Chapter 9

Self-Organizing Multi-Agent Systems

Event-based Sensor Processing, Feature Recognition, and Energy Management with Self-organizing Multi-Agent Systems for Sensor Processing

epubli, ISBN 9783746752228 (2018)

In this Chapter the principles of self-organizing Multi-agent systems and the relation to *AAPL* agents are discussed. Event-based sensor processing in large scale networks is one major use-case of Self-organizing MAS (SoMAS).

9.1 Introduction to Self-Organizing Systems

A common conceptual approach for building adaptive systems involves the design of such systems by using elements that find by themselves the solution of the problem to be solved [GER07]. Mobile Agents that are capable to adapt based on perception are well suited for the implementation of Self-organizing Systems (SoS).

Every dynamic and active system can be considered as agents that interact with each other and the agents are characterized by their behaviour and their goals. The behaviour of agents have influence of the future outcome of the behaviour of other agents and their aim to reach their goals or the selection of goals.

Considering [GER07] an agent is related to a goal satisfaction or fulfilment variable $\sigma \in [0,1]$. A system constructed of n agents is related to a system goal satisfaction function $f : \mathbb{R}^{2n+1} \rightarrow [0,1]$, depending on the weighted σ_i of each agent, defined in Equation 9.1. A weight w_i specifies the importance of the fulfilment of a particular agent for the system.

A system has a weight w_0 itself that can be considered as being a bias. This concept can be extended to hierarchical systems, where each system levels depends on the elements (sub-systems or agents) from which they are composed. If the system consists of elements with nearly linear interaction, the system satisfaction function can be approximated by a weighted sum. In heterogeneous systems, the system satisfaction function is non-linear.

$$\sigma_{sys} = f(\sigma_1, \sigma_2, ..., \sigma_n, w_0, w_1, ..w_n) \tag{9.1}$$

The robustness of an SoS is related to the system satisfaction function. If elements (sub-systems or agents) are removed or altered and σ_{sys} does change significantly, we can say that the system provides some kind of robustness or tolerance regarding failures of parts of the system, and if σ_{sys} does not change significantly if any one of the elements fail (i.e., $\sigma_i \rightarrow 0$ or $|\Delta\sigma_i| \gg |\Delta\sigma_{sys}|$), we can say there is no single point of failure in the whole system. For instance, a system goal can be the detection of a feature change of a sensing system, e.g., a significant change of sensor values in a topologically correlated region. On the other side, if a small change of any $|\Delta\sigma_i| \leq |\Delta\sigma_{sys}|$ affects the system satisfaction significantly then the system can be considered as fragile.

Destructive inference and friction between elements minimizes the overall system satisfaction σ_{sys}. Semi-centralized sub-domain coordination can

improve the overall performance and the probability of the goal satisfaction by introducing mediators maximizing cooperation and resolving especially conflicts between elements, e.g., resource conflicts. A mediator is related to observers defining constraints.

In the following sections three examples for Self-organizing MAS (SoMAS) are introduced. All three MAS classes can be deployed in a sensor network.

The first SoMAS is used for the feature recognition detecting a boundary of a correlated region of sensor values, the second SoMAS is used to distribute and deliver sensor data event-based relying on the results of the feature recognition SoMAS, and the third SoMAS is used for the distributed smart energy management with energy distribution.

The ontology consisting of the various agent classes is shown in Figure 9.1, based on the adapted Agent Modelling Language (*AML*) notation [CER07].

The *AML* ontology diagram shows the relationship of the three MAS classes and their deployment in the sensor network. The agent classes poses the principle behaviours of agents instantiated from these classes.

The behaviours are not necessarily the *AAPL* activities of the agent classes presented in the next sections. A behaviour can be implemented with different activities.

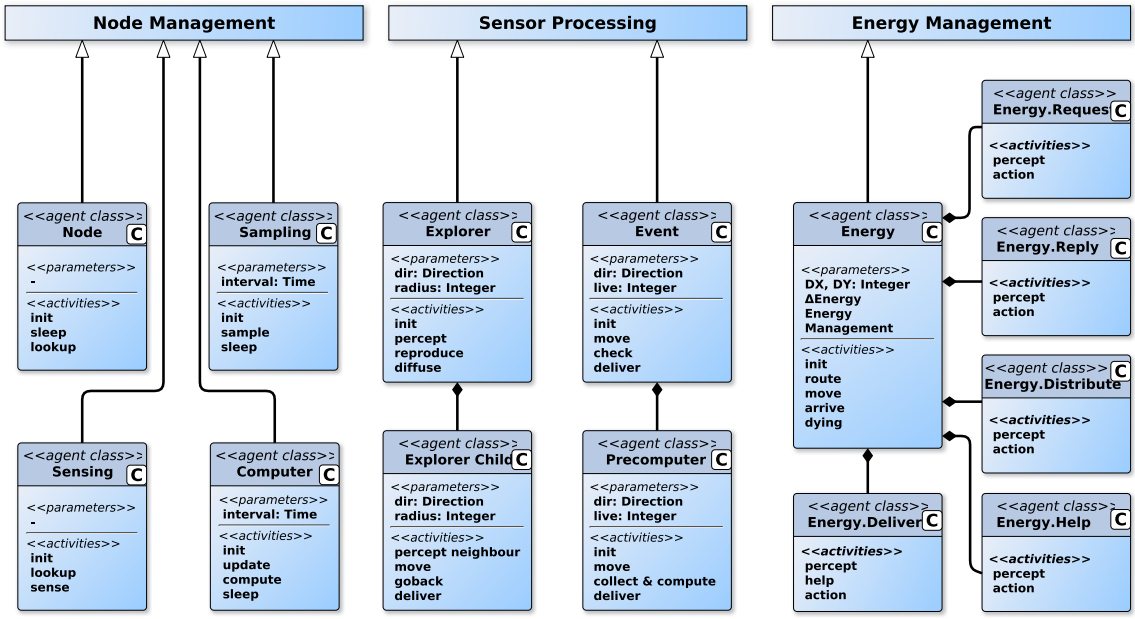

Fig. 9.1 *Ontology of the Sensor Network in AML notation composed of (a) node management, (b) event-based sensor processing, (c) feature recognition, and (d) smart energy management agent classes.*

In *AML*, an ontology class represents ontology concepts and frames. An ontology class is specified by its name, usually related to an agent behaviour class, a list of attributes, in the *AAPL* terminology the agent class parameters, a set of operations (e.g., the *AAPL* activities and functions), parts, and behaviours [CER07]. AML provides the modelling of instance-level and class-level ontologies (marked with a [C] icon), as shown in Figure *9.1* and later in Figure *9.9* (on page 314).

9.2 Self-organizing Distributed Feature Recognition

A small example implementing a distributed feature detection in an incompletely connected and unreliable mesh-like sensor network using mobile agents should demonstrate the suitability of self-organizing MAS for sensor data processing in distributed sensor networks. The sensor network consists of nodes with each node attached to a sensor used, for example, in a structural monitoring system (e.g. strain-gauge sensors), providing a scalar data value. The nodes can be embedded in a mechanical structure, for example, used in a robot arm. The goal of the MAS is to find the boundary of extended correlated regions of increased sensor stimuli (compared to the neighbourhood) due to mechanical deformation resulting from externally applied load forces. A distributed directed diffusion behaviour and self-organization (see Figure *9.2*) is used, derived from the approach proposed originally by [LIU01] for image processing feature recognition. A single sporadic sensor activity not correlated with the surrounding neighbourhood should be distinguished from an extended correlated region, which is the feature to be detected. There are three different agent classes used in the sensor network: an exploration, a deliver, and a node agent.

A *node agent* is immobile and is primarily responsible for sensor measurement, local preprocessing (filtering) and observation, and creating of exploration and deliver agents.

The feature detection is performed by the mobile *exploration agent*, which supports two main different behaviours: diffusion and reproduction. The explorer agent can be composed of the root agent class implementing diffusion and reproduction and an explorer child agent sub-class with a reduced behaviour set used for the exploration of the immediate neighbourhood relative to the current position of the explorer agent. The diffusion behaviour is used to move into a region, mainly limited by the lifetime of the agent, and to detect the feature, here the region with increased mechanical distortion (more precisely the edge of such an area). The detection of the feature enables the reproduction behaviour.

epubli, ISBN 9783746752228 (2018)

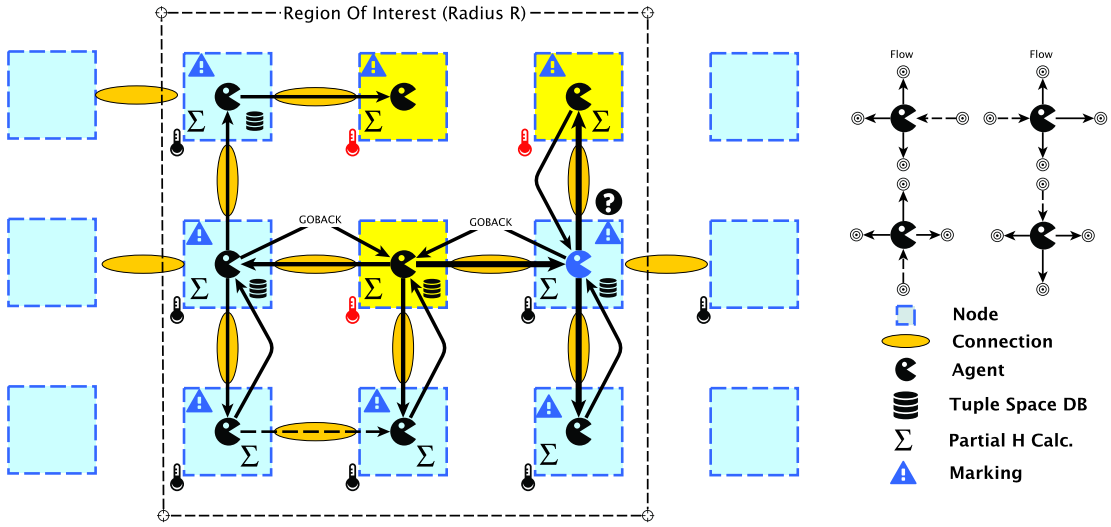

Fig. 9.2 *Distributed feature extraction in an unreliable and incomplete network by using distributed agents with migration and self-organization behaviour*

The reproduction behaviour induces the agent to stay at the current node, setting a feature marking and sending out more exploration agents in the neighbourhood. The local stimuli $H(i,j)$ for an exploration agent to stay at a specific node with coordinate (i,j) is given by Equation 9.2.

$$H(i,j) = \sum_{s=-R}^{R} \sum_{t=-R}^{R} \{ \| S(i+s, j+t) - S(i,j) \| \leq \delta \} \qquad (9.2)$$

S : Sensor Signal Strength

R : Square Region around (i,j)

The calculation of H at the current location (i,j) of the agent requires the sensor values within the rectangular area (the region of interest ROI) R around this location. If a sensor value $S(i+s,j+t)$ with $i,j \in \{-R,..,R\}$ is similar to the value S at the current position (difference is smaller than the parameter δ), H is incremented by one.

If the H value is within a parametrizable interval $\Delta=[\varepsilon_0,\varepsilon_1]$, the exploration agent has detected the feature and will stay at the current node to reproduce new exploration agents sent to the neighbourhood. If H is outside this interval, the agent will migrate to a neighbour different node and restarts exploration (diffusion).

The calculation of *H* is performed by a distributed calculation of partial sum terms by sending out child explorer agents to the neighbourhood, which itself can send out more agents until the boundary of the region *R* is reached. Each child agent returns to its origin node and hands over the partial sum term to his parent agent, shown in Figure 9.2. Because a node in the region *R* can be visited by more than one child agent, the first agent reaching a node sets a marking `MARK`. If another agent finds this marking, it will immediately return to the parent. This multipath visiting has the advantage of an increased probability of reaching nodes with missing (non operating) communication links (see Figure 9.2). A deliver agent, created by the node agent, finally delivers exploration results to interested nodes by using directed diffusion approaches, not discussed here.

9.2.1 Explorer Agent Behaviour Model

The explorer agent behaviour is partitioned in a main class and a child subclass. The ATG is shown in Figure 9.3. The different goals of the explorer agent (exploration, diffusion, reproduction) are served by the activities `percept`, `diffuse`, and `reproduce`. Perception inference requires the forking of child explorer agents, performed in the `percept` activity. Child agents create forked child agents (in activity `percept_neighbour`) until they reach the boundary of the ROI. The forked child agents will return to their parent location after perception (collecting of sensor data and computation of the partial term *h* of *H*), performed in the `goback` activity. Parents agents wait for their child agents until either all child agents returned or a time-out occurs. Each time a child agent delivers the percepted *h* value (in activity `deliver`) by updating the *H* tuple in the tuple-space it sends a signal `WAKEUP` that decreases a counter (`enoughinput`). The full *APPL* agent behaviour model is shown in Algorithm 9.1.

Alg. 9.1 *Definition of the Explorer agent behaviour class and the Explorer child subclass*

```
1   κ: { SENSORVALUE,FEATURE, H, MARK }      set of key symbols
2   ξ: { TIMEOUT, WAKEUP }                   set of signals
3   ω: { NORTH, SOUTH, WEST, EAST, ORIGIN }  set of directions
4   ε1 =3; ε2 = 6; MAXLIVE = 1;              some constant parameters
5
6   Ψ Explorer: (dir,radius) → {
7     Body Variables
8     Σ: { dx, dy, live, h, s0, backdir, group }  global persistent variables
9     σ: { enoughinput, again, die, back, s, v }  local  temporary  variables
10
11    Activities
12    α init: {
13      dx ← 0; dy ← 0; h ← 0; die ← false; group ← ℜ{0..10000};
14      if dir ≠ ORIGIN then
15        ⇔dir; backdir ← ϖ(dir)
```

```
16      else
17        live ← MAXLIVE; backdir ← ORIGIN
18      ∇⁺(H,$self,0);
19      found ← ∇^?%(0,SENSORVALUE,s0?)
20    }
21    α percept: {
22      enoughinput ← 0;
23      ∀{nextdir ∈ ω | nextdir ≠ backdir ∧ ?Λ(nextdir)} do
24        incr(enoughinput);
25        Θ→Explorer.child(nextdir,radius)
26      τ⁺(ATMO,TIMEOUT)
27    }
28    α reproduce: {
29      live--;
30      ∇ˣ(H,$self,?);
31      if ?∇(FEATURE,?) then ∇⁻(FEATURE,n?) else n ← 0;
32      ∇⁺(FEATURE,n+1);
33      if live > 0 then
34       π*(reproduce → init)
35       ∀{nextdir ∈ ω | nextdir ≠ backdir ∧ ?Λ(nextdir)} do
36         Θ→(nextdir,radius)
37       π*(reproduce → exit)
38    }
39    α diffuse: {
40      live--;
41      ∇ˣ(H,$self,?);
42      if live > 0 then
43       dir ← ℜ{nextdir ∈ ω | nextdir ≠ backdir ∧ ?Λ(nextdir)}
44      else
45       die ← true
46    }
47    α exit: { ⊗($self) }
48
49    inbound: (nextdir) → {
50      case nextdir of
51      | NORTH →  dy > -radius
52      | SOUTH →  dy < radius
53      | WEST  →  dx > -radius
54      | EAST  →  dx < radius
55    }
56
57    Signal handler
58    ξ TIMEOUT: {
59      enoughinput ← 0
60    }
61    ξ WAKEUP: {
62      enoughinput--;
63      if ?∇(H,$self,?) then ∇⁻(H,$self,h?);
64      if enoughinput < 1 then τ⁻(TIMEOUT);
65    }
66
```

```
67    Main Transitions
68    ∏: {
69      entry → init
70      init → percept | found
71      init → exit    | ¬found
72      percept → reproduce | (h ≥ ε1 ∧ h ≤ ε2) ∧ (enoughinput < 1)
73      percept → diffuse   | (h < ε1 ∨ h > ε2) ∧ (enoughinput < 1)
74      reproduce → exit
75      diffuse → init | die = false
76      diffuse → exit | die = true
77                                                  }
78    Explorer child sub-class
79    φ child: {
80      ↓ percept,exit      imported from root class
81      ↓ group,s,s0,h,backdir,dx,dy,dir,enoughinput,back
82      α percept_neighbour {
83        found ← ∇?%(0,SENSORVALUE,s?);
84        if found ∧ not ?∇(MARK,group) then
85          back ← false; enoughinput ← 0;
86          ∇τ(MTMO,MARK,group);
87          h ← (if |s-s0| ≤ DELTA then 1 else 0);
88          ∇+(H,$self,h);
89          π*(percept_neighbour → move)
90        ∀{nextdir ∈ ω | nextdir ≠ backdir ∧ ?Λ(nextdir) ∧ inbound(nextdir)} do
91            Θ→(nextdir,radius)
92          π*(percept_neighbour → goback | enoughinput < 1)
93          τ+(ATMO,TIMEOUT)
94        else back ← true
95      }
96      α move: {
97        backdir ← ϖ(dir); (dx,dy) ← (dx,dy) + δ(dir);
98        ⇔dir;
99      }
100     α goback: {
101       if ?∇(H,$self,?) then ∇-(MARK,$self,h?) else h ← 0;
102       ⇔backdir;
103     }
104     α deliver: {
105       ∇-(H,$parent,v?); ∇+(H,$parent,v+h);
106       ξWAKEUP ⇒ $parent;
107     }
108     π: {
109       percept → move
110       move → percept_neighbour
111       percept_neighbour → (enoughinput < 1) ∨ back
112       goback → deliver
113       deliver → exit
114     }
115   }
116 }
```

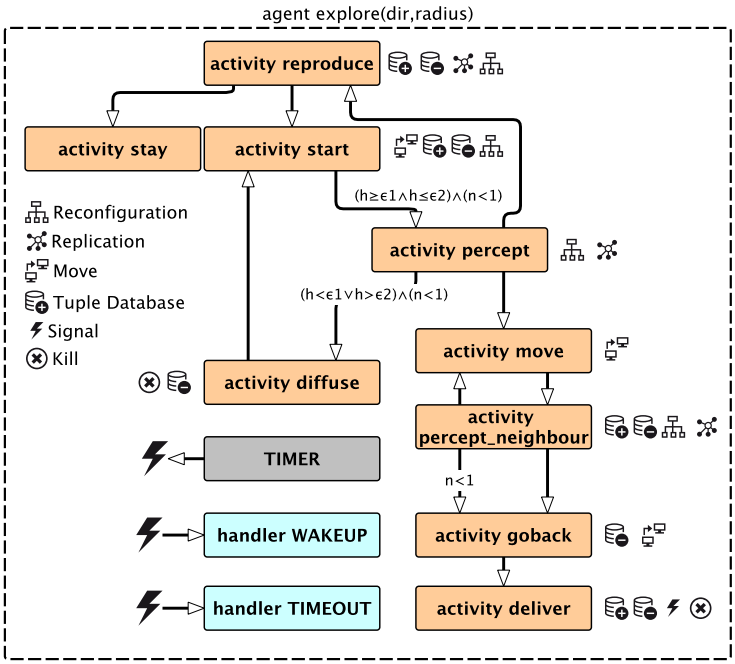

Fig. 9.3 *AAPL Behaviour Model of the explorer agent and the explorer child agent class branching from the percept activity.*

9.2.2 Some Simulation Experiments of a Sensor Network

To evaluate the capabilities of the feature marking SOMAS introduced in the previous section, the simulation environment described in Chapter *11* is used to carry out simulations with synthetic and real-world sensor data (though obtained from FEM simulation, the data sets are rather realistic including noise).

Figure *9.4* shows simulation results of a connectivity-incomplete 8x8 sensor network with a rectangular sensor stimuli region having a sharp boundary. The network had a communication connectivity of *CN*=70% (30% communication links are not operating). The creation of a root explorer agent involves three parameters: 1. The radius R and the size N_R of the square ROI (R=1 means 9 sensor values, R=2 means 27 sensor values contributing to the H calculation); 2. The lifetime L in node distance units; 3. The $\partial\varepsilon=[\varepsilon_0,\varepsilon_1]$ decision interval. In all simulations a $\partial\varepsilon=[3,6]$ setting was used.

With a parameter set {R=1, L=1} the sharp boundary of the sensor stimuli is detected reliable for a cluster size of 8 and 15 sensors shown in the plots (a)-(c) and (d). Surprisingly, the *CL*=15 cluster is not recognized with a parameter set {R=2, L=1} (e), in contrast to the smaller cluster with *CL*=8.

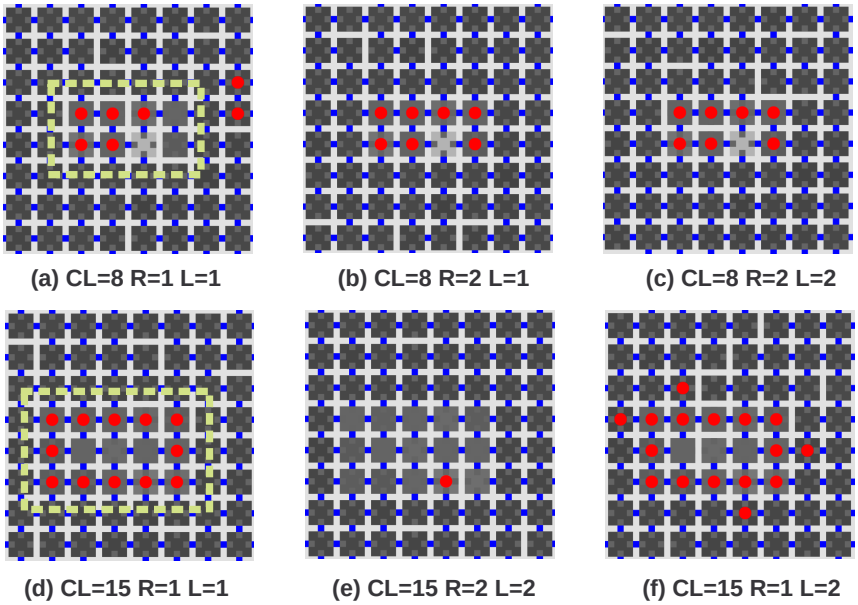

Fig. 9.4 *SOS Feature Marking (red circles) with a localized rectangular sensor stimuli region having a sharp boundary (yellow dotted line). CL: Cluster size, R: Exploration radius, L: Explorer lifetime, Network connectivity CN=70%*

Increasing the lifetime usually not increases the quality of feature recognition. In the case of the larger cluster size *CL*=15 (f) the fuzziness of the boundary increases if the lifetime is increased.

In Figure *9.5* the feature detection is applied to data sets retrieved from load and strain simulations of a steel plate using FEM simulation (see [BOS14C][BOS14F] for details), which leads to a more continuously sensor stimuli distribution without having a sharp boundary.

The first data set related to a specific load case has a significant increase of sensor values at the east side of the network. The boundary feature detection SOMAS reliable finds the west side of the region regardless of the different parameter settings, shown in the plots (a)-(c).

The second data set and load case with a smoother sensor value distribution and a lower sensor value gradient shows a totally different result. In plot (d) with the parameter set {R=1, L=1} the flat region is marked instead the sensor value gradient on the east side. This changes again with the parameter sets {R=2, L=1} and {R=2, L=2} shown in the plots (e)-(f), now detecting the gradient boundary correctly.

The third data set and load case with a nearly constant gradient of the sensor values shows again different results for R=1 and R=2 settings. The R=2 setting always marks the entire network, which is primarily a result of the

decision interval setting $\partial\varepsilon$. The $\boldsymbol{R}$=1 setting finds again the west side of the sensor stimuli related to the lowest sensor values.

To summarize the edge detection capabilities of the SoMAS are mostly suitable to recognize a stimulated sensor value region and can be used for triggering of the event-based sensor data distribution and processing described in Section 9.3. The quality of the feature detection depends on the parameter set {R,$\partial\varepsilon$}, which can be adjusted at run-time by using reinforcement learning performed by the agents based on a quality feedback from the computational nodes.

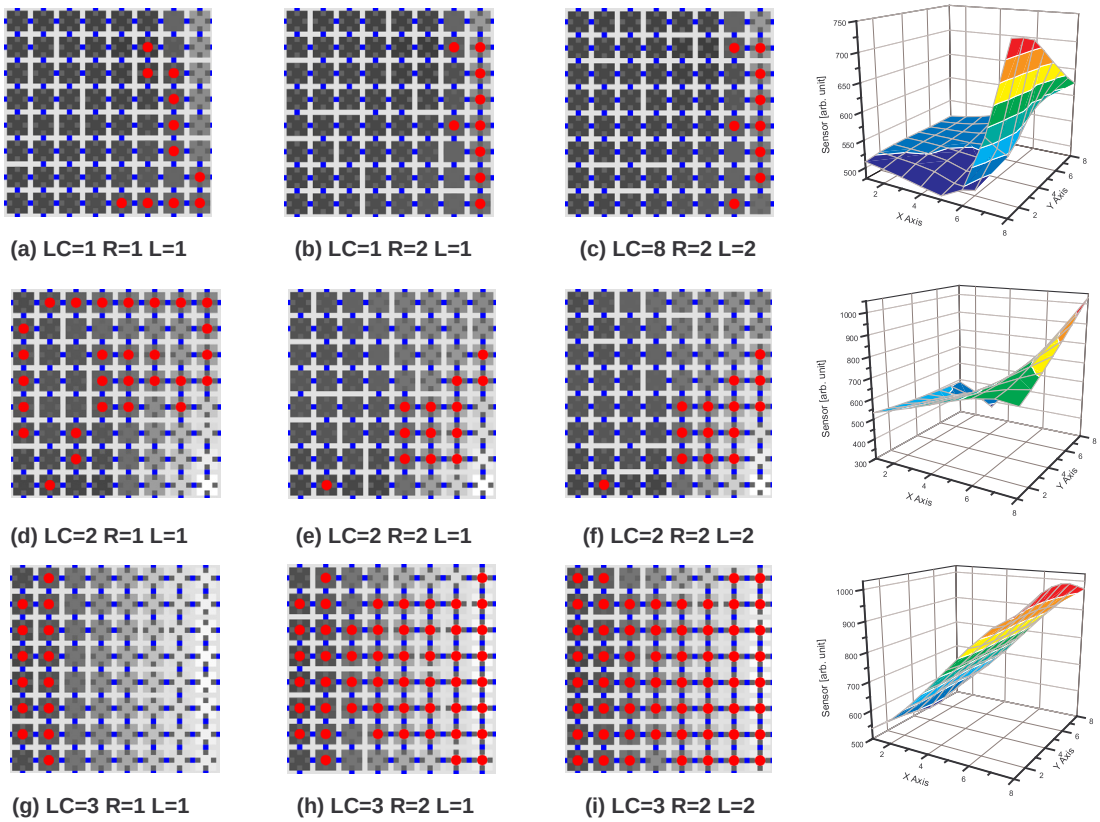

Fig. 9.5 *SOS Feature Marking (red circles) with a large area sensor stimuli region having no clearly defined boundary (continuous change). LC: Load case, R: Exploration radius, L: Explorer lifetime, Network connectivity CN=70%*

9.3 Self-organizing Event-based Sensor Data Processing and Distribution

Large scale sensor networks with hundreds and thousands of sensor nodes require smart data processing concepts far beyond the traditional centralized approaches. Multi-Agent systems can be used to implement smart and optimized sensor data processing in these distributed sensor networks.

Event-based sensor data distribution and pre-computation with agents reduces communication and overall network activity resulting in reduced energy consumption of single nodes and the entire network.

Different sensor data processing and distribution approaches are used and implemented with agents, leading to a significant decrease of network processing and communication activity and a significant increase of reliability and the Quality-of-Service:

1. An event-based sensor distribution behaviour is used to deliver sensor information from source sensor to computation nodes
2. Adaptive path finding (routing) supports agent migration in unreliable networks with missing links or nodes by using a hybrid approach of random and attractive walk behaviour
3. Self-organizing agent systems with exploration, distribution, replication, and interval voting behaviours based on feature marking are used to identify a region of interest (ROI, a collection of stimulated sensors) and to distinguish sensor failures (noise) from correlated sensor activity within this ROI. The feature SoS, already presented in the previous section, triggers the creation of sensor distribution agents.

It is assumed that sensor nodes arranged in a two-dimensional grid network (as shown in Figure *9.6*) providing spatially resolved and distributed sensing information of a surrounding technical structure, for example, a metal plate. Each sensor node shall sense mechanical properties of the technical structure nearby the node location, for example, by using strain gauge sensors. Usually a single sensor cannot provide any meaningful information of the mechanical structures. A connected area of sensors (complete sensor matrix or a part of it) is required to calculate the response of the material due to applied forces, i.e., computing the applied load vector from the sensor data vector either by using inverse numerical or machine learning approaches, discussed in Chapter *14*. The computation of the material response requires high computational power of the processing unit, which cannot offered by downscaled single micro-chip platforms. For these reasons, sensor nodes use mobile agents to deliver their sensor data to dedicated computational nodes located at the edges of the sensor network, shown in Figure *9.6*, discussed in

detail in the following sub-sections. The computational nodes arranged at the outside of the network are further divided in pre-computation and the final computation nodes (the four nodes located at the corners of the network). The pre-computational nodes can be embedded PCs or single micro-chips, and the computational nodes can be workstations or servers physically displaced from the material-embedded sensor network. Only the inner sensor nodes are micro-chip platforms embedded in the technical structure material, for example, using thinned silicon technologies.

The computation of the system response information requires basically the complete sensor signal matrix $\boldsymbol{S}$. In traditional sensor signal processing networks this sensor matrix is updated in regular time intervals, resulting in a high network communication and sensor node activities. In this approach presented here the elements of the sensor matrix are only updated if a significant change of specific sensors occurred. Only the four corner computational nodes store the complete sensor matrix and perform the load computations.

The sensor processing uses both stationary (non-mobile) and mobile agents carrying data, illustrated in Figure 9.7 on the left side. There are two different stationary (non-mobile) agents operating on each sensor node: the sampling agent which collects sensor data, and the sensing agent, which preprocesses and interprets the acquired sensor data.

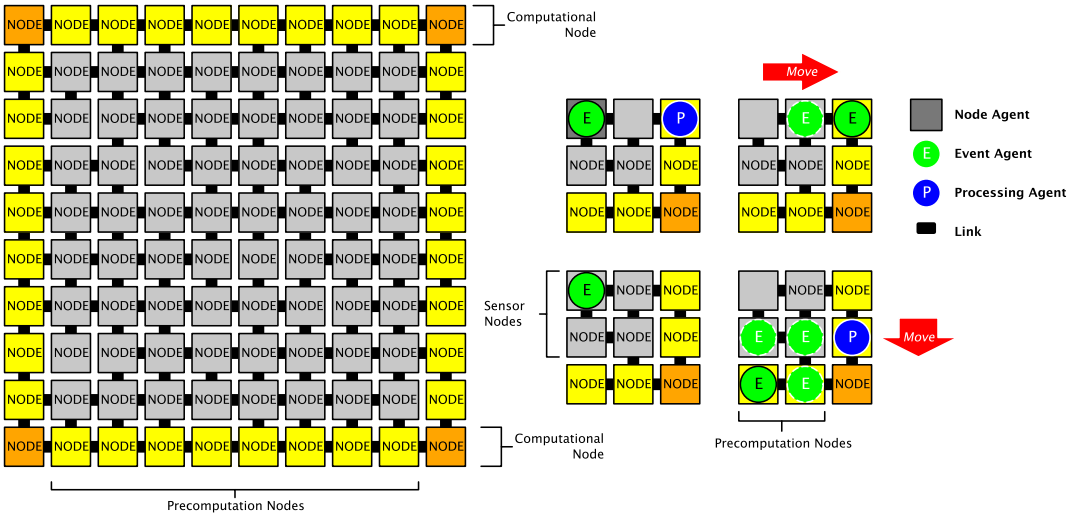

Fig. 9.6 *The logical view of a sensor network with a two-dimensional mesh-grid topology (left) and examples of the population with different mobile and immobile agents (right): event deliver, node, and computational processing agents. The sensor network can contain missing or broken links between neighbour nodes.*

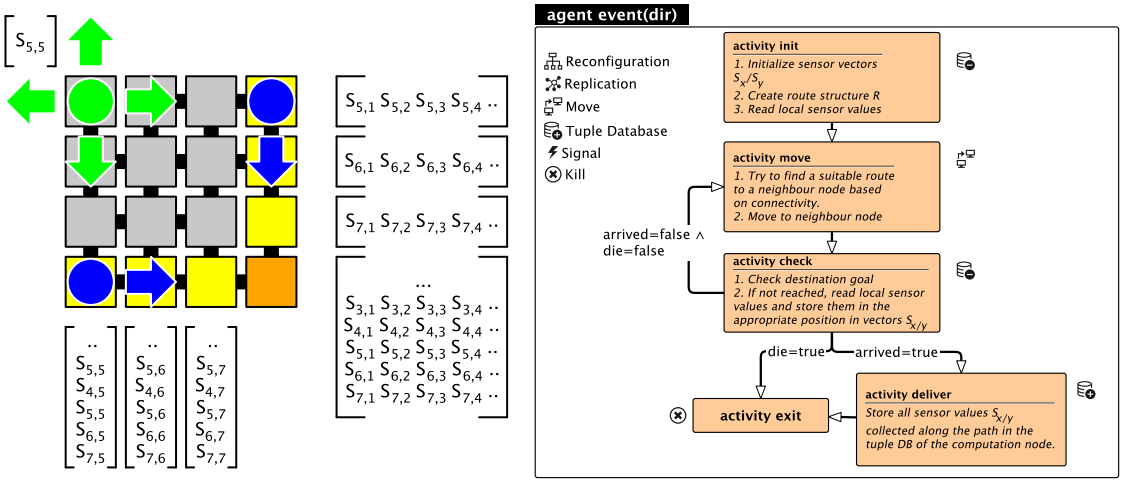

Fig. 9.7 *Left: Sensor data distribution with event (green) and preprocessing agents (blue): A sensor node which detected a significant change of the sensor values creates event agents which are sent in all four directions to the network boundary (pre-computation nodes). Right: ATG behaviour model of the event agent.*

If the sensing agent detects a relevant change in the sensor data, it sent out four mobile event agents, each in another direction. The event agent carries the sensor data and delivers it to the pre-computation nodes at the boundary of the sensor network. The agent behaviour is specified in Algorithms *9.2* and *9.3* (giving the routing behaviour), and an overview of the agent behaviour and the ATG can be found in Figure *9.7* on the right side.

9.3.1 Event Agent Behaviour

An event agent has a predefined path in the direction *dir* that is followed by the `move` activity as long as there is connectivity to the next neighbour node in this direction. Normally the agent travels to the outside of the network on the given direction by applying the `route_normal` routing strategy successfully. If it is impossible to migrate in the predefined direction, an alternative path is chosen by using the `route_opposite` routing strategy, which chooses a path away from the original destination to bypass unconnected nodes and missing communication links. Using the `route_relax` routing strategy the agent is directed again to the original planned path. Making routing decisions and migration are performed in the `move` activity of the agent, followed by the `check` activity, which collects sensor data from the current node and checks

the destination node goal, and if reached delivering the sensor values in the `deliver` activity.

Each pre-computation node stores a row or a column of the sensor matrix *S*. If their data changes, the pre-computation nodes will send out two mobile distribution agents in opposite directions, delivering a row or column of ***S*** to the final computation nodes, located at the edges of the sensor network.

Alg. 9.2 *Agent behaviour of the Event agent class offering a robust event-based and path tracking sensor data distribution*

```
1   κ: { SENSORVALUE,DISTRIBUTER }          set of key symbols
2   δ: { NORTH,SOUTH, WEST, EAST, ORIGIN }  set of directions
3   MAXFAILED = 4                           some constant parameters
4
5   type Route = (dir = ω, lastdir = ω, delta = Δ, gamma = Δ, routed = boolean);
6
7   Ψ Event: (dir) → {
8     Body Variables
9     Σ: { route, arrived, failed, die, SX=[0..DIMX-1], SY=[0..DIMY-1] }
        global persistent variables
10    σ: { vx, vy, index, found, row, col, rown, coln }
        local  temporary  variables
11
12    Activities
13    α init: {
14      arrived ← false;
15      ∀ i ∈ {0 .. DIMX-1} do SX[i] ← -1;
16      ∀ i ∈ {0 .. DIMY-1} do SY[i] ← -1;
17      route ← Route(dir,ORIGIN,(0,0),(0,0),false);
18      found ← ∇?%(0,SENSORVALUE,vx?,vy?);
19      if found then SX[0] ← vx; SY[0] ← vy;
20    }
21
22    α move: {
23      route.dir ← dir;  Try different routing strategies
                          to reach the destination
24      route ← route_relax(route);
25      if not route.routed then route ← route_normal(route);
26      if not route.routed then route ← route_opposite(route);
27      if route.routed then ⇔(route.dir) else failed++;
28    }
29
30    α check: {
31      found ← ?∇(DISTRIBUTER);
32      if found ∧ route.gamma=(0,0) then arrived ← true
33      else if route.gamma=(0,0) then
34        found ← ∇?%(0,SENSORVALUE,vx?,vy?); Collect all sensor values
                                                along delivery path
35        if found then
36          case dir of
```

```
37            | NORTH ⇒ index ← -route.delta.Y
38            | SOUTH ⇒ index ←  route.delta.Y
39            | WEST  ⇒ index ← -route.delta.X
40            | SOUTH ⇒ index ←  route.delta.X
41          SX[index] ← vx; SY[index] ← vy;
42      if failed > MAXFAILED then die ← true;
43    }
44
45    α deliver: {
46      ∇%(MATRIXDIM,row?,col?,rown?,coln?);
47      index ← 0;
48      case dir of  Deliver all sensor values collected along delivery path
49        | NORTH ⇒
50          ∀ row ∈ { -route.delta.Y-1 .. 0 } do
51            ∇+(SENSORVALUE,row,col,SX.[index],SY.[index]); index++;
52        | SOUTH ⇒
53          ∀ row ∈ { rown-route.delta.Y .. rown-1 } do
54            ∇+(SENSORVALUE,row,col,SX.[index],SY.[index]); index++;
55        | WEST ⇒
56          ∀ col ∈ { -route.delta.X-1 .. 0 } do
57            ∇+(SENSORVALUE,row,col,SX.[index],SY.[index]); index++;
58        | EAST ⇒
59          ∀ col ∈ { coln-route.delta.X .. coln-1 } do
60            ∇+(SENSORVALUE,row,col,SX.[index],SY.[index]); index++;
61    }
62
63    α exit: { ⊗($self) }
64
65    Main Transitions
66    ∏: {
67      entry → init
68      init → move
69      move → check
70      check → deliver | arrived = true
71      check → move    | arrived = false ∧ die = false
72      check → exit    | die = true
73      deliver → exit
74    }
75  }
```

Alg. 9.3 *Routing functions*

```
1   δ: { NORTH,SOUTH, WEST, EAST, ORIGIN }      set of directions
2   type Route = (dir = ω, lastdir = ω, delta = Δ, gamma = Δ, routed = boolean);
3
4   route_normal: (route) → {
5     if ?Λ(dir) ∧ route.last_dir ≠ ϖ(dir) then
6       route.routed ← true; route.lastdir ← dir;
7       route.delta ← route.delta+∂(route.dir);
8       case route.dir of
9       | NORTH ⇒
```

```
10        if route.gamma.Y ≠ 0 then route.gamma ← route.gamma+∂(dir);
11      | SOUTH ⇒
12        route.routed ← true; route.lastdir ← NORTH;
13        if route.gamma.Y ≠ 0 then route.gamma ← route.gamma+∂(dir);
14      | WEST ⇒
15        if route.gamma.X ≠ 0 then route.gamma ← route.gamma+∂(dir);
16      | EAST ⇒
17        if route.gamma.X ≠ 0 then route.gamma ← route.gamma+∂(dir);
18    ↑route
19  }
20
21  route_opposite: (route) → {
22    routes ← {d ∈ ω | ?Λ(d) ∧ route.lastdir ≠ ϖ(d) };
23    if routes ≠ ∅ then
24      route.routed ← true;
25      route.dir ← ℜ(routes);
26      route.lastdir ← route.dir;
27      route.delta ← route.delta+∂(route.dir);
28      route.gamma ← route.gamma+∂(dir);
29    ↑route
30  }
31
32  route_relax: (route) → {
33    nextdir ← ORIGIN;
34    if route.gamma ≠ (0,0) then
35      if route.gamma.X < 0 ∧ ?Λ(EAST) ∧ route.lastdir ≠ ϖ(EAST) then
36        nextdir ← EAST;
37      if route.gamma.X > 0 ∧ ?Λ(WEST) ∧ route.lastdir ≠ ϖ(WEST) then
38        nextdir ← WEST;
39      if route.gamma.Y < 0 ∧ ?Λ(SOUTH) ∧ route.lastdir ≠ ϖ(SOUTH) then
40        nextdir ← SOUTH;
41      if route.gamma.Y > 0 ∧ ?Λ(NORTH) ∧ route.lastdir ≠ ϖ(NORTH) then
42        nextdir ← NORTH;
43      if nextdir ≠ ORIGIN then
44        route.dir ← nextdir;
45        route.routed ← true; route.lastdir ← dir;
46        route.delta ← route.delta+∂(route.dir);
47    ↑route
48  }
49
50  ∂: (dir) → {
51    case dir of
52    | NORTH ⇒ (0,-1)
53    | SOUTH ⇒ (0,+1)
54    | WEST  ⇒ (-1,0)
55    | EAST  ⇒ (+1,0)
56  }
```

9.3.2 Simulation Experiments of a Sensor Network

Figure *9.8* shows the population of sensor and computation nodes with these agents retrieved from simulation using the Multi-Agent simulation environment shell *SeSAm* (see Chapter *11*). At the beginning there is an initial update of all sensor values, resulting in a fairly high number of event agents followed delayed by a high number of distribute agents (324 event and more than 500 distribute agents). The replicated sensor value delivery to four different computational nodes ensures a high reliability in the presence of node and link failures, which is likely in sensor networks embedded in technical structures and materials. But after this initial setup of the sensor network resulting in a flooding of the network, there are only few event and distribute agents (four event agents for each stimulated sensor node) required to update changes in the sensor matrix, shown in the simulation in Figure *9.8* (top row) at different time points t=100, 200, and so forth, for two different cluster sizes (the correlated area of stimulated sensors). The total number of distributed agents (maximal 8 for each stimulated sensor) depends on the time interval in which the pre-computation nodes send updated rows or columns to the computation nodes.

Simulation results obtained from different network situation using Monte-Carlo simulation are shown in the bottom row in Figure *9.8*. In the case of 30% broken links a slight increase of the travelling time of event and distribute agents can be observed by a broadening of the agent population curve, and in the case of 50% broken links the increased mean travelling time is significant, compared with the results shown on the top row in Figure *9.8*.

All agents can reach their intended destination if the probability for a broken link is below 10%. With increasing link failure probability not all event and distribute agents can reach their destination, which will die somewhere on the way after an upper limit of unsuccessful routing iterations.

Due to failed deliveries of sensor values and due to the temporal delay of different sensor values, resulting from different path lengths and node positions, the sensor matrix stored in each computational nodes can differ temporally or permanently from the real sensor matrix at a given time. Surprisingly, even with a large fraction of non-operational communication links, each cumulative image of the real sensor matrix stored in the computational nodes experience less deviation.

epubli, ISBN 9783746752228 (2018)

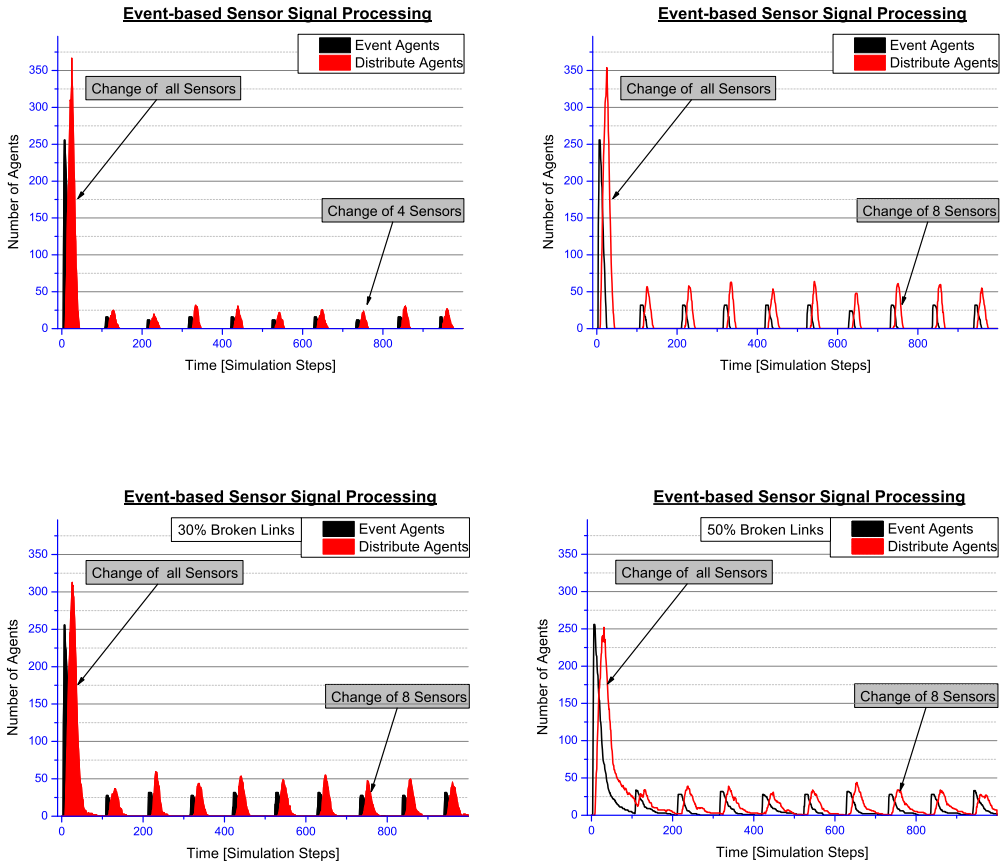

Fig. 9.8 *Analysis results of the agent population obtained from the multi-agent simulation of the event-based sensor data processing.*
(Top, left): clusters of four sensors are stimulated periodically with different centre position, (Top, right): cluster size is 8 sensors
(Bottom, left): With 30% broken links, (Bottom, right): With 50% broken links, (Both: cluster size is 8 sensors)

9.3.3 Interaction of Event and Explorer Agents

The following Figure *9.9* poses the relationship between feature recognition explorer, explorer child, event-based sensor distribution, and the node agents (sampling, sensing,..).

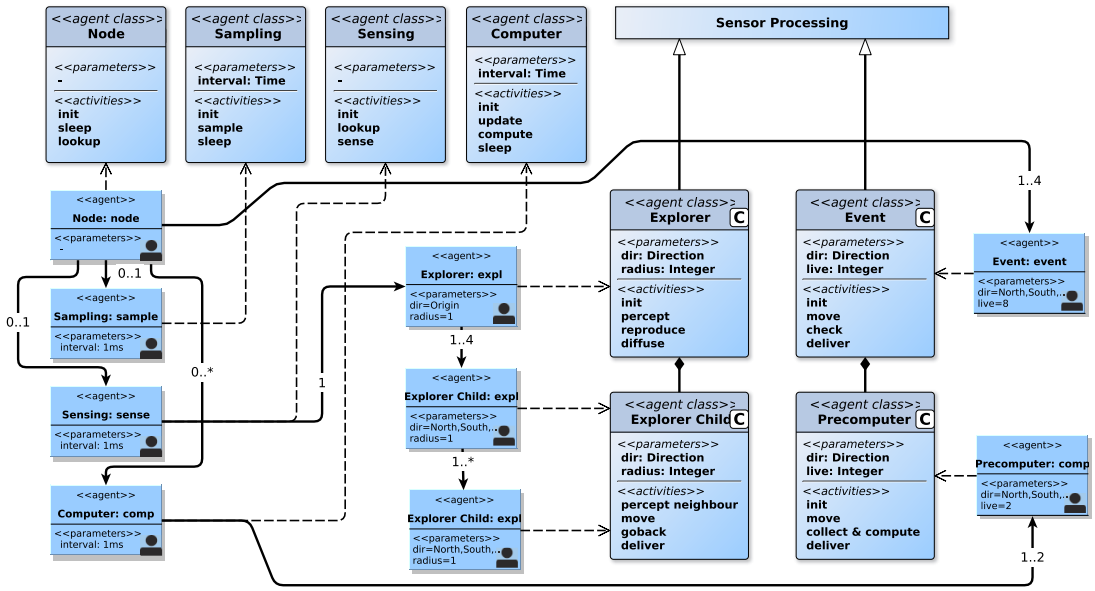

Fig. 9.9 *Ontology of the sensor network: Node sensing and sensor processing agent classes and the instantiation of agents*

9.4 Self-organizing Energy Management and Distribution

Having the technical ability to communicate data carrying energy with messages by using communication links, which will be introduced in the Section *13.2.2*, it is possible to use active messaging to transfer energy from "good" nodes having enough energy towards "bad" nodes, requiring energy. A mobile smart energy management (SEM) agent can be sent out by a bad node to explore and exploit the near neighbourhood. The agent examines sensor nodes during path travel or passing a region of interest (perception) and decides to send agents holding additional energy back to the original requesting node (action). Additionally, a sensor node is represented by a node energy management agent (SEN), too. The node and the energy management agents must negotiate the energy request.

9.4.1 The Mobile Smart Energy Management Agent

The behaviour of the SEM agent is composed of multiple sub-behaviours, each associated with its own subclass, shown in Algorithm *9.4*. The subclasses share the activities route and move. All subclass behaviours are entered from the main class `arrive` activity. The agent is capable to transfer energy from the current node to a neighbour node using the `transfer` operation. The behaviour and goals of each agent subclass are:

Request

Point-to-point agent: this agent requests energy from a specific destination node, returned with a `Reply` agent.

Reply

Point-to-point agent: Reply agent created by a `Request` agent, which has reached its destination node. This agent carries energy from one node to another.

Help

ROI agent: this agent explores a path starting with an initial direction and searches a good node having enough energy to satisfy the energy request from a bad node. This agent resides on the final good node for a couple of times and creates multiple deliver agents periodically in dependence of the energy state of the current node.

Deliver

Path agent: this agent carries energy from a good node to a bad node (response to `Help` agent). Depending on selected sub-behaviour (`HELPONWAY`), this agent can supply bad nodes first, found on the back path to the original requesting node.

epubli, ISBN 9783746752228 (2018)

Distribute

ROI agent: this agent carries energy from and is instantiated on a good node and explores a path starting with an initial direction and searches a bad node supplying it with the energy.

Initially the SEM agent is instantiated by the local SEN agent, either with the sub-class behaviour *help* (kind=`HELP`) or *distribute* (kind=`DISTRIBUTE`) selected, beginning at lines 148 and 184, respectively. The first action that the SEN agent performs is moving in the specified direction *Dx*/*Dy* (using the *route* and *move* activities), assuming a two-dimensional grid network topology. The initial direction was chosen randomly by the SEN parent agent. Moving of the agent consumes energy, which decrements the *ENERGY* tuple value (lines 34,35). Each time the agent moves it must previously transfer its virtual energy (stored in the body variable *Energy*) to the new destination node. Furthermore, the migration of the agent itself consumes energy (*EnergySend*). After the agent arrived at the destination node, it updates the local *ENERGY* tuple by the amount of the transferred energy (reduced by the transmission efficiency *EnergyEff*), shown in lines 45-47.

Help Agent

The percept activity of the *Help* agent subclass decides what happens with the agent. If the agent finds bad nodes along the path to the desired exploration boundary (limited by *Dx*/*Dy*), it will donate energy to these nodes to ensure the operational vividness of this bad node. If the boundary of the exploration region is reached (*Dx*/*Dy*=0), the agent dies (and leave its energy deposit at this last node). If the agent founds a good node, it will create a SEM agent of subclass *Deliver* to pass energy back to the root node, and then continues moving towards the boundary of the search range by entering the *route* activity again. Each good node found along its path increases the age, each bad node decreases the age of the agent. If the agent reaches its end of live (*Age*=0), then he dies, too.

Deliver Agent

The deliver agent propagates energy from a good node to a bad node, which destination is specified by *Dx*/*Dy*.If the help-on-way (*HelpOnWay*) behaviour is activated, it will deliver energy to bad nodes along its path first, and dies finally.

Distribute Agent

The distribute agent has the goal to deliver energy to bad nodes (whose energy is below *DistThres*) along the path to the destination or finally to the destination node specified by *Dx*/*Dy* (regardless if this is a bad or good node).

epubli, ISBN 9783746752228 (2018)

Request and Reply Agent

The request and reply subclass agents are used to transfer energy between nodes directly on a peer-to-peer basis. There is no help-on-way behaviour activated.

Alg. 9.4 *AAPL model of the SEM Agent Behaviour*

```
1   δ: { NORTH, SOUTH, WEST, EAST, ORIGIN }                set of directions
2   kind : { REQUEST, REPLY, DELIVER, HELP, DISTRIBUTE }   set of agent goals
3   state : { TRAVEL, AWAIT, DYING, HELP, IAMHERE, SLEEP } set of agent states
4
5   ψ SEM: (Dx, Dy, De, Energy, Kind,
6             SEMSet = ( record of SEM parameter settings
7               EnergieThres, EnergyEff, EnergyHelp, DistrThres, SendThres,
8               EnergyDeposit, HelpOnWay, HelpTime, AgingGood, AgingBad,
9               EnergySend )) → {
10    ξ: { TIMER }  set of signals
11
12    Σ: {Dir,Dx1,Dy1,Age,AgentState}
13    σ: {en,tryagain,qos}
14
15    α init: {
16      Dir ← ORIGIN; AgentState ← TRAVEL;
17    }
18
19    α route: {
20      if Dx > 0 & ?Λ(WEST) then
21        decr(Dx); Dir ← WEST;
22      elsif Dx < 0 & ?Λ(EAST) then
23        incr(Dx); Dir ← EAST;
24      elsif Dy < 0 & ?Λ(SOUTH) then
25        decr(Dy); Dir ← SOUTH;
26      elsif Dy > 0 & ?Λ(NORTH) then
27        incr(Dy); Dir ← NORTH;
28      else AgentState ← DYING;
29    }
30
31    α move: {
32      tryagain ← false
33      ∇%(ENERGY,en?)
34      if en > EnergieThres then
35        ∇⁻(ENERGY,en?)
36        ∇⁺(ENERGY,en-Energy-EnergySend)
37        transfer(DIR,Energy) transfer energy of this agent to neighbour node
38        ⇔(DIR)
39      else
40        state ← SLEEP;
41        τ+(TIMER,1sec);
42    }
43
```

```
44    α arrive: {
45      ∇⁻(ENERGY,en?)
46      en ← en+Energy*EnergyEff  consider energy loss
47      ∇⁺(ENERGY,en) update local ENERGY value,
48                    energy was already transferred
49    }
50
51    α dying: {
52      kill($self)
53    }
54
55    ξ TIMER: { tryagain ← true }
56
57    Π: {
58      init → route
59      route → dying | AgentState=DYING
60      route → move  | AgentState=TRAVEL
61      move → move   | AgentState=SLEEP and trygain
62      move → arrive | AgentState=TRAVEL
63    }
64
65    φ Request {
66      ↓: {en,Dx,Dy,Dx1,Dy1,De,SEMSet,AgentState,Kind}
67      ↓: {arrive,dying,route}
68
69      α percept: {
70        if (Dx,Dy) = (0,0) then AgentState <- IAMHERE;
71      }
72
73      α action: {
74        ∇%(ENERGY,en?)
75        if en-De > SendThres then
76          eval(new SEM(-Dx1,-Dy1,0,De,REPLY,SEMSet))
77          Age ← 0
78          AgentState ← DYNING
79        else
80          ∇%(QOS,qos?)
81          decr(Age)
82          if Age = 0 then AgentState ← DYING
83          if qos < 0.9 then AgentState ← DYING
84      }
85
86      π: {
87        arrive → percept | Kind=REQUEST
88        percept → action | AgentState = IAMHERE
89        percept → route | AgentState = TRAVEL
90        percept → dying | AgentState = DYING
91        action → dying | AgentState = DYING
92        action → route | AgentState ≠ DYING
93      }
94
```

```
95    }
96
97    φ Reply: {
98      ↓ Dx,Dy,AgentState,Kind
99      ↓ dying,route
100
101     α percept: {
102       if (Dx,Dy) = (0,0) then AgentState ← IAMHERE;
103     }
104
105     α action: {
106       Age ← 0
107       AgentState ← DYING
108     }
109
110     π: {
111       arrive → percept | Kind=REPLY
112       percept → action | AgentState = IAMHERE
113       percept → route | AgentState = TRAVEL
114       percept → dying | AgentState = DYING
115       action → dying | AgentState = DYING
116     }
117   }
118
119   φ Deliver: {
120     ↓: {en,duty,SEMSet,Energy,Dx,Dy,Age,Kind,AgentState}
121     ↓: {arrive,dying,route}
122
123     α help: {
124       ∇⁻(ENERGY,en?)
125       ∇%(DUTY,duty?)
126       if duty>0 and en < EnergyHelp then
127         incr(en,duty)
128         decr(Energy,duty)
129       ∇⁺(ENERGY,en)
130     }
131
132     α percept: {
133       ∇%(ENERGY,en?)
134       if (HelpOnWay ∧ en < EnergyHelp) ∨ (Dx,Dy)=(0,0) then
135         AgentState ← IAMHERE;
136     }
137
138     α action: {
139       Age ← 0
140       AgentState ← DYING
141     }
142
143     π: {
144       arrive → help | Kind=DELIVER
145       help → percept
```

```
146       percept → action | AgentState = IAMHERE
147       percept → route | AgentState = TRAVEL
148       percept → dying | AgentState = DYING
149       action → dying | AgentState = DYING
150     }
151   }
152
153   φ Help: {
154     ↓: {en,time,De,SEMSet,Dx,Dx1,Dy,Dy1}
155     ↓: {arrive,dying,route}
156
157     α percept: {
158       ∇%(ENERGY,en?)
159       if en > De+EnergyDeposit then
160         AgentState <- IAMHERE;
161         decr(Dx1,Dx)
162         decr(Dy1,Dy)
163       elsif (Dx,Dy) = (0,0) then
164         AgentState ← DYING
165         Age ← 0
166     }
167
168     α action: {
169       ∇%(ENERGY,en?)
170       if en > (De+EnergyDeposit) then
171         eval(new SEM(-Dx1,-Dy1,0,De,DELIVER,SEMSet))
172         incr(Age,AgingGood)
173       else
174         incr(Age,AgingBad)
175       if Age=0 then AgentState ← DYING
176     }
177
178     π: {
179       arrive → percept | Kind=HELP
180       percept → action | AgentState = IAMHERE
181       percept → route | AgentState = TRAVEL
182       percept → dying | AgentState = DYING
183       action → route | AgentState ≠ DYING
184       action → dying | AgentState = DYING
185     }
186
187   }
188
189   φ Distribute: {
190     ↓: {en,DistThres,Dx,Dy,Kind,AgentState}
191     ↓: {arrive,dying,route}
192
193     α percept: {
194       ∇%(ENERGY,en?)
195       if en < DistrThres ∨ (Dx,Dy)=(0,0) then
196         AgentState ← IAMHERE;
```

```
197       }
198
199       α action: {
200         AgentState ← DYING
201         Age ← 0
202       }
203
204       π: {
205         arrive → percept | Kind=DISTRIBUTE
206         percept → action | AgentState = IAMHERE
207         percept → route | AgentState = TRAVEL
208         percept → dying | AgentState = DYING
209         action → dying | AgentState = DYING
210       }
211     }
212 }
```

9.4.2 The Immobile Sensor Node Energy Management Agent

The mobile SEM agent is created by a non-mobile sensor node energy management agent SEN. The SEM agents interact with SEN agents through the tuple-space on each node, using the `ENERGY` tuple, as shown in Algorithm 9.5. The behaviour of the SEN agent consists mainly of collecting of local available energy by energy harvesting, and to monitor the local energy deposit in relation to the energy requirements, which determine the actually available service a node can provide (e.g., agent processing, routing of messages). There are two goals of the SEN agent to be fulfilled by instantiating a SEM agent:

Sensor Node Vividness
: The local node requires energy from the neighbourhood to ensure the service of the node, i.e., it is an individual goal.

Sensor Network Vividness
: The local node want to distribute energy to the neighbourhood to ensure the service quality of the sensor network, i.e., it is a system goal.

It is assumes that each sensor node is equipped with an energy harvester module, which is capable to collect energy from the environment. The harvested energy amount can be checked by the `harvest` function returning the amount of energy collected since the last call. If the energy is low, help agents are sent out (actually a bad node), if the energy is high (actually a good node) some portion of the energy deposit is distributed to the neighbourhood. The radius limiting the mobility of help and distribution agents is given by the SEN parameter *DistRange*. The help and distribute agents are created in the *service* activity (lines 34-53).

Alg. 9.5 *AAPL model of the non-mobile Sensor Node Energy Management (SEN) Agent Behaviour*

```
ψ SEN : (
    SENSet = (ProbeActivity,EnergyHigh,EnergyVeryHigh
              EnergyDonation,ActivityCost,
              EnergyThres,EnergySendThres,
              HelpEnergy,MessageEnergy,
              SleepTime,DistRange)) → {
  S: {Energy,EnergyFlow,Time,UpTime,DownTime}
  s: {dir,dirs,tryagain,lasttime}
  ξ: {SLEEP}

  SEMSet = ( record of SEM parameter settings
             EnergieThres, EnergyEff, EnergyHelp, DistrThres, SendThres,
             EnergyDeposit, HelpOnWay, HelpTime, AgingGood, AgingBad,
             EnergySend )

  α init: {
    ..
  }

  α collect: {
    rd(TIME,Time?);
    in(ENERGY,Energy?)
    incr(Energy,harvest())   Update Energy value
    out(ENERGY,Energy)
    if Energy < EnergySendThres then
      tryagain ← false
      incr(DownTime,Time-lasttime)
      τ+(SleepTime,SLEEP)
    else
      incr(UpTime,Time-lasttime)
    lasttime ← Time
  }

  α service: {
    in(ENERGY,Energy?)
    decr(Energy,ActivityCost)
    out(ENEGRY,Energy)
    dirs ← {}
    ∀ testdir ∈ ω do if ?Λ(testdir) then dirs ← dirs @ {testdir}
    if ℜ{0..100} < ProbeActivity then
      if Energy > EnergyHigh then
        dir ← ℜ(dirs)
        eval(new SEM(δ.x(dir)*DistRange,δ.y(dir)*DistRange,0,
                     EnergyDonation,DISTRIBUTE,SEMSet))
      if Energy > EnergyVeryHigh then
        dir ← ℜ(dirs)
        eval(new SEM(δ.x(dir)*DistRange,δ.y(dir)*DistRange,0,
                     EnergyDonation*2,DISTRIBUTE,SEMSet))
```

epubli, ISBN 9783746752228 (2018)

```
49          if Energy < EnergyThres then
50            dir ← ℜ(dirs)
51            eval(new SEM(δ.x(dir)*DistRange,δ.y(dir)*DistRange,HelpEnergy,
52                         MessageEnergy,HELP,SEMSet))
53      }
54
55      α evaluate: {
56        Evaluate energy harvesting, distribution, and collection
57        from other nodes and adapt SENSet parameters to fulfill
58        local and global lifeness goals
59      }
60
61      ξ SLEEP: { tryagain ← true }
62
63      Π: {
64        init → collect
65        collect → collect | tryagain
66        collect →» service | Energy > EnergySendThres
67        service → adapt
68        evaluate → collect
69      }
70  }
```

9.5 Further Reading

1. H. He, *Self-Adaptive Systems for Machine Intelligence*, John Wiley & Sons, Ltd, 2011, ISBN 9780470343968
2. C. Gershenson, *Design and Control of Self-organizing Systems*, Vrije Universiteit Brussel, CopIt ArXives, 2007.
3. J. Liu, *Autonomous Agents and Multi-Agent Systems - Explorations in Learning, Self-Organization and Adaptive Computation*, World Scientific Publishing, 2001, ISBN 9810242824

epubli, ISBN 9783746752228 (2018)