DOI: 10.1016/j.protcy.2014.09.039 **Procedia Technology, SysInt 2014**

Structural Health and Load Monitoring with Material-embedded Sensor Networks and Self-organizing Multi-Agent Systems

Stefan Bosse, Armin Lechleiter

University of Bremen, Department of Mathematics & Computer Science,
ISIS Sensorial Materials Scientific Centre, Germany

Abstract One of the major challenges in Structural Health Monitoring and load monitoring of mechanical structures is the derivation of meaningful information from sensor data. This work investigates a hybrid data processing approach for material-integrated SHM and LM systems by using self-organizing mobile multi-agent systems (MAS), with agent processing platforms scaled to microchip level which offer material-integrated real-time sensor systems, and inverse numerical methods providing the spatial resolved load information from a set of sensors embedded in the technical structure. Inverse numerical approaches usually require a large amount of computational power and storage resources, not suitable for resource constrained sensor node implementations. Instead, off-line computation is performed, with on-line sensor processing by the agent system.

Keywords: Structural Health Monitoring, Sensor Network, Mobile Agent, Heterogeneous Networks, Embedded Systems, Inverse Numerical Computation

1. Introduction

Structural Health Monitoring (SHM) of mechanical structures allows to derive not just loads, but also their effects to the structure, its safety, and its functioning from sensor data. A load monitoring system (LM) can be considered as a sub-class of SHM, which provides spatial resolved information about loads (forces, moments, etc.) applied to a technical structure.

One of the major challenges in SHM and LM is the derivation of meaningful information from sensor input. The sensor output of a SHM or LM system reflects the lowest level of information. Beside technical aspects of sensor integration the main issue in those applications is the derivation of a mapping function $F_m(S)$ which basically maps the raw sensor data input S, an n-dimensional vector consisting of n sensor values, to the desired information I, an m-dimensional result vector (see Fig. **1**).

Basically there are two different information extraction approaches: (I.) First those methods based on a mechanical and numerical model of the technical structure, the device under test (DUT), and the sensor, and (II.) second those without any or with a partial physical model. The latter class can profit from artificial intelligence which usually bases on classification algorithms derived from supervised machine learning or pattern recognition using, for example, self-organizing systems like multi-agent systems with less or no a-priori knowledge of the environment.

One common approach in SHM is the correlation of measured data resulting from an induced stimuli at run-time (system response) with data sets retrieved from an initial (first-hand) observation, which makes it difficult to select damage relevant features from the measurement results. Other variants are based on statistical methods or neural network approaches. We refer to **[12][13][14]** and **[15]** for examples illustrating the variety of possible approaches.

Inverse methods generally belong to the first class of approaches since they are based on a mechanical model T of the technical structure mapping loads to sensor signals. Given a sensor signal S, inverse methods try to stably "invert" the mapping T, that is, to find a stable solution x to the problem $Tx = S$. Since measured signals and the underlying physical model always contain numerical and modeling errors, inverse methods do not attempt to find an exact solution to the latter equation. Indeed, inversion problems, in particular those with incomplete data, are usually extremely ill-conditioned, meaning that small errors in the signals or the model lead to huge errors in any "solution" gained by such a naïve approach. Instead, inverse methods try to stabilize the inversion process, using, e.g., one of the following techniques: (1) Pick amongst all solutions to $Tx = S$ the one that minimizes a certain functional - the simplest functional to minimize would be the 2-norm of x, more complicated variants add a penalty term to this norm. (2) Alternatively, consider any iterative method driving the 2-norm of the residual Tx_k-S to zero during the iteration and stabilize the inversion by stopping the iteration when the norm of the residual is about the magnitude of the expected signal and modeling error.

A well-known inversion method in the first class is Tikhonov regularization minimizing a quadratic functional. A well-known iterative method belonging to the second class is for instance the Landweber iteration, but also the conjugate gradient iteration can be exploited for SHM or LM. The disadvantage of inverse methods usually is their cost in terms of computing time and memory requirements which definitely is a drawback for material-integrated SHM and LM systems. The possibly high computing time and memory requirements for pre-computations before actually launching the monitoring device due to the physical and numerical model are nowadays becoming less important due to advanced numerical methods and increasing computational power. Reliable distributed data processing in sensor networks using multi-agent systems (MAS) were recently reported in **[3]** and employed for SHM applications in **[4]**. Adaptive and learning behaviour of MAS, central to agent model, can aid to overcome technical unreliability and limitations **[7]**. Artificial intelligence and machine learning can be used in sensorial materials without a predictive mechanical model **[5]**.

This work investigates a hybrid data processing approach for material-integrated SHM and LM systems by using self-organizing and event-driven mobile multi-agent system (MAS), with agent processing platforms scaled to microchip level which offer material-integrated real-time sensor systems, and inverse numerical methods providing the spatially resolved load information from a set of sensors embedded in the technical structure. Inverse numerical approaches usually require a large amount of computational power and storage resources, not suitable for resource constrained sensor node implementations. Instead, off-line computation is performed, with on-line sensor processing by the agent system. Commonly off-line computation operates on a continuous data stream requested by the off-line processing system delivering sensor data continuously in fixed acquisition intervals, resulting in high communication and computational costs. In this work, the sensor pre-processing MAS delivers sensor data event-based if a change of the load was detected (feature extraction), reducing network activity and energy consumption significantly.

Multi-agent systems can be used for a decentralized and self-organizing approach of data processing in a distributed system like a sensor network **[9]**, enabling information extraction, for example, based on pattern recognition, decomposing complex tasks in simpler cooperative agents.

In this work the behaviour of mobile agents are modeled with an activity-transition graph (ATG) which is implemented with the Activity-based Agent Programming Language AAPL **[1]**, which can be compiled to machine code and executed on a virtual machine part of each sensor node. The ATG can be modified at run-time by the agent itself using dedicated AAPL transformation statements, implemented with code morphing techniques provide by the VM. Agents carrying the code, data, and already applied modifications, are capable to migrate in the network between nodes **[2]**.

The agent processing platform used for the execution of agents is a programmable stack-based virtual machine, with support for code morphing and code migration. This VM approach offers small sized agent program code, low system complexity, and high system performance. The agent platform VM can be implemented directly in hardware with a System-on-Chip design. Agents processed on one particular node can interact by using a tuple-space server provided by each sensor node. Remote interaction is provided by signals carrying data which can cross sensor node boundaries.

This approach provides a high degree of computational independency from the underlying platform and other agents, and enhanced robustness of the entire heterogeneous environment in the presence of node, sensor, link, data processing, and communication failures. Support for heterogeneous networks considering hardware (System-on-Chip designs) and software (microprocessor) platforms is covered by one design and synthesis flow including functional behavioural simulation.

The mechanical model of the structure under investigation allows in particular the pre-computation of a sufficiently accurate discretization of the forward mapping $\boldsymbol{T}$ linking loads with measured signals. Moreover, this pre-computation allows to associate to each sensor an individual signal level that might potentially be critical for the entire structure.

Hence, when a load change that is potentially critical is detected by one the material-integrated sensors, the signals measured by all sensors are propagated to an exterior CPU. An alternative way is that merely those sensors noting a critical load change start to propagate their signals to the exterior CPU. The propagated signals are then fed into a regularization scheme that is able to stably invert signals into loads. Several algorithms can be used at this point: A classical and well-known inversion method is Tikhonov regularization, minimizing the quadratic Tikhonov functional

$$x \mapsto \|Tx - S\|_2^2 + \alpha \|x - x_0\|_2^2, \qquad (1)$$

where x_0 is some a-priori guess and α is a (small) regularization parameter.

The minimum x is the unique solution of the linear system

$$T^{*}Tx + \alpha x = T^{*}g, \tag{2}$$

where ***T**** denotes the transpose of ***T***. The solution to this system is hence computed rapidly using a calculated singular value decomposition of the matrix ***T***. The disadvantage of this inversion scheme is that reconstructions of discontinuous loads, in particular with small support, are smoothed out which makes the precise location of the support of a load difficult. Several iterative inversion techniques such as the steepest descent method or the conjugate gradient method applied to ***T*Tx= T*g*** avoid this disadvantage. Further, they merely require the ability to compute matrix-vector products and a (cheap stopping) rule to stabilize the inversion. The class of iterative inversion methods also includes the so-called Landweber iteration and its variant, the so-called iterative soft shrinkage. The disadvantage of the latter two techniques is their slow convergence, and the huge number of iterations are necessary to compute accurate inversions **[10][11]**.

Combining self-organizing multi-agent systems (soMAS) with inverse numerical methods into a hybrid data processing approach has several advantages: First, the (possibly distinct) critical level for an individual sensor signal can be pre-computed for each sensor position individually. Second, depending on the a-priori knowledge on the expected loads on the structure, a suitable regularization technique can be chosen as inversion method, promoting specific features of the expected loads. Third, the sensor positions themselves might well be optimized with respect to the last two points, aiming for sensor positions that maximize the detectability of critical loads and/or sensor positions that maximize the quality of load reconstructions from sensor signals.

This work introduces some novelties compared to other data processing and agent platform approaches:

- Sensor signal pre-processing at run-time inside the sensor network by using multi-agent system.
- Agent mobility crossing different execution platforms in mesh-like networks and agent interaction by using tuple-space databases and global signal propagation aid solving data distribution and synchronization issues in the design of distributed sensor networks.
- Event-based sensor data distribution and pre-computation with agents reduces communication and overall network activity resulting in reduced energy consumption.
- Off-line inverse numerical computation of pre-processed sensor data allows the calculation of the system response based on data from prior FEM simulations of the technical structure.

The next sections introduce the activity-based agent processing model, available mobility and interaction, and the proposed agent platform architecture related to the programming model. Finally, the sensor signal processing algorithms and the used numerical methods are introduced and validated with simulation results.

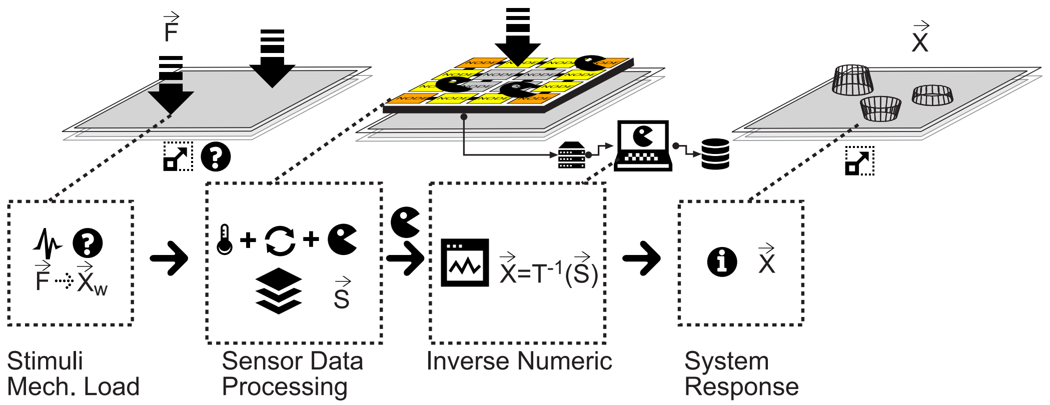


Fig. 1. External forces applied to a mechanical structure lead to an initially unknown deformation of the material based on the internal forces. A material-integrated active Sensor Network merging sensors, electronics, data processing, and communication, together with mobile agents can be used to monitor relevant sensor changes with an advanced event-based distribution behaviour. Inverse numerical methods

can compute finally the material response (X_w: unknown system response for externally applied load, S: measured sensor stimuli response, X: computed system response).

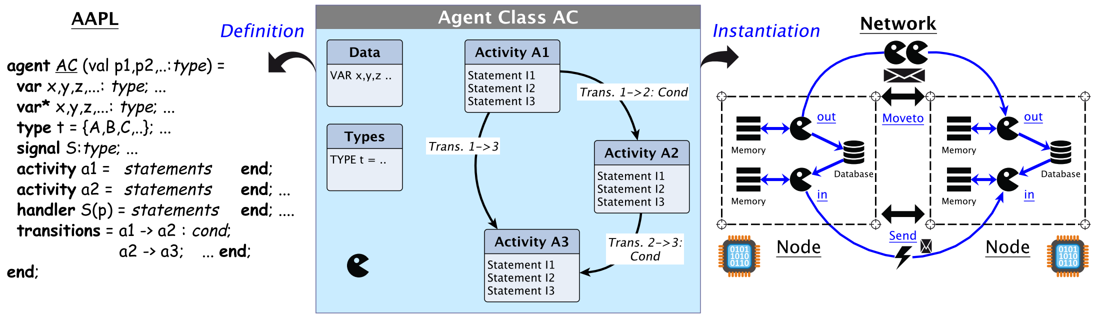
2. The Activity-based Agent Model

The implementation of mobile multi-agent systems for resource constrained embedded systems with a particular focus on microchip level is a complex design challenge. High-level agent programming and behaviour modelling languages can aid to solve this design issue. Activity-based agent models can aid to carry out multi-agent systems on hardware platforms.

The behaviour of an activity-based agent is characterized by an agent state, which is changed by activities. Activities perform perception, plan actions, and execute actions modifying the control and data state of the agent. Activities and transitions between activities are represented by an activity-transition graph (ATG). The activity-based agent-orientated programming language AAPL **[1]** was designed to offer modelling of the agent behaviour on programming level, defining activities with procedural statements and transitions between activities with conditional expressions (predicates). Though the imperative programming model is quite simple and closer to a traditional PL it can be used as a common source and intermediate representation for different agent processing platform implementations (hardware, software, simulation) by using a high-level synthesis approach.

Definition: There is a multi-agent system (MAS) consisting of a set of individual agents $\{A_1,A_2,..\}$. *There is a set of different agent behaviours, called classes* $\mathcal{C}=\{AC_1, AC_2,..\}$. *An agent belongs to one class. In a specific situation an agent* A_i *is bound to and processed on a network node* $N_{m,n}$ *(e.g. microchip, computer, virtual simulation node) at a unique spatial location (m,n). There is a set of different nodes* $\mathcal{N}=\{N_1, N_2,..\}$ *arranged in a mesh-like network with peer-to-peer neighbour connectivity (e.g. two-dimensional grid). Each node is capable to process a number of agents* $n_i(AC_i)$ *belonging to one agent behaviour class* AC_i*, and supporting at least a subset of* $\mathcal{C}' \subseteq \mathcal{C}$. *An agent (or at least its state) can migrate to a neighbour node where it continues working.*

Therefore, the agent behaviour and the action on the environment is encapsulated in agent classes, with activities representing the control state of the agent reasoning engine, and conditional transitions connecting and enabling activities. Activities provide a procedural agent processing by a sequential execution of imperative data processing and control statements. Agents can be instantiated from a specific class at run-time. A multi-agent system composed of different agent classes enables the factorization of an overall global task in sub-tasks, with the objective of decomposing the resolution of a large problem into agents in which they communicate and cooperate with one other.



```
agent AC (val p1,p2,..:type) =
  var x,y,z,...: type; ...
  var* x,y,z,...: type; ...
  type t = {A,B,C,..}; ...
  signal S:type; ...
  activity a1 =   statements     end;
  activity a2 =   statements     end; ...
  handler S(p) = statements     end; ....
  transitions = a1 -> a2 : cond;
                a2 -> a3;    ... end;
end;
```

Fig. 2. Agent behaviour programming level with activities and transitions (AAPL, left); agent class model and activity-transition graphs (middle); agent instantiation, processing, and agent interaction on the network node level (right) **[16]**.

The activity-graph based agent model is attractive due to the proximity to the finite-state machine model, which simplifies the hardware implementation.

An activity is activated by a transition depending on the evaluation of (private) agent data (conditional transition) related to a part of the agents belief in terms of *BDI* architectures, or using unconditional transitions (providing sequential composition), shown in Fig. **2**. Each agent belongs to a specific parameterizable agent class *AC*, specifying local agent data (only visible for the agent itself), types, signals, activities, signal handlers, and transitions.

The *AAPL programming language* (detailed description in **[1]**) offers statements for parameterized agent instantiation, like the creation of new agents and the forking of child agents, using the `new(args)` and `fork(args)` statements, respectively. Furthermore, agent interaction by using synchronized Linda-like tuple database space access and signal propagation (messages carrying simple data delivered to asynchronous executed signal handlers), agent mobility (migration using the `moveto` statement), and statements for ATG transformations and composition. Access of the tupel space is granted by using `in(TP)`, `rd(TP)`, `rm(TP)`, `exist?(TP)` and `out(T)` primitives (`T`: n-dimensional value tuple, `TP`: n-dimensional tuple with value patterns). A signal SIG can be sent to an agent with the ID identifier by using the `send(ID,SIG,ARG)` statement.

3. Sensing with Multi-Agent Systems

Large scale sensor networks with hundreds and thousands of sensor nodes require 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.

In this work, three different data processing and distribution approaches are used and implemented with agents, leading to a significant decrease of network communication activity and a significant increase of reliability and 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.

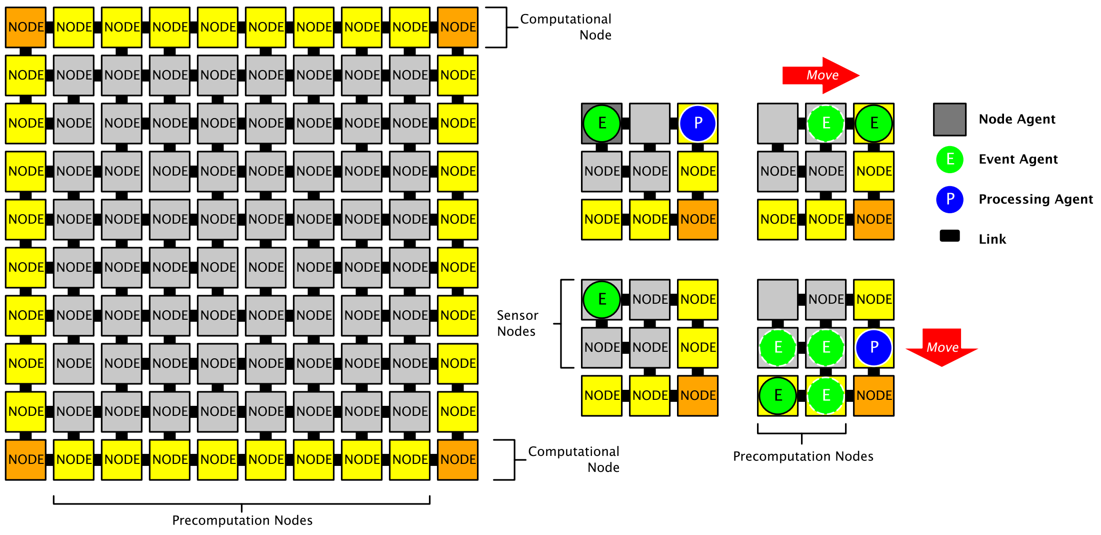


Fig. 3. The logical view of a sensor network with a two-dimensional mesh-grid topology (left) and examples of the population with different mobile and immobile agents (right): event deliver, node, and computational processing agents. The sensor network can contain missing or broken links between neighbour nodes.

Sensor nodes arranged in a two-dimensional grid network (as shown in Fig. **3**) provide spatially resolved and distributed sensing information of the surrounding technical structure, for example, a metal plate. 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. The computation of the material response requires high computational power of the processing unit, which cannot offered by down-scaled single micro-chip platforms. For these reasons, sensor nodes using mobile agents to deliver their sensor data to dedicated computational nodes located at the edges of the sensor network, shown in Fig. **3**, discussed in detail in the following sub-sections. The computational nodes 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 real sensor nodes are micro-chip platforms embedded in the technical structure material, for example, using thinned silicon technologies.

Fig. **4** gives an overview of the partition of the complete sensor data processing system in different agent classes, discussed in the following sections. Some classes are super classes composed of sub-classes (e.g. the computer and the explorer class). A sensor node is managed by a node agent, which creates and manages a sampling and a sensing agent, responsible for local sensor processing. The network class is only used in the simulation environment and has the purpose to create and initialize the sensor network world and to control the simulation using monte-carlo techniques.

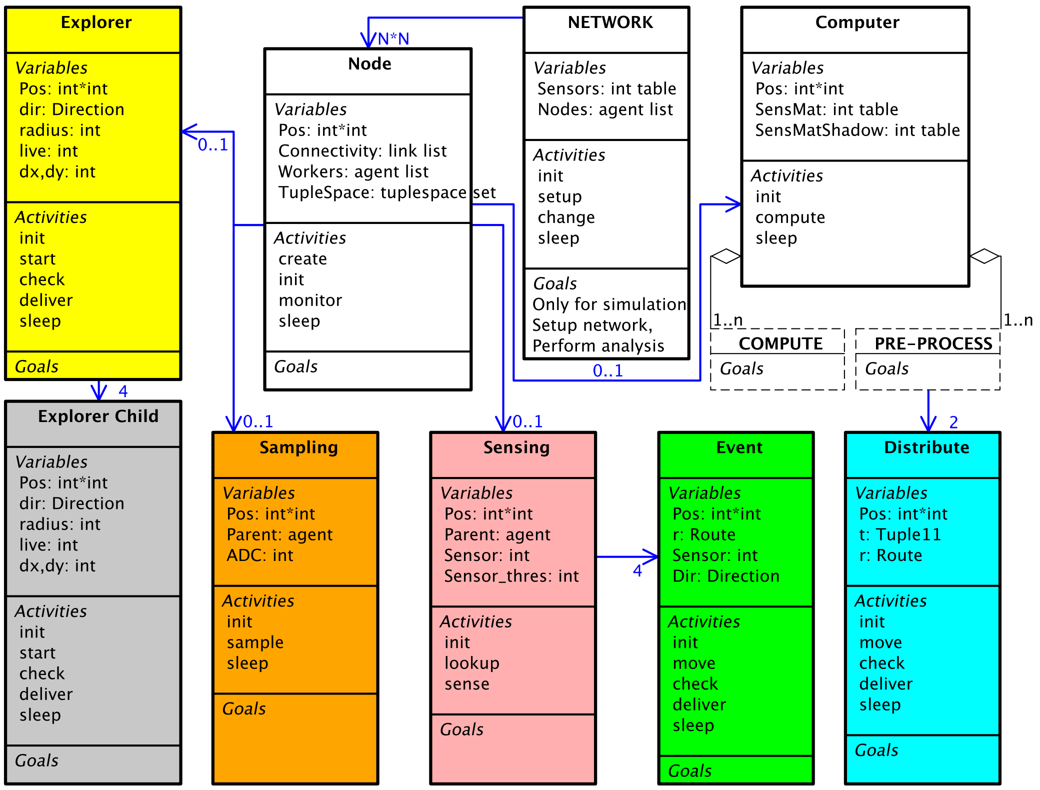


Fig. 4. Overview of different agent classes used for the sensor signal and data processing in the network and their relationships.

3.1. Event-based Sensor Data Distribution and Processing

The computation of the system response information requires basically the complete sensor signal matrix 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 inverse numerical computations, explained in Section **4.**.

The sensor processing uses both stationary (non-mobile) and mobile agents carrying data, illustrated in Fig. **5** (left). 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 pre-processes and interprets the acquired sensor data. 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. Each pre-computation node stores a row or a columns of the sensor matrix S. If their data changes, the pre-computation nodes will sent out two mobile distribution agents, delivering a row or column of *S* to the final computation nodes, located at the edges of the sensor network. The behaviour model for each agent class is summarized in Alg. **2** (see Appendix **A.**).

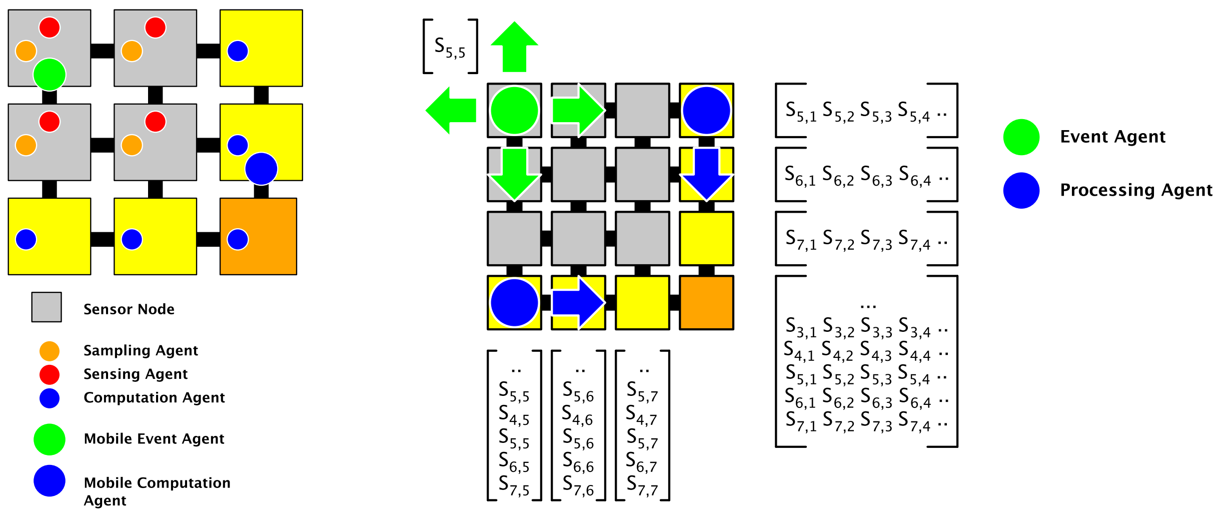


Fig. 5. Left: Different mobile and immobile agent classes used for the event-based sensor signal distribution in the sensor network. Right: Sensor data distribution with event and processing agents: A sensor node which detects a significant change of the sensor values creates event agents which are sent in all four directions to the network boundary (pre-computation nodes). Pre-computation nodes collect and process updated sensor data for a specific row or column of the sensor matrix S. The pre-computation nodes send out processing agents (in opposite direction) delivering updated rows/columns to the (four) computational nodes, finally performing the complex numerical computations.

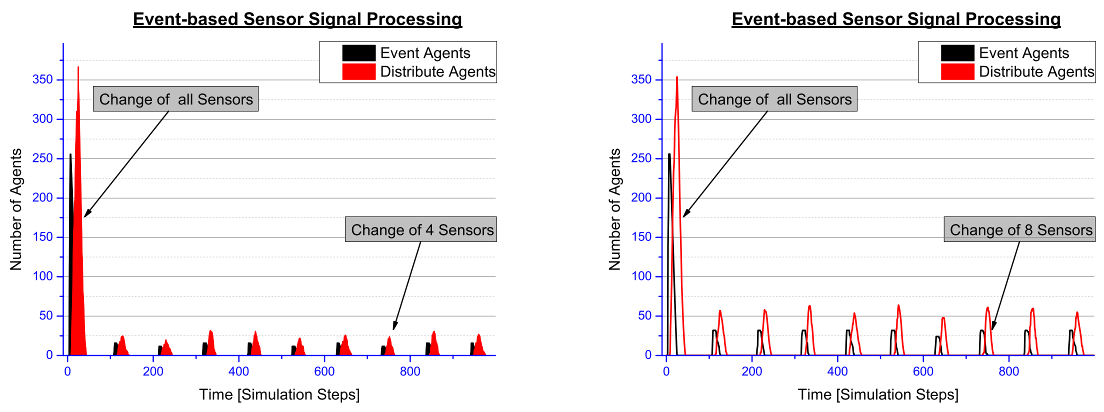


Fig. 6. Analysis results of the agent population obtained from the multi-agent simulation of the event-based sensor data processing. Left: clusters of four sensors are stimulated periodically with different center position, Right: cluster size is 8 sensors.

Fig. **5** (right) illustrates the aforementioned event-based sensor data distribution in the network, and the relationship of parts of the sensor matrix *S* with the nodes in the network.

Fig. **6** shows the population of sensor and computation nodes with these agents retrieved from simulation using the Multi-Agent simulation environment shell SeSAm **[6]**. 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 a 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 Fig. **6** 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.

3.2. Adaptive and robust Path finding with Random and Attractive Walk Behaviour

In the previous scenario all communication links between neighbour nodes were fully functional, and an event or distribute agent can travel to its destination along a straight line path. But under real conditions broken links (resulting from technical failures) can be expected. In this case, an alternative non-straight path from a source to a destination node must be found by each mobile agent.

A hybrid path finding approach consisting of three different routing behaviours is used, shown in Alg. **3** (see Appendix **A.**). First the normal routing is tried (using the `route_normal` function) with the goal to move the agent on a straight line in the originally given direction (attraction behaviour). The Δ-vector is updated accordingly each time an agent migrates to a neighbour node. If this is not possible due to a link failure (the `link?(DIR)` function returns a false value), the agent is moved to another direction which is selected randomly, performed in the `route_opposite` function, which leads to a travel away from the destination (divergence behaviour). The divergence is stored in the Γ-vector, similar to the Δ-vector, but only changed in the divergence case. The next time the normal routing is tried first again. If there is a significant divergence from the original path, the `route_relax` function is applied to bring the agent back to its original intended way (direction).

Simulation results obtained from different network situation using Monte-Carlo simulation are shown in Fig. **7**. In the case of 30% broken links a slight increase of the traveling 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 traveling time is significant, compared with the results from Fig. **6**.

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 a 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, observed by simulation results shown in Fig. **8**.

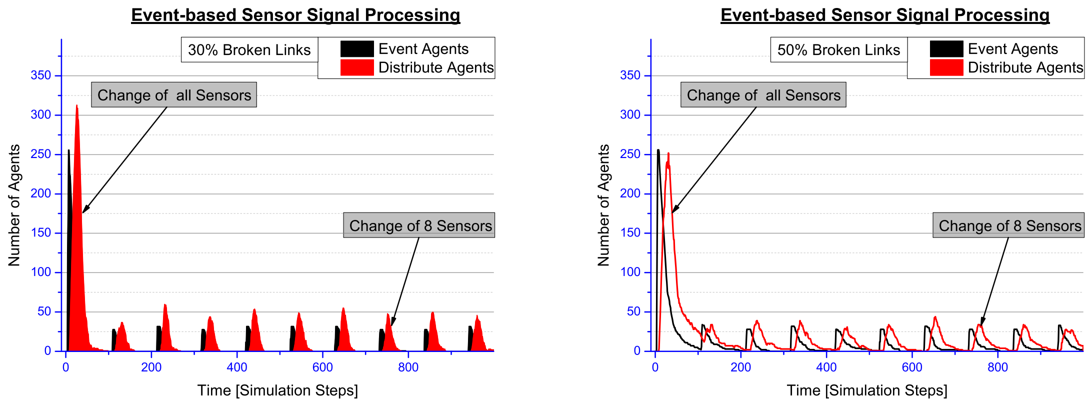


Fig. 7. Analysis results of the agent population obtained from the multi-agent simulation of the event-based sensor data processing. Left: With 30% broken links, Right: With 50% broken links, (Both: cluster size is 8 sensors)

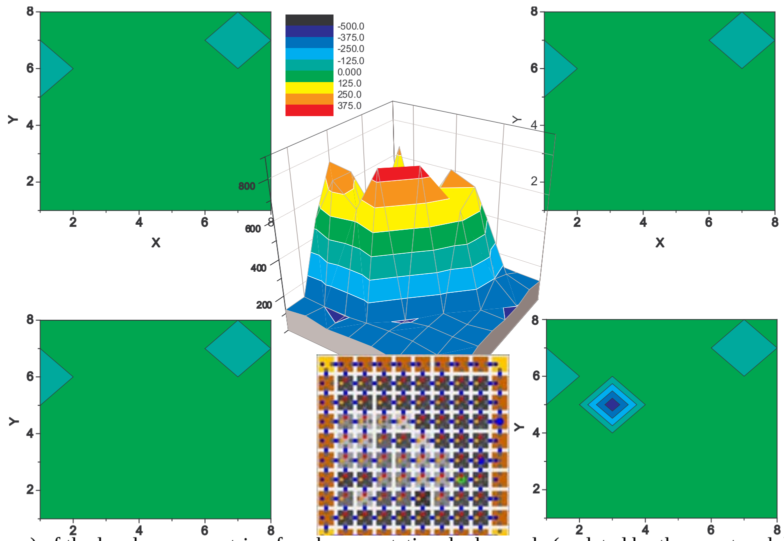


Fig. 8. Differences (error) of the local sensor matrix of each computational edge node (updated by the event and distribute agents) compared with the original sensor matrix (shown in the center) of the sensor network with 35% broken links (shown in the middle, bottom) and a cluster size of 8 sensors (after 9 cumulative stimulations).

3.3. Feature Detection with Self-organizing Multi-Agent Systems and Interval Voting

The event-based sensor data distribution assumes well operating sensors. Faulty or noisy sensors can disturb the further data processing algorithm significantly. It is necessary to isolate noisy from well operating sensors. Usually sensor values are correlated within a spatially close region. The goal of the following MAS is to find extended correlated regions of increased sensor intensity (compared to the neighbourhood) due to mechanical distortion resulting from externally applied load forces. A distributed directed diffusion behaviour and self-organization (see Fig. **9**) are used, derived from the image feature extraction approach (proposed by **[9]**). 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.

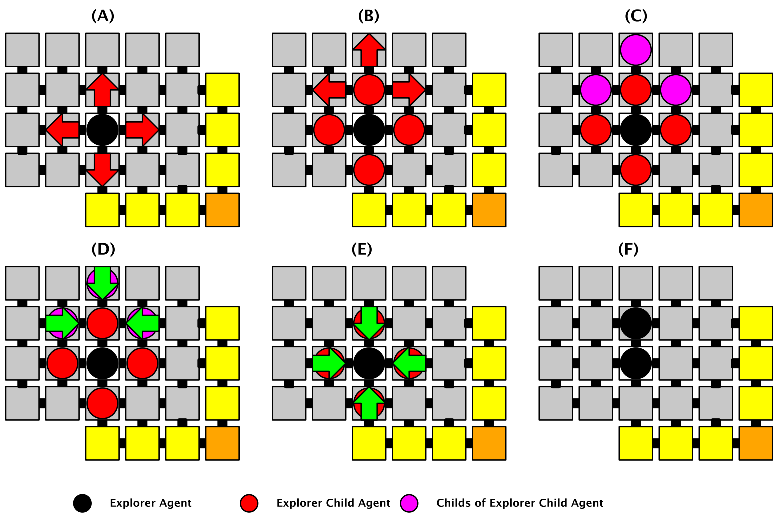


Fig. 9. Feature marking of stimulated sensor clusters by using explorer agents investigating the neighbourhood of their current position using forked child agents. On successful feature detection explorer agents are replicated, otherwise they diffuse to the neighbourhood to find features.

The feature detection is performed by the mobile *exploration agent*, which supports two main different behaviours: diffusion and reproduction. 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, which 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 Eq. **3**.

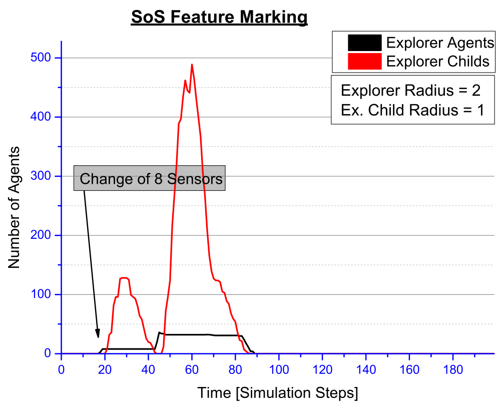
$$H(i,j) = \sum_{s=-R}^{R}\sum_{t=-R}^{R}\{\|S(i+s,j+t)-S(i,j)\| \le \delta\} \quad (3)$$

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 square 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 (diff. is smaller than the parameter δ), H is incremented by one.

If the H value is within a parameterized 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).



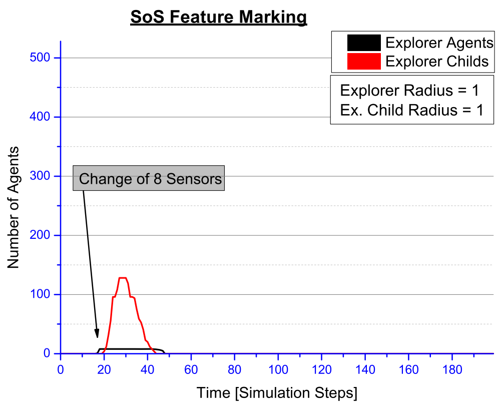
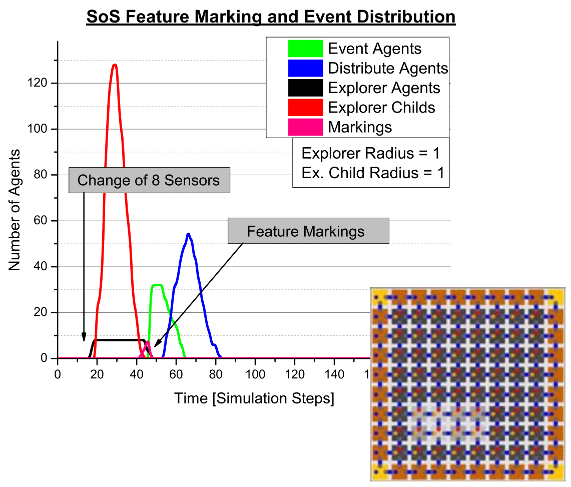


Fig. 10. Analysis results of the agent population obtained from the multi-agent simulation of the feature marking sensor data processing. Left: with explorer reproduction, Right: without (Both: contiguous cluster of 8 sensors)



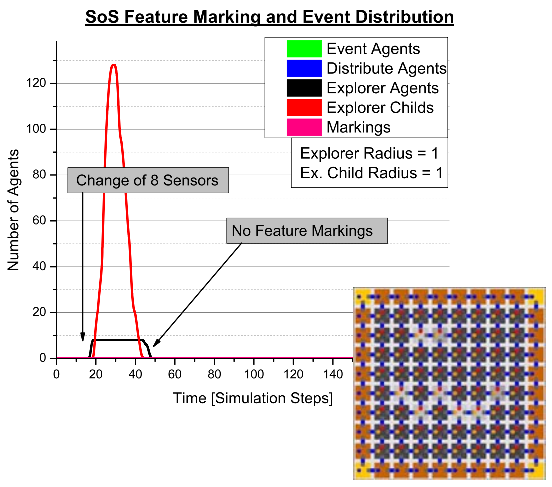


Fig. 11. Analysis results of the agent population obtained from the multi-agent simulation of the feature marking combined with the event-based data distribution. Left: with a contiguous cluster of 8 sensors (correlated), Right: 8 sensor excitation scattered in the network (no correlation).

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 Fig. **9**. 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 multi-path visiting has the advantage of an increased probability of reaching nodes with missing (non operating) communication links. An *event agent*, created by the sensing agent, finally delivers sensor values to computational nodes.

The behaviour of the explorer and its child agents is explained in Alg. **4**. Simulation results are shown in Fig. **10** and **11** for a sensor network stimulus of eight sensors. In Fig. **10** exploration with reproduction (exploration radius > 1) and without reproduction behaviour are compared. The reproduction after successful feature detection leads to a significant increase of the explorer child agent population, which has advantages only in the occurrence of large extended correlated regions (the feature is the boundary of the region). Fig. **11** compares the agent population in the case of correlated and uncorrelated occurrences of sensor stimulus. The latter case does not trigger the creation of event and distribute agents. The simulation results show that the temporal and spatial exploration does not depend on the presence of a feature (correlation) if the explorer radius is limited to one. Otherwise, either explorer reproduction or diffusion occurs, with a temporal broadening and/or increase in the agent population.

3.4. Agent Processing Platform

The requirements for the agent processing platform can be summarized to: 1. to be capable of microchip level (SoC) implementations, 2. supporting a standalone platform without any operating system, 3. performing efficient parallel processing of a large number of different agents, 4. to be scalable regarding to the number of agents processed concurrently, and 5. providing the capability to create, modify, and migrate agents at run-time. Migration of agents requires the transfer of the data and control state of the agent between different virtual machines (at different node locations). To simplify this operation, the agent behaviour based on an activity-transition graph model is implemented with program code, which embeds the (private) agent data as well as the activities, the transition network, and the current control state. It can be handled as a self contained execution unit. The execution of the program by a virtual machine (VM) is handled by a task. The program instruction set consists of zero-operand instructions, mainly operating on the stacks.

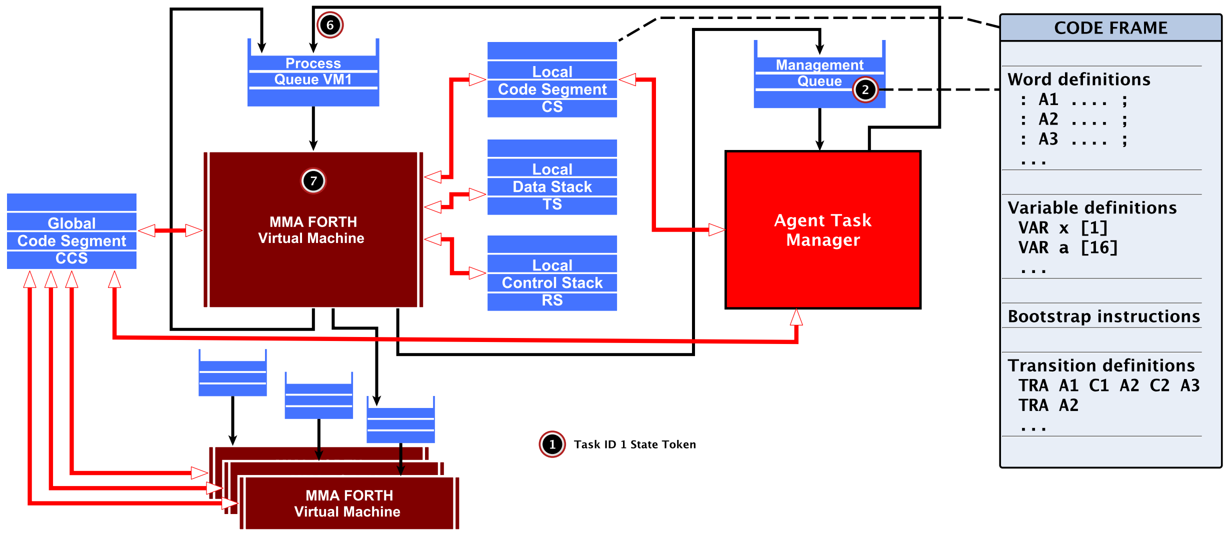


Fig. 12. The Agent processing architecture based on a pipelined stack-based Virtual Machine approach. Tasks are executing units of agent code, which are assigned to a token passed to the VM by using processing queues. A task points to the next instruction word to be executed. After execution the task token is either passed back to the input processing queue or to another queue of either the agent manager or a different VM. Right: the content and format of a code frame.

The virtual machine executing tasks is based on a traditional FORTH architecture: a data (TS) and a control (RS, return) stack, a code segment (CS) storing the program code with embedded data, shown in Fig. **12**. The program is mainly organized by a composition of words (functions). A word is executed by transferring the program control to the entry point in the CS, arguments and computation results are passed only by the stack(s). There are several virtual machines each attached to (private) stack and code segments. There is one global code segment storing global available functions accessed by all programs. This multi-segment architecture ensures high-speed program execution and. the local CS can be implemented with (asynchronous) dual-port RAM (the other side is accessed by the agent manager, discussed below), the stacks with simple single-port RAM. The global CS requires a Mutex scheduler to resolve competition by different VMs.

Commonly the number of agent tasks N_A executed on a node is much larger than the number of available virtual machines N_V. Thus efficient and well balanced multi-task scheduling is required to get proper response times of individual agents. To provide fine grained granularity of task scheduling, a token based pipelined task processing architecture was chosen. A task of an agent program is assigned to a token holding the agent-task identifier related with the code frame of the program to be executed. The token is stored in a queue and consumed by the virtual machine form the queue. After a (top-level) word was executed, leaving commonly with an empty stack, the token is either passed back to the processing queue or to another queue (processing queues of different VMs or the agent manager). This task scheduling policy allows fair and low-latency multi-agent processing.

The program code frame (shown on the right of Fig.**12**) of an agent consists basically of four parts: 1. word definitions defining agent activities (procedures without arguments and return values) and generic functions, 2. embedded agent body variable definitions, 3. bootstrap instructions which are responsible to setup the agent in a new environment (e.g, after migration or at first run), and 4. the transition network calling activity words (defined above) and branching to the next activity execution depending on the evaluation of conditions operating on private data (variables). The transition network section can be modified by the agent by using special instructions. Furthermore, new agents can be created by composing activities and transitions from existing agent programs, creating sub-classes of agent super classes with a reduced functionality. The program frame is stored in the local code segment of the VM executing the program task. The initial code frame loading and any modifications of the code are performed by the agent manager only. A migration of the program code between different VMs requires a copy operation. Each time a program task is executed and after returning from the current activity execution the stacks are assumed to be empty. Each VM has only one stack shared by all program tasks executed on the VM! This design significantly reduces the required hardware resources. Therefore, the return from an agent activity word execution is an appropriate task scheduling point for a different task transferred from the VM processing queue.

4. Inverse Methods for Structural Load Monitoring

Inversion methods for structural load monitoring are based on a physical model of the considered structure. This model allows to associate to a given structural load a response of the structure and hence a signal measured by sensors on or in the structure. Note that the selection of a physical and a sensor model always introduces a certain modeling error. Additionally, the computation of the structure's response to a given load and the resulting sensor signals are rarely computable analytically and hence need to be approximated numerically which introduces an additional simulation error. Finally, any sensor signal is affected by measurement errors due to, e.g., round-off errors due to finite numerical precision.

4.1. Computational Setting

In the numerical examples presented below we consider a thin steal plate of size 0.5m x 0.5m x 0.02m that is fixed at one of its four vertical sides and use the equations of linearized elasticity as a physical model. The structure's response to a load hence is deformation and the sensors measure surface strain on the (lower) surface of the plate. Using a finite element discretization, we simulate a finite number of loads caused by cylindrical weights placed at equidistant grid points on the upper side of the plate and compute the corresponding surface strains at sensor points on the lower side of the plate, and hence obtain a so-called load-strain matrix $\boldsymbol{T}$ mapping vectors modeling discretized loads to surface strains at the sensor positions. Obviously, this procedure additionally introduces a discretization error for any load that cannot be represented as a linear combination of the above-mentioned weights.

After computing the load-strain matrix $\boldsymbol{T}$ the inversion task is to stably compute load vectors x that satisfy the equation $\boldsymbol{Tx}=\boldsymbol{S}$ for given sensor measurements $\boldsymbol{S}$. (In contrast to the preceding section, we collect the sensor signals $\boldsymbol{S}$ here in a column vector rather than a matrix.) Since the number of columns of $\boldsymbol{T}$ equals the number of weight positions that is in general different from the number of measured strains, the matrix $\boldsymbol{T}$ cannot be inverted. Further, a solution of the corresponding least-squares problem should also be avoided since $\boldsymbol{T}$ is badly conditioned, as it is usually the case for inversion problems, meaning the the above-mentioned errors will lead to large errors in the solution. Instead, one has to stabilize the computation of $\boldsymbol{x}$ using so-called inverse (aka regularization or inversion) methods. We refer to **[10]** and **[11]** for an introduction to such methods.

4.2. Inverse Methods

In the sequel, we will introduce two different inversion methods, compare their applicability in the framework of the embedded sensor network introduced in Section 2 and test the stability of the two algorithms with respect to measurement errors and the number of sensors involved.

As a first technique, we consider the classical Tikhonov regularization, where one stabilizes the inversion by requiring that x does not minimize the 2-norm of the residual $\boldsymbol{Tx}$-$\boldsymbol{S}$ but the functional $x \rightarrow \|T(x)-S\|_2^2 + \alpha\|x\|_2^2$ where $\|y\|_2$ denotes again the 2-norm of a vector $\boldsymbol{y}$. The positive parameter α should to be coupled with the measurement error in the data and the modeling and computation error in the matrix $\boldsymbol{T}$, compare [Kirsch, 1996]. The minimizer of this quadratic problem solves the linear equation $\boldsymbol{T^*Tx}+\alpha\boldsymbol{x} = \boldsymbol{T^*S}$. An issue of this inverse method that might sometimes be critical for its application in a sensor network is the inversion of a dense linear system. However, in our context the matrix $\boldsymbol{T}$ must anyway be computed before the monitoring device is launched, and hence one can directly precompute a singular value decomposition of $\boldsymbol{T}$, boiling down the solution of the linear system to essentially three matrix multiplications. The disadvantage of Tikhonov regularization is the smoothing property of the scheme: In our context, loads with small spatial support or discontinuous loads typically will not be reconstructed with high accuracy but result in smoothed and smeared-out computational results. This makes a precise location of the support of a load difficult, in particular when several loads act simultaneously on the structure.

The second technique we consider for load monitoring is the conjugate gradient (cg) method applied to the normal equation $\boldsymbol{T^*Tx} = \boldsymbol{T^*S}$, an iterative technique where the k-th iterate $\boldsymbol{x_k}$ minimizes the discrepancy $\boldsymbol{x} \rightarrow \|\boldsymbol{T}(\boldsymbol{x})-\boldsymbol{S}\|_2^2$ in the so-called Krylov subspace spanned by the vectors

$$T^*S,(T^*T)T^*S,\ldots,(T^*T)^{k-1}T^*S. \tag{4}$$

This iterative scheme can, similar to Tikhonov regularization, be formulated explicitly, resulting in the following iterative scheme shown in Alg. **1**.

Alg. 1. The cg algorithm applied to the linear system $\boldsymbol{Tx}$=$\boldsymbol{S}$ (pseudo-notation)

```
x₀ := 0;
r := S - T • x₀;
d := T* • r;
m := |d|$_2^2$ ;
p := d;
for k=1,2,..., until r=0:
  q := T • p;
  a := m / |q|$_2^2$ ;
  $x_k$ := $x_{k-1}$ + a •p;
  r := r - a •q;
  d := T* •r;
  b := |d|$_2^2$ / m;
  m := |d|$_2^2$ ;
  p := d + b • p;
```

Due to the ill-conditioning of $\boldsymbol{T}$, the iteration has to be stopped whenever the discrepancy of the actual iterate reaches the expected noise level of the sensor signals or the modeling or discretization error of the matrix $\boldsymbol{T}$. The conjugate gradient iteration usually outperforms Tikhonov regularization when reconstruction loads with small support, which definitely is an advantage in the context of structural load monitoring. Further, this method is known to compute iterates reaching a given discrepancy level earliest among all Krylov subspace methods.

4.3. Pre-computations

All inverse methods introduced above do not directly act on the sensor signal $\boldsymbol{S}$ but rather on $\boldsymbol{T^*S}$. The signal of the k-th sensor is hence multiplied with the k-th row of $\boldsymbol{T}$. A critical signal level for the k-th sensor signal should hence incorporate both the signal strength and the 2-norm of the k-th row of $\boldsymbol{T}$. Indeed, if the norm of the k-th matrix row is small compared to the other rows, this means that structural loadings create a relatively small amount of strain at the corresponding sensor. A strong sensor signal at this sensor thus hints a possibly critical load. On the other hand, if a matrix row has large norm compared with the other rows, relatively large strains at the corresponding sensor have to be expected. The pre-processing scheme of the k-th sensor is then defined as follows: If the sensing agent detects that a signal strength multiplied with the quotient of the 2-norm of the k-th row of $\boldsymbol{T}$ and the arithmetic mean of the norms of all columns is above a certain threshold, then he sends out event agents to deliver the sensor data to the computational nodes. Otherwise, no agents are sent out and no data is propagated through the network. Of course, the assigned sensor threshold can vary from one sensor to the other - typically, one would assign a sensor situated in an important or vulnerable part of a structure a low threshold. Further, the individual threshold of a sensor could be changed by the computation nodes during monitoring, because of, e.g., several critical loadings in the vicinity of a sensor.

The Tikhonov regularization offers a further possibility for pre-computation of the sensor signals by the pre-computation nodes introduced above, see Fig. **5**. Assume that (U,D,V) is a singular value decomposition of the matrix, that is,

$$T = UDV^{*} \tag{5}$$

with orthogonal matrices U and V and a diagonal matrix D=diag(d_i) containing the singular values of T. Then the Tikhonov regularization x can be computed as

$$x = V\,\mathrm{diag}(d_i\,/\,(d_i^2 + \varepsilon))U^{*}S. \tag{6}$$

This technique hence multiplies the signal of the k-th sensor with the k-th column of the matrix V diag(d_i/(d_i^2+ε))U* and sums up all resulting column vectors to obtain x. If a pre-computation node hence stores all columns of this matrix corresponding to sensors in its row or its column in the two-dimensional sensor network shown in Figure 3, then this pre-computation node is able to compute the corresponding part of the Tikhonov regularization x and pass the result to the neighboring pre-computation node. The computation nodes in the corners of the sensor networks are then merely required to add up the results from the different pre-computations to obtain the load inversion x and decide whether the computed load is critical for the structure.

4.4. Numerical examples

The simulated experiments we present in the rest of this section follow the setting we already outlined above: Using a finite element method, we simulate a 2 mm thin quadratic steal plate with side length of 500 mm an elastic modulus of 210 kN/mm^2 and a Poisson's ration of 0.3 that is fixed at one of its four vertical front sides in a horizontal position. More precisely, using the equations of linear elasticity we simulate the displacement of this plate under the load of a cylindrical weight that is successively placed at all grid points of a regular grid on the upper horizontal surface of the plate. The finite element approximation used to compute the plate's displacement possesses about 11.5 million unknowns and is computed using piecewise linear and globally continuous basis functions on a regular tetrahedral mesh of the plate. For each grid point we compute the surface strain in two (fixed) orthogonal directions at the grid points of a second grid on the lower horizontal surface of the plate. Theses strains thus model the sensor measurements and the simulation yields the load-strain matrix T. The grid used to simulate the weights on the upper surface of the plate is in all examples a regular, quadratic grid of 15*15=225 points. The relative numerical approximation error of the computed strain data is less than one percent. Since our computing equipment was sufficiently strong and, moreover, the simulation time for the strain data and the load-strain matrix T play no role for the inversion time or memory requirements we preferred to discretize the full three-dimensional equations of linear elasticity instead of exploiting the small thickness of the plate to simulate a reduced model.

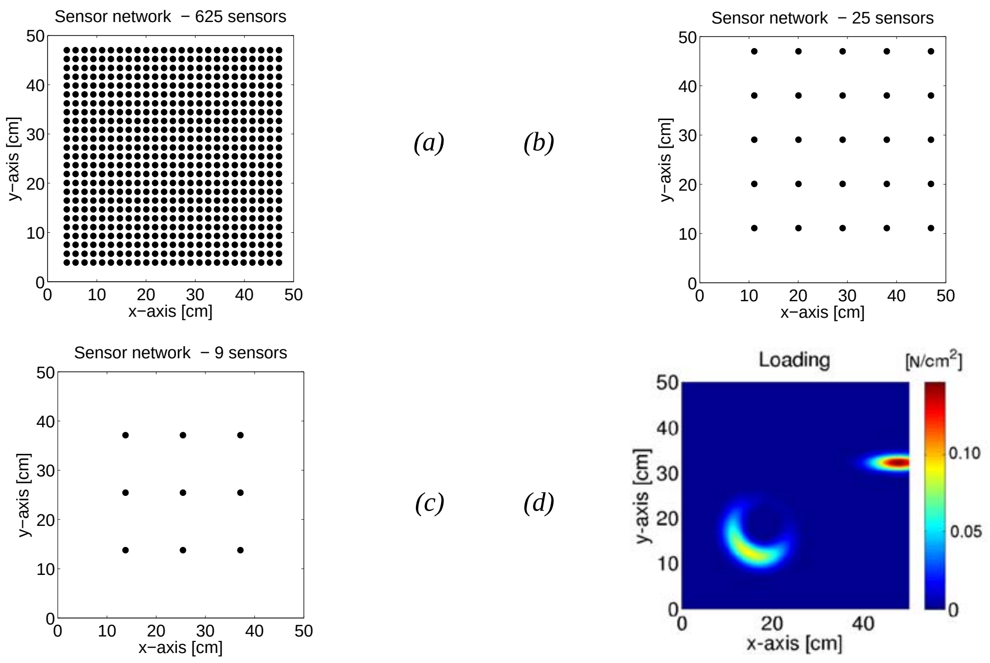


Fig. 13. The sensor grids and the load used to test the load monitoring inverse methods: (a) Equidistant grid with 676 points. (b) Equidistant grid with 25 points. (c) Equidistant grid with 9 grid points. (d) The numerical values of the load on the upper surface of the steal plate

To test the dependence of the quality of the load monitoring system on the number of sensors, we use different (quadratic) sensor grids on the lower surface of the plate (compare Fig. **13**(a)-(c)). Note, however, that our numerical model of the plate does not simulate the strain sensors: The strain values building the matrix $\boldsymbol{T}$ consist of point values of the simulated strain at the sensor positions. In particular, the sensors have no influence on the structure in our simulation, which definitely is a simplifying assumption. In all experiments we add random noise first to the simulated sensor signals $\boldsymbol{S}$ and second to the load-strain matrix $\boldsymbol{T}$. The random matrices and vectors added to $\boldsymbol{T}$ and $\boldsymbol{S}$, respectively, consist of independent random variables that are uniformly distributed in [-1,1] . Moreover, they are scaled to a certain relative noise level that we will be indicated for all examples below. Finally, in all examples, the threshold in the pre-processing step of the sensor signals described above equals the numerical noise level times the maximal signal strength observed when computing $\boldsymbol{T}$. Note that all plots below shows the top view of the steal plate. The top side of the plotted square corresponds to the front side of the plate that is fixed in the simulations.

To test the different inverse methods presented above, we choose a load ($\boldsymbol{L}$) consisting of two different parts, a bean-shaped load on the left and an ellipse-shaped load on the right, cf. Fig. **13**(d). In particular, this load is different from the cylindrical loads used to simulate the load-strain matrix $\boldsymbol{T}$.

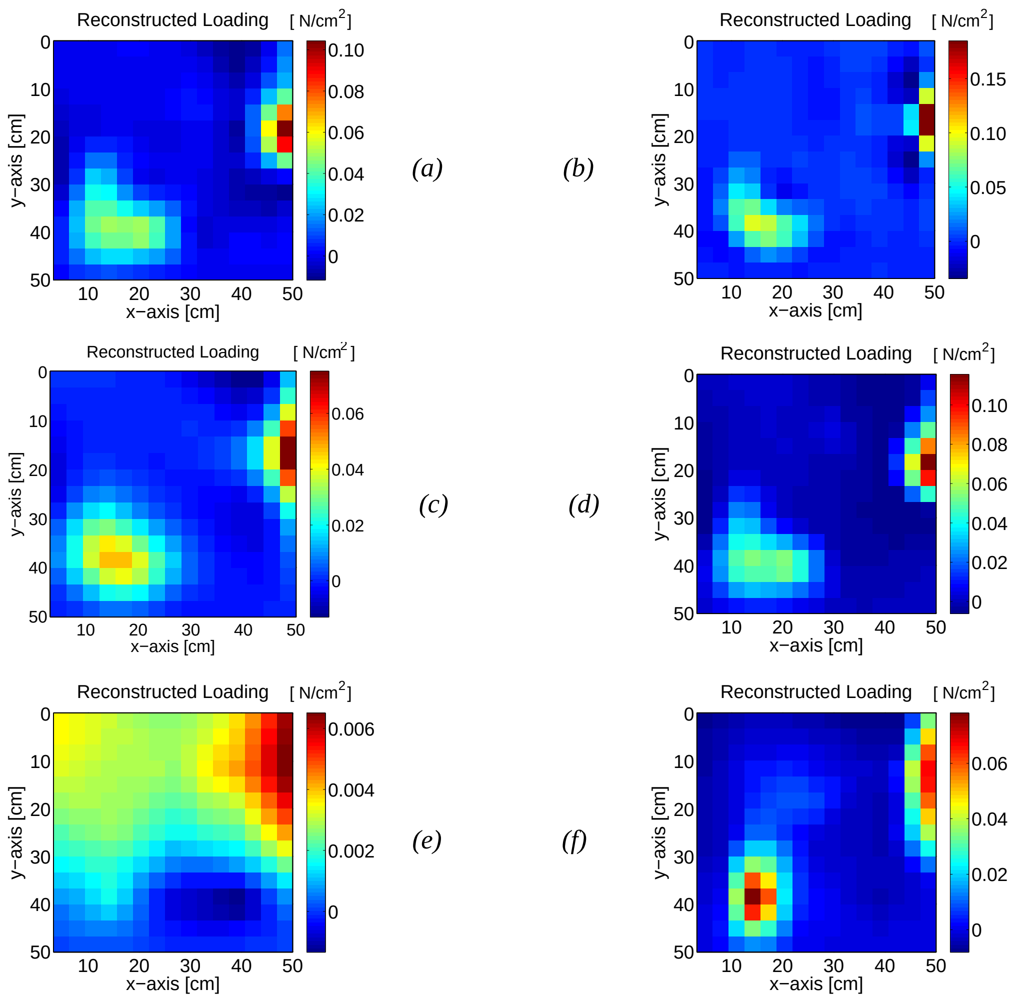


Fig. 14. Loads inverted using different sensor networks using simulated data without artificially added noise. (a) Tikhonov regularization, 625 sensors. (b) cg-method, 625 sensors. (c) Tikhonov regularization, 25 sensors. (d) cg-method, 25 sensors. (e) Tikhonov regularization, 9 sensors. (f) cg-method, 9 sensors.

In the first example, we do not add artificial noise to the simulated strains and the matrix $\boldsymbol{T}$. Fig. **14** (a)-(b) shows that both Tikhonov regularization and the cg-method locate the two disconnected part of the load ($\boldsymbol{L}$) at the correct places when the 625 sensor points from Fig. **13**(a) are used. However, the reconstructed load of the cg-method additionally arrives in reconstructing the correct magnitude of the loads and, additionally, reconstructs the shape of the beam-shaped left part of load ($\boldsymbol{L}$) much better. Both inverse methods have problems to correctly estimate the shape of the elliptic right part of load close to the boundary of the plate. When reducing the number of sensors to 25 (cf. Fig. **13**(b)), the quality of the reconstructed shapes stays about the same, but reconstructed numerical values of the loads become significantly smaller. When further reducing the number of sensors to merely 9 (cf. Fig. **13**(c)), the load reconstructed by Tikhonov regularization becomes essentially useless while the cg-method at least reconstructs the position of the two disconnected parts of the load correctly.

In a second example, we test the stability of the reconstructions under noise, adding noise with a relative noise level of one percent both the the load-strain matrix and to the simulated strains due to the load form Fig. **13**(d). The results, shown in Fig. **15**, show the cg-method still performs well for data collected in the sensor network from Fig. **13**(b) and (c) containing 25 and 9 sensors, respectively. However, Tikhonov regularization smoothes the inverted loads so strongly that the numerical values of the reconstruction are out of scale when using 25 sensors. As before, a network of 9 sensors is not sufficient to obtain meaningful results using this inverse method.

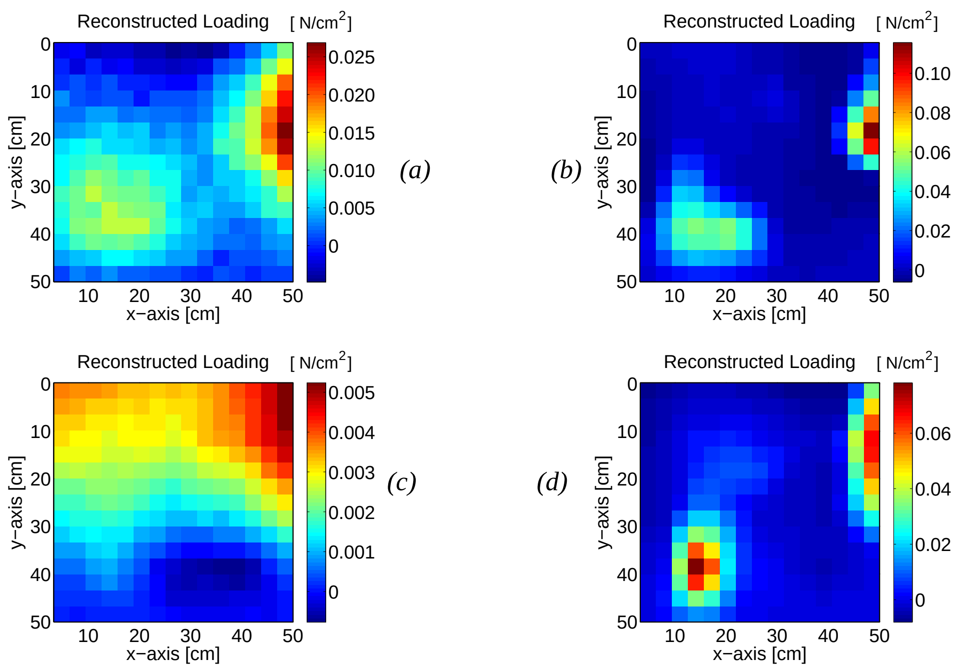


Fig. 15. Loads inverted using different sensor networks and simulated data with artificial additive noise and a relative noise level of 1 percent. (a) Tikhonov regularization, 25 sensors. (b) cg-method, 25 sensors. (c) Tikhonov regularization, 9 sensors. (d) cg-method, 25 sensors.

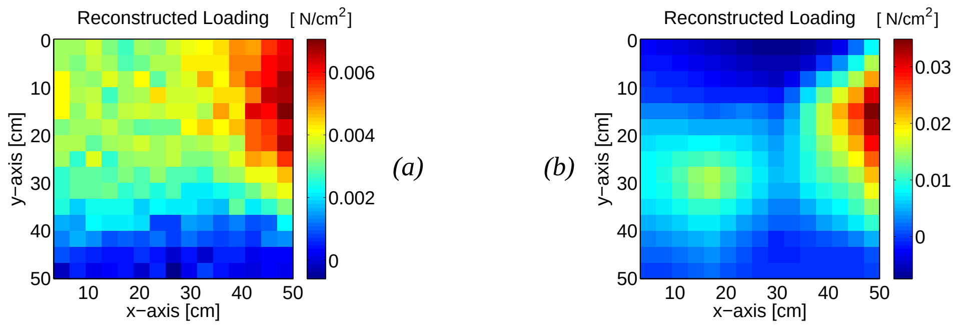


Fig. 16. Loads inverted using different sensor networks and simulated data with artificial additive noise and a relative noise level of 10 percent. (a) Tikhonov regularization, 625 sensors. (b) cg-method, 9 sensors

Finally considering a relative artificial noise level of 10 percent in Fig. **16** (a)-(b), Tikhonov regularization is no longer able to yield a reasonably accurate load reconstruction already for the largest sensor network with 625 sensors from Fig. **16** (a), whereas the cg-method remains surprisingly stable and separates the positions of the two parts of the load even for the sensor network involving 9 nodes.

5. Conclusions

A novel **sensor processing approach** using mobile agents for reliable distributed and parallel data processing in large scale networks of low-resource nodes was introduced, leading to a sensor signal pre-processing at run-time inside the sensor network by using a hybrid multi-agent system. Agent mobility crossing different execution platforms in mesh-like networks and agent interaction by using tuple-space databases and global signal propagation aid solving data distribution and synchronization issues in the design of distributed sensor networks. Event-based sensor data distribution and pre-computation with agents reduces communication and overall network activity resulting in reduced energy consumption.

Off-line inverse numerical computation of pre-processed sensor data allows the calculation of the system response based on data from prior FEM simulations of the technical structure.

The agent behaviour is partitioned in different agent classes and specified using the agent-orientated programming language AAPL which provides computational statements and statements for agent creation, inheritance, mobility, interaction, reconfiguration, and information exchange, based on agent behaviour partitioning in an activity graph. The agent behaviour is compiled to program code which is executed on a stack-based virtual machine, which can be directly synthesized to the microchip level by using a high-level synthesis approach. A high-level synthesis tool enables the synthesis of different processing platforms from a common AAPL program source, including standalone hardware and software platforms, as well as simulation models offering functional and behavioural testing. The migration of an agent to a neighbour node takes place by migrating the data and control state encapsulated in the program code of an agent using message transfers. Two different agent interaction primitives are available: signals carrying data and tuple-space database access with pattern templates. Configuration and (re-) composition of the activity graph offers agent behaviour adaptation (which can be inherited by children) at runtime and relax communication and storage requirements. Additionally, (re-) composition allows the derivation of agent sub-classes from a super class.

A case study demonstrated the suitability of the proposed smart sensor processing approach, using event-based sensor data propagation, adaptive path finding, and feature extraction with self-organizing exploration. The presented examples for load monitoring show that both the classical Tikhonov regularization method and the cg-method are suitable inverse algorithms for structural load monitoring. However, the cg-method produces load reconstructions that have significantly higher quality since the support of a loading is usually sharper and the reconstructed numerical values are in general more accurately found. All computations in these examples were carried out using floating point arithmetic in double precision. Sensing and pre-computations in a sensor network are inherently limited in memory and hence carried out in integer arithmetic.

We will address extensions of the presented work into this direction in the future, as well as other error models for the sensor network (e.g., a sensor blackout or malfunction). Network-adapted regularization schemes scaled to the sensor network's nodes computing capabilities, the modelling of the sensor network in the forward simulation and optimized sensor positions are two further crucial directions of investigation that will be considered in the future. For example, if the structural is a-priori known to be loaded by loads with small support on the surface of the structure, so-called sparsity-promoting regularization schemes potentially yield highly performant inverse methods for structural load detection.

Appendix A. Algorithms

Alg. 2. Simplified behaviour model for the Event, Sampling, Sensing, Computer and Distribution agents (mixed pseudo-notation and AAPL statements)

```
type Direction = {NORTH,SOUTH,WEST,EAST,ORIGIN}
type Position = record X:integer; Y:integer; end;

agent Event(Dir) =
  var Sensing,Delta;
  activity init = Delta:=(0,0); read sensor value from tuple space: rd(SENSORVALUE,Sensing?)
  activity move = move to a neighbour node in the specified direction Dir, change Delta
  activity check = if current node is a distribution node: exist?(DISTRIBUTER) then
                         arrived := true;
  activity deliver = calculte sensor matix position from Delta, store sensor value
                     in the tuple space: out(SENSORVALUE,i,j,Sensing);
  transitions = init→move; movec→check; check→deliver:arrived; check→move:not arrived;

agent Sampling(sample_intervall) =
  var adc;
  activity init = initializie ADC
  activity sample = read ADC value, store value in tuple space: out(ADC,adc)
  activity sleep = sleep for ’sample_intervall msec’
  transitions = init→sample; sample→sleep; sleep→sample;
```

```
agent Sensing(sensor_thres) =
  var sensor,sensor0;
  activity init = initialite sensing
  activity lookup = in(ADC,sensor?)
  activity sense = if |sensor-sensor0| > sensor_thres then
                           rm(SENSORVALUE,?); out(SENSORVALUE,sensor); sensor0 := sensor;
                           create event agents, one for each direction
                           eval(new Event(NORTH)); eval(new Event(SOUTH)); ...
  transitions = init→lookup; lookup→sense;sense→lookup;

agent Computer(Kind={KIND_DIST,KIND_COMP}) =
  var SensMat,SensMatShadow,Row,Col,row,col,v,v1,..,v8,Dir;
  activity init = Determine flow Dir(ection) and the Row or Col(umn) of Matrix S for this node
  activity compute =
    if Kind = KIND_DIST then
      SensMatShadow.[row,col] := v;
      create two distribute agents send in opposite directions carrying one row/column
     eval(new Distribute(Dir,Row,Col,v1,...,v8)); eval(new Distribute(-Dir,Row,Col,v1,...,v8));
    else -- Kind = KIND_COMP
      if row <> -1 then for col' = 1 to 8 do if vi <> 0 then SensMatShadow.[row,col'] := vi
      elsif col <> -1 then for row' = 1 to 8 do if vi <> 0 then SensMatShadow.[row',col] := vi

  activity sleep =
    if Kind = KIND_DIST then in(SENSORVALUE,row?,col?,v?)
                        else in(SENSORVALUE,row?,col?,v1?,...,v8?);
  transitions = init→sleep; sleep→compute ; compute→sleep;

agent Distribute(Dir,Row,Col,v1,v2,..,v8) =
  var arrived,Delta;
  activity init = Delta := (0,0);
  activity move = move to a neighbour node in the specified direction Dir, change Delta
  activity check = if current node is a computation node: exist?(COMPUTER) then
                         arrived := tue;
  activity deliver = out(SENSORVALUE,Row,Col,v1,..,v8);
  transitions = init→move; move→check; check→deliver:arrived; check→move: not arrived;
```

Alg. 3. Adpative routing functions used by Event and Distribute Agents

```
type Route = record dir:Direction; lastdir:Direction; delta:Position;
                    gamma:Position; routed:bool; end;
function route_relax(r:Route) =
  Goal: try to return to original planned path and minimize gamma vector until (0,0) is reached
  if r.gamma <> (0,0) then
    if r.gamma.X < 0 and link?(EAST) and r.lastdir <> WEST then
      r.routed:=true; r.delta.X:=r.delta.X+1; r.dir=EAST; r.gamma.X:=r.gamma.X+1
    if r.gamma.X > 0 and link?(WEST) and r.lastdir <> EAST then
      r.routed:=true; r.delta.X:=r.delta.X-1; r.dir=WEST; r.gamma.X:=r.gamma.X-1
    ...
function route_normal(r:Route) =
  Goal: try to follow the original planned path in specified direction
  case r.dir of
  | NORTH -> if link?(NORTH) and r.lastdir <> SOUTH then
    r.routed:=true; r.delta.Y := r.delta.Y-1; r.dir=NORTH;
  | WEST -> if link?(WEST) and r.lastdir <> EAST then
    r.routed:=true; r.delta.X := r.delta.X-1; r.dir=WEST;
  ...
function route_opposite(r:Route) =
  Goal: try to randomly select an alternative direction leaving the original planned path
  var dir,dirs;
  for dir in {NORTH,SOUTH,WEST,EAST} do if link?(dir) then dirs := dirs + {dir}; done;
  r.routed:=true; r.dir := Random(dirs); -- Select one new possible routing direction randomly
  case r.dir of
   | NORTH ->
```

```
    r.delta.Y := r.delta.Y-1; r.gamma.Y:=r.gammy.Y-1; r.lastdir:=NORTH;
  ..
```

Alg. 4. Behavioural model of the explorer agent (mixed pseudo-notation and AAPL statements)

```
agent Explorer(dir:Direction,radius:integer) =
  Goal: explore neighbourhood and try to find correlations to this local sensor value
  var s0,group,backdir,enoughinput,live,h;
  activity start = h:= 0;
    if dir <> ORIGIN then
      moveto(dir); case dir of | NORTH -> backdir := SOUTH; ...
    else live := MAXLIVE; backdir := ORIGIN;
    group := RandomInt(0,MAXGROUP-1); out(H,id(self),0); rd(SENSORVALUE,s0);
  activity percept = enoughinput := 0;
    for nextdir in {NORTH,SOUTH,WEST,EAST} do
       if nextdir <> backdir and link?(nextdir) then
         enoughinput++; eval(new ExplorerChild(nextdir,radius));
    timer+(TMO,TIMEOUT);
  activity diffuse = live--; rm(H,id(ME));
    if live > 0 then case backdir of | NORTH => dir := Random({SOUTH,EAST,WEST}); ...
    else  kill(ME);
  activity reproduce =
    rm(H,id(ME); if exist?(FEATURE,?) then in(FEATURE,n) else n := 0; out(FEATURE,n+1);
    if live > 0 then
     for nextdir in {NORTH,SOUTH,WEST,EAST} do
       if nextdir <> backdir and link?(nextdir) then eval(fork(nextdir,radius));
    kill(ME);
  handler WAKEUP = enoughinput--; try_rd(0,H,id(ME),h?);
    if enoughinput < 1 then timer-(TIMEOUT);
  handler TIMEOUT = enoughinput := 0;
  transitions = start→percept;
                percept→reproduce:h>=ETAMIN or h > ETAMAX and enoughinput < 1;
                percept→diffuse:h< ETAMIN and h <= ETAMAX and enoughinput < 1;
                diffuse→start:live>1;

agent ExplorerChild (dir:Direction,r:radius,group:integer,s0:integer,d:Delta) =
  var enoughinput,...;
  activity move = modify(dx,dy); moveto(dir);
  activity percept_neighbour =
    if not exist?(MARK,group) then
      mark(TMO,MARK,group); enoughinput := 0; rd(SENSORVALUE,s?);
      if abs(s-s0) <= DELTA then out(H,id(ME),1) else out(H,id(ME),0); end;
      for nextdir in {NORTH,SOUTH,WEST,EAST} do
        if nextdir <> backdir and inbound(nextdir,radius) and link?(nextdir) then
          enoughinput++; eval(fork(nextdir,radius));
      time+(TMO,TIMEOUT);
  activity goback = if exist?(H,id(ME),?) then in(H,id(self),h) else h := 0;
    moveto(backdir);
  activity deliver = in(H,id(PARENT),v?); out(H,id(PARENT),v+h); send(id(PARENT,WAKEUP);
    kill(ME);
  transitions =  move→percept_neighbour;
                 percept_neighbout→goback:enoughinput<1;
                 goback→deliver;
```

6. References

[1] S. Bosse, *Distributed Agent-based Computing in Material-Embedded Sensor Network Systems with the Agent-on-Chip Architecture*, IEEE Sensors Journal, Special Issue on Mateiral-integrated Sensing, DOI 10.1109/JSEN.2014.2301938

[2] S. Bosse, F. Pantke, *Distributed computing and reliable communication in sensor networks using multi-agent system*, Production Engineering, Research and Development, 2012, ISSN: 0944-6524, DOI:10.1007/s11740-012-0420-8

[3] M. Guijarro, R. Fuentes-fernández, and G. Pajares, *A Multi-Agent System Architecture for Sensor Networks*, Multi-Agent Systems - Modeling, Control, Programming, Simulations and Applications, 2008.

[4] X. Zhao, S. Yuan, Z. Yu, W. Ye, J. Cao. (2008), *Designing strategy for multi-agent system based large structural health* *monitoring*, Expert Systems with Applications, 34(2), 1154–1168. doi:10.1016/j.eswa.2006.12.022

[5] F. Pantke, S. Bosse, D. Lehmhus, and M. Lawo, *An Artificial Intelligence Approach Towards Sensorial Materials*, Future Computing Conference, 2011

[6] F. Klügel, *SeSAm: Visual Programming and Participatory Simulation for Agent-Based Models*, In: Multi-Agent Systems - Simulation and Applications, A. M. Uhrmacher, D. Weyns (ed.), CRC Press, 2009

[7] C. Sansores and J. Pavón, *An Adaptive Agent Model for Self-Organizing MAS*, in Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), May, 12-16., 2008, Estoril, Portugal, 2008, pp. 1639–1642.

[8] D. Lehmhus, S. Bosse, *Sensorial Materials*, in *Structural Materials and Processes in Transportation*, pp. 517-548, D. Lehmhus, M. Busse, A. S. Herrmann, K. Kayvantash (Ed.), Wiley-VCH, 2013, ISBN: 9783527327874

[9] J. Liu, *Autonomous Agents and Multi-Agent Systems*, World Scientific Publishing, 2001 (ISBN 981-02-4282-4)

[10] H. W. Engl, M. Hanke, A. Neubauer, *Regularization of inverse problems*, Kluwer Acad. Publ., Dordrecht, Netherlands, 1996

[11] A. Kirsch, *An introduction to the mathematical theory of inverse problems*, Springer, 1996

[12] M. Friswell, *Damage identification using inverse methods*, Phil. Trans. R. Soc. A, 365, 393–410, 2007.

[13] M.I. Friswell, J.E. Mottershead, *Inverse methods in structural health monitoring*, DAMAS 2001: 4th International Conference on Damage Assessment of Structures, Cardiff, June 2001, pp. 201-210

[14] C. Carn, P. Trivailo, *The inverse determination of aerodynamic loading from structural response data using neural networks*, Inverse Problems in Science and Engineering, 14, 379-395, 2006

[15] A. Huhtala, S. Bossuyt, *A Bayesian approach to vibration based structural health monitoring with experimental verification*, Rakenteiden Mekaniikka (Journal of Structural Mechanics), 44, 330-344, 2011

[16] S. Bosse, *Design of Material-integrated Distributed Data Processing Platforms with Mobile Multi-Agent Systems in Heterogeneous Networks*, Proc. of the 6'th International Conference on Agents and Artificial Intelligence ICAART 2014. DOI:10.5220/0004817500690080