goal_determination an agent determines its goals on the basis of information about its own characteristics, its current state and partial

world information. Selected goals are defined by the following information type:

```
information type agent_goal_it
        sorts Agent_goal;
        objects find_food, look_for_help, give_help: Agent_goal;
        relations selected_goal: Agent_goal;
end information type
```

See Section 11.7.2.3 for the complete knowledge base, which is a direct formalisation of Table 11.1.

11.4.2.4 Plan Determination

Based on an agent's goals the component plan_determination determines which plans are to be executed. The information type to express selected actions is as follows:

```
information type agent_action_it
        information types cell_it;
        sorts Agent_action
        objects
                pick_up_food, eat_food, change_appearance,
                feed_self, feed_help_seeker: Agent_action;
        functions
                move_to: Cell -> Agent_action;
        relations
                next_action: Agent_action;
end information type
```

As an example of action selection, consider the following rule:

```
if      selected_goal( give_help )
        and food_in_possession
        and closest( help_seeker_at( cell( X: ints, Y: ints )))
        and not X: ints = 0
then    next_action( move_to( cell( X: ints, Y: ints )));
```

This rule states that if the selected goal is to give help, and food is already in possession of an agent (so the agent does not have to look for food first), and the closest help seeking agent is observed at location (X,Y), relative to the agent, which is not our own position (and therefore, $X \neq 0$), then the agent decides to go to that cell. See Section 11.7.2.4 for the complete knowledge base.

11.4.3 Control Knowledge

Control knowledge is specified at all levels of abstraction. This subsection briefly describes control knowledge at the highest level of abstraction, namely the society level and one level lower, namely control knowledge within individual agents.

11.4.3.1 Control Knowledge at the Society Level

Control knowledge at the society level specifies how individual agents are activated. One option is to run all agents in parallel, the other is to predefine the

sequence of activation. In the first case only one control rule is needed:

```
if      start
then    next_component_state( external_world, awake )
        and next_component_state( agent00, awake )
        …
        and next_component_state( agent29, awake )
        and next_link_state( world_to_agent00, uptodate )
        …
        and next_link_state( world_to_agent29, uptodate )
        and next_link_state( world_to_agent00, uptodate )
        …
        and next_link_state( world_to_agent29, awake );
```

This control rule specifies that immediately after the system has started, the external world, each individual agent, and the links between the agents and the world, are all made awake. In this state, each agent, a link and the external world are continually ready to receive information and becoming active upon receipt of new information. Moreover, all components immediately start processing as specified by their default task control focus (as explained in Chapter 9). For the external world, this processing determines observations for all agents. For the agents, this processing consists of interpreting observations received from the external world. Thus, immediately after the system has started, the external world is the first component to actually run. After that, observations are transmitted to the agents, which become active concurrently at the moment the observations are received.

A second option is to exercise more control over the execution sequence between agents and the activation of links. In this case, the following control rules may be specified at the society level:

```
if      start
then    next_component_state( external_world, active )
        and next_task_control_focus( external_world, determine_observations );

if      evaluation( external_world, determine_observations, all_p, succeeded )
        and not previous_evaluation( external_world, determine_observations, all_p, succeeded )
then    next_component_state( agent00, active )
        and next_task_control_focus( agent00, default_focus )
        and next_link_state( world_to_agent00, uptodate );

if      evaluation( agent00, default_focus, all_p, succeeded )
        and not previous_evaluation( agent00, default_focus, all_p, succeeded )
then    next_link_state( agent00_to_world, uptodate )
        and next_component_state( agent01, active )
        and next_task_control_focus( agent00, default_focus )
        and next_link_state( world_to_agent01, uptodate );

…

if      evaluation( agent28, default_focus, all_p, succeeded )
        and not previous_evaluation( agent28, default_focus, all_p, succeeded )
then    next_link_state( agent28_to_world, uptodate )
```

```
        and next_component_state( agent29, active )
        and next_task_control_focus( agent29, default_focus )
        and next_link_state( world_to_agent29, uptodate );

if      evaluation( agent29, default_focus, all_p, succeeded )
        and not previous_evaluation( agent29, default_focus, all_p, succeeded )
then    next_link_state( agent29_to_world, uptodate )
        and next_component_state( external_world, active )
        and next_task_control_focus( external_world, determine_observations );
```

The first rule specifies that, immediately after the system is started, the external world becomes active, processes the available information in view of its task control focus, and becomes idle. In this case, a task control focus is set such that the external world makes observations available to each of the agents. The second rule specifies that if the external world has successfully met the evaluation criterion determine_observations, a specific agent, agent00, is made active, and observations from the world are transmitted to this agent by up-dating the link from the external world to agent00. The following rules specify that the agents are activated sequentially, and the relevant links between an agent and the external world, and the external world and the next agent to be activated, are up-dated. The agents transmit actions to be performed to the external world. The external world is activated once all agents have been given the opportunity to convey their own actions to the external world. After that the cycle is repeated. As the experimental system was designed to replicate (Cesta *et al.*, 1996a), option 2 has been implemented.

11.4.3.2 Control Knowledge at the Agent Level

Within an agent, the execution order of the components that correspond to the four tasks identified in (Cesta *et al.*, 1996a) should mirror the execution sequence described in (Cesta *et al.*, 1996a). This means that observation_information_interpretation, goal_determination, plan_determination, and action_execution_preparation should run in this order. However, these components are not at the agent level. Instead, they are subcomponents of the agent-level components own_process_control and world_interaction_management. The following agent-level control rules specify that these components are executed in the correct sequence:

```
if      start
then    next_component_state( world_interaction_management, active )
        and next_task_control_focus( world_interaction_management, interpret_observations );

if      evaluation( world_interaction_management, interpret_observations, all_p, succeeded )
        and not previous_evaluation( world_interaction_management, interpret_observations,
                        all_p, succeeded )
then    next_component_state( own_process_control, active )
        and next_task_control_focus( own_process_control, determine_plan );

if      evaluation( own_process_control, determine_plan, any, succeeded )
        and not previous_evaluation( own_process_control, determine_plan, any, succeeded )
```

```
        and previous_task_control_focus( own_process_control, determine_plan )
then    next_component_state( world_interaction_management, active )
        and next_task_control_focus( world_interaction_management, prepare_action_execution );
```

For each of the components own_process_control and world_interaction_management, more specific control rules at the component level are specified. These rules activate specific subcomponents depending on the current task control focus for the component. More specifically, control rules of own_process_control first activate goal_determination before plan_determination as a result of an activation with task control focus determine_plan.

Control rules at the agent level also determine the activation order of links between own_process_control and world_interaction_management. Moreover, control knowledge for the component own_process_control also specifies the activation order for the subcomponents own_resource_management and own_characteristics. For reasons of brevity, these control rules are not presented here.

11.5 Experimentation

The first goal of this exercise is to replicate the results presented in (Cesta *et al.*, 1996a), on the basis of a conceptual specification of agent behaviour, simulated in the DESIRE software environment. The second goal is to examine the effects of increasing the number of directions in which an agent can move from 4 to 8.

11.5.1 Method

One of the experiments discussed in (Cesta *et al.*, 1996a) has 15 social agents and 15 parasite agents with varying food energetic values in one world, with 500 steps per agent per run. The first experiment based on the DESIRE model used the same number and types of agents and the same number of steps. The second experiment consisted of re-running the first experiment in an environment in which agents could move in more than 4 directions: they could move in 8 directions. The experiments were carried out using an early version of the DESIRE prototype generator, which did not support a hierarchical component structure and which executed prototypes pseudo-concurrently on one processor. The prototype generator was therefore augmented with a custom tool that transformed the hierarchical model presented in this chapter to a non-hierarchical model.

For thes second experiment, extra rules had to be added to the different knowledge bases:

The following rules have been added to the knowledge base of component action_execution_preparation (Section 11.4.1.2):

```
if      next_action( move_to( cell( X: ints, Y: ints )))
        and X: ints > 0
        and Y: ints > 0
then    to_perform( go_northeast );
```

```
if      next_action( move_to( cell( X: ints, Y: ints )))
        and  X: ints < 0
        and  Y: ints > 0
then    to_perform( go_northwest );

if      next_action( move_to( cell( X: ints, Y: ints )))
        and  X: ints < 0
        and  Y: ints < 0
then    to_perform( go_southwest );

if      next_action( move_to( cell( X: ints, Y: ints )))
        and  X: ints > 0
        and  Y: ints < 0
then    to_perform( go_southeast );
```

The rules marked with a bar in the knowledge base of component action_execution_preparation (Section 11.4.1.2) have to be changed:

```
if      next_action( move_to( cell( 0, Y: ints )))
        and  Y: ints > 0
then    to_perform( go_north );

if      next_action( move_to( cell( 0, Y: ints )))
        and Y: ints < 0
then    to_perform( go_south );
```

The following rules have been added to the knowledge base of component own_resource_management (Section 11.4.2.2):

```
if      observed( prev_performance( go_northwest ))
then    to_increment_energy_level_with( -2 );

if      observed( prev_performance( go_northeast ))
then    to_increment_energy_level_with( -2 );
```

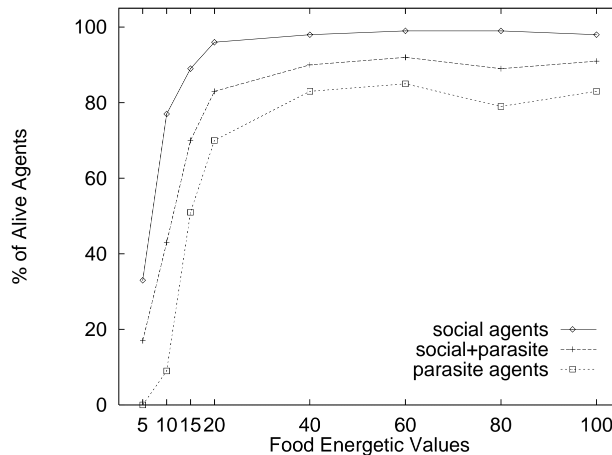

Figure 11.5: Comparison between Parasite and Social (from (Cesta et al., 1996a), p. 133).

```
if      observed( prev_performance( go_southwest ))
then    to_increment_energy_level_with( -2 );

if      observed( prev_performance( go_southeast ))
then    to_increment_energy_level_with( -2 );
```

The rule marked with a bar in the knowledge base of component own_resource_management (Section 11.4.2.2) has to be changed by adding the following conjuncts to the rule condition:

```
and not observed( prev_performance( go_northwest ))
and not observed( prev_performance( go_southwest ))
and not observed( prev_performance( go_northeast ))
and not observed( prev_performance( go_southeast ))
```

11.5.2 Results

Figure 11.5 depicts results averaged over 10 runs as presented in (Cesta *et al.*, 1996a), p. 133. Figure 11.6 depicts the DESIRE results on the basis of 2 runs. Figure 11.7 shows the results for the same experiment in the more flexible environment (thus, with 8 directions of movement), averaged over 5 runs.

11.5.3 Evaluation

The results acquired in the DESIRE simulation are comparable to the results in (Cesta *et al.*, 1996a): social agents survive more often than parasite agents in situations with low food energetic values. The same holds for the experiment in which agents have more degrees of freedom. In our experiments, the chance of survival increases with an increase in the degree of freedom for both types of agents. Additional experimentation with different situations have been reported by Cesta, Micelli and Rizo in (Cesta *et al.*, 1996b).

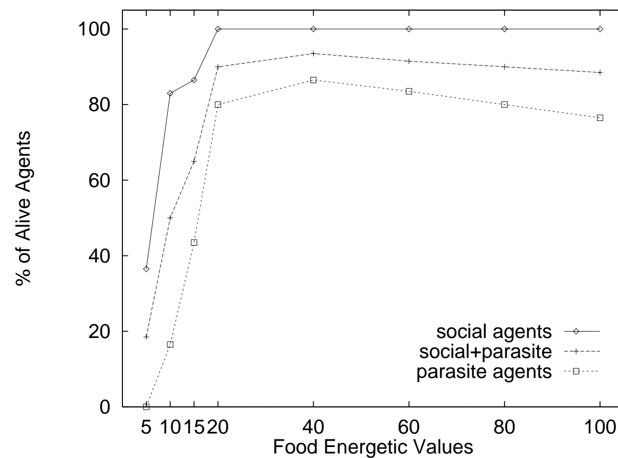

Figure 11.6: Comparison between Parasite and Social. Results from DESIRE prototype.

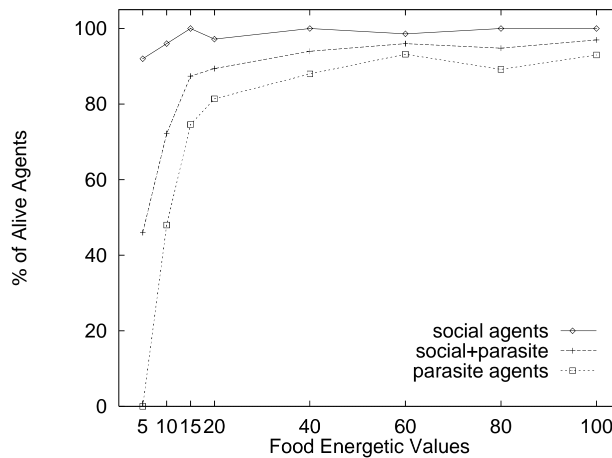

Figure 11.7: Comparison between Parasite and Social. 8 instead of 4 directions for movement.

11.6 Discussion

Much research concerning the design of multi-agent systems (at a conceptual level) addresses complex agents which exhibit complex interaction patterns. Due to this complexity, it is difficult to perform rigorous experimentation. On the other hand, systematic experimental work regarding behaviour of societies of more simple agents, while reporting valuable results, often lacks conceptual specification of the system under consideration.

In this chapter, the DESIRE modelling framework is not only successfully used to develop a conceptual specification of the simple agents discussed in (Cesta *et al.*, 1996a), but also to simulate the behaviour in a dynamical environment. In DESIRE, a conceptual specification, which provides a high-level view of an agent, has enough detail for automatic prototype generation. As stated in Section 9.5, support for knowledge-intensive domains is an advantage of DESIRE over conventional modelling frameworks such as UML. In the application presented in this chapter knowledge-intensity and complexity were not distinguishing factors; in applications with more complexity (for example more complex knowledge within cognitive agents) this advantage is more clear.

A social simulation environment that comes close to DESIRE is SDML (see (Moss, Gaylard, Wallis & Edmonds, 1998)). As in DESIRE, in SDML declarative specification is a central aim. Some of the differences are:

- In DESIRE time can be left implicit, whereas in SDML it is automatically attached to the information structures;
- In DESIRE a more strict form of modularity is used;

- DESIRE provides explicit constructs for control knowledge;
- DESIRE is supported by a distributed execution platform.

The prototype implementation of the conceptual specification of the simple agents has been used to replicate and extend one of the experiments reported in (Cesta *et al.*, 1996a). One of the advantages of conceptual specification has been explored, namely the ease with which existing specifications can be modified. The conceptual specification of a simple agent was modified as follows: the number of directions in which the simple agents can move was increased from 4 to 8, by two minor, local modifications to knowledge included in the specification. A new experiment was performed to compare the behaviour of these agents to the simple agents with only 4 directions of movement.

The ultimate goal of research in the Social Sciences is to develop a theory that explains how the behaviour of the society as a whole (in this chapter, the survival rate of the agents) emerges from the behaviour of the individual agents (which, in this chapter, is described by Table 11.1). Castelfranchi and Conte (1996) call such a theory a middle ground theory. As stated in Chapter 1, such a theory is also helpful in the area of multi-agent systems to predict the behaviour of a multi-agent system under development.

A possible use of the semantic structure developed in this thesis for the development of a middle ground theory can be illustrated using the multi-agent system described in this chapter. As the behaviour of the individual agents in the system is modelled using the DESIRE modelling framework, associated with each agent is a DESIRE structure hierarchy with control that represents the agent in the semantic structure. This DESIRE structure hierarchy with control contains the components presented in this chapter, but also control components associated with the composed components, as described in Chapter 9. For a primitive component or link in this control structure, a set $Beh_{loc}(S)$ is given by the standard dynamics of DESIRE knowledge bases. Together with the standard DESIRE compatibility relations, the behaviour of a single agent can be described by, among others, the white box view on the behaviour of the component that represents the agent. In this way, white box views can be obtained for each of the 30 agents in the society.

An element of the white box view on the behaviour of a simple agent, agent00, is depicted in Figure 11.8. (The name of this agent is abbreviated to a00.) Elements of the white box view are compatible multitraces that consist of local component and link traces for the components a00, $a00_{ctr}$, WIM and OPC, and for the links between these components. (Local link traces for the links are not depicted in Figure 11.8.) In Figure 11.8, boxes denote local component states. Horizontal, solid arrows between boxes denote state transitions. Diagonal, dashed arrows denote information transmission. A number of the input and output atoms for most states are also depicted in Figure 11.8. Atoms at the left side of the small vertical bars are input atoms, atoms at the right side are output atoms. The names of input and output atoms are abbreviated as follows: comp for component, tcf for task_control_focus,

obs_interpr for observation_interpretation, eval for evaluation, WIM for world_interaction_management, OPC for own_process_control, determ_plan for determine_plan, and prep_a_exe for prepare_action_execution.

In the upper left corner of Figure 11.8, agent00 receives new observations. Upon receipt of these new observations, task control at the agent level determines that world_interaction_management is the first subcomponent to activate. Task control focus for world_interaction_management is observation_interpretation, which directs world_interaction_management to activate its subcomponent observation_interpretation. (This is not visible in the white box view of the agent level. Task control rules for world_interaction_management specify that subcomponent observation_interpretation is activated whenever the task control focus of world_interaction_management is observation_interpretation.) In the example trace in Figure 11.8 observation_interpretation concludes that a help seeker is present. The next component activated by task control at the agent level is own_process_control. This component first activates its subcomponent goal_determination, and after that plan_determination. (In the white box view of the agent level, this subcomponent activation is not visible.) In the example trace, plan_determination decides to move to a specific cell. This result is transmitted to world_interaction_management. Task control at the agent level activates world_interaction_management with task control focus prepare_action_execution. As a result, subcomponent action_execution_preparation of observation_interpretation is activated, which determines that the next action to execute is go_south.

To support the development of a middle ground theory, the essential step consists of composing a description of the overall, emerging behaviour of the society from the separate descriptions of the behaviour of the agents. Proposition 5.26, presented in Chapter 5, enables such a composition: roughly speaking, this proposition states that a multitrace for a structure hierarchy that represents the society is an element of the glass box view on the behaviour of the society if, for each agent, the restriction of the multitrace to this agent is an element of the white box view on the behaviour of the agent. The three views on the behaviour of a compositional system presented in Chapter 5, together with the propositions that relate these three views, provide facilities for the development of theories on emergent behaviour. These facilities are flexible with respect to the amount of detail represented in the behaviour of individual agents or the society as a whole. Note that Proposition 5.26 is not itself a middle ground theory. However, this proposition facilitates the development of such a theory. Alternatively, as the complete society of 30 simple agents, together with the external world, is modelled using DESIRE, it is also possible to construct a DESIRE structure hierarchy with control that represents the society directly from the DESIRE model.

Figure 11.9 depicts an example element of the white box view on the behaviour of the entire society. All agents, agent00 to agent29, (abbreviated a00 to a29), together with a component that represents the world, are made subcomponent of a component called toplevel. The white box view on the behaviour of toplevel consists of local component and link traces for toplevel, agent00 to agent29, the world, and all

links between agents and the world. (The traces for links are not depicted in Figure 11.9.) In Figure 11.9, the world transmits observations to each agent, and receives actions to execute from each agent in turn. The element of the white box view depicted in Figure 11.9 is composed of elements of the white box views of the agents as described above. For example, in Figure 11.9, the local component trace for a00 consists of two copies of the trace for component a00 depicted in Figure 11.8.

A possible direction for future research is the design of more complex agents: agents capable of adapting their own characteristics to increase their chances of survival. These agents possess the capability to learn from the observed effects of their own behaviour and that of others. For some first steps in compositional modelling of adaptive animal behaviour, see (Jonker & Treur, 1998b). The explicit conceptual specification makes it possible to make such adaptations at a conceptual level. For example, the explicitly specified agent characteristic (the own_character(selfish) fact in the knowledge base of Section 11.4.2.2) can be replaced by knowledge that can be used to derive the characteristic in a dynamic manner. Another extension of this work is current research on agents that can design new agents on the basis of given requirements. Some results in this direction can be found in (Brazier, Jonker, Treur & Wijngaards, 2000).

11.7 Knowledge Bases

In this section, complete knowledge bases for each primitive component are presented. The structure of this section follows the structure of Section 11.4.

11.7.1 World Interaction Management

The knowledge bases for the (primitive) subcomponents of world_interaction_management are presented in Section 11.7.1.1 and Section 11.7.1.2.

11.7.1.1 Observation Information Interpretation

Knowledge base:

```
if    observed( closest_food_at( C: Cell ))
then  closest( food_at( C: Cell ));

if    observed( closest_help_seeker_at( C: Cell ))
then  closest( help_seeker_at( C: Cell ));

if    observed( help_seeker_at( C: Cell ))
then  help_seeker_present;

if    observed( self_looking_like_a_help_seeker )
then  looking_like_a_help_seeker;
```

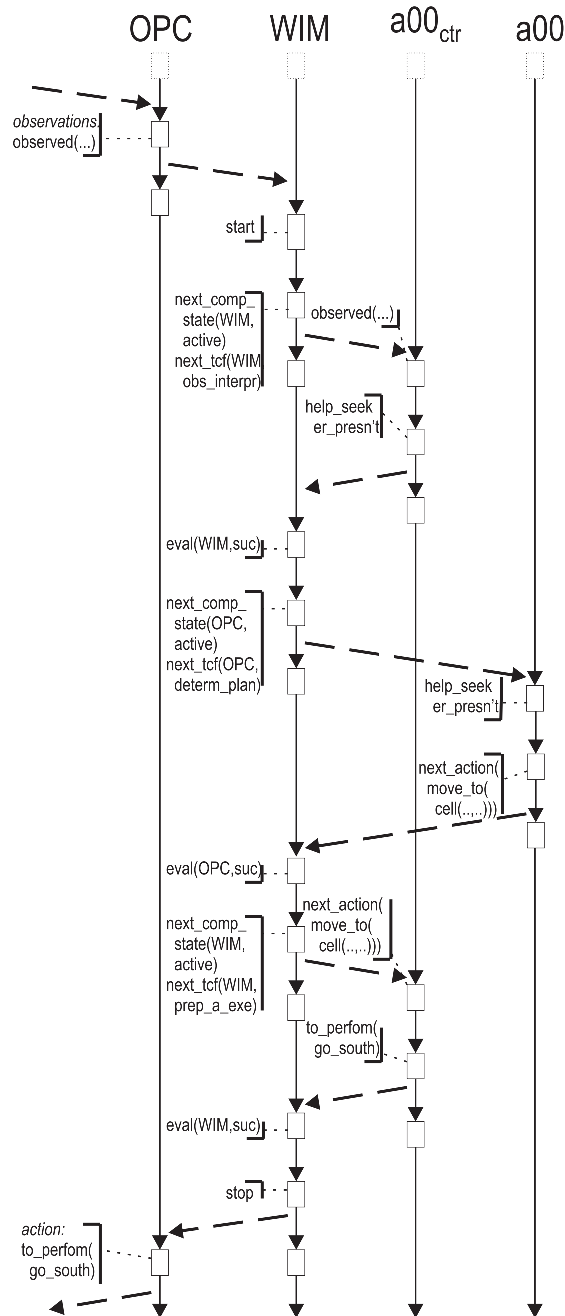

Figure 11.8: Element of the white box view on the behaviour of a simple agent.

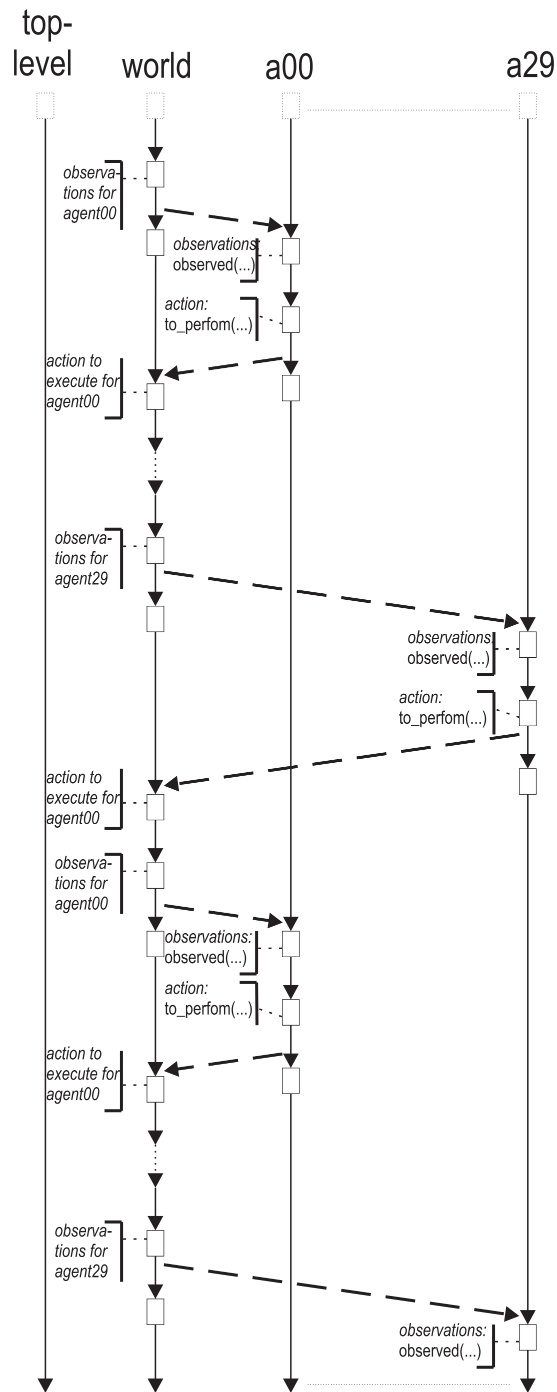

Figure 11.9: Element of the white box view on the behaviour of a society of simple agents.

```
if      not observed( self_looking_like_a_help_seeker )
then    not looking_like_a_help_seeker;

if      observed( prev_performance( pick_up_food ))
then    food_in_possession;

if      observed( prev_performance( feed_help_seeker ))
then    not food_in_possession;

if      observed( prev_performance( feed_self ))
then    not food_in_possession;
```

11.7.1.2 Action Execution Preparation

Knowledge base (some rules are marked with a bar in the margin for reference purposes):

```
if      next_action( change_appearance )
then    to_perform( change_appearance );

if      next_action( feed_self )
then    to_perform( feed_self );

if      next_action( pick_up_food )
        and observed( food_at( cell( 0, 0 )))
then    to_perform( pick_up_food );

if      next_action( feed_help_seeker )
        and observed( help_seeker_at( cell( 0, 0 )))
then    to_perform( feed_help_seeker );

if      next_action( eat_food )
        and observed( food_at( cell( 0, 0 )))
then    to_perform( eat_food );

| if    next_action( move_to( cell( X: ints, Y: ints )))
|       and  Y: ints > 0
| then  to_perform( go_north );

| if    next_action( move_to( cell( X: ints, Y: ints )))
|       and Y: ints < 0
| then  to_perform( go_south );

if      next_action( move_to( cell( X: ints, 0 )))
        and  X: ints > 0
then    to_perform( go_east );

if      next_action( move_to( cell( X: ints, 0 )))
        and  X: ints < 0
then    to_perform( go_west );
```

11.7.2 Own Process Control

The knowledge bases for the (primitive) subcomponents of own_process_control are presented in Section 11.7.2.1 to Section 11.7.2.4.

11.7.2.1 Own Characteristics

Knowledge base (for a selfish agent):

```
agent_character( selfish );

if      energy_level < 0
then    current_state( dead );

if      current_state( dead )
then    agent_died;

if      not energy_level < 0
        and energy_level < 20
then    current_state( in_danger );

if      not energy_level < 20
        and energy_level < 60
then    current_state( hungry );

if      not energy_level < 60
then    current_state( normal );
```

11.7.2.2 Own Resource Management

Knowledge base (for food value 40):

```
if      observed( prev_performance( eat_food ))
then    to_increment_energy_level_with( 40 );

if      observed( prev_performance( feed_self ))
then    to_increment_energy_level_with( 40 );

if      observed( prev_performance( go_north ))
then    to_increment_energy_level_with( -2 );

if      observed( prev_performance( go_south ))
then    to_increment_energy_level_with( -2 );

if      observed( prev_performance( go_west ))
then    to_increment_energy_level_with( -2 );

if      observed( prev_performance( go_east ))
then    to_increment_energy_level_with( -2 );

if      observed( prev_performance(
                      change_appearance ))
then    to_increment_energy_level_with( -1 );
```

```
if      observed( prev_performance( pick_up_food ))
then    to_increment_energy_level_with( -1 );

if      observed( prev_performance( feed_help_seeker ))
then    to_increment_energy_level_with( -1 );

if      not observed( prev_performance( eat_food ))
        and not observed( prev_performance( go_north ))
        and not observed( prev_performance( go_south ))
        and not observed( prev_performance( go_east ))
        and not observed( prev_performance( go_west ))
        and not observed( prev_performance( change_appearance ))
        and not observed( prev_performance( pick_up_food ))
        and not observed( prev_performance( feed_help_seeker ))
        and not observed( prev_performance( feed_self ))
then    to_increment_energy_level_with( -1 );

if      old_energy_level = V1: ints
        and delta_energy_level = V2: ints
        and not V1: ints + V2: ints > 100
then    energy_level = V1: ints + V2: ints;

if      old_energy_level = V1: ints
        and delta_energy_level = V2: ints
        and V1: ints + V2: ints > 100
then    energy_level = 100;
```

11.7.2.3 Goal Determination

Knowledge base:

```
if      agent_character( solitary )
then    selected_goal( find_food );

if      agent_character( parasite )
then    selected_goal( look_for_help );

if      agent_character( selfish )
        and current_state( in_danger )
then    selected_goal( look_for_help );

if      agent_character( selfish )
        and current_state( hungry )
then    selected_goal( find_food );

if      agent_character( selfish )
        and current_state( normal )
then    selected_goal( find_food );

if      agent_character( social )
        and current_state( in_danger )
then    selected_goal( look_for_help );

if      agent_character( social )
        and current_state( hungry )
```

```
then    selected_goal( find_food );

if      agent_character( social )
        and current_state( normal )
        and help_seeker_present
then    selected_goal( give_help );

if      agent_character( social )
        and current_state( normal )
        and not help_seeker_present   /* Obtained by CWA */
then    selected_goal( find_food );
```

11.7.2.4 Plan Determination

Knowledge base:

```
if      selected_goal( look_for_help )
        and not looking_like_a_help_seeker
then    next_action( change_appearance );

if      selected_goal( find_food )
        and looking_like_a_help_seeker
then    next_action( change_appearance );

if      selected_goal( give_help )
        and looking_like_a_help_seeker
then    next_action( change_appearance );

if      selected_goal( find_food )
        and food_in_possession
then    next_action( feed_self )

if      selected_goal( find_food )
        and not food_in_possession
        and closest( food_at( cell( 0, 0 )))
then    next_action( eat_food );

if      selected_goal( find_food )
        and not food_in_possession
        and closest( food_at( cell( X: ints, Y: ints )))
        and not X: ints = 0
then    next_action( move_to( cell( X: ints, Y: ints )));

if      selected_goal( find_food )
        and not food_in_possession
        and closest( food_at( cell( X: ints, Y: ints )))
        and not Y: ints = 0
then    next_action( move_to( cell( X: ints, Y: ints )));

if      selected_goal( give_help )
        and not food_in_possession
        and closest( food_at( cell( X: ints, Y: ints )))
        and not X: ints = 0
then    next_action( move_to( cell( X: ints, Y: ints )));
```

```
if      selected_goal( give_help )
        and not food_in_possession
        and closest( food_at( cell( X: ints, Y: ints )))
        and not Y: ints = 0
then    next_action( move_to( cell( X: ints, Y: ints )));

if      selected_goal( give_help )
        and not food_in_possession
        and closest( food_at( cell( 0, 0 )))
then    next_action( pick_up_food )

if      selected_goal( give_help )
        and food_in_possession
        and closest( help_seeker_at( cell( X: ints, Y: ints )))
        and not X: ints = 0
then    next_action( move_to( cell( X: ints, Y: ints )));

if      selected_goal( give_help )
        and food_in_possession
        and closest( help_seeker_at( cell( X: ints, Y: ints )))
        and not Y: ints = 0
then    next_action( move_to( cell( X: ints, Y: ints )));

if      selected_goal( give_help )
        and food_in_possession
        and closest( help_seeker_at( cell( 0, 0 )))
then    next_action( feed_help_seeker );
```