Delta framework cookbook

Danny Weyns *Elke Steegmans* *Tom Holvoet*
Els Helsen *Koen Deschacht*

Report CW 473, January 2007



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Delta framework cookbook

Danny Weyns *Elke Steegmans* *Tom Holvoet*
Els Helsen *Koen Deschacht*

Report CW 473, January 2007

Department of Computer Science, K.U.Leuven

Abstract

The Delta framework was built to help software developers to develop situated multi-agent systems. This framework is based on the reference-architecture for situated multi-agent systems that was developed by the AgentWise taskforce at DistriNet labs, K.U.Leuven. The Delta framework can be used for very different kinds of multi-agent systems. Multi-agent systems using a software environment as well as multi-agent systems in the physical world are supported.

A framework is organized in two parts: a core (also called frozen-spot) that is common to all applications derived from the framework, and hot-spots that represent the variable parts which allow a framework to be instantiated for a particular application. Because the Delta framework contains a high number of hot spots, its application is not an easy task. This cookbook gives a general description of the Delta framework, together with a set of recipes. Each recipe explains how a particular hot-spot of the framework can be applied. Illustrative examples show how the developer can instantiate the various hot-spots for an application at hand.

AMS(MOS) Classification : Primary : D.2.

Contents

I Overview of the Delta framework **7**

1 Introduction **8**
1.1 Introduction . . . 8
1.2 Overview of this cookbook . . . 8
1.2.1 Part I : Design . . . 8
1.2.1.1 Agents . . . 9
1.2.1.2 Ongoing activities . . . 9
1.2.1.3 Environment . . . 10
1.2.2 Part II : Recipes . . . 11
1.2.2.1 Packetworld . . . 11

2 State **13**
2.1 Overview . . . 13
2.2 Items . . . 13
2.2.1 Agents . . . 14
2.2.2 Ongoing activities . . . 15
2.2.3 Static item . . . 15
2.3 State of the environment . . . 16
2.3.1 Relations . . . 16

3 Synchronization and synchronization schemes **17**
3.1 synchronization scheme's . . . 17
3.1.1 Look, Talk and Do . . . 17
3.1.2 Look, Conversate and Do . . . 17
3.1.3 Choosing a synchronization scheme . . . 18
3.2 Synchronization . . . 18
3.2.1 Different synchronization types . . . 18
3.2.2 Choosing between synchronization . . . 19

4 Active Perception **20**
4.1 Overview . . . 20
4.2 Foci and filters . . . 21
4.3 Sensing . . . 21
4.3.1 Creating a representation in a physical environment . . . 21
4.3.2 Creating a representation in a software environment . . . 21
4.4 Interpreting the representations of the environment . . . 22
4.5 Filtering the percept . . . 22

5 Action **23**
5.1 Some basic concepts . . . 23
5.2 Overview of action . . . 25
5.3 The free-flow architecture . . . 26

6 Communication **28**
6.1 Communication protocols . . . 28
6.1.1 Introduction . . . 28
6.1.2 Building communication protocols . . . 29
6.2 Communication in the Delta framework . . . 29

II Practical implementation guide: the recipes **32**
1 EnvironmentFactory . . . 34
2 OngoingActivityFactory . . . 37
3 PhysicalAgentFactory . . . 38
4 SoftwareAgentFactory . . . 41
5 SystemCreator . . . 44
6 SystemManager . . . 45
7 AgentState . . . 47
8 OngoingActivityState . . . 48
9 StaticItemState . . . 49
10 Region . . . 50
11 Relation . . . 52
12 State . . . 53
13 KnowledgeObject . . . 56
14 Focus . . . 57
15 Filter . . . 57
16 Representation . . . 58
17 VirtualRepresentation . . . 59
18 Sensor . . . 62
19 Percept . . . 63
20 Description . . . 65
21 PerceptualLaw . . . 66
22 ItemRepresentation . . . 68
23 RelationRepresentation . . . 69
24 AgentDecision . . . 71
25 Execution . . . 72
26 SoftwareExecution . . . 74
27 ConsumptionHandler . . . 75
28 OngoingActivityDecision . . . 76
29 Effect . . . 77
30 ActionLaw . . . 78
31 Consumption . . . 80
32 LockType . . . 81
33 Influence . . . 82
34 Operator . . . 83
35 FreeflowDecision . . . 84
36 Activity . . . 86
37 Stimulus . . . 87
38 ActionNode . . . 88
39 CombinationFunction . . . 89
40 IndividualFFCommitment . . . 90
41 MutualFFCommitment . . . 92
42 Protocol . . . 94
43 ExoInitiation . . . 95
44 EndoInitiation . . . 96
45 MessageStep . . . 98
46 Continue . . . 99

47 ExoTerminate . . . 100
48 EndoTerminate . . . 101
49 TimeOutStep . . . 102
50 Conversation . . . 103
51 Externalizable . . . 104
52 Ontology . . . 106
53 CommunicationLaw . . . 107

Bibliography **109**

List of code examples

1 The PWEnvironmentFactory part 1 . . . 35
2 The PWEnvironmentFactory part 2 . . . 36
3 Creating a thrown packet . . . 38
4 The IslandRobotFactory . . . 40
5 Creating and starting an island agent . . . 41
6 The PWAgentFactory . . . 43
7 Creating the packetworld agent . . . 44
8 Use of the class SystemCreator . . . 45
9 Use of the class SystemManager . . . 46
10 The PWAgentState . . . 47
11 The class ThrownPacket . . . 49
12 The class Packet . . . 50
13 The region Position . . . 51
14 The relation HoldRelation . . . 53
15 The state GridState part 1 . . . 54
16 The state GridState part 2 . . . 55
17 The knowledge-object EnergyOfAgent . . . 56
18 The focus VisualFocus . . . 57
19 The filter OnlyPacketsFilter . . . 58
20 The representation ImageRepresentation . . . 59
21 The virtual representation GridStateRepresentation . . . 61
22 The sensor CameraSensor . . . 62
23 The percept GridStatePercept part 1 . . . 64
24 The percept GridStatePercept part 2 . . . 65
25 The description ItemDescription . . . 66
26 The perceptual law PerceptualLawObstacle . . . 67
27 The item representation PWAgentRepresentation . . . 69
28 The relation representation HoldRelationRepresentation . . . 70
29 The decision class RandomWalkDecision . . . 72
30 The execution class ExampleExecution . . . 73
31 The SoftwareExecution class PWExecution . . . 74
32 The PWConsumptionHandler . . . 75
33 The OngoingActivityDecision ThrownPacketDecision . . . 76
34 The effect ChangePositionEffect . . . 77
35 The law StepLaw part 1 . . . 79
36 The law StepLaw part 2 . . . 80
37 The consumption ThrownPacketConsumption . . . 81
38 The type of lock SharedLockType . . . 82
39 The influence StepInfluence . . . 83
40 The operator StepOperator . . . 84
41 The FreeFlowDecision class PWFreeFlow . . . 85
42 The activity VectorActivity . . . 87
43 The stimulus HoldingPacketStimulus . . . 88

44 The action node CatchPacket . . . 89
45 The combination function DoubleMultiply . . . 90
46 The individual commitment ChargeCommitment . . . 91
47 The individual commitment ChargeCommitment . . . 92
48 The protocol ThrowPacketProtocol . . . 94
49 The ExoInitiation protocol step HandleRequestThrow . . . 96
50 The EndoInitiation protocol step InitiateThrowPacketProtocol . . . 97
51 The protocol step ExampleMessageStep . . . 99
52 The continue step ExampleContinue . . . 100
53 The ExoTerminate step HandleAcceptThrowPacket . . . 101
54 The EndoTerminate step ExampleEndoTerminate . . . 102
55 The TimeOut step ExampleTimeOutStep . . . 103
56 The TimeOut step ExampleTimeOutStep . . . 104
57 The Externalizable class Person . . . 106
58 The use of the class Ontology . . . 107
59 The communication law CommunicationRangeLaw . . . 108

Part I

Overview of the Delta framework

Chapter 1

Introduction

1.1 Introduction

The Delta framework was built to help software developers create situated multi-agent systems (SMAS). This framework is based on the reference-architecture for situated multi-agent systems that was developed by the AgentWise research group of KULeuven [2]. The Delta framework can be used for very different kinds of multi-agent systems. Multi-agent systems using a software environment as well as multi-agent systems in the physical world are supported.

The hot spots of the framework define those parts of the framework that can be adapted to a specific application. To create a specific application, a developer has to implement the different hot spots. Because of the amount of hot spots, this implementation is not an easy task. This cookbook aims at helping a developer implementing all the hot spots for a specific application. Dutch readers can find a lot more information about the reference-architecture and the design of this framework in our thesis [5].

1.2 Overview of this cookbook

This cookbook is structured in 2 parts. In this first part we give an overview of the design of this framework. The second part of this cookbook contains all the recipes.

1.2.1 Part I : Design

In the first part of this cookbook we describe the design of the framework and how the different parts of the framework work together. We'll focus our attention here on the hot spots of the framework that are important to the developer. The frozen spots, that remain hidden to the developer when implementing an application, are discussed briefly. You can find a full description of the design of the framework in [5].

In the Delta framework we distinguish three important abstractions: agents, ongoing activities and the environment. They constitute complementary parts of a situated multi-agent system and encapsulate their own responsibilities. In this section we give an overview of the design of the three abstractions in this framework.

1.2.1.1 Agents

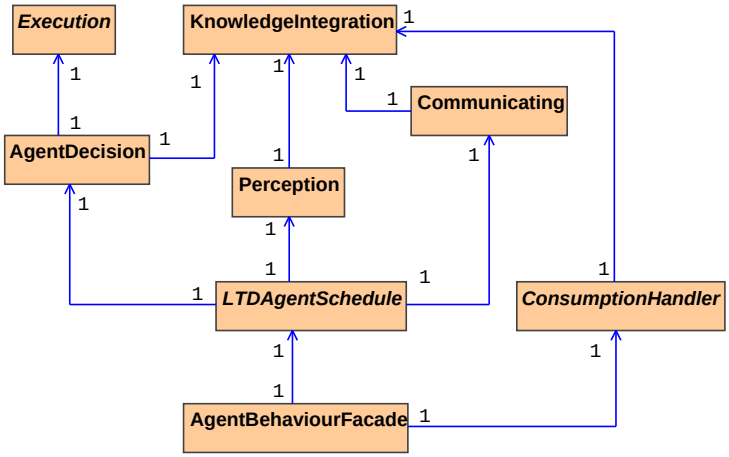


Figure 1.1: Overview of the design of an agent

The AgentSchedule (chapter 3) is the running thread of an agent. It determines when the different activities (perception, action selection or communication) are executed.

An agent is able to perceive the local state of the environment. Perception maps the local state of the environment onto a percept for the agent. In the framework, a model for active perception is used. Active perception enables an agent to direct its perception at the most relevant aspects in the environment according to its current task. We discuss perception in chapter 4.

Decision is responsible for action selection. In every action cycle, the agent selects an operator to execute. This operator represents a particular action to execute by the agent.

Execution is responsible for the execution of the operator. In a physical environment, Execution activates certain effectors to perform the action in the physical world. In a software environment, Execution sends an influence to the environment. An influence is an attempt to modify the state the environment. The environment will determine whether this attempt succeeds. If the attempt succeeds, the environment changes the state and sends a consumption to the agent. A consumption is an effect of the reaction of the environment for a particular agent. The ConsumptionHandler handles this consumption. We'll discuss all classes with regard to action in chapter 5.

Communication takes care of communication between two or more agents. Communication processes incoming messages and produces outgoing messages according to communication protocols. We discuss communication in chapter 6.

KnowledgeIntegration holds the current knowledge of the agent. We explain the KnowledgeIntegration in chapter 2

1.2.1.2 Ongoing activities

Ongoing activities model other processes in the environment. An ongoing activity produces influences according to the state of the environment. Examples of ongoing activities are a moving object or an evaporating pheromone. The OngoingActivitySchedule (chapter 3) determines when a new influence can be send to the environment. OngoingActivityDecision (chapter 5) selects an operator and SoftwareExecution maps this operator on an influence and sends this influence to the environment.

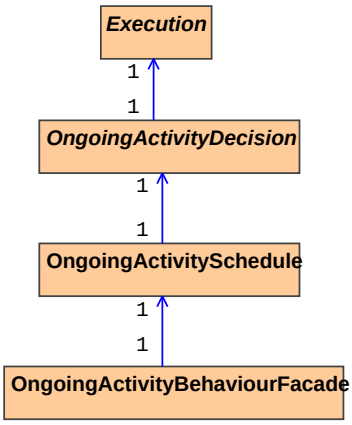


Figure 1.2: Overview of the design of an ongoing activity

1.2.1.3 Environment

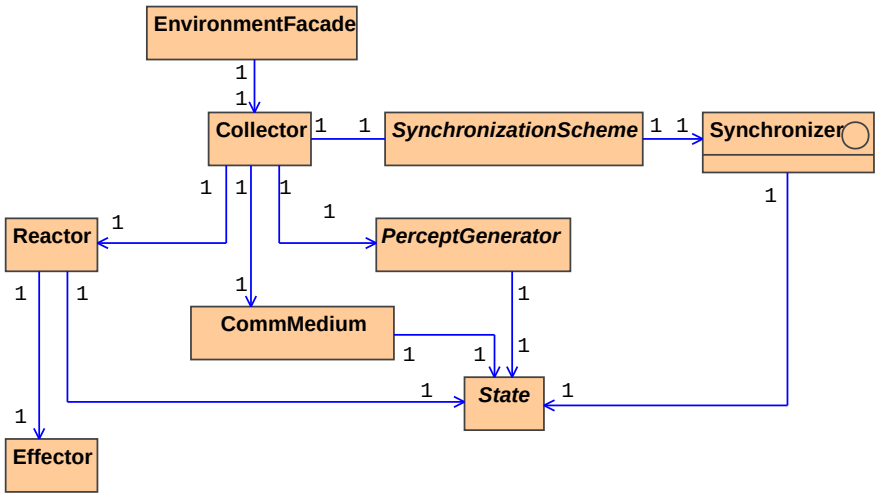


Figure 1.3: Overview of the design of the environment

Different agents and ongoing activities can execute activities simultaneous. The activities send by the agents and ongoing activities are first stored in the COLLECTOR. The SYNCHRONIZATIONSCHEME uses the SYNCHRONIZER (chapter 3) to determine what agents and ongoing activities are synchronized. When all the synchronized items have produced the same activity, the set of activities is passed to the correct handler: the REACTOR, the MESSAGEDELIVERING or the REPRESENTATIONGENERATOR.

MESSAGEDELIVERING handles communication message transport through the environment (chapter 6). The REPRESENTATIONGENERATOR (chapter 4) generates representations of the local state of the environment for the agents. The REACTOR (chapter 5) calculates, according to a set of domain specific laws, the reaction, i.e. state changes in the environment and consumptions for the agents as a consequence of the actions that the agents and ongoing activities execute in the environment. The REACTOR calculates the effects and consumptions and the EFFECTOR (chapter 5) is responsible for the execution of the effects and the delivery of the consumptions at the agents.

1.2.2 Part II : Recipes

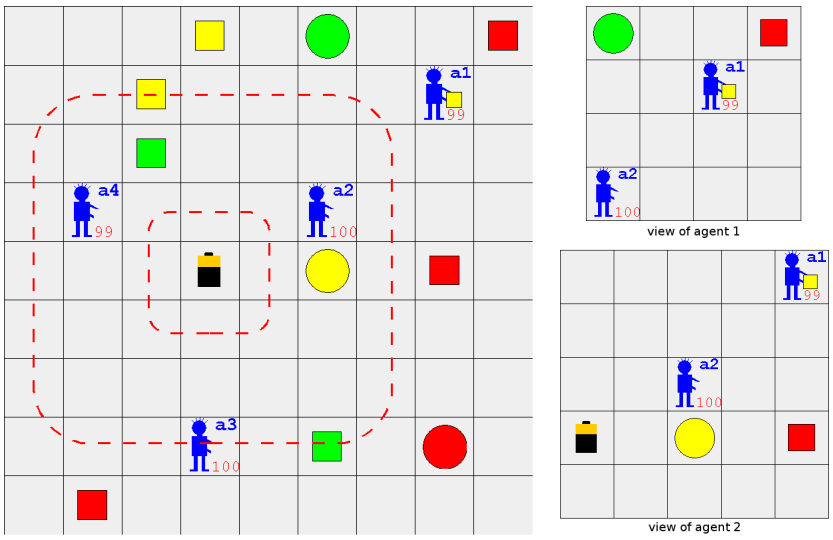
The second part of this cookbook contains all the recipes. Every recipe explains one hot spot of the framework. Every recipe has the following structure:

- **Purpose**: This section gives some more information about the hot spot
- **Used in**: This section tells whether the hot spot has to be implemented in a software MAS, a physical MAS or both
- **Working instructions**: In this section we explain step by step the procedure to implement the hot spot
- **Example**: This last section gives a concrete example of the how hot spot is implemented. The examples are taken from the implementation of the packetworld application, which is developed with this framework. Some examples are entirely fictitious, but if we use such a fictitious example, we describe the fictitious situation where it could be used. We'll explain the packetworld in the next section.

1.2.2.1 Packetworld

The packetworld consists of a number of different colored packets that are scattered over a rectangular grid. Agents that live in this virtual world have to collect those packets and bring them to their corresponding colored destination. Fig. 1.4(a) shows an example of the packetworld with size 9x9 with 4 agents.

Each agent of the packetworld has only a limited view on the world. The view-range of the world expresses how far, i.e. how many squares, an agent can see in its neighborhood. Figure 1.4(b) illustrates the limited view of agent 1 and 2, in this example the view-range is 2.



(a) De pakjeswereld (b) Gezichtsveld

Figuur 1.4: The packetworld

Agents in the packetworld can execute different actions: step to a free neighbouring field, pick up a packet, drop a packet, throw a packet to another agent, catch a flying packet, charge their battery and, if there's nothing useful to do for a while, do nothing.

Performing actions requires energy. Therefore agents are equipped with a battery. The energy level of the battery is of vital importance to the agents. The battery can be charged at one of the available battery chargers. In the example of figure 1.4 there is only one charger, indicated by a battery symbol.

Besides acting in the environment, agents can also send messages to each other. In particular agents can request each other for information about packets or destinations or ask to set up collaborations.

To find the way to the battery chargers and the destinations, they all emit a gradiënts. Agents can follow these gradients.

Chapter 2

State

2.1 Overview

The state of the framework is divided in 2 different kinds of states: the state of the items in the environment and the state of the environment itself. In the following sections, we explain how the state of the items and the environment are modeled in the framework.

2.2 Items

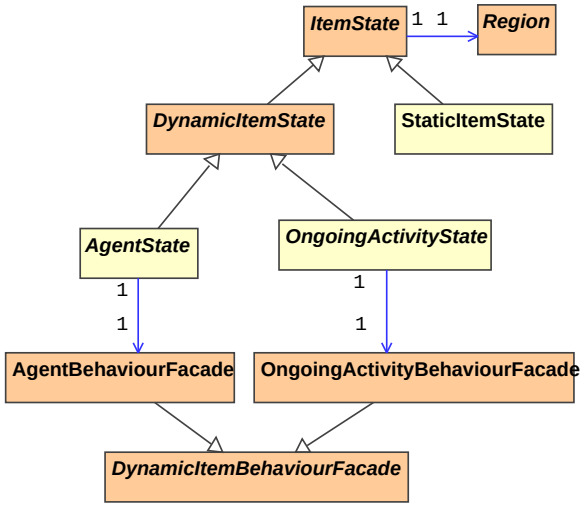


Figure 2.1: Design of the items in a MAS

Fig. 2.1 gives an overview of the design of items in the framework We explain this diagram:

In the framework, we distinguish two types of items: dynamic items and static items. All the items have an external state. The external state of an item contains the external characteristics of the item. By external characteristics we mean (1) the characteristics of the item that are subject to the action laws of the MAS and (2) the observable characteristics of the item. The position and the energy level of an agent are examples of external characteristics.

Static items are items that only have an external state, dynamic items have, next to the external state, a behaviour. We divide the dynamic items into agents and ongoing activities. The external state of a dynamic entity references the behaviour of the entity.

2.2.1 Agents

Agents have an internal and an external state. Recipe 7 on page 47 explains how to create the external state of agent.

The internal state contains the internal characteristics and the knowledge of the agent. Internal characteristics are characteristics that are (1) not subject to the action laws of the MAS and (2) not observable by other agents. Examples of internal characteristic are the situated commitments and roles of an agent. The percept of an agent is an example of the knowledge of an agent. The internal state is part of the behaviour of the agent and is kept in the KnowledgeIntegration class of the behaviour of the agent.

The class KnowledgeObject is used to represent all objects in the internal state (knowledge-objects). Recipe 13 on page 56 explains how to create the different knowledge-objects. KnowledgeIntegration keeps all the knowledge-objects of the agent and offers the following methods to access the knowledge-objects and to store them:

- *public void addKnowledgeObject(KnowledgeObject object):* This method adds a knowledge-object to the KnowledgeIntegration.
- *public <T extends KnowledgeObject> Vector<T> getAllKnowledgeObjectsOfType(Class<T> knowledgeObjectType)*: This method retrieves all objects of a given type
- *public <T extends KnowledgeObject> void addUniqueKnowledgeObjectType(Class<T> knowledgeObjectType):* This method registers a given type as a "unique-object type". Every object that is added to the KnowledgeIntegration, is checked to see whether it belongs to any of the registered unique-object types. If this is the case, all other objects of this class will be removed, and the specified object added. An example of a unique object class is the Percept class. This ensures that only one percept can exist at the time in the KnowledgeIntegration.
- *public <T extends KnowledgeObject> void removeUniqueKnowledgeObjectType(Class<T> knowledgeObjectType)* This method removes a given unique-object type.
- *public <T extends KnowledgeObject> T getUniqueKnowledgeObject(Class<T> knowledgeObjectType)* This method returns the unique knowledge-object of the specified type. For example, you can retrieve the most recent percept with *getUniqueKnowledgeObject(Percept.class)*.
- The java language already offers several classes to implement the Observer pattern. In addition, the KnowledgeIntegration offers a method to register an object as an observer of every object (in the knowledge-integration of that agent) of a specified type.
  public <T extends KnowledgeObject> void addObserverOfKnowledgeObjectType(Class<T> knowledgeObjectType, Observer observer): This method will register a given observer as an observer of the given knowledge-object type. Every time a new object is added, the KnowledgeIntegration checks if the object belongs to the given type. If this is the case, the given observer is registered as an observer of the new object and updated. This can for example be used to register an observer to the class Percept. This observer will be notified every time a new percept is added to the KnowledgeIntegration.

In recipe 13 on page 56 we'll explain how to create a custom knowledge-object. Some knowledge-objects are already defined for you! Before creating your own type of knowledge-objects, check this list to see whether the type you want to create is already implemented in the framework:

- FilterSelection: This class holds the current selection of filters and is registered as a unique-object class. The FilterSelection object is used every time a percept is created through perception. See recipe 20 on page 65 for more information on instantiating this object.
- FocusSelection: This class holds the current selection of foci and is registered as a unique-object class. The FocusSelection object will be used to determine what sensors to activate. See recipe 20 on page 65 for more information on instantiating this object.

- PERCEPT: One of the most important knowledge-objects is the percept. This type of knowledge-object is registered as a unique-object type. See recipe 19 on page 63 for more information on creating a percept.
- ROLE: A role covers a logical functionality of an agent. See recipe 35 on page 84 for more information on creating a role in the free-flow architecture.
- INDIVIDUALSITUATEDCOMMITMENT: A situated commitment defines a relationship between one role (the goal role) and a non-empty set of other roles (the source roles). An INDIVIDUALSITUATEDCOMMITMENT is a commitment of an agent to himself. It will be activated when certain activation conditions hold. See recipe 40 on page 90 for more information on creating an individual commitment in the free-flow architecture.
- MUTUALSITUATEDCOMMITMENT: A situated commitment defines a relationship between one role (the goal role) and a non-empty set of other roles (the source roles). A MUTUALSITUATEDCOMMITMENT is activated in collaboration with one or more other agents. The commitment is set after making an agreement using communication between two or more agents. See recipe 41 on page 92 for more information on creating a mutual commitment in the free-flow architecture.
- AGENTID: This class holds the ID of this agent and will be set upon creation of the agent. This type of knowledge-object is also a unique-object type, because an agent can only have one ID. An object of this class is already added to the KNOWLEDGEINTEGRATION when starting the agent! This object can't be added or changed by the KNOWLEDGEINTEGRATION by the user.
- CONVERSATION: For ever active conversation of an agent, an object of the class CONVERSATION is added to the KNOWLEDGEINTEGRATION. A conversation is removed from the KNOWLEDGEINTEGRATION after it is terminated. See recipe 50 on page 103 for more information on creating conversations.
- PROTOCOL: Every communication protocol of an agent is added to the KNOWLEDGEINTEGRATION. See recipe 42 on page 94 for more information on creating communication protocols.
- STRINGKNOWLEDGEOBJECT: This class holds a single String. Objects of this class are often used in communication to send a message to another agent, but it can also be used to save a String in the KNOWLEDGEINTEGRATION.

2.2.2 Ongoing activities

Ongoing activities are other processes in the environment that produce influences in the environment. They produce influences in the environment depending on the current state of the environment, so they only have an external state. An example of an ongoing activity in the packetworld is a packet that is thrown by an agent (THROWNPACKET). The agent throws the packet to a certain destination. During different action cycles the packet "flies" to its destination. When it reaches the destination, it lands. The decision to fly or to land depends only on the state of the environment and its own external state: if the packet has arived on its destination, it decides to land, otherwise it flies in the direction of the destination.

Recipe 8 on page 48 explains how to create the external state of an ongoing activity.

2.2.3 Static item

Static items do not have a behaviour, they only have an external state and do not produce influences. The only reason of existance of a static object is to model the domain of application. Examples of static items in the packetworld and a destination.

Recipe 9 on page 49 explains how to create the external state of a static item.

2.3 State of the environment

Not only the items have an external state, also the environment has one. The state of the environment (STATE) is an important hot spot, the entire domain of application is modeled in here. The state of the environment contains the external state of all items, as wel as the relations between the items. The state can have certain characteristics like temperature or current time. Furthermore the state of the environment defines a topology. The topology of the environment of the packetworld for example is a grid of positions. In a MAS where the agents are spread over a network it's a graph of nodes in the network. Recipe 12 on page 53 explains how to create the state of the environment for your application.

2.3.1 Relations

In a multi-agent system the items are not isolated but have certain relations (RELATION) between them. An example of a relation is the HOLDRELATION, that expresses an agent is holding a packet. In general two or more items can be involved in a relation. For example, an item in the environment (for example a bag) can contain different other items. The CONTAINSRELATION can be a relation between the bag and the contained items. A relation can have certain characteristics. The HOLDRELATION for example could have as a characteristic the force by which the agent holds the packet. Just like the external state of the items, relations are kept in the state of the environment.

Relations model the domain of the application, so they are a hot spot. Recipe 11 on page 52 explains how to create a relation.

Chapter 3

Synchronization and synchronization schemes

Agents and ongoing activities execute actions in the environment. This framework offers support for simultaneous actions. Simultaneous actions are actions that are executed at the same time and can possibly interfere with each other. For example when two agents want to pick up the same packet at the same time, the actions interfere, because only one agent can pick up the packet.

Only synchronized agents can execute simultaneous actions. Synchronization determines which agents are synchronized.

Another issue with respect to synchronization is the synchronization scheme. This scheme determines when the agents in the framework can execute an activity and when these activities are handled by the environment.

3.1 synchronization scheme's

In the framework two synchronization scheme's are supported: Look, Talk and Do and Look, Conversate and Do. In this section we discuss them both and explain how a developer can choose a synchronization scheme for his application.

3.1.1 Look, Talk and Do

The LT/D-synchronization scheme (Look, Talk and Do) [4] divides the activity in the multi-agent system in action cycli. Every cycle consists of two phases. The first phase is the perception phase, during which the agents perceive their local environment. The second phase is the action phase. During this phase every agent is allowed to perform one action. We distinguish between two kinds of actions:

First an agent may choose to communicate one message with other agents. Communication exists of two sub-phases. During the first sub-phase an agent can send one message. In the second sub-phase agents that have not send a message in the first sub-phase may respond to a message. Communication closes when all responses are delivered.

The second option for acting in the environment is reserved for agents that have not sent a message. Each such agent has to send an influence to the environment. The reaction of the environment to all performed influences concludes the action cycle. Ongoing activities send an influence to the environment every action cycle.

3.1.2 Look, Conversate and Do

The LCD-synchronization scheme (Look, Conversate and Do) also divides the activity in the multi-agent system in action cycli. Every cycle consists of three phases. The first phase is the perception phase, during which the agents perceive their local environment. The second phase is the communication phase. During this phase every agent is allowed to communicate. Communication exists of

two sub-phases. During the first sub-phase every agent can start one ore more conversations. In the second sub-phase the agents can send an arbitrary number of messages, until all their conversations are terminated. The third phase starts when all conversations are terminated. In this phase all the agents and ongoing activities send an influence to the environment. The reaction of the environment to all performed influences concludes the action cycle.

3.1.3 Choosing a synchronization scheme

The developer of an application has to choose between the LT/D- or LCD-synchronization scheme. Both schemes are implemented and as explained in recipes 1 on page 34, 3 on page 38 and 4 on page 41, a developer only has to choose the correct scheme when implementing the environment and the agents.

3.2 Synchronization

Agents and ongoing activities execute actions in the environment. This framework offers support for simultaneous actions. Simultaneous actions are actions that are executed at the same time and can possibly interfere with each other [7]. For example when two agents want to pick up the same packet at the same time, the actions interfere, because only one agent can pick up the packet. Only synchronized agents can execute simultaneous actions. Synchronization determines which agents are synchronized.

In the Delta framework we offer support for three kinds of synchronization: no synchronization, global synchronization and regional synchronization. In this section we explain each kind of synchronization and how a developer has to choose the kind of synchronization for his application.

3.2.1 Different synchronization types

No synchronization

Without synchronization the agents and ongoing activities act asynchronous, each on their own rhythm. The agents can not act simultaneous.

Global synchronization

With global synchronization all the agents and ongoing activities act at one global pace. They are all synchronized with each other and perform actions simultaneously. In each cycle, every agent and ongoing activity performs one action, after which the actions of all agents are processed and the cycle restarts.

Regional synchronization

Regional synchronization [8] divides the agents of the MAS into regions. A region exists of the agent himself, the agents within his perceptual range, the agents in the perceptual range of those agents and so on.

Applied to an example with a view size of 2 in Fig. 7, we have three regions of synchronized agents, indicated by different colors: region 1 consisting of agent 1, agent 4 and agent 5, region 2 consisting of agent 2, agent 3 and agent 6, and finally agent 7 forms region 3 on its own, since it is currently out of range of all other agents. All agents within the same region are synchronized with each other and act simultaneously, performing actions at the same pace. Different regions on the other hand act asynchronous: the agents within region 1 act at their own pace, independent of the pace of the agents in region 2 or region 3. Note that the regions are dynamic and have to be kept up-to-date as the agents move. For instance, if agent 7 performs a step in the northern direction, it enters the perceptual range of agent 3, and hence

Compared to global synchronization, regional synchronization allows synchronization to be more selective, increasing the efficiency of acting for the agents. Regional synchronization is based on the characteristic that agents within a situated MAS typically perceive and act locally.

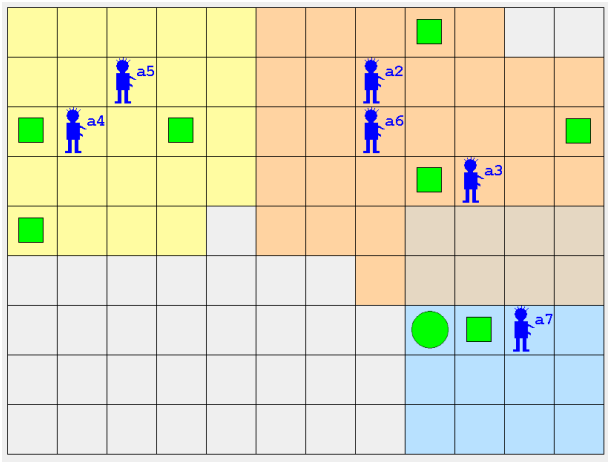


Figure 3.1: Regional synchronization

3.2.2 Choosing between synchronization

The developer of an application has to choose a certain type of synchronization The three synchronization types are implemented and as explained in recipe 1 on page 34 the developer has to choose between the different types when implementing the environment.

Chapter 4

Active Perception

An agent is able to perceive the local state of the environment. Perception maps the local state of the environment onto a percept for the agent. In this chapter we'll describe how perception is implemented in this framework.

4.1 Overview

In this framework, agents use active perception to perceive the environment [9]. Active perception enables an agent to focus its attention to the most relevant aspect of the environment according to its current task. To sense the environment for only specific types of information, an agent selects certain foci. An agent can also filter the percept for certain types of information. To this end the agent selects filters that remove elements of the percept that do not fit certain conditions. We discuss foci and filters and the selection of them in section 4.2.

Active perception divides perception into three functionalities: sensing, interpreting and filtering. Figure 4.1 illustrates the perception of an agent. Perception starts when the class AgentSchedule calls the method *doPerception()* of the class Perception (1). Perception retrieves the selection of foci from the KnowledgeIntegration (1.1), takes the foci from the selection (1.2) and passes them to the class Sensing (1.3) which is responsable for creating the representations of the local environment of the agent (see section 4.3). These representations are passed to the class Interpreting (1.4) who interpretes these representations to create the percept (see section 4.4). Finally this percept is filtered. Perception retrieves the selection of filters from the KnowledgeIntegration (1.5), takes the filters from this selection (1.6) and passes them to the class Filtering (1.7). This class filters the percept (see section 4.5), which is finally stored in the KnowledgeIntegration of this agent (1.8).

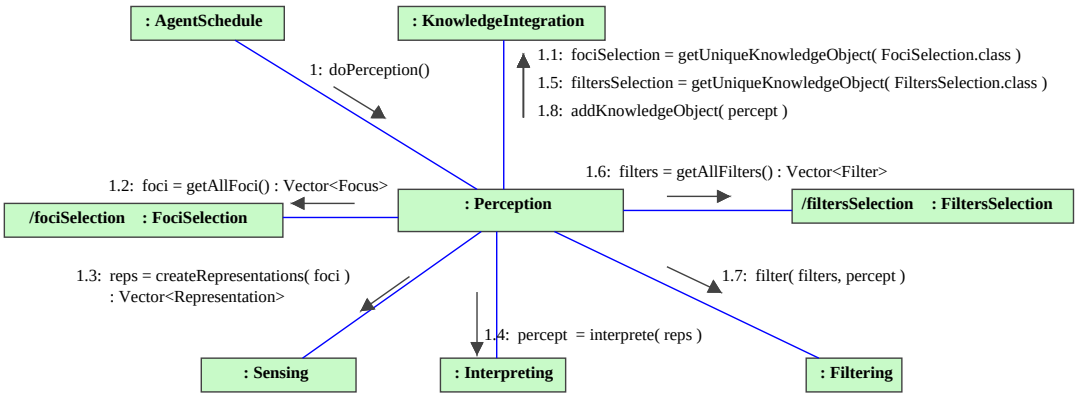


Figure 4.1: Overview of the creation of a percept

4.2 Foci and filters

An agent can select foci to direct its perception, it allows him to sense the environment for only specific types of information. For example, in an ant-like MAS, one agent may be interested in a 'visible' perception of this environment, while another agent may be interested in 'smelling' pheromones. To sense the desired type of information, both agents have to select the appropriate foci. Each focus can have certain properties such as an operating range, a resolution, etc. In recipe 14 on page 57 we explain how to create foci.

When the percept is created the agent can filter the new percept for information he is interested in at that moment. By selecting a set of filters an agent is able to select only those elements of the percept that match specific selection criteria. Each filter imposes conditions on the elements of the percept. These conditions determine whether the element of the percept can pass the filter or not. For example, an agent that has selected a focus to visually perceive its environment and who is only interested in viewing packets within his perceptual range, can select an appropriate filter that matches only packets for his percept. Elements that do not pass all the filters, are removed from the percept. In recipe 15 on page 57 we explain how to create filters.

The agent can select new foci and filters every action cycle to create a percept that's suited for its current task. The class DECISION is responsible for selecting new foci and filters. It does this in the methods *getFociSelection()* and *getFiltersSelection()*. These methodes are called every time the agent has performed an action. See recipe 24 on page 71 for more information on creating a DECISION class for the agent.

4.3 Sensing

SENSING maps the state of the environment to representations. This is done in a different way for a physical or a software agent. In section 4.3.1 we explain how sensing works for a physical agent, then we explain it for a software agent in section 4.3.2.

4.3.1 Creating a representation in a physical environment

A physical agent lives in a physical world, and perception is probably far from trivial. A physical agent has several sensors that are responsable for the creation of the representations of the environment. These sensors can be camera's, touch sensors, light sensors, ... Every sensor generates a raw representation (this can be an image, a sound recording or any other sensor reading). Every sensor can create a representation for one or multiple foci. The sensor will only be activated if the agent selects the corresponding kinds of foci.

We explain how to how to create a sensor in recipe 18 on page 62 and how to create a representation in recipe 16 on page 58.

4.3.2 Creating a representation in a software environment

Fortunately, implementing perception for agents in the software world isn't hard at all. In a software MAS the environment is responsable for the creation of the representations of the state of the environment for the agents. The representations of the state of the environment (VIRTUALREPRESENTATION) are composed by the REPRESENTATIONGENERATOR according to a set of perceptual laws. Perceptual laws are domain specific constraints on perception. They determine what the agents can observe. Whereas physical sensing naturally incorporates such constraints, in software multi-agent systems, the constraints have to be modeled explicitly. The REPRESENTATIONGENERATOR first creates the representation of the state of the environment and then enforces all the perceptual laws of the environment on it. The created representations are automatically fetched from the environment and passed to the SENSING class of the agents.

WWe explain how to create perceptual laws in recipe 21 on page 66 and in recipe 17 on page 59 we'll explain how to create a virtual representation of the environment.

4.4 Interpreting the representations of the environment

The second functionality of active perception is interpreting. INTERPRETING maps the representations to a percept of the environment. A percept is a description of the local environment of an agent and contains representations of the various items in the environment. These representations can be easily understood and used by the other modules of the agent.

To map the representations to a percept, INTERPRETING uses a number of descriptions. Each description extracts some kind of information of the raw representations (E.g, a description to recognize agents in an image and a description to recognize these walls in this image). Each description can update the percept. Finally, the fully build percept is passed to passed to Filtering.

We explain how to create a percept in recipe 19 on page 63 and how to create a new description in section 20 on page 65.

4.5 Filtering the percept

The final functionality of active perception is filtering. By selecting a set of filters an agent is able to select only those elements of the percept that match specific selection criteria. Each filter imposes conditions on the elements of the percept. These conditions determine whether the element of the percept can pass the filter or not (see section 4.2). FILTERING passes the percept through all the filters the agent has selected. Elements that do not pass all the filters, are removed from the percept. After the filtering, the modified percept is stored in the KNOWLEDGEINTEGRATION of the agent. In recipe 15 on page 57 we explain how to create a filter.

Chapter 5

Action

Agent execute actions in the environment to achieve their goals. The framework offers support for taking a decision on what action to execute, for executing that action and for handling actions in the environment. We'll start this chapter with some basic concepts, then we'll explain how the concern action is implemented in this framework.

5.1 Some basic concepts

In this section we explain what roles and situated commitments are. Roles and situated commitments are both used in action selection and in communication.

A role is an agent's functionality in the context of an organization. A role is an abstract representation of a functionality of the agent.

Fig. 5.1 shows the roles of a simple agent. One of the roles is *search packet.* In this role, an agent searches for a packet, and if he finds one, he picks it up.

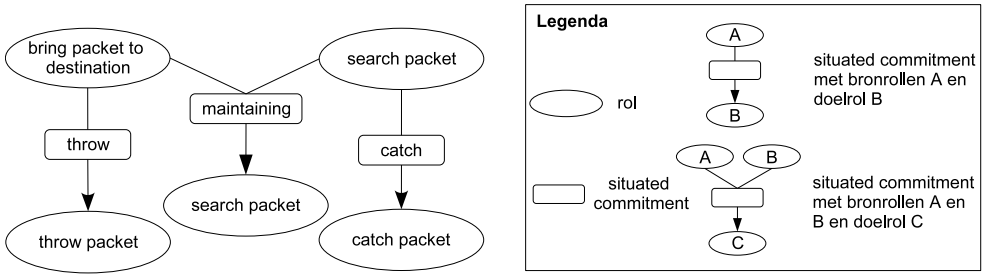
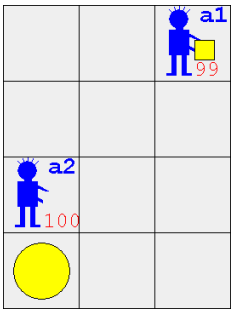


Figure 5.1: Roles and commitments of a simple agent in the packet world

A situated commitment defines a relationship between one role (the goal role) and a non-empty set of other roles (the source roles). The behaviour of an agent tends to prefer the goal role of the commitment over the source roles when the situated commitment is activated. Next to the goal role and source roles, a situated commitment defines a relationship between one role, called the goal role, and a non-empty set of other roles, i.e. the source roles, of an agent. Situated commitments are characterized by a well-known name, a relations set, a context, an activation condition and deactivation condition and a status (activated or deactivated). The name of a situated commitment is used during communication to refer explicitly to the commitment. The relations set contains the identity of the related agent(s) in the situated commitment. The context describes contextual properties of the situated commitment such as descriptions of objects in the local environment. When the activation condition is satisfied, the situated commitment is gets activated. It is deactivated when the deactivation condition is satisfied.



Figuur 5.2: A part of the packetworld

We given an example of the use of situated commitments on the basis of fig. 5.2 (also take a look at fig 5.1 for the roles and situated commitments of these agents). Agent 2 notices he is situated next to a yellow destination. He also notices that agent 1 holds a yellow packet and the yellow destination is beyond agent 1's range. Agent 2 tries to collaborate with agent 1 to throw the packet. He uses direct communication to do so. When the two agents reach an agreement, agent 1 activates the commitment *throw* en agent 2 the commitment *catch*. We discuss the commitment *throw* in more detail.

The situated commitment *throw* has as source role the role *bring packet to destination* en as goal role the role *throw packet*. If the commitment *throw* is activated, agent 1 prefers to throw the packet (*throw packet*) to agent 2, in stead of bringing it himself to the destination (*bring packet to destination*). The relations set of the *throw* commitment contains the names of the all the agents in the collaboration, which are in this case a1 en a2. The context could for example be the color of the packet. The activation condition is satisfied when agent 1 receives a request for collaboration of agent 2, and replies positive on this request. The deactivation condition of the commitment *throw* is satisfied at the moment agent 1 throws the packet to agent 2.

The Delta framework makes a distinction between a commitment of an agent to himself (IndividualSituatedCommitment) and a commitment between multiple agents (MutualSituatedCommitment). An IndividualSituatedCommitment has an activation and deactivation condition. The commitment is activated if the activation condition evaluates true and the deactivation condition evaluates false. The commitment is deactivated if the deactivation condition evaluates true. The *maintaining* commitment (see fig 5.1) is an individual commitment that is activated if the battery level of an agent drops below a critical value. The commitment is deactivated if the battery is fully charged.

Mutual situated commitments are commitments that express a social attitude of a situated agent. Each mutual situated commitment has a relations set and a deactivation condition. A mutual situated commitment will be activated by communication. When two agents reach an agreement about cooperation, each agent will set the correct commitment, and will add the other agent to the relations set of that commitment. The commitment *throw* as explained above is an example of a mutual situated commitment The commitment remains active until the deactivation condition evaluates true.

5.2 Overview of action

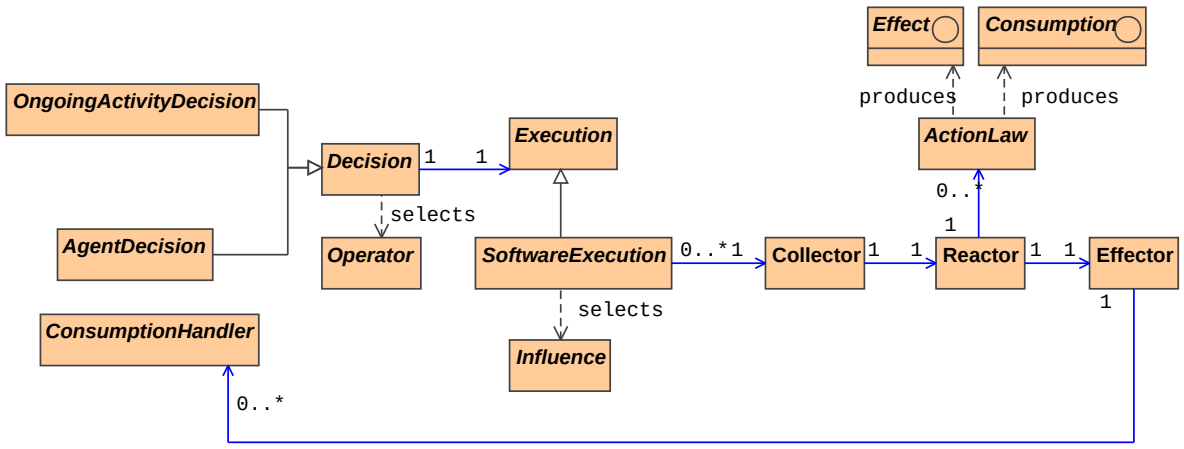


Figure 5.3: Overview action

Figure 5.3 gives an overview of the modules that concern with action in the framework.

Agents and ongoing activities execute actions in the environment, they select the operator that represents the best action at a certain moment, to execute in the environment. DECISION is responsible for selecting this operator. AGENTDECISION (see recipe 24 on page 71) selects an operator for an agent and ONGOINGACTIVITYDECISION (see recipe 28 on page 76) for an ongoing activity. The selected operator is passed to the EXECUTION class. A lot of mechanisms exist for action selection. Examples are the subsumption architecture [3] or the free-flow architecture [6]. The free-flow architecture is supported in the framework. We explain this architecture in the next section.

EXECUTION in a physical environment is different from execution in a software environment. EXECUTION (recipe 25 on page 72) in a physical environment is responsible for starting certain effectors like motors from a physical agent. Execution in a software environment is handled by the class SOFTWAREEXECUTION (recipe 26 on page 74). This class maps the selected operator on a corresponding influence (recipe 33 on page 82) and sends it to the environment. An influence is a representation of an action that an agent or an ongoing activity wants to execute.

The influences of the agents and ongoing activities in the MAS are collected in the collector. When all the agents and ongoing activities in a synchronization-set have produced an influence, the set of influences from these agents and ongoing activities is passed to the reactor. The reactor calculates the effects of the influences. To calculate these changes, it uses the action laws of the environment (ACTIONLAW, see recipe 30 on page 78).

Every action law needs a number of influences of a specific type. If the influence-set contains all the needed influences from the right types, the conditions of the law are checked. If the conditions hold for the given influences, the law produces effects (EFFECT, recipe 29 on page 77) and consumptions (CONSUMPTION, recipe 31 on page 80).

We give an example of a law here: THROWLAW. This law needs one influence of type THROWINFLUENCE. The conditions for this law hold if the agent is holding a packet. If this is the case, new consumptions and effects are created.

When all the effects and consumptions are calculated they are passed to the EFFECTOR. This class executes the effects and passes the consumptions to the agent. The agent handles consumptions in his CONSUMPTIONHANDLER (recipe 27 on page 75).

5.3 The free-flow architecture

The Freeflow architecture was first proposed by Rosenblatt and Payton in [6]. T. Tyrell demonstrated that hierarchical free-flow architectures are superior to flat decision structures, especially in complex and dynamic environments. An example of the free-flow architecture is depicted in fig. 5.4.

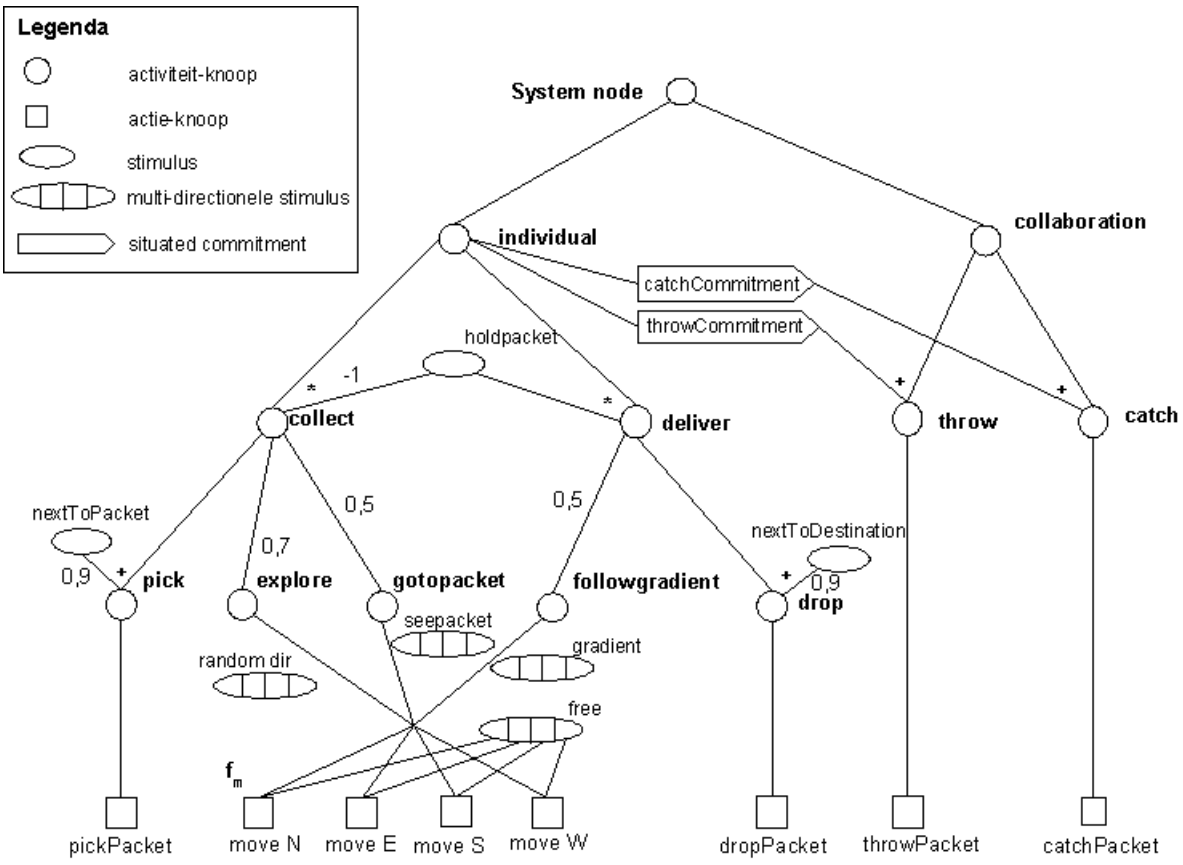


Figure 5.4: An example of a free-flow tree

The hierarchy is composed of activity nodes (in short: nodes) wich receive information from stimuli (recipe 37 on page 87) in the form of activity (ACTIVITY). In a free-flow tree, nodes that represent a functionality that is more favorable at a certain moment have a higher activity then nodes wich are less favorable at that moment. You can use different types of activity in a free-flow tree, but the use of a double (DOUBLEACTIVITY) to represent the activity is the most natural choice. In this framework however, other sorts of activities are also supported. You can find more information on creating new types of activity in recipe 36 on page 86.

The activity nodes have to combine the activity received from their stimuli and parent node. This is done by the COMBINATIONFUNCTION. See recipe 39 on page 89 for more information on creating a combination function. The nodes feed their activity down through the hierarchy until the activity arrives at the leaf nodes of the three (ACTIONNODE, recipe 38 on page 88) where a winner-takes-it-all process decides which action is selected. The activity nodes are connected to each other and to the action nodes by links. Every link has a weight. This weight determines how much activity is passed from the parent node to the child node.

A situated commitment in the free-flow architecture connects a non-empty set of source roles with a goal role. When a situated commitment is activated, it passes an extra amount of activity to the goal role. In the free-flow architecture we also make a distinction between a commitment of an

agent to himself (IndividualFFCommitment, recipe 40 on page 90) and between multiple agents (MutualFFCommitment, recipe 41 on page 92).

Figure 5.4 gives an example of a free-flow tree. This tree is a simplified version of the tree that is used in the packetworld to select an operator. The *System node* feeds its activity to the *individual* and the *collaboration* node. The *individual* node feeds its activity to the *collect* and *deliver. Collect* stands for the functionality of retrieving a packet, *deliver* for delivering that packet in the destination. *Collect* is divided in *explore* (search for a packet), *gotopacket* (go to a perceived packet) and *pick* (pick up a packet next to the agent). *Deliver* is divided in *followgradient* (follow a gradient towards the destination) and *drop* (drop packet in destination next to packet). *Collaboration* stands for the actions agents take in cooperation: *throw* (throw the packet to another agent) and *catch* (catch a thrown packet).

The free-flow tree in figure 5.4 also consists of two commitments: throwCommitment and catchCommitment. These commitments are activated when two agents agree on a collaboration by direct communication.

In recipe 35 on page 84 we'll see how to use the created free-flow tree to select a concrete operator for the agent.

Chapter 6

Communication

Agents in a multi-agent system are social entities. They cooperate with each other to achieve their goals. This cooperation is achieved by communication. We make a distinction between indirect and direct communication. Indirect communication takes place when an agent interacts with another agent via the environment, for example by means of virtual pheromones. Agents need direct communication to make an agreement about cooperation. Direct communication is typically achieved by sending messages between the agents.

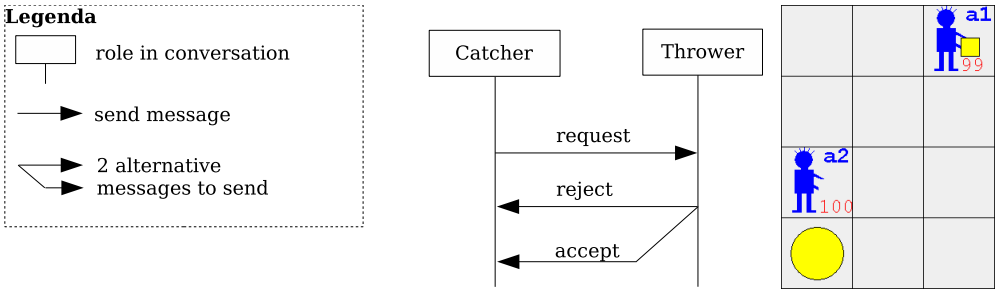
In this chapter we'll explain communication specified in terms of protocols (section 6.1) and we'll see how communication is implemented in the Delta framework (section 6.2).

6.1 Communication protocols

6.1.1 Introduction

In the Delta framework, agents interact by sending and receiving messages. A communication protocol specifies the order of the messages. Protocol based communication is the interaction between agents based on the exchange of messages according to a specific protocol. We use the notion of a conversation to refer to such an ongoing interaction.

An example of a protocol is the ThrowPacketProtocol (fig. 6.1(a)).



(a) interaction diagram (b) example situation

Figure 6.1: The ThrowPacketProtocol

We give an example of the use of this protocol in figure 6.1(b). Agent 2 notices he's standing next to a yellow destination and notices agent 1, who carries a yellow packet and doesn't see the yellow destination (we assume the agents have a perception range of 2 fields). Agent 2 takes the role of *Catcher* in this conversation and sends a *request* to agent 1, who takes the role of *Thrower*. Agent 1 checks whether his ThrowCommitment is not already activated (meaning he's got an agreement to throw a packet with another agent). If this is not the case, he sends an *accept*-message to agent

2. Otherwise he sends a *reject*-message. When agent 2 receives the accept message, he activates his CatchCommitment. After this the conversation is finished.

6.1.2 Building communication protocols

In general, an agent can have one or more communication protocols (PROTOCOL, see recipe 42 on page 94). Every protocol consists of different protocol steps. We use the term protocol step to refer to an action in a conversation, such as starting, continuing or terminating a conversation or reacting to an incoming message.

	Exogenous	**Endogenous**
start of conversation	ExogenousInitiation	EndogenousInitiation
during conversation	MessageStep	Continue
end of conversation	ExoTerminate	EndoTerminate

Table 6.1: The different protocol steps

We make a distinction between protocol steps that are activated when conditions internal to the agent are satisfied (Endogenous steps) and protocol steps that react to an incoming message (Exogenous steps). Every endogenous step has a method that checks whether the conditions for that protocol step are satisfied. Every exogenous step will hold a performative and the step is executed when a message with that performative from a conversation using the protocol of this protocol step is received.

We further divide the endogenous steps in EndogenousInitation (recipe 44 on page 96) that starts a new conversation, Continue (recipe 46 on page 99) that continues a conversation after a break and EndoTerminate (recipe 47 on page 100) that terminates a conversation.

The exogenous steps are divided in ExoInitiation (recipe 43 on page 95) that starts a new conversation after receiving the first message of that conversation, MessageStep (recipe 45 on page 98) reacts to an incoming message and ExoTerminate (recipe 47 on page 100) terminates a conversation after receiving a message.

There's still one protocol step we haven't discussed: the TimeOutStep (recipe 49 on page 102). When you create a conversation, you can supply a time out. When a conversation hasn't send or received a message for a time greater than the time out, the TimeOutStep of the protocol of that conversation is executed and the conversation is removed. If you don't supply a time out when creating a conversation the conversation will not be checked for timing out.

To implement a communication protocol, you should implement the correct protocol steps and add them to a communication protocol. This is explained in recipe 42 on page 94.

6.2 Communication in the Delta framework

In this section we explain how protocol based communication is implemented in the Delta framework. Figure 6.2 gives an overview of the design of the communication module of an agent.

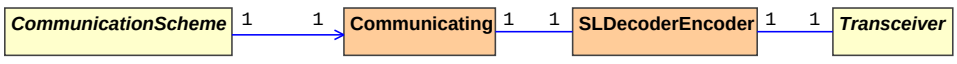


Figure 6.2: Overview design communication

We first give a brief overview of the different classes in the design of communication (see fig. 6.2), then we'll explain every class in more detail.

The communiction scheme (CommunicationScheme) determines when the agent starts the communication phase. Communicating handles communication for an agent and is responsible for starting new conversations, continuing conversations after a break, terminating conversations and reacting to incoming messages. When Communicating decides to send a new message to another agent, it creates a new AgentMessage and passes it to the SLDecoderEncoder. This class converts this AgentMessage into a StringMessage and passes this StringMessage to the Transceiver. The Transceiver sends this messsage to the receiver of this message through the environment.

CommunicationScheme

Every environment has a synchronization scheme that determines when an agent can execute the different activities (perception, communication or action). The AgentSchedule implements this synchronization scheme and selects the correct activity for an agent to execute. If the AgentSchedule determines the agent can execute communication, it passes control to the communication scheme (CommunicationScheme). The communication scheme handles the communication phase of a synchronization scheme. This communication phase is divided in different sub-phases. In every subphase, CommunicationScheme executes one or more methods of Communicating to execute the different parts of the communication phase.

For example a subphase in communication is checking whether new conversations can be started. CommunicationScheme instructs Communicating to see whether new conversations can be started after which Communicating checks all the protocols of the agent to see whether a new conversation can be started. If this is the case, a new conversation is started.

Communicating

Communicating executes the different parts of communication. It holds an inbox and an outbox. If the agent receives a message, the message is stored in the inbox. The message is only passed to the corresponding protocol steps when the CommunicationScheme instructs Communication to do so. Also, when a certain protocol step sends a new AgentMessage it is not send immediately, but stored in the outbox. The CommunicationScheme determines when the messages in the outbox of an agent can effectively be send to the receiver of that message.

Communicating does not only offer different methods to the CommunicationScheme, but also to the protocol steps. The methods that can be used in a protocol step are collected in CommunicatingInterface. This interface consists of all the methods of the Communicating class that can be used by a concrete protocol step:

- *public Vector<Conversation> getActiveConversations()* : Every time the agent starts a new conversation, this conversation is added to the KnowledgeIntegration of that agent. A conversation is removed after it is terminated. This method retrieves all conversations from the KnowledgeIntegration.
- *public int getNumberOfMessagesInInbox()* : this method returns the number of messages in the inbox of this agent.
- *public int getNumberOfMessagesInOutbox()* : this method returns the number of messages in the outbox of this agent.
- *public void sendMessage(AgentMessage message)* : this method adds a message to the outbox of the agent.

When a protocol step wants to send a message, it creates a new AgentMessage. An AgentMessage is the internal representation of a message. This class is easy to handle for the protocol steps, but isn't the best representation to send a message through the environment. To send a message through the environment, we use the class StringMessage.

SLDecoderEncoder

The SLDecoderEncoder is responsible for transforming an AgentMessage to a StringMessage and vice versa. The content of an AgentMessage is a java object and the content of a StringMessage is a string. We'll first give an overview of this conversion, and then explain the different steps in detail

Overview When converting an object to a string, the values of the attributes of that object are extracted from that object and converted to strings. These strings have a syntax similar to the SL-language defined by the fipa [1]. The strings representing the different attributes of that object are concatenated and set as the content of a StringMessage. This message is send to another agent (using the Transceiver, see below). This receiver must reconstruct the object. Therefore it splits the received string, converts every substring to java-objects (or primitive types) and uses these values to reconstruct the original object.

Detailed explanation Classes that are used as the content of an AgentMessage implement the Externalizable interface (see recipe 51 on page 104). The Externalizable interface defines the different slots for a class. Slots are the attributes of that class that are extracted from an object of that class when converting it to a string. The values of these slots are collected in an InternalState object.

The conversion of the slots in the InternalState object to strings is handled as follows:

- If the slot is a primitive type (or a wrapper class of a primitive type): The value of the slot is converted to a string by the calling the *toString*() method of it's wrapper class.
- If the slot is a list or array: Every value is converted to a string. These strings are concatenated, together with brackets and comma's to seperate the different values.
- If the slot is an object of a class implementing the Externalizable interface: The slots are extracted from the object and each slot is converted to a string. These strings are concatenated, together with brackets and comma's to seperate the different slots.

The conversion from string to the different slots of the InternalState object is handled as follows:

- A string representing a primitive type: The method *convertToInteger(String string)*, *convertToFloat(String string)* .. from the corresponding wrapper class is called
- A string representing a list or array: The different substrings are retrieved, converted to objects and placed in a list or array.
- A string representing a class implementing the Externalizable interface: The substrings representing the different slots are extracted and used to create a new InternalState object.

This InternalState object is passed to the constructor of the class of the original object, where it initializes the values for the different slots of that class (see recipe 51 on page 104)

Transceiver

The transceiver is responsible for sending and receiving the StringMessage to another agent through the environment. This class is a hot spot in a physical environment where it can use for example radio communication to send the message to other transceiver.

In a software environment, the transceiver sends the message to the MessageDelivering who passes it to the correct receiver(s).

Part II

Practical implementation guide: the recipes

Developing a concrete application

This part of the cookbook explains how to develop a concrete application using the Delta framework. First we explain how to build a concrete application and then we'll discuss how to run this application. To build an application you just have to follow the recipes, they will guide you to implement your application.

An application in a physical environment

Developing the application

In a physical environment there is no need to model the environment or ongoing activities, because they are already present as physical entities. The only classes needed here are the classes to control the agents. These classes are collected in the behaviour of the agent. PhysicalAgentFactory is the factory needed to create the behaviour of an agent. This factory is described in recipe 3 on page 38.

Running the application

After creating a PhysicalAgentFactory, you can create different agents using this factory. The execution of every agent is started as shown in recipe 3 on page 38.

An application in a software environment

Developing the application

In a software environment, the behaviour of the agent also has to be created. SoftwareAgent-Factory is responsable for the creation of it. This factory is described in recipe 1 on the following page. Also ongoing activities and the environment need to be modeled in a software environment. The creation of an ongoing activity is done by OngoingActivityFactory (recipe 2 on page 37) and the creation of the environment by EnvironmentFactory (recipe 1 on the following page.

To create the entire system you have to use the SystemCreator. We'll explain how to do this in recipe 5 on page 44.

Running the application

When you create the application as described in recipe 5 on page 44, you return the SystemManager of the application. It offers methods to control the execution of the application and to add new items and relations to the system and to remove existing ones. See recipe 6 on page 45.

1 EnvironmentFactory

Purpose

The class SystemCreator (see recipe 5 on page 44) is responsible for building the system. It will create the environment, initialize its modules and place the items and relations between the items in the environment. Some parts are application dependent, f.e the laws, perceptual laws and communication laws of the MAS, the state of the environment, the items that exist in the environment, the relations between them, etc. The only thing you have to do to build your application, is specify those application dependent objects by creating a class that implements the interface EnvironmentFactory. This class is passed to the SystemManager that makes sure the laws, perceptual laws, communication laws, items and relations are added to the system.

Used in

Software MAS

Working instructions

(1) Create a class that implements the interface EnvironmentFactory.

(2) Implement the method *public State getEmptyState()*. Create the state of your environment and return an instance of it. How to create a state you can read in recipe 12 on page 53.

(3) Implement the method *public Vector<Law> getActionLaws(SystemManager systemManager)*. Create the different types of action laws of the environment and return a vector containing an instance of each type. How to create an action law, you can read in recipe 30 on page 78.

(4) Implement the method *public Vector<PerceptualLaw> getPerceptualLaws(State state)*. Create the different types of perceptual laws of the environment and return a vector containing an instance of each type. How to create a perceptual law, you can read in recipe 21 on page 66.

(5) Implement the method *public Vector<ItemState> getItems(EnvironmentFacade environment)*. Create the types of static items (see recipe 9 on page 49), and the factories for creating the agents (see recipe 4 on page 41) and ongoing activities (see recipe 2 on page 37) that exist in the environment and return a vector containing all the items that are initially present in the state of the environment.

(6) Implement the method *public Vector<Relation> getRelations()*. Create the different types of relations between the items in your environment and return a vector containing the relations that are initially present in the state of the environment. How to create a relation, you can read in recipe 11 on page 52.

(7) Implement the method *public Vector<CommunicationLaw> getCommunicationLaws(State state)*. Create the different types of communication laws of the environment and return a vector containing an instance of each type. How to create a communication law, you can read in recipe 53 on page 107.

(8) Implement the method *public SynchronizationChoice chooseSynchronization()*. Return the type of synchronization you want in your application. The enumeration SynchronizationChoice presents the different possibilities, return one of them.

(9) Implement the method *public SynchSchemeChoice chooseSynchronizationScheme()*. Return the type of synchronizationscheme you want in your application. The enumeration SynchSchemeChoice presents the different possibilities, return one of them.

(10) Implement the method *public <T extends VirtualRepresentation> Class<T> getRepresentationType()*. Create the represenation of the state of the environment and return the class of it. (see recipe 17 on page 59).

Example

Code 1 The PWEnvironmentFactory part 1

```
public class PWEnvironmentFactory implements EnvironmentFactory {                                    (1)
    //some parameters to fill in from gui:
    private GridState state;                                                                          (2.2)
    private int nbAgents, nbPacketColors, nbPacketsPerKind, nbDestinationsPerKind, initialEnergy ;   (5.1)
    private SynchronizationChoice synch;                                                              (8.2)

    //the colors for packets and destinations that can be used
    private String[] colors = {"green", "yellow", "red", "magenta", "cyan", "blue"};

    public State getEmptyState() {                                                                    (2)
        return state;
    }
    public void makeState(int width, int height, int perceptionRange) {                               (2.1)
        state = new GridState(width, height, perceptionRange);
    }

    public Vector<ActionLaw> getActionLaws(SystemManager systemManager) {                             (3)
        Vector<ActionLaw> laws = new Vector<ActionLaw>();
        laws.add(new PickLaw(systemManager));
        laws.add(new DropLaw(systemManager));
        laws.add(new StepLaw(systemManager));
        laws.add(new ThrowLaw(systemManager));
        laws.add(new PacketFlyLaw(systemManager));
        laws.add(new PacketLandLaw(systemManager));
        laws.add(new LandCatchLaw(systemManager));
        laws.add(new ChargeLaw(systemManager)) ;
        return laws;
    }

    public Vector<PerceptualLaw> getPerceptualLaws(State state) {                                     (4)
        Vector<PerceptualLaw> laws = new Vector<PerceptualLaw>();
        laws.add(new PerceptualLawObstacle((GridState)state));
        return laws;
    }

    public Vector<ItemState> getItems(EnvironmentFacade env) {                                        (5)
        Vector<ItemState> items = new Vector<ItemState>();
        //First create charge station
        ChargeStation chargeStation = new ChargeStation(getRandomPosition(),20) ;                     (5.2)
        items.add(chargeStation) ;

        //for each color:
        for (int i=0; i<nbPacketColors ; i++) {
            //make the number of destinations                                                         (5.3)
            for (int j=0; j<nbDestinationsPerKind; j++) {
                Position pos = getRandomPosition();
                Destination dest = new Destination(pos, colors[i]);
                items.add(dest);
            }
            //make the number of packets                                                              (5.4)
            for (int j=0; j<nbPacketsPerKind; j++) {
                pos = getRandomPosition();
                Packet p = new Packet(pos, colors[i]);
                items.add(p);
            }
        //make the right number of agents                                                           (5.5)
        PWAgentFactory factory = new PWAgentFactory() ;
        for (int i=0; i<nbAgents; i++) {
            pos = getRandomPosition();
            AgentState agent = factory.createNewAgent("a" + (i+1), env, pos);
            items.add(agent);
        }
        return items;
    }

    private Position getRandomPosition() {
        //get a random position that is still free
    }
}
```

Code 2 The PWEnvironmentFactory part 2

```
    public Vector<Relation> getRelations() {                                                    (6)
        return new Vector<Relation>();
    }

    public Vector<CommunicationLaw> getCommunicationLaws(State state) {                         (7)
        Vector<CommunicationLaw> laws = new Vector<CommunicationLaw>() ;
        laws.add(new CommunicationRangeLaw((GridState)state)) ;
        return laws ;
    }

    public SynchronizationChoice chooseSynchronization() {                                      (8)
        return synch;
    }
    public void setSynchronization(SynchronizationChoice synch) {                             (8.1)
        this.synch = synch;
    }

    public SynchSchemeChoice chooseSynchronizationScheme() {                                    (9)
        return SynchSchemeChoice.LTD;
    }

    public <T extends VirtualRepresentation> Class<T> getRepresentationType() {                 (10)
        return (Class<T>)GridStateRepresentation.class;
    }
}
```

The factory to initialize the packetworld is the PWEnvironmentFactory. Some things that have to be filled in are asked to the user of the application by a GUI.

(2) The state of the environment of the packetworld is GridState. This class is developed. The gui of the packetworld application calls the method *makeState(...)* to instantiate the state of the environment (2.1). The user of the application can define the characteristics of the state in the gui. The created state is saved in the attribute *state* (2.2) which is returned in the method *getState()*.

(3) The packetworld has to following action laws: PickLaw, DropLaw, StepLaw, ThrowLaw, PacketFlyLaw, ThrowLaw, PacketLandLaw, LandCatchLaw, ChargeLaw. The action laws are developed and a vector containing an instance of each action law is returned

(4) The packetworld has one perceptual law, PerceptualLawObstacle. This perceptual law is first developed and a vector containing an instance of that law is returned

(5) The packetworld has three types of static items: Packet, Destination and ChargeStation. First these are developed. Next to static items the packetworld has one type of agents: PacketWorldAgent. The factory for this type of agent (PWAgentFactory) is developed. There is one type of ongoing activity: ThrownPacket. The factory for this type (ThrownPacketFactory) is developed.
The user of the packetworld application can specify in the gui the number of packets, destinations, ... All these characteristics are filled in the attributes in (5.1). In the method *getItems(...)* first one charge station is created (5.2) by instantiating the class ChargeStation. Next the right number of destinations (5.3) and packets is created (5.4) again by instantiation respectively the classes Destination and Packet. Next the agents are created by using the method *createNewAgent(...)* of an instance of the PWAgentFactory (5.5). All the created items are returned in a vector.

(6) The packetworld has one relation, the HoldRelation. First this relation is developed. When the packetworld is created initially it does not contain any relations, so an empty vector is returned.

(7) The packetworld has one communication law, CommunicationRangeLaw. This communication law is developed and a vector containing an instance of that law is returned.

(8) The user of the packetworld application can choose the type of synchronization in the gui. The gui calls the method *setSynchronization(...)* (8.1). The type of synchronization is saved in the attribute *synch* (8.2), which is returned in the method *chooseSynchronization()*.

(9) The choice for synchronizationscheme for the packetworld is SynchSchemeChoice.LTD.

(10) The representation of the state of the environment is GridStateRepresentation. This representation is developed and the class of this representation is returned

2 OngoingActivityFactory

Purpose

An OngoingActivityFactory creates the different components of the behaviour and the external state of an ongoing activity and joins these components together to create the ongoing activity. When creating a new ongoing activity you can pass certain arguments. These arguments are passed to every method that's implemented by the developer.

This recipe explains how to create an OngoingActivityFactory and how to create a new ongoing activity using this factory.

Used in

Software MAS

Working instructions

(1) Create a subclass of OngoingActivityFactory.

(2) Implement the method *protected OngoingActivityState getOngoingActivityState(Region region, Object[] args)*. Create the external state of the ongoing activity using recipe 8 on page 48 and return an instance of it.

(3) Implement the method *protected OngoingActivityDecision getDecision(OngoingActivityBehaviourFacade beh, SoftwareExecution exec, State state, Object args[])*. Create the Decision class of the ongoing activity using recipe 28 and return an instance of it.

(4) Implement the method *protected SoftwareExecution getExecution(EnvironmentFacade env, OngoingActivityBehaviourFacade beh, Object args[])*. If the execution class is not create yet, create the Execution class of the ongoing activity using recipe 26 and return an instance of it.

(5) Use this factory to create a thrown packet. The thrown packet can be created with the method *public AgentState createNewAgent(EnvironmentFacade env, State state, Region region, Object... args)*. The parameter *env* is the facade of the environment, *state* is the state of the environment and *region* is the region this object occupies. The parameter *args* can be used to pass any number of arguments, these arguments will be passed to the methods discussed above.

Example

```
public class ThrownPacketFactory extends OngoingActivityFactory {                                    (1)

    protected ThrownPacket getOngoingActivityState(Region pos, Object[] args) {                       (2)
        if (args.length != 3)
            throw new IllegalArgumentException("ThrownPacket must have other arguments");
        int speed = (Integer)args[2];
        LinkedList<Position> path = (LinkedList<Position>)args[0];
        String color = (String)args[1];
        return new ThrownPacket((Position) pos, path, speed, color);
    }

    protected ThrownPacketDecision getDecision(OngoingActivityBehaviourFacade beh,                   (3)
                                SoftwareExecution exec, State state, Object[] args) {
            return new ThrownPacketDecision(beh, exec, state);
    }

    protected SoftwareExecution getExecution(EnvironmentFacade env,                                   (4)
                                OngoingActivityBehaviourFacade beh, Object[] args) {
        return new PWExecution(env, beh);
    }
}
```

ThrownPacketFactory is the factory by which thrown packets can be created in the packetworld

(2) The external state of a thrown packet is ThrownPacket. An instance of this class is returned. The parameters to instantiate this class with are passed in the list with arguments.

(3) The Decision class of a thrown packet is ThownPacketDecision. A instance of this class is returned.

(4) The Execution class of a thrown packet is PWExecution. A instance of this class is returned.

Code 3 Creating a thrown packet

```
Position pos = getRandomPosition();
OngoingActivityState tp = new ThrownPacketFactory().createNewOngoingActivity(                        (5)
          env, state, pos, path,"green", 1);
```

(5) The argument *env* holds a reference to the environment, *state* holds a reference to the state of the environment, the position of the thrown packet is a random free position in the grid. Next to these argument there are extra arguments: the path of the packet, the color ant the speed.

3 PhysicalAgentFactory

Purpose

A PhysicalAgentFactory creates the different components of the behaviour of a physical agent and joins these components together to create an agent. This recipe describes how to implement a PhysicalAgentFactory, how to use this subclass to create an agent and how to start the execution of this agent. When creating a new agent (method *public AgentSchedule createAgentBehaviour(String ID, Object... args)*) you can pass certain arguments. These arguments are passed to every method that's described below and has to be implemented by the developer. We'll see an example of the use of these arguments below.

Used in

Physical MAS

Working instructions

(1) Create a subclass of PhysicalAgentFactory.

(2) Implement the method *public AgentScheduleChoice chooseSchedule(Object[] args)*. This method returns a choice in the enumeration AgentScheduleChoice. This enumeration represents the choice of AgentSchedule: the developer has to choose between LTDAgentSchedule and LCDAgentSchedule.

(3) Implement the method *protected Vector<Focus> createInitialFoci(Object[] args)*: This method returns the initial foci for this agent. The initial foci are the foci that are used when the agent performs perception for the first time. See recipe 14 on page 57 for more information on creating foci.

(4) Implement the method *protected Vector<Filter> createInitialFilters(Object args[])*. This method returns the initial filters for this agent. See recipe 15 on page 57 for more information on creating filters.

(5) Implement the method *protected Vector<Sensor> createSensors(Object[] args)*. This method returns the sensors of this agent. See recipe 18 on page 62 for more information on creating these sensors.

(6) Implement the method *protected <T extends Percept> Class<T> createPerceptType(Object[] args)*. This method returns the class that represents the percept for this agent. See recipe 19 on page 63 for more information on creating a percept.

(7) Implement the method *protected Vector<Description> createDescriptions(Object[] args)*. This method returns the descriptions for this agent. See recipe 20 on page 65 for more information on creating descriptions.

(8) Implement the method *protected AgentDecision createDecision(Execution execution, KnowledgeIntegration knowi,Object[] args)*. This method creates the Decision class for this agent. See recipes 24 on page 71 and 35 on page 84 for more information on creating the Decision class of an agent.

(9) Implement the method *protected Execution createExecution(Object[] args)*. This method creates the Execution class for this agent. See recipes 25 on page 72 for more information on creating the Execution class of an agent.

(10) Implement the method *protected Vector<Protocol> createCommunicationProtocols(Object[] args)*. This method creates the communication protocols for this agent. See recipe 42 on page 94 for more information on creating communication protocols.

(11) Implement the method *protected Ontology createOntology(Object[] args)*. This method creates the ontology of this agent. See recipes 52 on page 106 and 51 on page 104 for more information on creating an ontology.

(12) Implement the method *protected abstract Transceiver createTransceiver(Object[] args)*. This method creates a Transceiver for this agent.

(13) Use this factory to create the behaviour of an agent. This behaviour can be created by calling the method *public Schedule createAgentBehaviour(Object... args)*. The parameter *args* can be used to pass any number of arguments, these arguments will be passed to the methods discussed above. This method returns the Schedule for the agent.

(14) Start the AgentSchedule by calling the method *public void start()*.

Example

Code 4 The IslandRobotFactory

```
public class IslandRobotFactory extends PhysicalAgentFactory {                                    (1)

    protected AgentScheduleChoice chooseSchedule(Object[] args) {                                  (2)
        return AgentScheduleChoice.LTDAgentSchedule;
    }

    public Vector<Focus> createInitialFoci(Object[] args) {                                        (3)
        Vector<Focus> result = new Vector<Focus>();
        result.add(new IslandWorldFocus());
        return result;
    }

    protected Vector<Filter> createInitialFilters(Object[] args) {                                 (4)
        return new Vector<Filter>();
    }

    protected Vector<Sensor> createSensors(Object[] args) {                                        (5)
        Vector<Sensor> result = new Vector<Sensor>();
        InfraRedSensor comm = (InfraRedSensor) args[0];
        result.add(comm);
        return result;
    }

    protected <T extends Percept> Class<T> createPerceptType(Object[] args) {                      (6)
        return (Class<T>) IWStatePercept.class;
    }

    protected Vector<Description> createDescriptions(Object[] args) {                              (7)
        Vector<Description> result = new Vector<Description>();
        return result;
    }
    protected AgentDecision createDecision(Execution execution, KnowledgeIntegration knowi,
                                                                    Object[] args) {               (8)
        return new IslandAgentDecision(execution,knowi) ;
    }
    protected Execution createExecution(Object[] args) {                                           (9)
        InfraRedSensor comm = (InfraRedSensor) args[0];
        return new MindstormExecution(comm);
    }
    protected Vector<Protocol> createCommunicationProtocols(Object[] args) {                       (10)
        Vector<Protocol> result = new Vector<Protocol>();
        Protocol prot = new Protocol("passPacketProtocol");
        prot.addStep(new PacketReady()) ;
        prot.addStep(new HandlePacketReady()) ;
        prot.addStep(new ReleasePacket()) ;
        prot.addStep(new HandleReleasePacket()) ;
        prot.addStep(new PacketReleased()) ;
        prot.addStep(new HandlePacketReleased()) ;
        prot.addStep(new PacketPassed()) ;
        prot.addStep(new HandlePacketPassed()) ;
        result.add(prot) ;
        return result;
    }
    protected Ontology createOntology(Object[] args) {                                             (11)
        Ontology ontology = new Ontology();
        return ontology;
    }
    protected Transceiver createTransceiver(Object[] args) {                                       (12)
        IWTransceiver transc = new IWTransceiver((TwoAgentCommunicationMedium)args[1]) ;
        ((TwoAgentCommunicationMedium)args[1]).setSecondAgent(transc) ;
        return transc ;
    }
    protected KnowledgeIntegration createKnowledgeIntegration(String name, Object args[]) {
        KnowledgeIntegration knowi = super.createKnowledgeIntegration(name,args) ;
        knowi.addKnowledgeObject(new IslandAgentCurrentAction(IslandAgentCurrentAction.WAIT)) ;
        knowi.addUniqueKnowledgeObjectType(IslandAgentCurrentAction.class) ;
        return knowi ;
    }
}
```

The IslandRobotFactory is used to create an island robot. An island robot is a robot build in Lego Mindstorms. This robot is used in an application with two robots: the island robot and the sea robot. The sea robot is positioned in the "sea" and the island robot is positioned on the "island". The sea robots collects the packets from the sea and brings them to the island. There, the sea robot passes the packet to the island robot. Finally, the island robot drops the packet in the destination on the island.

(2) This agent uses the LTDAgentSchedule and thus returns AgentScheduleChoice.LTDAgentSchedule

(3) The only focus used in this application is the IslandWorldFocus, so we select this focus as initial focus

(4) We do not use filters in this application, so we return an empty selection.

(5) This robot has one sensor, the InfraredSensor. Because this sensor is created outside this factory, it is passed as an argument. This method retrieves it from the list of arguments and returns it as the only sensor.

(6) The type of percept used in this application is the IWStatePercept. This method returns this type.

(7) We do not use any descriptions in this application, so an empty vector is returned.

(8) The Decision class used for this robot is IslandAgentDecision.

(9) The Execution used for this robot is MindStormExecution

(10) The robot has one communication protocol, the PassPacketProtocol.

(11) The robot doesn't send any communication messages with a content in the passPacketProtocol. Therefore this robot can return an empty ontology.

(12) The island robot communicates with the sea robot using the class TwoAgentCommunicationMedium. An object of this class is created outside this factory an passed in the list of arguments.

Code 5 Creating and starting an island agent

```
IslandRobotFactory islandRobotFactory = new IslandRobotFactory() ;
AgentSchedule islandRobot = islandRobotFactory.createAgentBehaviour("islandAgent",sensor,medium) ;   (13)
islandRobot.start() ;                                                                                 (14)
```

(13) Creating the island robot is easy: first create the factory, then call the method *public AgentSchedule createAgentBehaviour(String agentID, Object... args)*. This method is called with two arguments, the infrared sensor and the communication medium.

(14) Finally, the robot is started using the *public void start()* method.

4 SoftwareAgentFactory

Purpose

A SoftwareAgentFactory creates the different components of the behaviour and the external state of a software agent and joins these components together to create a new agent. When creating a new agent you can pass certain arguments. These arguments are passed to every method that's implemented by the developer. We don't use these arguments in the example in this recipe, but an example of the use of these arguments is given in recipe 3. This recipe will also explain how to create a new agent using this factory.

Used in

Software MAS

Working instructions

(1) Create a subclass of SoftwareAgentFactory.

(2) Implement the method *public AgentScheduleChoice chooseSchedule(Object[] args).* This method returns the choice in the enumeration AgentScheduleChoice. This enumeration represents the choice of AgentSchedule: the developer has to choose between LTDAgentSchedule and LCDAgentSchedule.

(3) Implement the method *protected Vector<Focus> createInitialFoci(Object[] args).* This method returns the initial foci for this agent. The initial foci are the foci that are used when the agent performs perception for the first time. See recipe 14 on page 57 for more information on creating foci.

(4) Implement the method *protected Vector<Filter> createInitialFilters(Object args[]).* This method returns the initial filters for this agent. See recipe 15 on page 57 for more information on creating filters.

(5) Implement the method *protected <T extends Percept> Class<T> createPerceptType(Object[] args).* This method returns the type of percept that will be used for this agent. See recipe 19 on page 63 for more information on creating a percept.

(6) Implement the method *protected Vector<Description> createDescriptions(Object[] args).* This method returns the descriptions of this agent. See recipe 20 on page 65 for more information on creating descriptions.

(7) Implement the method *protected AgentDecision createDecision(Execution execution, KnowledgeIntegration knowi,Object[] args).* This method creates the Decision class of this agent. See recipe 24 on page 71 and 35 on page 84 for more information on creating the Decision class of an agent.

(8) Implement the method *protected Execution createExecution(Object[] args).* This method creates the Execution class of this agent. See recipe 25 on page 72 for more information on creating the Execution class of an agent.

(9) Implement the method *protected Vector<Protocol> createCommunicationProtocols(Object[] args).* This method creates the communication protocols of this agent. See recipe 42 on page 94 for more information on creating communication protocols.

(10) Implement the method *protected Ontology createOntology(Object[] args).* This method creates the ontology of this agent. See recipe 52 on page 106 and recipe 51 on page 104 for more information on creating the ontology.

(11) Implement the method *protected ConsumptionHandler createConsumptionHandler(KnowledgeIntegration knowi).* This method creates the consumption handler for this agent. See recipe 27 on page 75 for more information on creating the consumption handler.

(12) Implement the method *protected AgentState createAgentState(String ID, Region region, Object... args).* This method creates the external state of this agent. See recipe 7 on page 47 for more information on creating the external state of an agent.

(13) Use this factory to create the agent. The agent can be created with the method *public AgentState createNewAgent(String ID, EnvironmentFacade env, Region region, Object... args).* The parameter *ID* is the ID of the agent that's beeing created, *env* is the facade of the environment and *region* is the region this object occupies. The parameter *args* can be used to pass any number of arguments, these arguments will be passed to the methods discussed above.

Example

Code 6 The PWAgentFactory

```
public class PWAgentFactory extends SoftwareAgentFactory {                                    (1)

    protected AgentScheduleChoice chooseSchedule(Object[] args) {                              (2)
        return AgentScheduleChoice.LTDAgentSchedule ;
    }

    protected Vector<Focus> createInitialFoci(Object[] args) {                                 (3)
        Vector<Focus> foci = new Vector<Focus>();
        foci.add(new VisualFocus());
        return foci;
    }

    protected Vector<Filter> createInitialFilters(Object[] args) {                             (4)
        return new Vector<Filter>() ;
    }

    protected <T extends Percept> Class<T> createPerceptType(Object[] args) {                  (5)
         return (Class<T>)GridStatePercept.class;
    }

   protected Vector<Description> createDescriptions(Object[] args) {                          (6)
        Vector<Description> descriptions = new Vector<Description>() ;
        descriptions.add(new ItemDescription()) ;
        descriptions.add(new GradientDescription()) ;
        descriptions.add(new RelationDescription()) ;
        return descriptions ;
    }

    protected AgentDecision createDecision(Execution exec,
                               KnowledgeIntegration knowI, Object[] args) {                    (7)
        return new PWFreeFlowDecision(exec,knowI) ;
    }

    protected SoftwareExecution createExecution(EnvironmentFacade env,
                                  AgentBehaviourFacade beh, Object[] args) {                   (8)
        return new PWExecution(env, beh);
    }

   protected Vector<Protocol> createCommunicationProtocols(Object[] args) {                    (9)
        Vector<Protocol> protocols = new Vector<Protocol>();
        Protocol p = new Protocol("ThrowPacketProtocol");
        p.addStep(new InitiateThrowPacketProtocol()) ;
        p.addStep(new HandleRequestThrow());
        p.addStep(new HandleAcceptThrowPacket());
        p.addStep(new HandleRejectThrowPacket());
        p.addStep(new TerminateConversation()) ;
        protocols.add(p);
        return protocols;
    }
    protected Ontology createOntology(Object[] args) {                                         (10)
        Ontology onto = new Ontology();
        try {
            onto.addToVocabularium(PacketRepresentation.class, "packet");
            onto.addToVocabularium(PositionRepresentation.class, "position");
        }
        catch (IncorrectClassException e) {
            e.printStackTrace();
        }
        return onto;
    }

    protected ConsumptionHandler createConsumptionHandler(KnowledgeIntegration knowi) {        (11)
         return new PWConsumptionHandler(knowi);
    }

    protected AgentState createAgentState(Region region, Object... args) {                     (12)
         return new PWAgentState((Position) region);
    }
}
```

Code 7 Creating the packetworld agent

```
Position pos = getRandomPosition();
AgentState agent = factory.createNewAgent("a1", env, pos);                    (13)
```

The PWAgentFactory creates the agents used in the packetworld. We'll explain the implementation of the different methods here:

(2) The agents in the packetworld use the LTDAgentSchedule, thus this method returns the AgentScheduleChoice.LTDAgentSchedule.

(3) The agents only use one focus, the VisualFocus, so a selection with one VisualFocus is returned.

(4) Initially, no filters are used. An empty selection of filters is returned.

(5) The type of percept used in the packetworld is the GridStatePercept. This method returns this type.

(6) Three descriptions are used in the packetworld: ItemDescription (to interprete items in the representation), GradientDescription (to interprete the gradient of the destionations and the battery) and RelationDescription (to interprete the relations in this representation). One description of every type is returned.

(7) The agents use a free-flow tree to select an operator. This free-flow tree is implemented in PWFreeFlowDecision.

(8) The mapping of operators to influences in the packetworld is implemented by the PWExecution.

(9) The agents have one communication protocol: ThrowPacketProtocol. This protocol is used to agree on throwing a packet from one agent to another. The different protocol steps are added to this protocol and the protocol is returned.

(10) The ontology of these agents contains two classes: PacketRepresentation and PositionRepresentation. These classes are added and the ontology is returned.

(11) The consumption handler used in the packetworld is the PWConsumptionHandler. An object of this class is initiated and returned.

(12) The external state of an agent in the packet world is represented by PWAgentState. An object of this class is created with the given region (casts to a position, because agents in the packetworld occupy a certain position).

(13) The position of this agent is a random free position in the grid and the id of this agent is "a1". The variable *env* holds a reference to the environment.

5 SystemCreator

Purpose

After you have implemented the factories to create the items, ongoing activities and environment, it's time to build the entire application. The class SystemCreator is responsible for this. It will create the environment, initialize its modules and place the items and relations between the items in the environment. This recipe explains how you can create and start the entire system by using the SystemCreator.

Used in

Software MAS

Working instructions

(1) Create an instance of the class SystemCreator

(2) Run the method *public SystemManager createSystem(EnvironmentFactory userFactory)*: this method creates the entire application and returns the SystemManager of your application. You use this manager to run you application. The parameter *userfactory* is an instance of the EnvironmentFactory you have created (see recipe 1 on page 34). This method does only creates the application, it doesn't start it yet. How to start the application you can read in recipe 6.

Example

Code 8 Use of the class SystemCreator

```
SystemCreator systemCreator = new SystemCreator();                                          (1)
SystemManager systemManager = systemCreator.createSystem(new PWEnvironmentFactory());      (2)
```

(2) The EnvironmentFactory used to create the environment of the packetworld is PWEnvironmentFactory

6 SystemManager

Purpose

The SystemManager of the system is central class of the system that is the interface of the application for the developer. It offers methods to control the execution of the application and to add new items and relations to the system and to remove existing ones. This recipe explains how you can use the SystemManager in your application

Used in

Software MAS

Working instructions

In this recipe you don't have to follow all the instructions. It are just all the methods of the SystemManager offers.
To control the execution of your application you can use the following methods:

(1) To start your application, run the method *public void StartSystem()*. This method starts the execution of the entire system. It starts the execution of the modules of the environment and the execution of the behaviours of the dynamic items.

(2) To terminate your application, run the method *public void TerminateSystem()*. This method terminates the execution of the entire system. It terminates the execution of the modules of the environment and the execution of the behaviours of the dynamic items. All the items and relations are removed from the system.

(3) To suspend your application, run the method *public void SuspendSystem()*. This method suspends the execution of the entire system. It suspends the execution of the modules of the environment and the execution of the behaviours of the dynamic items.

(4) To resume your application after you suspended it, run the method *public void ResumeSystem()*. This method resumes the execution of the entire system. It resumes the execution of the modules of the environment and the execution of the behaviours of the dynamic items.

To add new items and relations to the system and to remove existing ones, you can use the following methods:

(5) To add an item to your application, run the method *public void addItem()*. This method adds the item to the system. It is added to the state of the environment and if it is a dynamic item, the behaviour of the item is started. The modules of the environment are then notified that the item has been added.

(6) To remove an item from your application, run the method *public void removeItem()*. This method removes the item from the system. It is removed from the state of the environment and if it is a dynamic item, the behaviour of the item is terminated. The modules of the environment are then notified that the item has been removed.

(7) To add a relation to your application, run the method *public void addRelation()*. This method adds the relation to the system, it is added to the state of the environment.

(8) To remove a relation from your application, run the method *public void removeRelation()*. This method removes the relation from the system, it is removed from the state of the environment.

Example

Code 9 Use of the class SystemManager

```
systemManager.startSystem();                                                    (1)
systemManager.addItem(new Packet(new Position(0,0), "green"))                   (2)
```

As an example we will start the system we created in recipe 5 on page 44 (1) and add an extra packet to it (4). The other methods can be used in the same way.

7 AgentState

Purpose

Every software agent has an external state and a behaviour. The AgentState represents the external state of an agent. This recipe explains how you can create the external state of an agent.

Used in

Software MAS

Working instructions

(1) Create a subclass of AgentState.

(2) If necessary, add attributes that represent the external state of the agent and the according getters and setters.

(3) Add a constructor with as parameter *Region region* and pass it to the superclass. You can add other parameters to initialize the attributes you've defined in (2)**.** *Region* is the region the agent occupies in the state of the environment.

(4) Implement the method *public <T extends ItemRepresentation> T getRepresentation()*. The external state of an agent is used in the perception of agents, this method returns the representation of the external state of the agent (see recipe 22 on page 68).

Example

Code 10 The PWAgentState

```
public class PWAgentState extends AgentState {                                    (1)

    private int energyLevel;                                                      (2)
    public static int initialEnergyLevel = 100;

    public PWAgentState(Position position) {                                      (3)
        super(position);
        setEnergyLevel(initialEnergyLevel);
    }

    public int getEnergyLevel() {                                                 (2)
        return energyLevel;
    }
    public void setEnergyLevel(int energyLevel) {
        this.energyLevel = energyLevel;
    }

    public PWAgentRepresentation getRepresentation() {                            (4)
        return new PWAgentRepresentation(pos.getRepresentation(),
                                         getID(), getEnergyLevel());
    }
}
```

As an example we give PWAgentState, that represents the external state of an agent in the packetworld.

(2) The attribute *energyLevel* represents the energy of the battery of the agent, *initialEnergyLevel* represents the initial level of energy an agent has when he is created.

(3) The constructor has the argument *Position position*, which is passed to the constructor of the superclass. *Position* is the position of the agent in the packetworld. The *energyLevel* is set to the *initialEnergyLevel*.

(4) The method *getRepresentation()* returns a PWAgentRepresentation. This is the representation of the external state of an agent in the packet world.

8 OngoingActivityState

Purpose

Every ongoing activity has external state and a behaviour. The OngoingActivityState represents the external state of the ongoing activity. This recipe explains you how to create the external state of an ongoing activity in your application.

Used in

Software MAS

Working instructions

(1) Create a subclass of OngoingActivityState.

(2) If necessary, add attributes that represent the ongoing activity and add according getters and setters.

(3) Add a constructor with as parameter *Region region* and pass it to the superclass. The parameter *region* is the region of the ongoing activity in the state of the environment. Next to the region you can add other parameters to initialize the attributes you have defined in (2).

(4) Implement the method *public <T extends ItemRepresentation> T getRepresentation()*. Ongoing activities are used in the perception of agents, this method returns the representation of the ongoing activity (see recipe 22 on page 68).

Example

An example of an ongoing activity is ThrownPacket, that represents a packet that was thrown by an agent.

(2) The attribute *speed* represents the speed by which the ThrownPacket can fly, *color* is the color of the packet

(3) The class ThrownPacket has one constructor, it has as parameters *Position pos*, *int speed* and *String color*. *Pos* is the position of the ongoing activity in the state of the environment. It is passed to the superclass. The parameter *speed* is used to initialize the attribute *speed* and the parameter *color* is used to initialize the attribute *color*.

(4) The method *getRepresentation()* returns a ThrownPacketRepresentation. This is the representation of a ThrownPacket. For more information about creating representations see recipe 22 on page 68.

Code 11 The class ThrownPacket

```
public class ThrownPacket extends OngoingActivityState {                                (1)

    private int speed;                                                                   (2)
    private String color;

    public ThrownPacket(Position pos, int speed, String color) {                        (3)
        super(pos);
        setSpeed(speed);
        setColor(color);
    }

    public int getSpeed() {                                                              (2)
        return speed;
    }
    private void setSpeed(int speed) {
        this.speed = speed;
    }

    public String getColor() {                                                           (2)
        return color;
    }
    private void setColor(String color) {
        this.color = color;
    }

    public ThrownPacketRepresentation getRepresentation() {                              (4)
        Position pos = (Position) getRegion();
        return new ThrownPacketRepresentation(pos.getRepresentation());
    }
}
```

9 StaticItemState

Purpose

StaticItemState represents the external state of a static item. A static item is an item in the environment that models the domain of application. Examples of static items in the packetworld are packets and destinations. This recipe explains you how to create the external state of a static item in your application.

Used in

Software MAS

Working instructions

(1) Create a subclass of StaticItemState.

(2) If necessary, add attributes that represent the item and the according getters and setters.

(3) Add a constructor with as parameter *Region region* and pass it to the superclass. The parameter *region* is the region of the static item in the state of the environment. Next to the region you can add other parameters to initialize the attributes you have defined in (2).

(4) Implement the method *public <T extends ItemRepresentation> T getRepresentation()*. Static items in the environment are used in the perception of agents, this method returns the representation of the static item (see recipe 22 on page 68).

Example

Code 12 The class PACKET

```
public class Packet extends StaticItemState {                                    (1)

    private String color;                                                        (2)

    public Packet(Position position, String color)  {                            (3)
        super(position) ;
        setColor(color);
    }

    public String getColor() {                                                   (2)
        return color;
    }
    private void setColor(String color) {                                        (2)
        this.color = color;
    }

    public PacketRepresentation getRepresentation() {                            (4)
        Position pos = (Position) getRegion();
        return new PacketRepresentation(pos.getRepresentation(),getColor()) ;
}
```

An example of a static item in the packetworld is PACKET, that represents a packet in the packetworld.

(2) The packet has, next to its position one attribute, the color of the packet, with the according getter and setter.

(3) The class PACKET has one constructor, it has as parameters the position of the new packet and its color. These are used to initialize the attributes of the packet.

(4) The method *getRepresentation()* returns a PACKETREPRESENTATION. This is the representation of a packet in the packetworld. For more information about creating representations see recipe 22 on page 68.

10 Region

Purpose

This framework is a framework for situated multi-agent systems, so every item in the multi-agent system is situated in the environment. It occupies some region in the state of the environment. This recipe explains how you can create the type of region for the items of your application.

Used in

Software MAS

Working instructions

(1) Create a class that implements the interface REGION.

(2) If necessary, add attributes that represent the item and the according getters and setters.

(3) Add a constructor with parameters to initialize the attributes you have defined in (2).

(4) Implement the method *public <T extends RegionRepresentation> T getRepresentation()*. This method returns the representation of the region.

(5) Add methods to inspect and modify the region.

Example

Code 13 The region POSITION

```
public class Position implements Region  {                                                    (1)

    private int x, y;                                                                         (2)
    private Vector<ItemState> items;

    public Position(int x, int y) {                                                           (3)
        setX(x);
        setY(y);
        setItems(new Vector<ItemState>());
    }

    public PositionRepresentation getRepresentation() {                                       (4)
        PositionRepresentation pos = new PositionRepresentation(this.getX(), this.getY());
        return pos ;
    }

    public boolean containsItems(){                                                         (5.1)
        return ! (getItems().isEmpty());
    }

    public boolean contains(ItemState item){
        return getItems().contains(item);
    }

    public <T extends ItemState> Vector<T> getItemsOfType(Class<T> type) {
        Vector<T> items = new Vector<T>();
        for (ItemState item : getItems())
           if (item.getClass() == type)
              items.add((T)item);
        return items;
    }

    public void addItem(ItemState item){                                                    (5.2)
        getItems().add(item);
    }

    public void removeItem(ItemState item){
        getItems().remove(item);
    }
}
```

POSITION is the type of region of the items in the packetworld. The state of the environment in the packetworld consists of a grid of positions on which the items are situated. Every position contains the items that are positioned on it.

(2) The attributes x and y represent the coordinates of the position. *Items* holds a reference to the items that are positioned on the position. In this example we left out the getters and the setters. We refer the interested reader to the application code of the packetworld.

(3) The constructor of the class initializes the attributed defined in (2).

(4) The method *getRepresentation()* returns a POSITIONREPRESENTATION. This is the representation of a position in the packetworld.

(5) Methods are added to inspect and modify the position. There are methods to retrieve information about the items on the position (2.1) and methods to add and remove items to the position (5.2).

11 Relation

Purpose

Items that exist in the environment can be related to each other. These relations are explicitly modeled in the environment. This recipe describes how you can model the relations between the items in your environment.

Used in

Software MAS

Working instructions

(1) Create a subclass of RELATION.

(2) Add attributes that represent the relation, and the according getters and setters.

(3) Add a constructor with parameters to initialize the attributes you have defined in (2).

(4) Implement the method *public <T extends ItemState> Vector<T > getItems()*. Return a vector containing the items involved in the relation.

(5) Implement the method *public <T extends RelationRepresentation> T getRepresentation()*. Relations are used in the perception of agents, this method returns the representation of the relation. First you have to create the representation of the relation (see recipe 23 on page 69) en in this method you should return an instance of that representation.

Example

An example of a relation in the packetworld is the HOLDRELATION, a relation that expresses that an agent is holding a packet.

(2) The agent and the packet involved in the relation represent the relation.

(3) The constructor of the class has as parameters the agent that is holding a packet and the packet. These parameters are used to initialize the attributes defined in 2.

(4) The items involved in the relation are the agent and the packet.

(5) The method *getRepresentation()* returns a HOLDRELATIONREPRESENTATION. This is the representation of the HOLDRELATION in the packetworld.

Code 14 The relation HoldRelation

```
public class HoldRelation extends Relation {                                                    (1)

    private AgentState agent ;                                                                  (2)
    private PacketState packet ;

    public HoldRelation(AgentState agent, PacketState packet) {                                 (3)
        setAgent(agent) ;
        setPacket(packet) ;
    }

    public AgentState getAgent() {                                                              (2)
        return agent;
    }
    public void setAgent(AgentState agent) {
       this.agent = agent;
    }

    public PacketState getPacket() {                                                            (2)
        return packet;
    }
    public void setPacket(PacketState packet) {
       this.packet = packet
    }

    public <T extends ItemState> Vector<T> getItems() {                                         (4)
        Vector<ItemState> items = new Vector<ItemState>();
        items.add(agent);
        items.add(packet);
        return items;
    }

    public HoldRelationRepresentation getRepresentation() {                                     (5)
        return new HoldRelationRepresentation(
           getAgent().getRepresentation(), getPacket().getRepresentation());
    }
}
```

12 State

Purpose

The state of the environment is the part of the environment that holds the items and the relations between them. It represents the physical state of the environment. The state of the environment is used very frequently by the different laws (action, perceptual and communication laws) of the MAS.

Used in

Software MAS

Working instructions

(1) Create a subclass of State.

(2) Add the attributes that represent the state and the according getters and setters.

(3) Add a constructor. Add the parameters needed to initialize the state.

(4) Implement the method *public void addItem(ItemState item)*. Implement in this method how an item is added to the state.

(5) Implement the method *public void removeItem(ItemState item)*. Implement in this method how an item is removed from the state

(6) Implement the method *public Vector<ItemState> getAllItems()*. Return all the items that exist in the state of the world.

(7) Implement the method *public Vector<ItemState> getItemsInPerceptionRange(DynamicItemState observer)*. Return all the items in the perception range of a given agent

(8) Add methods to inspect the state of the world. These are the methods that are needed by the different laws of the environment. The action laws for example need the state to to check if certain conditions in the state of the world are fulfilled and to execute some effects (see recipe 30).

Example

Code 15 The state GridState part 1

```
public class GridState extends State {                                                          (1)

    private Position[][] grid;
    private int perceptionRange;                                                                (2)

    public GridState(int width, int height, int perceptionRange) {                              (3)
        setPerceptionRange(perceptionRange);
        setGrid(width, height);
        for (int i = 0; i < width; i++)
           for (int j = 0; j < height; j++)
              getGrid()[i][j] = new Position(i, j);
    }

    public void addItem(ItemState item) {                                                       (4)
        Position p = (Position) item.getRegion();
        getGrid()[p.getX()][p.getY()].addItem(item);
    }

    public void removeItem(ItemState item) {                                                    (5)
        Position p = (Position) item.getRegion();
        getGrid()[p.getX()][p.getY()].removeItem(item);
    }

    public Vector<ItemState> getAllItems() {                                                    (6)
        Vector<ItemState> items = new Vector<ItemState>();
        for (int x = 0; x < getWidth(); x++) {
           for (int y = 0; y < getHeight(); y++) {
              items.addAll(getGrid()[x][y].getItems());
           }
        }
        return items;
    }

    public Vector<ItemState> getItemsInPerceptionRange(DynamicItemState observer) {             (7)
        return getItemsInRange(observer, getPerceptionRange());
    }

    public Vector<ItemState> getItemsInRange(DynamicItemState item, int range) {                (7.1)
        //this method returns a vector containing all the items that are positioned
        //within the given range surrouning the given item
    }

    public void changePos(ItemState item, Position newPos) {                                    (8)
        //this method changes the position in the state of the environment
    }
```

Code 16 The state GridState part 2

```
    public HoldRelation agentHoldsPacket(AgentState agent) {                                    (9)
        for (HoldRelation rel : getRelations(HoldRelation.class))
           if (rel.containsItem(agent))
              return rel;
           return null;
    }

    public Position getPositionInDirection(Direction direction, Position pos)
                                                        throws OutOfGridException {
       //this method returns the position in the grid that is positioned
       //on distance one in the given direction of the given position.
    }

    public Position getPosition(int x, int y) {
        //this method returns the position in the grid with <x> as x-coordinate and
        //<y> as y-coordinate
    }

    public int distance(Position pos1, Position pos2) {
        //this method returns the distance between <pos1> and <pos2>
    }


    ...

}
```

The GridState is the state of the packetworld. To manipulate the state and its items in an efficient way, the state is divided as a grid of positions. Each item is positioned on one position in the grid. The position has a reference to the item and the item has a reference back to the position. In this example, we show the methods that have to be implemented plus some extra methods that are added because the different laws need them. Not all methods are inserted, we refer the interested reader to the application code of the packetworld.

(2) The attribute *grid* represents the grid of positions of the state. The attribute *perceptionRange* represents the perception range of the agents in the packetworld.

(3) The constructor of the GridState has as parameters: *int height*, *int width*, and *int percepetionRange*. Height and width are respectively the height and the width of the grid. They are used to initialize the attribute *grid* with the right dimensions. The grid is filled with new positions. The parameter *perceptionRange* is used to initialize the attribute *perceptionRange.*

(4) When an item is added to the state, it is positioned on the according position in the grid (the position is linked with the item).

(5) When an item is removed from the state, it is removed from its position in the grid (the position does not contain the item anymore, they are unlinked).

(6) To return all the items in the state of the environment, all the positions of the grid are traversed. All the items present on the different positions are returned.

(7) A vector, containing all the items that are positioned on the positions within the perception range surrounding the observer, is returned

(8-9) We have added the methods that are needed by the different laws. There are methods to manipulate the state of the environment (8) and methods to inspect the state of the environment (9).

13 KnowledgeObject

Purpose

The interface KnowledgeObject represents a knowledge-object in the KnowledgeIntegration of the agent. Every element in the internal state of an agent is a knowledge-object.

Used in

Software and physical MAS

Working instructions

(1) Create a class that implements the interface KnowledgeObject.

(2) Add attributes that represent the knowledge-object and the according getters and setters.

(3) Create a constructor. You can add parameters to initialize the attributes defined in (2).

Example

Code 17 The knowledge-object EnergyOfAgent

```
public class EnergyOfAgent extends Object implements KnowledgeObject, Observer {                    (1)

    private double energyLevel ;                                                                     (2)

    public EnergyOfAgent() {                                                                         (3)
        setEnergyLevel(1) ;
    }

    public double getEnergyLevel() {                                                                 (2)
        return energyLevel;
    }
    private void setEnergyLevel(double energyLevel) {                                                (2)
        this.energyLevel = energyLevel;
    }
}
```

An example of a knowledge-object is EnergyOfAgent, that holds the energy in the battery of an agent.

(2) The attribute *energyLevel* represents the energy in the battery of the agent.

(3) This constructor initializes the attribute *energyLevel* on value 1.

14 Focus

Purpose

An agent can select foci to direct its perception, it allows him to sense the environment for only specific types of information. In a physical MAS the focus determines what sensors are activated (see recipe 18 on page 62). In a software MAS the foci influence the construction of the representation of the environment (see recipe 21 on page 66). This recipe explains how you can create the foci for the agents in your application.

Used in

Software and physical MAS

Working instructions

(1) Create a class that implements the interface FOCUS.

(2) If necessary, add the attributes that represent the focus and the according getters and setters.

(3) If you've defined attributes in (2), you can add a constructor to initialize these attributes.

Example

Code 18 The focus VISUALFOCUS

```
public class VisualFocus extends Focus {                                    (1)

    private int range ;                                                      (2)

    public VisualFocus(int range) {                                          (3)
        setRange(range);
    }

    public int getRange() {                                                  (2)
        return range;
    }
    private void setRange(int range) {
        this.range = range
    }
}
```

We give as an example the VISUALFOCUS. This focus is selected by the agents in the packetworld if they want a visual perception of the environment.

(2) The attribute *range* indicates the range in which the agent wants to perceive its environment

(3) This constructor has one parameter: *int range*. This parameter initializes the attribute *range*.

15 Filter

Purpose

The interface FILTER represents a filter of an agent. A filter removes certain elements from a percept. An agent selects the filters according to its current task, they will remove any information that's not necessary for the agent when executing it's current task.

Used in

Software and physical MAS

Working instructions

(1) Create a class that implements the interface FILTER.

(2) If necessary, add the attributes that represent the filter and the according getters and setters.

(3) If you've defined attributes in (2), you can add a constructor to initialize these attributes.

(4) Implement the method *public void filterPercept(Percept percept)*: this method removes all elements that do not match the conditions of the filter from the percept.

Example

Code 19 The filter ONLYPACKETSFILTER

```
public class OnlyPacketsFilter implements Filter {                                    (1)

    public void filterPercept(Percept percept) {                                      (4)
        GridStatePercept gPercept = (GridStatePercept) percept ;
        for(ItemRepresentation item : gPercept.getItems())
           if (! (item instanceof PacketRepresentation))
              gPercept.removeItem(item);
    }
}
```

The ONLYPACKETSFILTER is used in the implementation of the packetworld when the agent only wants to perceive the packets in his neighbourhood and is not interested in the other items.

(4) This method removes all items except packets from the percept.

16 Representation

Purpose

The interface REPRESENTATION represents a representation of the environment. A representation is for example an image or a sound recording. This hot spot only needs to be implemented in a physical MAS. Recipe 17 on the next page explains how to create a representation in a software MAS.

Used in

Physical MAS

Working instructions

(1) Create a class that implements the interface REPRESENTATION.

(2) If necessary, add the attributes that represent the representation and the according getters and setters.

(3) If you've defined attributes in (2), you can add a constructor to initialize these attributes.

Example

Code 20 The representation ImageRepresentation

```
public class ImageRepresentation {                                          (1)

    private Image image ;                                                   (2)

    public ImageRepresentation(Image image) {                               (3)
        setImage(image) ;
    }

    public Image getImage() {                                               (2)
        return image ;
    }
    public void setImage(Image image) {                                     (2)
        this.image = image ;
    }
}
```

The representation ImageRepresentation is used by a robot that lives in a room. Every robot has a single camera that is responsable for his representation of the state of the environment. The CameraSensor (see recipe 18 on page 62) creates an ImageRepresentation when activated. This representation holds an image of the environment.

(2) The attribute *image* holds an image of the environment.

(3) This constructor has one parameter: *Image image.* This parameter initializes the attribute *image.*

17 VirtualRepresentation

Purpose

The interface VirtualRepresentation is the representation of the state of the environment. This hot spot only needs to be implemented in a Software MAS, recipe 16 on the previous page explains how to create a representation in a physical MAS.

As in physical sensing, the agents do not see the real state of the environment, only a representation of it: the agents do not see the entire topology of the state of the environment. Items that are behind a wall for example can not be seen. Of everything present in the state of the environment, the agents can only perceive a representation. It is for example possible that all the items in the environment have a certain color, but the agent can only see black-and-white. Then the representation of an item does not contain the color of that item. This recipe explains you how to create the representation of the state of your environment. Recipes 22 on page 68 and 23 on page 69 explain how to create representations that can be used in the VirtualRepresentation of the state of your application.

Used in

Software MAS

Working instructions

(1) Create a class that implements the interface VirtualRepresentation.

(2) Add the attributes that represent the VirtualRepresentation and the according getters and setters.

(3) Create a constructor. This constructor must have to following parameters: *State state, Vector<Focus> foci, AgentBehaviourFacade observer. State* is the state of the environment, *foci* is

the set of foci selected by the observer, *observer* is the facade of the behaviour of the agent for who the representation is created. This constructor can't have any other parameters.

(4) Add methods that the descriptions can use to build up the percept (see recipe 20 on page 65).

(5) Add methods that the perceptuals laws can use to modify the representation (see recipe 21 on page 66).

Example

The representation GridStateRepresentation is the representation of the state of the environment of the packetworld . It consists of a representation of the topology of the area of the grid that the agent (for who the representation is created) can perceive. It also contains the representations of all the items and relations in the area the agent can perceive.

(2) (2.1) The attribute observer holds the representation of the agent for who the representation is
(2.2) The attributes *startX*, *startY*, *endX*, *endY* represent the topology of the area of the grid the observer can perceive. *startX* and *startY* are the coordinates of the upper left position of the observable area, *endX* and *endY* are the coordinates of the lower right position.
(2.3) The attribute *items* holds a reference to a vector containing the representations of all the items that are present in the area of the grid the observer is able to perceive .
(2.4) The attribute *relations* holds a reference to a vector containing the representations of all the relations between the items in the area of the grid the observer is able to perceive.

(3) The GridStateRepresentation is a visual representation of the environment, so a VisualFocus is needed (3.1). The VisualFocus contains the range of the area the observer can perceive. The agents in the packetwold have a limited perception range, when the range in the focus is bigger then the perception range of the agent, it is reduced to that perception range (3.2). Next the GridStateRepresentation is actually created (3.3): all the attributes of the GridstateRepresentation are initialized (see 3.4).

(4) The GridstateRepresentation contains methods to inspect the representation. It contains methods to inspect the representations of the items and relations (4.1) and to inspect the representation of the topology (4.2). These methods are used by the descriptions to create the percept of the environment.

(5) There are also methods to manipulate the GridStateRepresentation. New representations of items and relations can be added (5.1) and existing ones can be removed from the GridStateRepresentation (5.2).

Code 21 The virtual representation GridStateRepresentation

```
public class GridStateRepresentation implements VirtualRepresentation {                                    (1)

    private PWAgentRepresentation observer ;                                                               (2.1)
    private int startX, endX, startY, endY;                                                                (2.2)
    private Vector<ItemRepresentation> items ;                                                             (2.3)
    private Vector<RelationRepresentation> relations ;                                                     (2.4)

    public GridStateRepresentation (State state, Vector<Focus> foci,                                        (3)
                                                        AgentBehaviourFacade observer) {
        if (foci.size() != 1 || !(foci.get(0) instanceof VisualFocus))
            throw new IllegalArgumentException("This RepresentationGenerator expects one                  (3.1)
                                                          focus of type VisualFocus !");
        VisualFocus focus = (VisualFocus) foci.get(0);
        AgentState obs = (AgentState) state.getItem(observer);
        GridState gridState = (GridState) state;
        int range;
        if (focus.getRange() >= 0 && focus.getRange()<= gridState.getPerceptionRange())                    (3.2)
            range = focus.getRange();
        else
            range = gridState.getPerceptionRange();
        createRepresentation(gridState, obs, range);                                                       (3.3)
    }

    public void createRepresentation(GridState state, AgentState observer, int range) {                    (3.4)
       //startX, startY, endX, endY: these attributes represent the
        //coordinates of the start position and and endposition of the
        //represenation, they depend on the position of the observer and the range
        //that the observer is able to see. First these are calculated and initialized.

        //the observer of this representation is set to the representation of the given observer.
       // the representations of all the items in the state in the range that the observer
       // can perceive are added to the vector with items
       // the representations of the relations between the items are added to the vector with
       // relations.
    }

    public Vector<ItemRepresentation> getItems() {                                                         (4.1)
        return items;
    }
    public Vector<RelationRepresentation> getRelations() {
        return relations;
    }
    public int getStartX() {                                                                               (4.2)
        return startX;
    }
    public int getEndX() {
        return endX;
    }
    public int getStartY() {
        return startY;
    }
    public int getEndY() {
        return endY;
    }
    public void addItemRepresentation(ItemRepresentation item) {                                           (5.1)
        items.add(item) ;
    }
    public void addRelationRepresentation(RelationRepresentation relation) {
        relations.add(relation) ;
    }
    public void removeItemRepresentation(ItemRepresentation item) {                                        (5.2)
        items.remove(item) ;
    }
    public void removeRelationRepresentation(RelationRepresentation relation) {
        relations.remove(relation) ;
    }

}
```

18 Sensor

Purpose

The interface SENSOR represents a sensor that is used in a physical MAS. The sensors create a representation of the environment (see recipe 16 on page 58). This hot spot is only implemented in a physical MAS. In a software MAS, the representation generator creates the representations of the environment

Used in

Physical MAS

Working instructions

(1) Create a class that implements the interface SENSOR.

(2) If necessary, add attributes that are used for the sensor and the according getters and setters.

(3) If you've defined attributes in (2), you can add a constructor to initialize these attributes.

(4) Implement the method *public boolean canSenseFocus(Focus focus).* This method determines whether this sensor can create a representation for a given focus.

(5) Implement the method *public Representation sense(Vector<Focus> foci).* This method creates a representation of the environment according to the given foci. The set of foci is a subset of the foci that are selected by the agent, it contains those foci that the sensor can sense.

Example

Code 22 The sensor CAMERASENSOR

```
public class CameraSensor implements Sensor {                                    (1)

    Camera camera ;                                                              (2)

    public CameraSensor(CameraSensor camera) {                                   (3)
       this.camera = camera ;
    }

    public boolean canSenseFocus(Focus focus) {                                  (4)
        return focus instanceof ColorFocus ;
    }

    public Representation sense(Vector<Focus> foci) {                            (5)
        ColorFocus cFocus = (ColorFocus) foci.get(0) ;
        if(cFocus.isBlackAndWhite())
            camera.setCaptureType(camera.BLACKANDWHITE) ;
        else
            camera.setCaptureType(camera.COLOR) ;
        return new ImageRepresentation(camera.captureImage()) ;
    }
}
```

The CAMERASENSOR is used by a robot that lives in a room. Every robot has a single camera that is responsable for his representation of the state of the environment. The CAMERASENSOR creates an IMAGEREPRESENTATION when activated (see recipe 16 on page 58).

(2) The attribute *camera* holds a reference to an object that operates the camera.

(3) This constructor has one parameter: *Camera camera.* This parameter initializes the attribute *camera.*

(4) This methods returns true when the given focus is a ColorFocus. A ColorFocus indicates whether the image of the environment should be a black-and-white image or a color image.

(5) The only foci that can be handled by the sensor CameraSensor are foci of type ColorFocus, so when the method *sense(Vector<Focus> foci)* is called, you can be sure that the given list of foci contains one focus, which is of type ColorFocus. If the focus indicates that the agent wants to receive a black-and-white image, the camera is told to capture a black-and-white image. Otherwise the camera is told to capture a color image. Finally, this method returns a new ImageRepresentation holding the captured image.

19 Percept

Purpose

The interface Percept represents the percept of the local environment for an agent. A percept describes the environment in a way that's easy to use for an agent. It is built by the descriptions of the agent (see recipe 20 on page 65).

Used in

Software and physical MAS

Working instructions

(1) Create a class that implements the interface Percept.

(2) If necessary, add attributes that represent the percept and the according getters and setters.

(3) Create a constructor. This constructor can only have one parameter: *Vector<Representation> representations*. The representations that are passed to the constructor are the representations created by the activated sensors (see recipe 18 on the previous page). This constructor can't have any other parameters. In this constructor you should create an empty percept. This constructor is called by the interpreting module. It first creates an empty percept and then the different descriptions (see recipe 20 on page 65) fill and adapt the percept by using the methods defined in (4).

(4) Add methods the agent can use to inspect the percept

(5) Add methods that can be used by the descriptions to update the percept.

Example

Code 23 The percept GridStatePercept part 1

```
public class GridStatePercept implements Percept {                                                    (1)

    private PositionRepresentation[][] grid;                                                           (2.1)
    private Vector<RelationRepresentation> relations;                                                  (2.2)

    public GridStatePercept(Vector<Representation> representations) {                                   (3)
        GridStateRepresentation repr = (GridStateRepresentation) representations.get(0);
        createGridEmptyPercept(repr.getStartX(),repr.getStartY(), repr.getEndX(), repr.getEndY()) ; (3.1)
    }

    public void createEmptyPercept(int startX, int startY, int endX, int endY) {
        //the percept must be empty, so no relations are added initially                               (3.2)
        setRelations(new Vector<RelationRepresentation>());
        //initialization of the grid:                                                                  (3.3)
        setGrid(endX - startX + 1, endY - startY + 1);
        int x = startX;
        int y = startY;
        for (int i = 0; i <= endX - startX; i++) {
            for (int j = 0; j <= endY - startY; j++) {
                getGrid()[i][j] = new PositionRepresentation(x, y);
                y++;
            }
            y = startY;
            x++;
        }
    }

    public Vector<ItemRepresentation> getItems() {                                                     (4.1)
        Vector<ItemRepresentation> items = new Vector<ItemRepresentation>();
        for (int x = 0; x < getWidth(); x++)
            for (int y = 0; y < getHeight(); y++)
                items.addAll(getGrid()[x][y].getItems());
        return items;
    }

    public Vector<RelationRepresentation> getRelations() {                                             (4.2)
        return relations;
    }

    public PositionRepresentation[][] getGrid() {                                                      (4.3)
        return grid;
    }
    private int getHeight() {
    }
    private int getWidth() {
        return getGrid().length;
    }
    private boolean isFree(int x, int y) {
       //this method checks whether the position reprsentation with coordinates x an y
       // is free
        try {
            return (!getGrid()[x][y].containsItems());
        }
        catch (ArrayIndexOutOfBoundsException e) {
            return false;
        }
    }
   ...
```

Code 24 The percept GridStatePercept part 2

```
    public void addItem(ItemRepresentation itemRep) {                                    (5.1)
        //the item represenation is added on the right position represenation in the grid
        PositionRepresentation posItem = (PositionRepresentation) itemRep.getRegion();
        getGrid()[posItem.getX()][posItem.getY()].addItem(itemRep);
    }
    public void addRelation(RelationRepresentation relationRep) {
        getRelations().add(relationRep);
    }
    public void removeItem(ItemRepresentation itemRep) {                                 (5.2)
       //the item represenation is removed from the right position represenation in the grid
        PositionRepresentation posItem = (PositionRepresentation) itemRep.getRegion();
        getGrid()[posItem.getX()][posItem.getY()].removeItem(itemRep);
    }
    public void removeRelation(RelationRepresentation relationRep) {
        getRelations().remove(relationRep);
    }

    ...
}
```

The percept GridStatePercept is the percept of the state of the environment of the packetworld . It consists of a representation of the topology of the area of the grid that the agent (for who the representation is created) can perceive. It also contains the representations of all the relations between the items present in the area the agent can perceive.

(2) (2.2) The attribute *grid* represent the topology of the area of the grid the observer can perceive. *Grid* is a two-dimensional array that holds the representations of the positions of the observable area of the grid. For each position the representation contains the representations of the items present on that position.
(2.4) The attribute *relations* holds a reference to a vector containing the representations of all the relations between the items in the area of the grid the observer is able to perceive.

(3) An empty GridStatePercept is created by using the specifications of the topology (startX, startY, endX, endY) of the first representation in the vector of representations, which is a GridStateRepresentation (3.1). The percept to be created should be empty, so the attribute relations is initialized with an empty vector (3.2). Next the attribute grid is initialized (3.3): startX,startY, endX and endY are the coordinates of the upper left and lower right position of the observable area. These are used to create the grid: it contains representations of all the positions with there coordinates between the given coordinates. The position representations do not contain any item representations yet.

(4) The GridstatePercept contains methods to inspect the representation. It contains methods to inspect the representations of the items (4.1) and relations (4.2) and to inspect the representation of the topology (4.3). For the sake of simplicity we did not show all the methods implemented, just some. We refer the interested reader to the application code of the packetworld.

(5) There are also methods to manipulate the GridStatePercept. New representations of items and relations can be added (5.1) and existing ones can be removed (5.2) from the GridStatePercept.

20 Description

Purpose

The interface Description represents a description to interprete the percept. A description extracts some kind of information from the representations and updates the percept with that information.

Used in

Software and physical MAS

Working instructions

(1) Create a class that implements the interface DESCRIPTION.

(2) If necessary, add attributes that represent the description and the according getters and setters

(3) If you've defined attributes in (2), you can add a constructor to initialize these attributes.

(4) Implement the method *public boolean canInterprete(Representation representation).* Not every description can interprete every kind of representation. This method indicates whether this description can interprete the given representation.

(5) Implement the method *public void interprete(Percept percept, Representation representation).* The first parameter of this method is the percept that's under construction. The descriptions that are executed before this description have already added some representations to the percept and the descriptions that are executed after this description will also add representations the percept. This description will update the percept according the given representation. The representation is of a type that can be interpreted by this description.

Example

An example of a description in the packetworld is ITEMDESCRIPTION. It is used to add the items that are present in a GRIDSTATEREPRESENTATION (see recipe 17 on page 59) to a GRIDSTATEPERCEPT (see recipe 19 on page 63).

(4) This methods returns true when the given representation is a GRIDSTATEREPRESENTATION, the representation of the state of the environment of the packetworld.

(5) All the items that are present in the representation are added to the percept.

Code 25 The description ITEMDESCRIPTION

```
public class ItemDescription implements Description {                                    (1)

    public boolean canInterprete(Representation representation) {                        (4)
        return representation instanceof GridStateRepresentation ;
    }

    public void interprete(Percept percept, Representation representation) {             (5)
        GridStatePercept gPercept = (GridStatePercept) percept ;
        GridStateRepresentation repr = (GridStateRepresentation) representation ;
        for(ItemRepresentation item : repr.getItems())
            gPercept.addItem(item) ;
    }
}
```

21 PerceptualLaw

Purpose

The representation of the state of the environment is composed by the representation-generator according to a set of perceptual laws. Perceptual laws are domain specific constraints on perception. They determine what the agents can observe. Whereas physical sensing naturally incorporates such constraints, in software multi-agent systems, the constraints have to be modeled explicitly. This recipe explains how to create specific perceptual laws for your application.

Used in

Software MAS

Working instructions

(1) Create a subclass of PerceptualLaw

(2) Add a constructor with as parameter *State state,* the state of the environment and pass it to the superclass.

(3) Implement the method *public VirtualRepresentation enforce(AgentState observer, VirtualRepresentation repr, Vector<Focus> foci).* This method enforces the perceptual law on the given representation. Implement the execution of the law. *Observer* is the agent that is doing perception, for him the representation is created. *Foci* is the set of foci the observer selected.

Example

Code 26 The perceptual law PerceptualLawObstacle

```
public class PerceptualLawObstacle extends PerceptualLaw {                                    (1)

    public PerceptualLawObstacle(GridState state) {                                            (2)
        super(state);
    }

    public VirtualRepresentation enforce(AgentState observer,                                  (3)
                                         VirtualRepresentation repr, Vector<Focus> foci) {
        //this law is only applicable on a GridStateRepresentation and a VisualFocus
        if (repr instanceof GridStateRepresentation && fociOk(foci)) {                         (3.1)
           //check for every item whether there is obstacle between it and the observer
           for (int i=0; i<((GridStateRepresentation)repr).getItems().size(); i++) {           (3.2)
              ItemRepresentation item = ((GridStateRepresentation)repr).getItems().get(i);
              PositionRepresentation posRep = (PositionRepresentation)item.getRegion();
              Position pos = getState().getPosition(posRep.getX(), posRep.getY());
              if (getState().obstacleBetween((Position)observer.getRegion(), pos))
                 removeItemFromRepresentation((GridStateRepresentation)repr, item);
           }
        }
        return repr;
    }

    public void removeItemFromRepresentation(GridStateRepresentation repr,                     (3.3)
                                             ItemRepresentation item) {
        //the item is removed from the representation toghether with all the
       //relations the item is involved in
    }

    private boolean fociOk(Vector<Focus> foci)  {                                              (3.4)
        //if <foci> contains a focus of type VisualFocus, this method returns true
       //if not, this method returs false
    }
}
```

An example of a perceptual law is PerceptualLawObstacle. This kind of perceptual law does not allow agents to perceive items that are situated behind another item (obstacle) from their point of view. Those items will be removed from the representation. We say that there is an obstacle between the observer and an item if both are on the same line (horizontal, vertical or diagonal) and on that line there is another item that has a smaller distance than the item with respect to the observer.

(2) The state of the environment of the packetworld is the GridState, so the constructor of the law has this state as parameter and passes it to its superclass

(3) (3.1) The law is only applicable on a GridStateRepresentation (the representation used in the packetworld, see recipe 17) and when the observer has selected the right focus (see 3.4). If this is not the case, the representation is returned as is.
(3.2) All the items for which there is an item between it and the observer, are removed from the representation (see 3.3)
(3.3) When an item has to be removed from the representation, the item itself and all the relations the item is involved in, are removed from the representation.
(3.4) This law needs a VisualFocus, so this method checks whether <foci> contains an instance of the class VisualFocus

22 ItemRepresentation

Purpose

Agents can perceive their environment. In their perception they do not see the items itself but representations of the items. For every type of item in the environment you have to create the according representation that the agents can perceive in their perception. The representation is not necessarily the same as the item itself, it is possible that the agent is not able to perceive everything of the item, or that he can see some characteristics of the item slightly different.

Used in

Software MAS

Working instructions

(1) Create a subclass of ItemRepresentation.

(2) Add attributes that represent representation of the item and the according getters and setters.

(3) Add a constructor with as parameter *RegionRepresentation region* and pass it to the superclass. *Region* is the representation of the region of the item for which this is the representation. Next to the representation of the region you can add other parameters to initialize the attributes you have defined in (2).

(4) Implement the method *public boolean equals(ItemRepresentation other)*. Return in this method whether this representation is equal to another one.

Example

PWAgentRepresentation is the representation of the external state of an agent in the packetworld (see recipe 7 on page 47).

(2) The attribute *id* represents the Id of the agent, the attribute *energyLevel* represents the energy of the battery of the agent,

(3) The constructor has the as parameters *PositionRepresentation pos*, *String id*, *int energyLevel*. *Pos* is the representation of the position (region in the state of the world) of the agent. It is passed to the constructor of the superclass. The parameter *id* is used to initialize the attribute *id* and the parameter *energyLevel* is used to initialize the attribute *energyLevel*.

(4) The method *equals()* returns true if the other representation is a also a PWAgentRepresentation. And the Id in the other representation is equals to own the own Id.

Code 27 The item representation PWAgentRepresentation

```
public class PWAgentRepresentation extends ItemRepresentation {                                    (1)

    private String id;                                                                              (2)
    private int energyLevel;
    public PWAgentRepresentation(PositionRepresentation pos, String id, int energyLevel) {         (3)
        super(pos);
        setId(id);
        setEnergyLevel(energyLevel) ;
    }

    public int getEnergyLevel() {                                                                   (2)
        return energyLevel;
    }
    private void setEnergyLevel(int energyLevel) {
        this.energyLevel = energyLevel;
    }

    public String getId() {                                                                         (2)
        return id;
    }
    private void setId(String id) {
        this.id = id;
    }

    public boolean equals(ItemRepresentation other) {                                               (3)
        if (!(other instanceof PWAgentRepresentation))
            return false;
        return (getRegion().equals(other.getRegion()) &&
                            getId().equals(((PWAgentRepresentation) other).getId()));
    }
}
```

23 RelationRepresentation

Purpose

Agents can perceive their environment. In their perception they do not see the relations itself but representations of the relations. For every type of relation in the environment you have to create the according representation that the agents can perceive in their perception. The representation is not necessarily the same as the relation itself, it is possible that the agent is not able to perceive everything of the relation, or that he can see some characteristics of the relation slightly different.

Used in

Software MAS

Working instructions

(1) Create a subclass of RelationRepresentation.

(2) Add attributes that represent representation of the item and the according getters and setters.

(3) Add a constructor with as parameters to initialize the attributes you defined in (2).

(4) Implement the method *public Vector<ItemRepresentation> getItems()*. Return a vector containing the representations of the items involved in the relation for which this is the representation.

Example

Code 28 The relation representation HoldRelationRepresentation

```
public class HoldRelationRepresentation extends RelationRepresentation {                    (1)
    private ItemRepresentation agent ;                                                       (2)
    private PacketRepresentation packet ;

    public HoldRelationRepresentation(ItemRepresentation agent, PacketRepresentation packet) {   (3)
        this.agent = agent;
        this.packet = packet;
    }
    public ItemRepresentation getAgent() {                                                   (2)
        return agent;
    }
    private void setAgent(ItemRepresentation agent) {
        this.agent = agent;
    }

    public PacketRepresentation getPacket() {                                                (2)
        return packet;
    }
    private void setPacket(PacketRepresentation packet) {
        this.packet = packet;
    }

    public Vector<ItemRepresentation> getItems() {                                           (4)
        Vector<ItemRepresentation> result = new Vector<ItemRepresentation>() ;
        result.add(getAgent()) ;
        result.add(getPacket()) ;
        return result ;
    }
}
```

HoldRelationRepresentation is the representation of the HoldRelation (see recipe 11 on page 52). The representation is slightly different from the external state itself, it does not contain the initial energylevel of the agent, and it contains the id of the agent. This representation holds the representations of the agent en the packet that are involved in the HoldRelation.

(2) The attributes *agent* and *packet* respectively hold the representations of the agent and the packet of the HoldRelation.

(3) The constructor has the a by which the attributes defined in (2) are initialized.

(4) The method *getItems()* returns a vector containing the agent and packet.

24 AgentDecision

Purpose

In every action cycle, agents have to decide what action to execute. Therefore they select an operator. The class AgentDecision is responsible for this selection. It selects the operator that corresponds to the best action. This recipe explains how you can create the decision class for an agent in your application.

Used in

Software and physical MAS

Working instructions

(1) Create a subclass of AgentDecision

(2) If necessary, add attributes that represent this class and add the according getters and setters.

(3) Create a constructor with parameters *Execution execution* and *KnowledgeIntegration knowI*. These parameters should be passed to the superclass. You can add other parameters to initialize the attributes defined in (2).

(4) Implement the method *public Operator selectOperator()*. This method returns the operator that represents the best action to be executed for the agent How to create a concrete operator is explained in recipe 34 on page 83.

(5) Implement the method *public fociSelection getFociSelection()*. This method returns the selection of foci for the agent. To implement this method you have to create a new FociSelection object and add the desired foci to this object. These foci will be used for perception in the next action cycle.

(6) Implement the method *public filtersSelection getfiltersSelection()*. This method returns the selection of filters for the agent. To implement this method you have to create a new FiltersSelection object and add the desired filters to this object. These filters will be used in the next action cycle to filter the new percept.

Example

An agent using the RandomWalkDecision class walks random through the environment.

(4) This method selects a StepOperator with a random direction. Every direction has 25% chance of being selected.

(5) This agent only want a visual representation of the packetworld, so he selects one focus, the VisualFocus.

(6) This agent selects one filter, the OnlyPacketsFilter.

Code 29 The decision class RandomWalkDecision

```
public class RandomWalkDecision extends AgentDecision {                                    (1)

    public RandomWalkDecision(Execution execution, KnowledgeIntegration knowi) {           (3)
        super(execution, knowi);
    }

    public Operator selectOperator() {                                                      (4)
        double random = Math.random()
        if(random < 0.25)
            return new StepOperator(Direction.north) ;
        else if(random < 0.5)
            return new StepOperator(Direction.east) ;
        else if(random < 0.75)
            return new StepOperator(Direction.south) ;
        else
            return new StepOperator(Direction.west) ;
    }

    public FociSelection getFociSelection() {                                           (5)
        FociSelection foci = new FociSelection() ;
        foci.addFocus(new VisualFocus()) ;
        return foci ;
    }

    public FiltersSelection getFiltersSelection() {                                     (6)
        FiltersSelection filters = new FiltersSelection() ;
        filters.addFilter(new OnlyPacketsFilter()) ;
        return filters ;
    }

}
```

25 Execution

Purpose

Execution is the execution class of an agent in a physical MAS. It receives a selected operator from the decision class of the agent and executes this operator by activating the correct effectors of the robot in the physical world. This recipe explains how you can create the execution class of an agent in a physical MAS.

Used in

Physical MAS

Working instructions

(1) Create a subclass of SoftwareExecution.

(2) If necessary, add the attributes that represent this commitment and the according getters and setters.

(3) If you've defined attributes in (2) you can add a constructor with parameters to initialize these attributes.

(4) Implement the method *public void execute(Operator operator).* This method executes the different operators by activating the different effectors of the robot.

Example

Code 30 The execution class ExampleExecution

```
public class ExampleExecution extends Execution {                                  (1)

    Motor leftMotor;                                                                (2)
    Motor rightMotor;

    public ExampleExecution(Motor leftMotor, Motor rightMotor) {                    (3)
        setRightMotor(rightMotor);
        setLeftMotor(leftMotor);
    }

    public void execute(Operator operator) {                                        (4)
        if (operator instanceof MoveForward) {
            getLeftMotor().start();
            getRightMotor().start();
            wait(500);
            getLeftMotor().stop();
            getRightMotor().stop();
        }
        else if (operator instanceof TurnLeft) {
            getRightMotor().start();
            wait(500);
            getRightMotor().stop();
        }
        else {
            // Turn right
            getLeftMotor().start();
            wait(500);
            getLeftMotor().stop();
        }
    }

    public Motor getLeftMotor() {                                                   (2)
        return leftMotor;
    }
    private void setLeftMotor(Motor leftMotor) {
        this.leftMotor = leftMotor;
    }

    public Motor getRightMotor() {                                                  (2)
        return rightMotor;
    }
    private void setRightMotor(Motor rightMotor) {
        this.rightMotor = rightMotor;
    }
}
```

The class ExampleExecution is an entirely fictitous example that steers the two motors of a robot. It can steer the robot forward, turn left or turn right.

(2) This class has two attributes : *leftMotor* and *rightMotor*. These attributes hold the two motors of this agent, the left one and right one .

(3) This constructor has two parameters to initialize the two motors.

(4) This method steers the robot according to the type of the received operator. If the received operator is a MoveForward operator, both motors are activated for half a second and the robot moves forward. If the received operator is MoveLeft operator, the right motor is activated for half a second and the robot steers left. Finally, the last option is to steer right. For this purpose the left motor is activated for half a second and the robot steers left.

26 SoftwareExecution

Purpose

SoftwareExecution is the execution class of the agents and ongoing activities in a software MAS. It receives a selected operator from the decision class and executes this operator by sending the corresponding influence to the environment.

Used in

Software MAS

Working instructions

(1) Create a subclass of SoftwareExecution.

(2) If necessary, add the attributes that represent the SoftwareExecution and the according getters and setters.

(3) Add a constructor. This constructor has two parameters: *EnvironmentFacade env* and *DynamicItemBehaviourFacade behaviour*. These parameters should be passed to the superclass. The constructor can have other parameters to initialize the attributes defined in (2).

(4) Implement the method *public Influence getInfluence(Operator op)*. This method maps an operator on the corresponding influence. Recipe 33 on page 82 explains how to create the influences for your application.

Example

Code 31 The SoftwareExecution class PWExecution

```
public class PWExecution extends SoftwareExecution{                                    (1)

    public PWExecution(EnvironmentFacade env, DynamicItemBehaviourFacade behaviour) {   (3)
         super(env, behaviour);
    }
    public Influence getInfluence(Operator op) {                                        (4)
        if (op instanceof StepOperator)
            return new StepInfluence((AgentBehaviourFacade)getBehaviour(),
                                     ((StepOperator)op).getDirection()) ;
        else if (op instanceof PickOperator)
            return new PickInfluence((AgentBehaviourFacade)getBehaviour(),
                                     ((PickOperator)op).getDirection()) ;
        else if (op instanceof DropOperator)
            return new DropInfluence((AgentBehaviourFacade) getBehaviour(),
                                     ((DropOperator)op).getDirection()) ;
        else if (op instanceof ThrowOperator)
            return new ThrowInfluence((AgentBehaviourFacade) getBehaviour(),
                                     ((ThrowOperator)op).getPosition());
        else if (op instanceof CatchOperator)
            return new CatchInfluence((AgentBehaviourFacade) getBehaviour()) ;
        else if (op instanceof ChargeOperator)
            return new ChargeInfluence((AgentBehaviourFacade)getBehaviour()) ;
        else
            throw new IllegalArgumentException("Illegal operator");
        }
}
```

The class PWExecution is the execution class of the agents and ongoing activities in the packetworld and thus matches every operator of the packetworld to the corresponding influence.

(3) This constructor just passes its parameters to its superclass

(4) This method maps all the operators in the packetworld to the corresponding influence. If necessary, the arguments of the operators are passed to the influence.

27 ConsumptionHandler

Purpose

The ConsumptionHandler of an agent handles the consumptions an agent receives from the environment. This recipe explains how you can create the ConsumptionHandler of an agent in your application.

Used in

Software MAS

Working instructions

(1) Create a subclass of ConsumptionHanlder.

(2) If necessary, add the attributes that represent the ConsumptionHandler and the according getters and setters.

(3) Add a constructor with as parameter *KnowledgeIntegration knowi.* These parameter should be passed to the superclass. The constructor can have other parameters to initialize the attributes defined in (2).

(4) Implement the method *public void handleConsumption(Consumption consumption).* This method handles a consumption that the agent receives from the environment.

Example

Code 32 The PWConsumptionHandler

```
public class PWConsumptionHandler extends ConsumptionHandler {                    (1)

    public PWConsumptionHandler(KnowledgeIntegration knowi) {                      (3)
        super(knowi);
    }

    public void handleConsumption(Consumption consumption) {                       (4)
        if (consumption instanceof ThrownPacketConsumption)
            getKnowi().getUniqueKnowledgeObject(ThrowCommitment.class).deactivate();
        else
            throw new IllegalArgumentException("The consumption can not be handled!");
   }
}
```

The PWConsumptionHandler is the consumptionHandler of the agents in the packetworld. The only consumption used in the packetworld is ThrownPacketConsumption (see recipe on page 80) that is given to the agent if he has thrown his packet to another agent.

(4) The PWConsumptionHandler can only handle a ThrowPacketConsumption. If the received consumption is such a consumption, the ThrowCommitment of the agent is deactivated.

28 OngoingActivityDecision

Purpose

In every action cycle, ongoing activities have to execute an action. Therefore they select an operator. The class OngoingActivityDecision is responsible for this selection. It selects the operator to be executed. This recipe explains how you can create the decision class for an ongoing activity in your application.

Used in

Software MAS

Working instructions

(1) Create a subclass of OngoingActivityDecision.

(2) If necessary, add the attributes that represent the OngoingActivityDecision and the according getters and setters.

(3) Add a constructor with as parameters: OngoingActivityBehaviourFacade beh, SoftwareExecution exec, and State state and pass them to the superclass. You can add extra parameters to initialize the attributes defined in (2).

(4) Implement the method *public Operator selectOperator()*. Return the operator to be executed in the state of the environment

Example

Code 33 The OngoingActivityDecision ThrownPacketDecision

```
public class ThrownPacketDecision extends OngoingActivityDecision {                    (1)

   public ThrownPacketDecision(OngoingActivityBehaviourFacade beh,                      (3)
                                        SoftwareExecution exec, State state) {
        super(beh, exec, state);
   }

   public Operator selectOperator() {                                                   (4)
       ThrownPacket me  = (ThrownPacket) getState().getItem(getBehaviour());
       int speed = me.getSpeed();
       Position newPos = null;
       int dist=0;
       while (dist < speed && ! me.getPath().isEmpty()) {
           newPos = me.getPath().removeFirst();
           dist++;
       }
       if (! me.getPath().isEmpty()) {
           //packet is not on its destination yet
           return new FlyOperator(newPos.getX(), newPos.getY());
       }
       //packet is on its destination
       return new LandOperator(newPos.getX(), newPos.getY());
   }
}
```

ThrownPacketDecision is the decision class of a thrown packet in the packetworld.

(3) A thrown packet can execute two actions: or it flies to a certain position, or it lands an a certain position. First the new position is calculated according to the speed of the thrown packet and the path it still has to traverse. If after the calculation the path to be traversed is empty,

this means the thrown packet is arrived on its destination and it may land. In that case a LandOperator is returned. If the path is not empty, the packet has to fly further through the air, so a FlyOperator is returned.

29 Effect

Purpose

Agents execute actions in the environment, the result of the actions are changes in the state of the environment. These changes in the state are modeled by the class Effect. This recipe explains how you can create the effects of the actions in your application.

Used in

Software MAS

Working instructions

(1) Create a class that implements the interface Effect.

(2) Add the attributes that represent the effect.

(3) Add a constructor with parameters to initialize the attributes you have defined in 2.

(4) Implement the method *public void execute()*. This method implements the execution of the effect. After the execution, the state of the environment will have changed.

Example

Code 34 The effect ChangePositionEffect

```
public class ChangePositionEffect implements Effect {                              (1)

    private ItemState item;                                                         (2)
    private Position newPos;
    private GridState state;

    public ChangePositionEffect(ItemState item, Position newPos, GridState state) {  (3)
        setItem(item);
        setNewPos(newPos);
        setState(state);
    }

    public void execute() {                                                         (4)
        state.changePos(item, newPos);
    }

    ...
}
```

An example of an effect in the packetworld is the ChangePositionEffect, an effect to change the position of an item in the state of the environment of the packetworld.

(2) The aim of the effect is to change the position of an item to a new one, and this has to happen by way of the state of the environment. So this are the attributes of the class. These attributes are then used when executing the effect (see 4). For the aim of simplicity we leave out the getters and setters of the different attributes of this class.

(3) In the constructor of the class all the attributes are initialized, for every attribute the constructor has a parameter to initialize it with.

(4) For this effect the execution means that the position of theitem has to be changed into a new one. There is a method *changePos(ItemState item, Position pos)* in the state of the environment that realizes this (see recipe 24 on page 71), so this method is called to execute the effect.

30 ActionLaw

Purpose

The action laws of the environment are rules that determine which effects are executed in the state of the environment and which consumptions are send to the agents as a consequence of a set of influences. They model the constraints of the domain of the application.

Used in

Software MAS

Working instructions

(1) Create a subclass of ActionLaw.

(2) Add the attributes that represent the law.

(3) Add a constructor with as parameter *SystemManager systemManager* and pass it to the superclass. You may add other parameters.

(4) Implement the method *public <T extends Influence> Vector<Class<T>> getInfluenceTypes()*. A law is applicable on a certain set of influences, in this method you should return a vector containing the classes of the influences for this law. IMPORTANT: If your law needs a certain influence multiple times, you also have to add the class multiple times. For example, if you have a law that determines that two agents clash when they step to the same place, then that law needs two time a StepInfluence and will have to return the class StepInfluence twice.

(5) Implement the method *public boolean checkConditions(Vector<Influence> infs)*. The effects of the law are only executed if certain conditions in the state of the world are met. Return a boolean indicating whether the conditions are met and the effects can be executed. You can be sure that the vector with influences contains, for each type of influence the law needs (this is defined in *getInfluenceTypes()*), just one influence.

(6) Implement the method *public Vector<Lock> getLocks()*. To make sure there are no conflicts between the laws, every law has to lock the objects it uses. Return in this method a vector containing for every object a lock. A lock holds the object that is locked and the type of lock that is present on it. To create locks you have to instantiate the class Lock with as parameters the type of lock and the object to be locked. How to create locktypes you can read in recipe 32 on page 81.

(7) Implement the method *public Vector<Effect> getEffects(Vector<Influence> influences)*. Return a vector containing the effects that will be executed in the state of the world. The vector with influences is the same vector that was passed as argument to the method *checkConditions(...)*. How to create an effect, you can read in recipe 29 on the preceding page.

(8) Implement the method *public Vector<Consumption> getConsumptions(Vector<Influence> infs)*. Return a vector containing the consumptions for the agents that are a result of the execution of the law. The vector with influences is the same vector that was passed as argument to the method *checkConditions(...)*. How to create a consumption, you can read in recipe 31 on page 80.

Example

Code 35 The law StepLaw part 1

```
public class StepLaw extends ActionLaw {                                                    (1)

    private PWAgentState agent;                                                              (2)
    private Position pos;

    public StepLaw(SystemManager systemManager) {                                            (3)
        super(systemManager);
    }

    protected GridState getState(){
        return (GridState) getSystemManager().getState();
    }

    public <T extends Influence> Vector<Class<T>> getInfluenceTypes() {                      (4)
        Vector<Class<T>> infs = new Vector<Class<T>>();
        infs.add((Class<T>)StepInfluence.class);
        return infs;
    }

    public boolean checkConditions(Vector<Influence> infs) {                                 (5)
        StepInfluence inf = (StepInfluence)infs.elementAt(0);
        agent = (PWAgentState)getState().getItem(inf.getItemBehaviour());
        Direction direction = inf.getDirection();
        try {
           pos = getState().getPositionInDirection(direction, (Position)agent.getRegion());
        }
        //you can only step in the grid and not outside it
        catch (OutOfGridException e) {
           return false;
        }
        //the place where you want to step to, may not contain an item yet,
        //except a thrown packet
        if (pos.containsItems()) {
           Vector<ItemState> itemsOnPos = pos.getItems();
           for (ItemState item : itemsOnPos) {
              if (! (item instanceof ThrownPacket)) {
                 return false;
              }
           }
         }
         return true;
    }

    public Vector<Lock> getLocks() {                                                         (6)
        ExclusiveLockType lock = new ExclusiveLockType();
        Vector<Lock> toLock = new Vector<Lock>();
        toLock.add(new Lock(lock, pos));
        return toLock;
    }
```

Code 36 The law StepLaw part 2

```
    public Vector<Effect> getEffects(Vector<Influence> influences) {                          (7)
        Vector<Effect> effects = new Vector<Effect>();
        //the position of the packetWorldAgent is changed
        effects.add(new ChangePositionEffect(agent, pos, getState()));                        (7.1)

        //if the packetWorldAgent holds a packet, the position of the packet
        //has to be changed too, and the energy decrease of the agent is a big one
        HoldRelation hold = getState().agentHoldsPacket(agent);
        if (hold != null) {
           effects.add(new ChangePositionEffect(hold.getPacket(), pos, getState()));          (7.2)
           effects.add(new ChangeEnergyEffect(agent,-PWAgentState.bigEnergyDecrease)) ;       (7.3)
        }
         //the energy decrease of the agent is a normal one
        else
            effects.add(new ChangeEnergyEffect(agent,-PWAgentState.energyDecrease)) ;         (7.3)
        return effects;
    }

    public Vector<Consumption> getConsumptions(Vector<Influence> infs) {                      (8)
        return new Vector<Consumption>();
    }
}
```

An example of an action law is StepLaw, a law that controls the movement of the agents in the packetworld.

(2) The agent that wants to make the step and the position the agent walks to, are the attributes of this class. These attributes are added because of efficiency reasons. They are needed multiple times, spread over the different methods (for example *checkConditions(...)*, *getEffects(...)* and *getConsumptions(...)*). By using attributes for them, they have to be calculated only once.

(4) This law needs only one influence, a StepInfluence. Therefore a vector containing the StepInfluence-class is returned.

(5) In this method we determine under which conditions an agent can make a step. There are two conditions. The position the agent wants to step to must be in the grid and it may not contain any items yet, except a flying packet. If these conditions are fulfilled for the incoming influence, true is returned. If not, false is returned.

(6) The law needs an exclusive lock on the position the agent wants to step to. This is necessary because when the law does not lock the position, another law can for example also put an agent on the position. This leads to a conflict situation, there may never be two agents on the same position.

(7) There are three effects that are executed in the state of the environment when the conditions defined in 5 are fulfilled: the position of the agent is changed to the new one (7.1), if the agent is holding a packet, the position of this packet is also changed to the position the agents has walked to (7.2), the energy of the agent is decreased (7.3).

(8) There are no consumptions that have to be send to the agent, so an empty vector is returned in this method.

31 Consumption

Purpose

A consumption is an effect from the environment reserved for a particular agent. Such consumption is a result from the reaction of the environment to the most recently produced influences for that agent. This recipe explains how to create the consumptions for your application.

Used in

Software MAS.

Working instructions

(1) Create a class that implements the interface CONSUMPTION.

(2) If necessary add the attributes that represent the consumption and the according getters and setters.

(3) If you defined attributes in (2), you can add a constructor with parameters to initialize those attributes.

(4) Implement the method *public AgentBehaviourFacade getAgentBehaviour()*. Return the facade of the behaviour of the agent the consumption is intended for.

Example

Code 37 The consumption THROWNPACKETCONSUMPTION

```
public class ThrownPacketConsumption implements Consumption {                    (1)

    private AgentBehaviourFacade agent;                                          (2)

    public ThrownPacketConsumption(AgentBehaviourFacade agent) {                 (3)
        setAgentBeaviour(agent);
    }

    public AgentBehaviourFacade getAgentBehaviour() {                            (4)
        return agent;
    }
    public void setAgentBehaviour(AgentBehaviour agent) {                        (2)
      this.agent = agent;
    }
}
```

An example of a consumption is a THROWNPACKETCONSUMPTION. After the successful execution of the THROWLAW (The law that determines how an agent can throw a packet), this consumption is send to the agent that has thrown the packet.

(2) To be able to return the facade of the behaviour of the agent the consumption is intended for, it is added as an attribute of the consumption.

32 LockType

Purpose

The class LOCKTYPE represents the type of lock that can be set on elements in the environment. By using locks the access to an element is limited. In the framework 2 types of locks are already present, a shared locktype (SHAREDLOCKTYPE) and an exclusive one (EXCLUSIVELOCKTYPE). When an exclusive lock is present on an object, no other locks can be set anymore. When a shared lock is present, other shared locks can still be set, but no exclusive locks anymore.

You can use these types of locks or you can create your own locktypes. This recipe explains you how you can create your own locktypes.

Used in

Software MAS

Working instructions

(1) Create a class that implements the interface LockType.

(2) Implement the method *public boolean canSetLock(LockType currentLockType)*. Return a boolean to indicate whether the type of lock can be set on a given object if *currentLockType* is the type of the lock that is already present on the object.

Example

We give as example the class SharedLockType, that represents a shared lock.

Code 38 The type of lock SharedLockType

```
public class SharedLockType implements LockType {                                    (1)

    public boolean canSetLock(LockType currentLockType) {                            (2)
        if (! (currentLockType instanceof SharedLockType) ||
            ! (currentLockType instanceof ExclusiveLockType) )
           throw new IllegalArgumentException("only read and write locks may exist in
                                              the application");
        if (currentLockType instanceof ExclusiveLockType)
           return false;
        return true;
    }
}
```

(2) A shared lock can only be combined with another shared lock or with an exclusive lock. When <currentLock> is not a shared or an exclusive lock, an exception is thrown. When there is already an exclusive lock on the object, the lock can not be set anymore and this method returns false. Otherwise it can be set and true this method returns true.

33 Influence

Purpose

Agents and ongoing activities (dynamic items) execute actions in the state of the world. Because different dynamic items can produce actions at the same time, these actions can possibly interfere with each other, so the result is not always the same as what the dynamic item intended to do. This is why they have to produce influences. An influence represents the intention of a dynamic item to execute an action in the state of the world. The reification of actions as influences enables the environment to combine simultaneously performed activity in the MAS.

This recipe explains how to create the influences that the dynamic items of your application can produce.

Used in

Software MAS

Working instructions

(1) Create a subclass of Influence.

(2) Add the attributes that represent the influence and the according getters and setters.

(3) Add a constructor with as parameter DynamicItemBehaviourFacade *item* and pass it to the superclass. Next to that parameter, you can add other parameters to initialize the attributes you have defined in (2).

Example

Code 39 The influence StepInfluence

```
public class StepInfluence extends Influence {                                    (1)

    private Direction direction;                                                   (2)

    public StepInfluence(AgentBehaviourFacade agent, Direction direction) {        (3)
        super(agent);
        setDirection(direction);
    }

    public Direction getDirection() {                                              (2)
        return direction;
    }
    private void setDirection(Direction  direction) {                              (2)
        this.direction = direction;
    }
}
```

An example of an influence in the packetworld is StepInfluence, an influence that represents the intention of an agent to make a step in a certain direction.

(2) The direction in which the agent wants to make a step is added as an attribute because this characterizes the step. The according getter and setter are defined to.

(3) In the constructor of the class, the facade of the agent that produces the influence is added as a parameter, together with the direction. The direction is initialized, the facade is passed to the superclass.

34 Operator

Purpose

An operator represents an action that a dynamic item can perform in the environment. This recipe explains how you can create the operators of your application.

Used in

Software MAS

Working instructions

(1) Create a class that implements the interface Operator.

(2) Add the attributes that represent the operator and the according getters and setters.

(3) Add a constructor with as parameter *DynamicItemBehaviourFacade item* and pass it to the superclass. Next to that parameter, you can add other parameters to initialize the attributes you have defined in (2).

Example

Code 40 The operator StepOperator

```
public class StepOperator extends Operator {                                    (1)

    private Direction direction;                                                (2)

    public StepOperator(Direction direction) {                                  (3)
        setDirection(direction);
    }

    public Direction getDirection() {                                           (2)
        return direction;
    }
    private void setDirection(Direction  direction) {                           (2)
        this.direction = direction;
    }
}
```

An example of an operator in the packetworld is StepOperator, an operator that represents the action of an agent making a step.

(2) The direction in which the agent wants to make a step is added as an attribute because this characterizes the step. The according getter and setter are defined to.

(3) The constructor of the class has as parameter the direction the agent wants to step in. The attribute <direction> is initialized with this parameter.

35 FreeflowDecision

FreeFlowDecision is used to select an operator using the free-flow architecture. This class feeds the activity through the free-flow tree and selects the action node with the highest activity. This action node returns the operator to be executed. This recipe explains how you can create a free-flow decision class for your application.

Used in

Software and physical MAS

Working instructions

(1) Create a subclass of FreeFlowDecision

(2) Add the attributes that represent this class and the according getters and setters.

(3) Create a constructor with parameters *Execution execution*, *KnowledgeIntegration knowI*. These parameters should be passed to the superclass. You can add other parameters to initialize the attributes defined in (2).

(4) Implement the method *public FreeFlowTree createFreeFlowTree*. This method returns the free-flow tree that is used by this class to select an operator

(5) Implement the method *public fociSelection getFociSelection()*. This method returns the selection of foci for this agent. To implement this method you have to create a new FociSelection object and add the desired foci to this object. These foci will be used for perception in the next action cycle.

(6) Implement the method *public filtersSelection getfiltersSelection()*. This method returns the selection of filters for this agent. To implement this method you have to create a new FiltersSelection object and add the desired filters to this object. These filters will be used in the next action cycle to filter the new percept.

Example

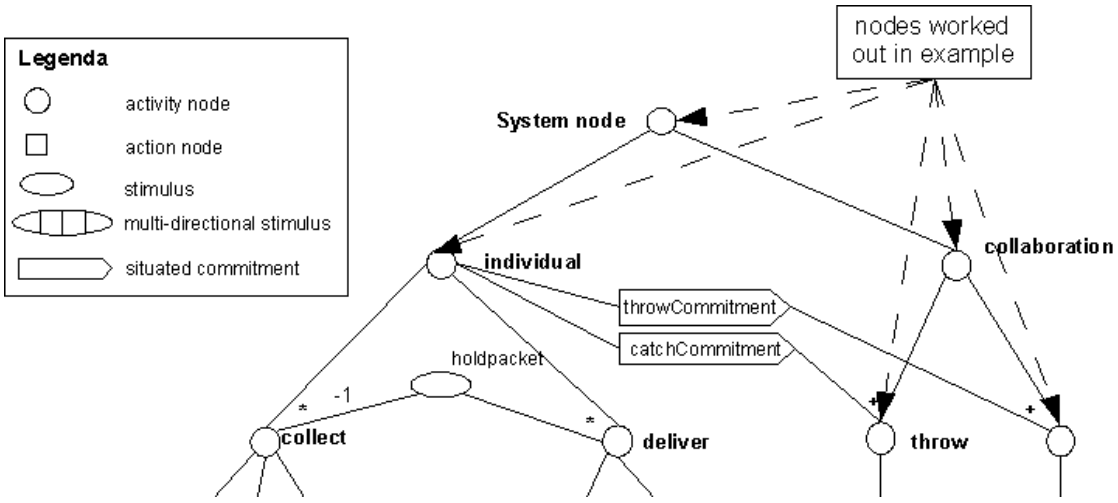
Code 41 The FreeFlowDecision class PWFreeFlow

```
public class PWFreeFlow extends FreeflowDecision {                                          (1)

    public PWFreeFlow(Execution execution, KnowledgeIntegration knowi) {                    (3)
        super(execution, knowi);
    }

    public FreeflowTree createFreeFlowTree() {                                              (4)
        // Create all activity nodes, stimuli, action nodes and commitments
        ActivityNode systemNode
                    = new ActivityNode("root", new ConstValue(new DoubleActivity(1.0)));
        ActivityNode indiv = new ActivityNode("individual", new DoubleSum());
        ActivityNode collab = new ActivityNode("collaboration", new DoubleSum());
        ActivityNode _catch = new ActivityNode("catch", new DoubleSum());
        Vector<ActivityNode> sourceRoles = new Vector<ActivityNode>();
        sourceRoles.add(indiv);
        CatchCommitment catchCommitment = new CatchCommitment(_catch, sourceRoles);
       ....
        // Link all elements of the tree defined above
        new DirectedLink(systemNode, indiv);
        new DirectedLink(systemNode, collab);
        new DirectedLink(collab, _catch);
        new DirectedLink(indiv, catchCommitment);
        new DirectedLink(catchCommitment, _catch);
        ....
    }

    public FociSelection getFociSelection() {                                               (5)
        GridStatePercept percept = getKnowledgeIntegration()
                                   .getUniqueKnowledgeObject(GridStatePercept.class);
        FociSelection foci = new FociSelection();
        if (percept.agentCarriesPacket(percept.getObserver()))
            foci.addFocus(new VisualFocus(1));
        else
            foci.addFocus(new VisualFocus());
        return foci;
    }

    public FiltersSelection getFiltersSelection() {                                         (6)
        GridStatePercept percept = getKnowledgeIntegration()
                                    .getUniqueKnowledgeObject(GridStatePercept.class);
        FiltersSelection filters = new FiltersSelection();
        if (!percept.agentCarriesPacket(percept.getObserver()))
            filters.addFilter(new RemoveGradientsFilter());
        return filters;
    }
}
```

The class PWFreeFlow is used to select the operator for the agents in the packetworld.

(3) This constructor passes the parameters *Execution execution* and *KnowledgeIntegration knowI* to the superclass.

(4) This method creates the free-flow tree. Because the tree is very elaborate we only show part of the tree. First we initialize alle the elements of the tree and then we link them together. For more information on creating the various elements of a tree, see recipes 39 on page 89, 37 on page 87, 38 on page 88, 40 on page 90 and 41 on page 92.



(5) If the agent holds a packet it only needs the gradients of the surrounding positions to find its way to the destination, so when the agent holds a packet, a VisualFocus with range 1 is selected. Otherwise a VisualFocus with standard range is selected.

(6) If this agent doesn't carry a packet, it select the RemoveGradientsFilter because it doesn't need the gradients from the destinations.

36 Activity

Purpose

The class Activity represents the activity in a free-flow tree. The framework offers DoubleActivity, but you can define other types of activities if desired.

Used in

Software and physical MAS

Working instructions

(1) Create a subclass of Activity.

(2) Add the attributes that represent the activity and the according getters and setters.

(3) If you've defined attributes in (2) you can add a constructor with parameters to initialize these attributes.

(4) Implement the method *public boolean greaterThen(Activity other)*. This method compares the activity with another activity and returns true if this activity has a higher value.

Example

Code 42 The activity VectorActivity

```
public class VectorActivity extends Activity {                                          (1)

    private Vector<Double> values;                                                      (2)

    public VectorActivity(int size) {                                                   (3)
        this.values = new Vector<Double>(size);
        for (int i = 0; i < size; i++) {
            values.add(new Double(0));
        }
    }

    public double getMaxValue() {
        ... //return the maximum of the values in the vector
    }

    public boolean greaterThen(Activity other) {                                        (4)
        if (!(other instanceof DoubleActivity || other instanceof VectorActivity))
            throw new IllegalArgumentException("activities can not be compared");
        if (other instanceof DoubleActivity)
            return getMaxValue() > ((DoubleActivity) other).getValue();
        else
            return getMaxValue() > ((VectorActivity) other).getMaxValue();
    }

    public Vector<Double> getValues() {                                                 (2)
        return values;
    }
    private void setValues(Vector<Double> values) {                                     (2)
        this.values = values ;
    }
}
```

An example of a custom type of activity is VectorActivity. This activity holds a list of values. Every value expresses the tendency of an agent to move in a certain direction. The first value holds the tendency to go north, the second value the tendency to go south, ...

(2) This class has one attribute: *Vector<Double> values*. This attribute holds the list of values.

(3) This constructor initializes every value of a list of values with given size, with value 0.

(4) This method compares this activity with other activities. If the other activity is an activity of type VectorActivity, the maximum values of both activities are compared. If the activity is an activity of type DoubleActivity, the given activity and the maximum value of this activity are compared.

37 Stimulus

Purpose

The class Stimulus represents the stimulus of a free-flow tree. The stimuli adds a certain amount of activity in the activity nodes. The amount of activity depends on conditions internal to the agent.

Used in

Software and physical MAS

Working instructions

(1) Create a subclass of Stimulus.

(2) Add the attributes that represent the stimulus and the according getters and setters.

(3) Add a constructor. This constructor can have parameters to initialize the attributes defined in (2) and passes the name of this stimulus to his superclass.

(4) Implement the method *public Activity calcActivity(KnowledgeIntegration knowI)*. This method calculates the activity of the stimulus. This activity depends on the internal state of the agent.

Example

Code 43 The stimulus HoldingPacketStimulus

```
public class HoldingPacketStimulus extends Stimulus {                                    (1)

   public HoldingPacketStimulus() {                                                       (3)
       super("holdingPacket") ;
   }

   public Activity calcActivity(KnowledgeIntegration knowI) {                             (4)
        GridStatePercept percept = knowI.getUniqueKnowledgeObject(GridStatePercept.class);
        if (percept.agentCarriesPacket(percept.getObserver()))
             return new DoubleActivity(1);
        else
             return new DoubleActivity(0);
   }
}
```

An example of a stimulus used in the packetworld is the HoldingPacketStimulus.

(3) This constructor passes the name of this stimulus (holdingPacket) to the superclass

(4) This method returns a DoubleActivity with value 1 if the agent holds a packet, and with value 0 otherwise.

38 ActionNode

Purpose

The class ActionNode represents an action node of a free-flow tree. If an action node has the maximum activity from all the action nodes, it selects the operator for the agent.

Used in

Software and physical MAS

Working instructions

(1) Create a subclass of ActionNode.

(2) Add the attributes that represent this node and the according getters and setters.

(3) Add a constructor. This constructor can have parameters to initialize the attributes defined in (2) and passes the name of this action node and the combination function to its superclass

(4) Implement the method *public Operator getOperator()*. This method returns the operator that is executed by the agent if this action node has the maximum activity

Example

Code 44 The action node CatchPacket

```
public class CatchPacket extends ActionNode {                                    (1)

    public CatchPacket() {                                                       (3)
        super("catchPacket", new DoubleSum())
    }

    public Operator getOperator() {                                              (4)
        return new CatchOperator();
    }
}
```

The CatchPacket action node is used in the packetworld and represents the action "catch a packet".

(3) This constructor passes the name (*catchPacket*) and the combination function (DoubleSum) to the superclass.

(4) This operator that is executed when this action node is executed is a CatchOperator.

39 CombinationFunction

Purpose

The interface CombinationFunction represents a combination function in the free-flow architecture. A combination function combines the activities of different stimuli and activity nodes into one activity

Used in

Software and physical MAS

Working instructions

(1) Create a class that implements the interface CombinationFunction

(2) Add the attributes that represent this function and the according getters and setters.

(3) If you've defined attributes in (2) you can add a constructor with parameters to initialize these attributes.

(4) Implement the method *public Activity calculateActivation(Node node).* This method calculates the activation of a given node. This is done by retrieving the activities of the parents of this node and combining them.

Example

Code 45 The combination function DoubleMultiply

```
public class DoubleMultiply implements CombinationFunction {                                    (1)

    public Activity calculateActivation(Node node) {                                             (4)
        if(node.getLinksToParents().size()==0)
            throw new IllegalArgumentException("Can't calculate activity if no parent!") ;
        Vector<DirectedLink> linksToParents = node.getLinksToParents() ;
        double val  = ((DoubleActivity) linksToParents.get(0).getParent().getActivity()).getValue() ;

        for(int i=1 ; i<linksToParents.size() ; i++) {
            val *= ((DoubleActivity) linksToParents.get(i).getParent().getActivity()).getValue()
                * linksToParents.get(i).getWeight() ;
        }
        return new DoubleActivity(val) ;
    }
}
```

The combination function DoubleMultiply combines two ore more activities of type DoubleActivity

(4) This method returns the activity for a given node: For the first parent the activity of that parent is retrieved and multiplied by the weight of the link to that parent. This value is multiplied by the activity of the second parent times the weight of the link, and this value is again multiplied ... The calculated activity is returned.

40 IndividualFFCommitment

Purpose

IndividualFFCommitment represents an individual commitment that is used in the free-flow architecture. This commitment links a non-empty set of source roles with a goal role, and if it's activated injects an extra amount of activity in the goal role.

Used in

Software and physical MAS

Working instructions

(1) Create a class extending IndividualFFCommitment.

(2) Add the attributes that represent this commitment and the according getters and setters.

(3) Add a constructor. This constructor can have parameters to initialize the attributes defined in (2) and passes the name of this commitment, the goal and the source roles to the superclass.

(4) Implement the method *protected boolean activationCondition(KnowledgeIntegration knowI)*. This method checks whether the activation conditions of the commitment hold. If the conditions hold, it returns true, otherwise it returns false.

(5) Implement the method *protected boolean deactivationCondition(KnowledgeIntegration knowI)*. This method checks whether the deactivation conditions of the commitment hold. If the conditions hold, it returns true, otherwise it returns false.

(6) Implement the method *public Activity calcActivity(KnowledgeIntegration knowI)*. This method calculates the amount of activity this commitment adds to its goal goal. This activity is typically bigger when the commitment is activated then when it's deactivated.

Example

Code 46 The individual commitment ChargeCommitment

```
public class ChargeCommitment extends IndividualFFCommitment {                                    (1)

    private int toChargeThreshold;                                                                 (2)
    private int chargedThreshold;

    public ChargeCommitment(ActivityNode goalRole, Vector<ActivityNode> sourceRoles,
                                            int toChargeThreshold, int chargedThreshold) {         (3)
        super("charge", goalRole, sourceRoles);
        this.toChargeThreshold = toChargeThreshold;
        this.chargedThreshold = chargedThreshold;
    }

    protected boolean activationCondition(KnowledgeIntegration knowi) {                            (4)
        if (knowi.getUniqueKnowledgeObject(GridStatePercept.class).getObserver()
                                                .getEnergyLevel() < getToChargeThreshold())
            return true;
        else
            return false;
    }

    protected boolean deactivationCondition(KnowledgeIntegration knowi) {                          (5)
        if (knowi.getUniqueKnowledgeObject(GridStatePercept.class).getObserver()
                                                .getEnergyLevel() >= getChargedThreshold())
            return true;
        else
            return false;
    }

    public Activity calcActivity(KnowledgeIntegration knowi) {                                     (6)
        if (isActivated()) {
            return new DoubleActivity(1);
        } else
            return new DoubleActivity(0);
    }

    private int getToChargeThreshold() {                                                           (2)
        return toChargeThreshold;
    }

    private int getChargedThreshold() {                                                            (2)
        return chargedThreshold;
    }
}
```

The commitment ChargeCommitment makes sure the agent doesn't runs out of energy : the commitment is activated when the energy falls below a certain threshold. The commitment is deactivated when the battery is fully charged.

(2) This commitment has two attributes: *toChargeThreshold* and *chargedThreshold*. These attributes are respectively the threshold to start charging the battery and to stop charging.

(3) This constructor has four parameters: *ActivityNode goalRole*, *Vector<ActivityNode> sourceRoles*, *int toChargeThreshold* and *int chargedThreshold*. The first two parameters are passed to the superclass, together with the name of this commitment ("charge"). The other parameters are used to initialize the attributes.

(4) This method checks the activation condition of this commitment. These conditions hold if the energylevel of the agent falls below *toChargeThreshold*.

(5) This method checks the deactivation condition of this commitment. These conditions hold if the energylevel of the agent falls below *toChargeThreshold*.

(6) This method returns the amount of activity this commitment adds to its goal goal. If the commitment is activated it adds a DoubleActivity with amount 1, otherwise a DoubleActivity with value 0.

41 MutualFFCommitment

Purpose

MutualFFCommitment represents a mutual commitment that is used in the free-flow architecture. This commitment links a non-empty set of source roles with a goal role, and if it's activated injects an extra amount of activity in the goal role.

Used in

Software and physical MAS

Working instructions

(1) Create a class extending MutualFFCommitment.

(2) Add the attributes that represent this commitment and the according getters and setters.

(3) Add a constructor. This constructor can have parameters to initialize the attributes defined in (2) and passes the name of this commitment, the goal and the source roles to the superclass.

(4) Implement the method *protected boolean deactivationCondition(KnowledgeIntegration knowI)*. This method checks whether the deactivation conditions of the commitment hold. If the conditions hold, it returns true, otherwise it returns false.

(5) Implement the method *public Activity calcActivity(KnowledgeIntegration knowI)*. This method calculates the activity this commitment adds to its goal goal. This activity is typically bigger when the commitment is activated then when it's deactivated.

Example

Code 47 The individual commitment ChargeCommitment

```
public class CatchCommitment extends MutualFFCommitment {                                    (1)

    public CatchCommitment(Role goalRole, Vector<? extends Role> sourceRoles) {              (3)
        super("acceptCommitment", goalRole, sourceRoles);
    }

    protected boolean deactivationCondition(KnowledgeIntegration knowi) {                    (4)
        GridStatePercept percept = knowi.getUniqueKnowledgeObject(GridStatePercept.class);
        Vector<ThrownPacketRepresentation> thrownPackets =
                                percept.getItemsOfClass(ThrownPacketRepresentation.class);
        if (thrownPackets.isEmpty()) {
            return true;
        }
        return false;
    }

    public Activity calcActivity(KnowledgeIntegration knowi) {                               (5)
        if (isActivated())
            return new DoubleActivity(1);
        else
            return new DoubleActivity(0);
    }
}
```

The commitment ChargeCommitment is activated by communication. It is used to throw a packet from one agent to another.

(3) This constructor has two parameters: *ActivityNode goalRole* and *Vector<ActivityNode> sourceRoles*. These parameters are passed to the superclass, together with the name of this commitment ("catchCommitent").

(4) This method checks the deactivation condition of this commitment. These conditions hold if the agent cannot perceive a thrown packet, indicating that the thrown packet landed somewhere.

(5) This commitment adds a DoubleActivity with amount 1 if the commitment is activated, and a DoubleActivity with amount 0 if it's deactivated.

42 Protocol

Purpose

Communication in this framework is specified in terms of communication protocols. A protocol defines the sequence of messages that can be send during a conversation between agents. A protocol consists of different protocol steps. There are different types of protocol steps: ExoInitiation (recipe 43 on the next page), EndoInitiation (recipe 44 on page 96), MessageStep (recipe 45 on page 98), Continue (recipe 46 on page 99), ExoTerminate (recipe 47 on page 100), EndoTerminate (recipe 48 on page 101) and to create a TimeOutStep (recipe 49 on page 102). You should first create the different protocol steps and then add them to the communication protocol.

Used in

Software and physical MAS.

Working instructions

(1) Instantiate a new object of the class Protocol. The constructor has one argument: *String name*. This argument represents the name of this protocol.

(2) Add all the protocol steps of this protocol to the created object

Example

Code 48 The protocol ThrowPacketProtocol

```
Protocol protocol = new Protocol("ThrowPacketProtocol");                    (1)
protocol.addStep(new InitiateThrowPacketProtocol()) ;                       (2)
protocol.addStep(new HandleRequestThrow());                                 (2)
protocol.addStep(new HandleAcceptThrowPacket());                            (2)
protocol.addStep(new HandleRejectThrowPacket());                            (2)
protocol.addStep(new TerminateConversation()) ;                             (2)
```

As an example we give the protocol ThrowPacketProtocol (see fig. 6.3). This protocol is used between two agents to reach an agreement about throwing a packet.

(1) This creates *protocol*, a new object of the class Protocol with name "ThrowPacketProtocol".

(2) These lines add the different protocol steps of this protocol.
InitiateThrowPacketProtocol starts the conversation and sends a "request"-message to the Thrower (see recipe 44 on page 96).
HandleRequestThrow handles this request and sends an "accept"-message or "reject"-message to the Catcher (see recipe 43 on the next page).
HandleAcceptThrow handles the arrival of the "accept"-message at the first agent and terminates the conversation at the Catcher (see recipe 47 on page 100).
HandleRejectThrow handles the arrival of the "reject"-message at the first agent and terminates the conversation at the Catcher.
TerminateConversation terminates the conversation at the Thrower (also see recipe 48 on page 101).

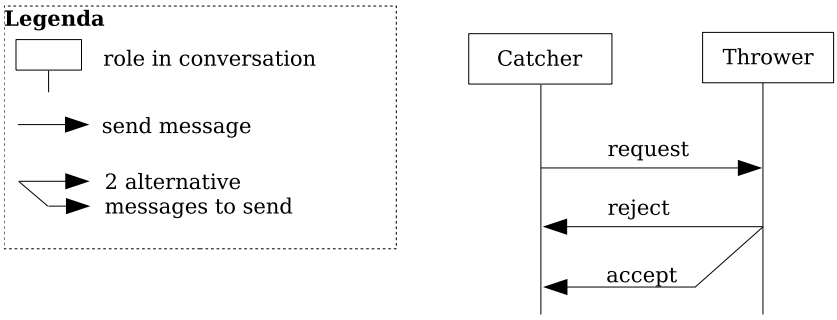


Figure 6.3: The ThrowPacketProtocol

43 ExoInitiation

Purpose

The class ExoInitiation represents a protocol step that starts a conversation when a certain message is received. This message must have the same performative as this step, and the conversation the message belongs to must use the protocol of this step.

This step is part of a protocol, see recipe 42 on the preceding page for more information on how to add this protocol step to a protocol.

Used in

Software and physical MAS.

Working instructions

(1) Create a subclass of ExoInitiation

(2) If necessary, add attributes that represent the protocol step and add according getters and setters.

(3) Create a constructor where you can add parameters to initialize the attributes defined in (2). Inside this constructor, you have to pass the performative of this protocol step to the superclass (using the method *super(String performative)*).

(4) Implement the method *public void execute(AgentMessage firstMessageOfConversation, KnowledgeIntegration knowi, CommunicatingInterface commI)*. This method is executed when a message that has the same performative as this step is received. The message must be part of a conversation using the protocol of this step and that conversation may not yet be registered as an active conversation of this agent.

Example

As an example we create the protocol step HandleRequestThrow. This step is activated when an agent receives the first message, witch has performative "request". If the conditions to accept the cooperation are satisfied it sends a message with performative "accept", otherwise it will send a message with performative "reject".

(3) Create a constructor with one argument, the performative, and passes it to the superclass.

(4) The method *execute(...)* executes this protocol step. This protocol step first retrieves the ThrowCommitment (4.1) and the most recent GridStatePercept (4.2) from the KnowledgeIntegration. If the agent holds a packet and the ThrowCommitment is not activated it retrieves

the representation of the packet from the percept (4.3) and searches for a destination in the percept with the same color as the packet (4.5-4.8). If it doesn't find such a destination it activates the commitment (4.9), adds the sender of the message to the relation set (4.10), replies an "accept"-message (4.11) and ends this method (4.12).
The agent sends a "reject"-message (with no content) if one of the above conditions failed (4.13).

Code 49 The ExoInitiation protocol step HandleRequestThrow

```
public class HandleRequestThrow extends ExoInitiation {                                    (1)

    public HandleRequestThrow() {                                                           (3)
        super("request");
    }

    public void execute(AgentMessage mess, KnowledgeIntegration knowI,
                                                CommunicatingInterface communicating) {     (4)
        ThrowCommitment commitment = knowI.getUniqueKnowledgeObject(ThrowCommitment.class);     (4.1)
        GridStatePercept percept = knowI.getUniqueKnowledgeObject(GridStatePercept.class);      (4.2)
        boolean holdingPacket = percept.agentCarriesPacket(percept.getObserver()) ;
        if (holdingPacket && !commitment.isActivated()) {
            PacketRepresentation packet = percept.getPacketCarriedByAgent(percept.getObserver()) ;  (4.3)
            boolean seeDestinationOfRightColor = false;                                     (4.4)
            Vector<DestinationRepresentation> dests =
                                        percept.getItemsOfClass(DestinationRepresentation.class); (4.5)
            for (DestinationRepresentation dest : dests)                                    (4.6)
                if (dest.getColor().equals(packet.getColor()))                              (4.7)
                    seeDestinationOfRightColor = true;                                      (4.8)
            if (!seeDestinationOfRightColor) {
                commitment.activate();                                                      (4.9)
                commitment.addAgentToRelationsSet(mess.getSenderID());                     (4.10)
                communicating.sendMessage(mess.createReply("accept", null));               (4.11)
                return;                                                                     (4.12)
            }
        }
        communicating.sendMessage(mess.createReply("reject, null));                        (4.13)
    }
}
```

44 EndoInitiation

Purpose

The class EndoInitiation represents a protocol step that starts a conversation when certain conditions internal to the agent hold. Every communication cycle these conditions will be checked and if they hold the step will be executed.

This step is part of a protocol, see recipe 42 on page 94 for more information on how to add this protocol step to a protocol.

Used in

Software and physical MAS.

Working instructions

(1) Create a subclass of EndoInitation.

(2) Add attributes that represent the protocol step and add according getters and setters.

(3) If you've defined attributes in (2) you can add a constructor with parameters to initialize these attributes.

(4) Implement the method *public void checkConditions(KnowledgeIntegration knowI)*. This method checks the conditions and returns true if they hold and false otherwise.

(5) Implement the method *public Conversation getNewConversation(KnowledgeIntegration knowI)*. This method is executed when the conditions to start the conversation hold (tested in (4)) and returns the conversation that is started. See recipe 50 on page 103 for more information on creating conversations.

(6) Implement the method *public void execute(Conversation conversation, KnowledgeIntegration knowI, CommunicatingInterface interface)*. This method executes the first action of a conversation and will typically send the first message of this conversation to the responders of this conversation.

Example

Code 50 The EndoInitiation protocol step InitiateThrowPacketProtocol

```
public class InitiateThrowPacketProtocol extends EndoInitiation {                                  (1)

    PWAgentRepresentation packetHolder ;                                                            (2)

    public boolean checkConditions(KnowledgeIntegration knowI) {                                    (4)
        GridStatePercept perc = knowI.getUniqueKnowledgeObject(GridStatePercept.class);            (4.1)
        CatchCommitment commitmnt = knowI.getUniqueKnowledgeObject(CatchCommitment.class);         (4.2)
        boolean conditions = !commitmnt.isActivated();                                             (4.3)
        conditions &= !perc.getItemsOfClass(DestinationRepresentation.class).isEmpty();            (4.4)
        conditions &= !perc.agentCarriesPacket(perc.getObserver());                                (4.5)
        conditions &= knowI.getAllKnowledgeObjectsOfType(Conversation.class).isEmpty() ;           (4.6)
        if (conditions) {
            DestinationRepresentation dest =
                            perc.getItemsOfClass(DestinationRepresentation.class).get(0);          (4.7)
            for(PWAgentRepresentation agent = perc.getItemsOfClass(PWAgentRepresentation.class)) {
                boolean agentCond = !agent.equals(perc.getObserver());                             (4.8)
                agentCond &= perc.agentCarriesPacket(agent);                                       (4.9)
                agentCond &= perc.distance(destination, agent) > perc.getGrid().length / 2;       (4.10)
                if (agentCond &&
                           perc.getPacketCarriedByAgent(agent).getColor().equals(dest.getColor()) {
                    packetHolder = agent;                                                          (4.11)
                    return true ;                                                                  (4.12)
                }
            }
        }
        return false;                                                                              (4.13)
    }

    public Conversation getNewConversation(KnowledgeIntegration knowI) {                           (5)
        String myName = knowI.getUniqueKnowledgeObject(AgentName.class).getAgentName() ;
        return new BinaryConversation(getProtocol().getName(),myName, this.packetHolder.getId(), 0L);
    }

    public void execute(Conversation conversation, KnowledgeIntegration knowI,
                                                   CommunicatingInterface handler) {               (6)
        BinaryConversation conv = (BinaryConversation) conversation;
        handler.sendMessage(new AgentMessage(conv.getInitiatorName(),
                                            conv.getResponderName(), "request", conv, null));
    }
}
```

As an example we create the protocol step InitiateThrowPacketProtocol. This step is the first step of the ThrowPacketProtocol. If certain conditions hold it starts a conversation by sending a "request"-message to another agent.

(2) This protocol step has one attribute : *PWAgentRepresentation packetHolder* that's used in (4), (5) and (6). This is the agent that holds the packet (*Thrower*) and is asked to throw it to the agent that executes this step (*Catcher*)

(4) This method checks the conditions for this protocol step. First, the most recent percept and the CatchCommitment are retrieved from the KnowledgeIntegration (4.1 and 4.2). Next this method checks whether it should consider starting a new conversation: the commitment must be deactivated (4.3) and the agent must see a destination in its percept (4.4), cannot carry a packet (4.5) or have any other conversations (4.6). The first destination from the destinations of the percept is retrieved (4.7), and an agent is searched that is not the observer (4.8), carries a packet (4.9) and can't see the destination itself (4.10). If these conditions hold and the agent carries a packet from the same color as the destination, the attribute *packetHolder* is set to the agent we've found (4.11) and true is returned (4.12). Otherwise, false is returned (4.13).

(5) This method returns a new BinaryConversation with the name of the protocol, the name of this agent, the name of the agent that's holding the packet and no timeout.

(6) This method sends the first message in the conversation defined in (5) to the agent holding the packet.

45 MessageStep

Purpose

The class MessageStep represents a protocol step that is executed when the agent receives a message with the same performative as this protocol step during a conversation.

This step is part of a protocol, see recipe 42 on page 94 for more information on how to add this protocol step to a protocol.

Used in

Software and physical MAS.

Working instructions

(1) Create a subclass of MessageStep.

(2) Add attributes that represent the protocol step and add according getters and setters.

(3) Create a constructor where you can add parameters to initialize the attributes defined in (2). Inside this constructor, you have to pass the performative of this protocol step to the superclass (using the method *super(String performative)*).

(4) Implement the method public void *execute(AgentMessage message, KnowledgeIntegration knowI, CommunicatingInterface commI)*. This method executes this protocol step by reacting to the incoming message.

Example

Code 51 The protocol step ExampleMessageStep

```
public class ExampleMessageStep extends MessageStep {                                    (1)

    public ExampleMessageStep() {                                                         (3)
        super("hello");
    }

    public void execute(AgentMessage mess, KnowledgeIntegration knowI,
                                            CommunicatingInterface commI) {               (4)
        commI.sendMessage(mess.createReply("good morning")) ;
    }
}
```

In this example, we show an entirely fictitious protocol step: the ExampleMessageStep. This message step receives an incoming message with performative "hello" and replies a message with performative "good morning". This protocol step doesn't address any of the difficulties associated with checking certain conditions. For a protocol step with more realistic condition checking, see recipes 43 on page 95, 44 on page 96 and 47 on the following page.

(3) This constructor passes the performative of this message ("hello") to the superclass.

(4) This method executes this protocol step: it replies "good morning" to the received "hello"-message.

46 Continue

Purpose

The class Continue represents a protocol step that continues the conversation after a break when certain condition internal to the agent hold.

This step is part of a protocol, see recipe 42 on page 94 for more information on how to add this protocol step to a protocol.

Used in

Software and physical MAS.

Working instructions

(1) Create a subclass of Continue.

(2) Add attributes that represent the protocol step and add according getters and setters.

(3) If you've defined attributes in (2) you can add a constructor with parameters to initialize these attributes.

(4) Implement the method *public boolean checkConditions(Conversation conv, KnowledgeIntegration knowI)*. This method returns true if the conditions to execute this protocol step hold and false otherwise.

(5) Implement the method *public void execute(Conversation conv, KnowledgeIntegration knowI, CommunicatingInterface commI)*. This method executes this protocol step. One or more messages to another agent will typically be send in a this method.

Example

Code 52 The continue step ExampleContinue

```
public class ExampleContinue extends ContinueStep {                                                    (1)

    PWAgentRepresentation otherAgent;                                                                   (2)

    public boolean checkConditions(Conversation conv, KnowledgeIntegration knowI) {                     (4)
        GridStatePercept perc = knowI.getUniqueKnowledgeObject(GridStatePercept.class);
        Vector<PWAgentRepresentation> agents = perc.getItemsOfClass(PWAgentRepresentation.class);
        boolean seeOtherAgent = false;
        for (PWAgentRepresentation agent : agents) {
            if (!agent.equals(perc.getObserver())) {
                seeOtherAgent = true;
                otherAgent = agent;
                break;
            }
        }
        return seeOtherAgent;
    }

    public void execute(Conversation conv, KnowledgeIntegration knowI,                                  (5)
                                                        CommunicatingInterface commI) {
        String thisAgentID = knowI.getUniqueKnowledgeObject(AgentID.class).getAgentID() ;
        commI.sendMessage(new AgentMessage(thisAgentID,otherAgent.getId(),"hello",conv)) ;
    }
}
```

We create an entirely fictitious protocol step: the ExampleContinue. This continue step checks whether the agent perceives another agent in its percept, and when it does, it sends this agent a "hello"-message.

(2) This class has one attribute *otherAgent*. This attribute is used in (4) and (5)

(4) This method checks whether the agent perceives an agent that is not equal to this agent.

(5) This method executes this step by sending a message to the other agent with performative "hello" and no content.

47 ExoTerminate

The class ExoTerminate represents a protocol step terminates a conversation after receiving the last message of the conversation. This step is executed when a message with the same performative as this protocol step is received.

Purpose

This step is part of a protocol, see recipe 42 for more information on how to add this protocol step to a protocol.

Used in

Software and physical MAS.

Working instructions

(1) Create a subclass of ExoTerminate.

(2) Add attributes that represent the protocol step and add the according getters and setters.

(3) Create a constructor where you can add parameters to initialize the attributes defined in (2). Inside this constructor, you have to pass the performative of this protocol step to the superclass (using the method *super(String performative)*).

(4) Implement the method *public void execute(AgentMessage message, KnowledgeIntegration knowI, CommunicatingInterface commI)*: This method executes this protocol step. This method performs the last actions of this conversation. When this method is finished, the conversation is terminated and removed.

Example

Code 53 The ExoTerminate step HandleAcceptThrowPacket

```
public class HandleAcceptThrowPacket extends ExoTerminate {                                    (1)

    public HandleAcceptThrowPacket() {                                                         (3)
        super("accept");
    }

    public void execute(AgentMessage message, KnowledgeIntegration knowI,
                                                    CommunicatingInterface handler) {          (4)
        CatchCommitment commitment = knowI.getUniqueKnowledgeObject(CatchCommitment.class);
        commitment.addAgentToRelationsSet(message.getSenderID()) ;
        commitment.activate();
    }
}
```

The step HandleAcceptThrowPacket is a step used in the ThrowPacketProtocol. This step is activated when the agent in the role *Catcher* receives an "accept"-message from the agent in the role *Thrower*. It adds the sender of the message to the relation set of the CatchCommitment of the agent in the role *Catcher* and activates that commitment.

(3) This constructor passes the performative of this protocol step ("accept") to its superclass.

(4) This method retrieves the CatchCommitment of the agent from the KnowledgeIntegration, adds the sender of the message to the relation set and activates the commitment.

48 EndoTerminate

Purpose

The class EndoTerminate represents a protocol step that terminates a conversation when some conditions internal to the agent hold. The conditions for this step are checked once every communication cycle and if the conditions for this step hold, the step is executed.

This step is part of a protocol, see recipe 42 on page 94 for more information on how to add this protocol step to a protocol.

Used in

Software and physical MAS.

Working instructions

(1) Create a subclass of EndoTerminate.

(2) Add attributes that represent the protocol step and add according getters and setters.

(3) If you've defined attributes in (2) you can add a constructor with parameters to initialize these attributes.

(4) Implement the method *public boolean checkConditions(Conversation conv, KnowledgeIntegration knowI)*. This method returns true if the conditions to execute this protocol step hold and false otherwise.

(5) Implement the method *public void execute(Conversation conv, KnowledgeIntegration knowI, CommunicatingInterface commI)*. This method executes this protocol step. This method performs the last actions of the conversation. When this method is finished, the conversation is terminated and removed.

Example

Code 54 The EndoTerminate step ExampleEndoTerminate

```
public class ExampleEndoTerminate extends EndoTerminate {                                    (1)

   public boolean checkConditions(Conversation conv, KnowledgeIntegration knowI) {            (4)
       GridStatePercept perc = knowI.getUniqueKnowledgeObject(GridStatePercept.class);
       Vector<PWAgentRepresentation> agents = perc.getItemsOfClass(PWAgentRepresentation.class);
       return agents.size()>=1 ;
    }

    public void execute(Conversation conv, KnowledgeIntegration knowi,
                                                CommunicatingInterface commI) {          (5)
        System.out.println("Terminating conversation : "+conv) ;
    }
}
```

In this example, we create an entirely fictitious protocol step: the ExampleEndoTerminate. This protocol step ends the conversation when the agent doesn't see any agents in his neighbourhood (apart from itself).

This protocol step doesn't address any of the difficulties associated with checking certain conditions. For a protocol step with more realistic condition checking, see recipe 44 on page 96.

(4) This method checks the conditions for this step: it checks whether the agent perceives other agents by checking whether the number of agents in the percept is greater than 1 (the agent always perceives itself).

(5) This method warns the user the conversation is terminated. After this method is finished, the conversation is terminated and removed from the KnowledgeIntegration of this agent.

49 TimeOutStep

Purpose

The class TimeOutStep represents a protocol step that is executed when a conversation times out. When you create a conversation, you can supply a timeout. When a conversation hasn't send or received a message for a time greater than the timeout, the TimeOutStep of the protocol of that conversation is executed and the conversation is removed. If you don't supply a time out when creating a conversation the conversation will not be checked for timing out. See recipe 50 on the following page for more information on how to create a conversation.

This step is part of a protocol, see recipe 42 on page 94 for more information on how to add this protocol step to a protocol.

Used in

Software and physical MAS.

Working instructions

(1) Create a subclass of TimeOutStep.

(2) Add attributes that represent the protocol step and add according getters and setters.

(3) If you've defined attributes in (2) you can add a constructor with parameters to initialize these attributes.

(4) Implement the method *public void execute(Conversation conv, KnowledgeIntegration knowI, CommunicatingInterface commI)*: This method executes this protocol step. This method performs the last actions of this conversation. When this method is finished, the conversation is terminated and removed.

Example

Code 55 The TimeOut step ExampleTimeOutStep

```
public class ExampleTimeOutStep extends TimeOutStep {                                (1)

    public void execute(Conversation conv, KnowledgeIntegration knowI) {              (4)
        System.out.println("Conversation "+conv+" has timed out !") ;
    }
}
```

In this example, we create a timeout protocol step: the ExampleTimeOutStep. This protocol step is called when the conversation has timed out and will simply print a warning to the user.

(4) This method warns the user that the conversation has timed out.

50 Conversation

Purpose

The class Conversation represents an ongoing conversation for an agent. When an agent starts a new conversation (either in a EndoInitiation or ExoInitiation protocol step, see recipes 43 on page 95 and 44 on page 96) the conversation is registered in the KnowledgeIntegration of that agent. When the conversation is terminated, the conversation is removed from the KnowledgeIntegration.

In this recipe we'll see how to create a new type of conversation. If you're conversation is between two agents, you don't need to implement this hot spot but can use BinaryConversation, which is already provided in the framework.

Used in

Software and physical MAS.

Working instructions

(1) Create a subclass of Conversation.

(2) Add attributes that represent this conversation and add according getters and setters. A conversation is an Externalizable class, so you need to annotate every getter with *@Slot* (see recipe 51 on the following page).

(3) Create a constructor with arguments : *String protocolName, String initiatorName* and *long timeOut*. These arguments represent respectively the name of the protocol, the name of the initiator of the conversation and the time out. You can add arguments to initialize the attributes defined in (2).

(4) Create a constructor with one argument : *InternalState internalState*. This constructor is needed because a conversation is an EXTERNALIZABLE class (see recipe 51). This constructor should pass *iternalState* to its superclass, and initialize the attributes defined in (2) using *internalstate*.

Example

Code 56 The TIMEOUT step EXAMPLETIMEOUTSTEP

```
public class BinaryConversation extends Conversation implements Externalizable {                    (1)

    private String responderName;                                                                    (2)

    public BinaryConversation(String protocolName, String initiatorName,
                                        String responderName, long timeOut) {                        (3)
        super(protocolName, initiatorName, timeOut);
        setResponderName(responderName);
    }

    public BinaryConversation(InternalState internalState) throws IncorrectSlotException {           (4)
        super(internalState);
        setResponderName((String) internalState.get("respondername"));
    }

    public @Slot String getResponderName() {                                                         (2)
        return responderName;
    }

    private void setResponderName(String responderName) {                                            (2)
        this.responderName = responderName;
    }
}
```

In this example we create BINARYCONVERSATION. BINARYCONVERSATION is provided in the framework and is used to represent a conversation between two agents, the initiator and the responder.

(2) A binary conversation is a conversation between two agents. The superclass CONVERSATION already holds the name of the protocol, the name of the initiator and the timeout, so the only attribute we define in BINARYCONVERSATION is the name of the responder : *String responderName*

(3) This constructor passes the name of the protocol, the name of the initiator and the time out to the superclass and initializes the attribute *responderName*.

(4) This constructor passes the internal state to the superclass and initializes the attribute *respondername*, using the value of *internalState*.

51 Externalizable

Purpose

An AGENTMESSAGE is used as the internal representation of a message. This class is easy to handle for an agent, but it is not the best representation to send a message through the environment. To send a message through the environment, we use the class STRINGMESSAGE. The SLDECODERENCODER is responsible for transforming an AGENTMESSAGE to a STRINGMESSAGE and vice versa.

The content of an AgentMessage is an java-object of a class implementing the Externalizable interface. This interface makes it possible for the SLDecoderEncoder to create the InternalState object that contains all the values for the different slots of a class and to convert this InternalState object to a string. This string is used as the content of a StringMessage that's passed to another agent. The receiver takes the content of this message, and converts this string into an InternalState object. This InternalState object is used to reconstruct the original java-object.

Used in

Software and physical MAS.

Working instructions

(1) Create a class that implements the interface Externalizable. (This interface is provided in the framework, don't confuse this interface with the interface java.io.Externalizable!)

(2) Add attributes that represent that the class and add according getters and setters.

(3) From the attributes defined in (2), select the attributes that are the slots of this class. A slot is an attribute of a class that will be externalized. Every slot should belong to one of these types :

Primitive types Integer, int, Character, char, Boolean, boolean, Byte, byte, Short, short, Long, long, Float, float, Double, double, String

Externalizable Any class implementing the Externalizable interface

List Any class implementing the List interface. The attribute *elementType* of Slot describes the type of the elements in the list, see below.

Array Any array of the above classes

Create for every slot with name *Nnnn* a public getter of the form *getNnnn()* and annotate this method with the Slot annotation. If the slot belongs to a class implementing the List interface, you have to supply the *elementType* parameter of the slot. This parameter describes the type of the elements of the list. This type can be a primitive type or a class implementing the Externalizable interface.

(4) If you've defined attributes in (2) you can add a constructor with parameters to initialize these attributes.

(5) Create a constructor with one parameter : *InternalState internalState.* This constructor is used when reconstructing an object from a String. The constructor should retrieve the different slots from the *internalState* object (method *public Object get(String slotName)*) and assign the returned values to the correct slots.

Example

Code 57 The EXTERNALIZABLE class PERSON

```
public class Person implements Externalizable {                                    (1)

    private int age ;                                                               (2)
    private Vector<Person> children ;

    public void setAge(int age) {                                                   (2)
        this.age = age;
    }

    public void setChildren(Vector<Person> children) {
        this.children = children;
    }

     public @Slot int getAge() {                                                    (2-3)
        return age ;
    }

    public @Slot(elementType=Person.class) Vector<Person> getChildren() {           (2-3)
        return children ;
    }

    public Person(int age) {                                                        (4)
        this.age = age ;
        children = new Vector<Person>() ;
    }

    public Person(InternalState state) throws IncorrectSlotException {              (5)
        this.age = (Integer) state.get("age") ;
        this.children = (Vector<Children>) state.get("children") ;
    }

}
```

We have already seen an example of an externalizable class in recipe 50 on page 103. The fictitious example given here is the class PERSON. This class represents a person with a certain age and a collection of children.

(2) This class has attributes *int age* and *Vector<Person> children*.

(3) Every attribute of this class is a slot. We've added *@Slot* to their getters, *children* attribute is a vector that contains elements of the class PERSON, so this class is the elementType.

(4) This constructor creates a new person with given age and an empty set of children.

(5) This constructor retrieves the values from the slots from the internal state, casts them and assigns them to the correct attributes.

52 Ontology

Purpose

The ontology represent the vocabulary that the agents use to communicate with each other. Every class you want to use as the content of a message, needs to be part of the ontology of both agents.

Used in

Software and physical MAS.

Working instructions

(1) Instantiate a new object of the class ONTOLOGY.

(2) Add all the classes that are used as the content of a message. These classes must implement the EXTERNALIZABLE (recipe 51 on page 104)

Example

Code 58 The use of the class ONTOLOGY

```
Ontology onto = new Ontology();                                    (1)
onto.addToVocabularium(Person.class, "person");                    (2)
```

(1) An instance of the class ONTOLOGY is created.

(2) The class PERSON is added to the vocabularium. This class implements the EXTERNALIZABLE interface.

53 CommunicationLaw

Purpose

The communication laws of the environment enforce constraints on the communication of agents. A communication law can modify or delete a message an agent sends. This recipe explains how to create specific communication laws for your application.

Working instructions

(1) Create a subclass of COMMUNICATIONLAW

(2) If necessary, add attributes that represent the COMMUNICATIONLAW and the according getters and setters

(2) Add a constructor with as parameter the state of the environment and pass it to the superclass. You can add extra parameters to initialize the attributes defined in (2).

(3) Implement the method *public ExternalMessage enforce(AgentState sender, AgentState receiver, ExternalMessage message)*. This method enforces the communication law on the given message, it returns the message after applying the law on it. *Message* is the message on which the law is applied, *sender* is the agent that sends the message, *receiver* is the receiver of the message. If the message should be deleted, return null.

Example

Code 59 The communication law CommunicationRangeLaw

```
public class CommunicationRangeLaw extends CommunicationLaw {                                    (1)

    public CommunicationRangeLaw(GridState state) {                                              (2)
        super(state);
    }

    public ExternalMessage enforce(AgentState sender, AgentState receiver,                       (3)
                                                        ExternalMessage message) {
        if (sender.equals(receiver))
           return null;
        if (! ((GridState)getState()).getItemsInPerceptionRange(sender).contains(receiver))
           return null;
        else
            return message;
    }
}
```

An example of a communication law is the CommunicationRangeLaw. This communication law does not allow agents to send messages outside their perception range.

(2) The state of the environment of the packetworld is the GridState, so the constructor of the law has this state as parameter and passes it to its superclass

(3) This method enforces the range law on the incoming message. If the sender equals the receiver or the receiver is not in the perception range of the sender, the message should be deleted (null is returned). Otherwise the message is returned unchanged.

Bibliography

[1] *http://www.fipa.org/.*

[2] *K.U.Leuven, Dept. of Computerscience, DistriNet Research Group, http://www.cs.kuleuven.ac.be/cwis/research/distrinet/.*

[3] R. A. Brooks. Intelligence without representation. *Artificial Intelligence Journal*, 47:139–159, 1991.

[4] D.Weyns and T.Holvoet. Look, talk and do, a synchronization scheme for situated multi-agent systems. In *Workshop notes of Workshop on Multi-agent Systems, UKMAS*, pages 1–8, 2002.

[5] Els Helsen and Koen Deschacht. Het delta raamwerk voor gesitueerde multi-agent systeem. Master's thesis, K.U.Leuven, 2005.

[6] K. Rosenblatt and D. Payton. A fine grained alternative to the subsumption architecture for mobile robot control. In *International Joint Conference on Neural Networks, Proceedings*, volume 2, pages 317–324, 1989.

[7] D. Weyns and T. Holvoet. Model for simultaneous actions in situated multi-agent systems. In *Multiagent System Technologies, First German Conference, MATES 2003, Proceedings*, volume 2831 of *Lecture Notes in Computer Science*, pages 105–118, 2003.

[8] D. Weyns and T. Holvoet. Regional synchronization for simultaneous actions in situated multi-agent systems. In *Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Proceedings*, volume 2691 of *Lecture Notes in Computer Science*, pages 497–510, 2003.

[9] D. Weyns, E. Steegmans, and T. Holvoet. Towards active perception in situated multi-agent systems. *Journal on Applied Artificial Intelligence*, 18(8-9):867–883, 2004.