

A Hybrid Approach for Structural Monitoring with Self-organizing Multi-Agent Systems and Inverse Numerical Methods in Material-embedded Sensor Networks

Stefan Bosse, Armin Lechleiter

University of Bremen, Dept. of Mathematics & Computer Science, Robert Hooke Str. 5, 28359 Bremen, Germany

Abstract: One of the major challenges in Structural Monitoring of mechanical structures is the derivation of meaningful information from sensor data. This work investigates a hybrid data processing approach for material-integrated Structural Health and Load Monitoring systems by using self-organizing mobile multi-agent systems (MAS), and inverse numerical methods providing the spatial resolved load information from a set of sensors embedded in the technical structure with low-resource agent processing platforms scalable to microchip level, enabling material-integrated real-time sensor systems. The MAS is deployed in a heterogeneous environment and offers event-based sensor preprocessing, distribution, and pre-computation. Inverse numerical approaches usually require a large amount of computational power and storage resources, not suitable for resource constrained sensor node implementations. Instead, the computation is partitioned into spatial off-line (outside the network) and on-line parts, with on-line sensor processing performed by the agent system. A unified multi-domain simulation framework is used to profile and validate the proposed approach.

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

1. Introduction

Structural Monitoring of mechanical structures allows to derive not just loads by using Load Monitoring (LM), but also their effects to the structure, its safety, and its functioning from sensor data, offering some kind of Structural Health Monitoring (SHM). A Load Monitoring system is a basic component of a SHM system, which provides spatial resolved information about loads (forces, moments, etc.) applied to a technical structure, with applications ranging from robotics to building monitoring.

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 information mapping function $Map(s, E) : s \times E \rightarrow i$ that basically maps the raw sensor data input s , a n -dimensional

vector consisting of n sensor values, on the desired information i , a m -dimensional result vector. The result of the computed information commonly depends on some abstract environmental setting E (see Fig. 1) arising in all technical systems, i.e., the disturbance of data caused by communication, data processing, energy supply, or temporal and spatial data distribution. The goal of the mapping function is to reduce the data dimension significantly, i.e., $m \ll n$.

This work investigates a hybrid data processing approach for material-integrated 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. Such inverse approaches usually require a considerable amount of computational power and storage resources, not very well matching 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 preprocessing MAS delivers sensor data event-based if a change of the load was detected (feature extraction), reducing network activity and energy consumption significantly. Inverse numerical algorithms use matrix computations extensively, so it is in principle possible to distribute and perform some of the matrix computations in the sensor network offering an on-line pre-computation by the MAS. This is a main advantage over Machine Learning methods used in LM and SHM systems [1][2][3], which are more difficult to distribute efficiently due to long distance data dependencies.

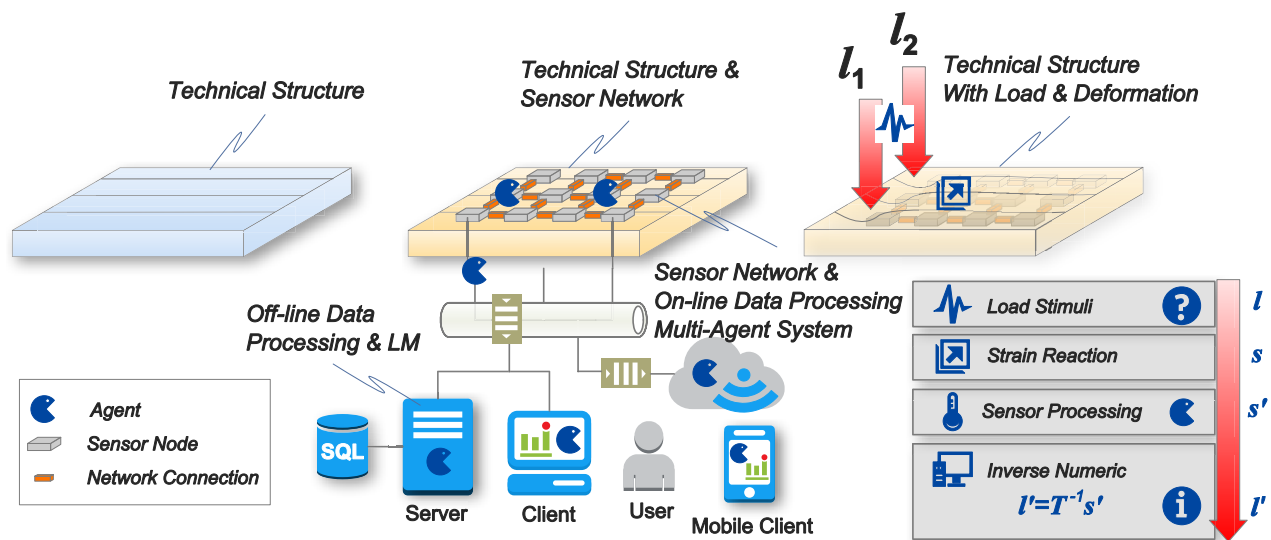


Figure 1. Initially unknown external forces acting on a mechanical structure lead to an deformation of the material based on the internal forces. A material-integrated active sensor network composed of sensors, electronics, data processing, and communication, together with mobile agents can be used to monitor relevant sensor changes with an advanced event-based information delivery behaviour. Inverse numerical methods can compute finally the material response. The unknown system response for externally applied load l is measured by the strain sensor stimuli response s' (a function of s), and finally inverse numerical methods compute an approximation l' to the applied load.

Basically there are two different information extraction approaches for material-integrated LM systems and a possible optimization of sensor positions: (I) Those methods based on a mechanical and numerical model of the technical structure, the Device under Test (DUT), and the sensor; (II) Those without any or with a partial physical model. The latter class can profit from artificial intelligence, which is usually based 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, data analysis using Fourier- or wavelet techniques, or neural network approaches. We refer to [4][5][6][7] and [8] (Chapter 12) posing 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 \mathbf{T} of the technical structure mapping loads to sensor signals. In this study, we consider measurements of surface strains and aim to compute the associated spatially varying (discretized) loads \mathbf{l} on the structure. The mechanical model \mathbf{T} is gained from linear elasticity and can, in a discretized setting, be represented by a matrix. Given a sensor signal vector \mathbf{s} (serialization of a two-dimensional sensor matrix \mathbf{S} , which is approximately linear depending on the measured strain), inverse methods try to stably "invert" the mapping \mathbf{T} , that is, to find a stable solution \mathbf{l} to the problem $\mathbf{T}\mathbf{l} = \mathbf{s}$. Since measured signals and the underlying physical model always contain numerical and modelling 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:

- A classical and well-known inversion method is Tikhonov regularization. Pick amongst all approximated solutions to $\mathbf{T}\mathbf{l} = \mathbf{s}$ the one that minimizes a certain functional - the simplest functional to minimize would be the Tikhonov functional

$$\mathbf{l} \mapsto \|\mathbf{T}\mathbf{l} - \mathbf{s}\|_2^2 + \alpha \|\mathbf{l}\|_2^2, \quad (1)$$

where $\alpha > 0$ and $\|\cdot\|_2$ is the 2-norm of a vector defined by Eq. (2), but different and more complicated variants exist and might also be convenient choices. The latter vector norm is defined for any dimension $n \in \mathbb{N}$ and any vector $\mathbf{v} = (v_1, \dots, v_n)^T \in \mathbb{R}^n$ by

$$\|\mathbf{v}\|_2 = \left(\sum_{j=1}^n |v_j|^2 \right)^{1/2}. \quad (2)$$

- Alternatively, consider any iterative method that minimizes the residual $\mathbf{T}\mathbf{l} - \mathbf{s}$ and stabilize the inversion by stopping the iteration when the norm of the residual is about the magnitude of the expected signal and modelling error. Examples for such iterative techniques include the Land-

weber iteration, the conjugate gradient iteration, but also recent soft shrinkage techniques, see [9] and [10].

If the mechanical model T is linear, then the best known inversion method in the first class is the Tikhonov regularization minimizing the quadratic functional (Eq. (1)) by solving the equivalent linear system $(T^*T + \alpha)l = T^*s$, where T^* denotes the transpose matrix of T . The most powerful algorithm in the second class is, arguably, the conjugate gradient iteration. The disadvantage of inverse methods with regard to applications based on sensor networks 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 simulation methods and increasing computational power.

Reliable distributed data processing in sensor networks using multi-agent systems (MAS) were recently reported in [11] and employed for SHM applications in [12]. An adaptive and learning behaviour of MAS, which is a fundamental principle in the agent model, can aid to overcome technical unreliability and limitations [13]. Artificial intelligence and machine learning can be used in sensorial materials without a predictive mechanical model at hand [1], which is a definite advantage for complex materials.

Multi-agent systems can be used for a decentralized and self-organizing approach of data processing in a distributed system like a sensor network (that is already applied in macro-scale applications, e.g., in [14]), enabling information extraction, for example, based on pattern recognition [15], by decomposing complex tasks in simpler cooperative agents.

In this work the behaviour of mobile agents is related to a state and is modelled with an activity-transition graph (ATG) which is implemented with the Activity-based Agent Programming Language *AAPL* [16]. An activity is related with a state and actions performed by activating an activity by a transition. *AAPL* models can be compiled to various agent processing platform architectures including programmable code-based stack machines [17] and non-programmable, part of each sensor node. The non-programmable agent processing platform implements the ATG behaviour directly [16], enabling very low-resource single microsystem platforms. In the case of the programmable agent platform the program code implements the agent behaviour entirely. The ATG can be modified at run-time by the agent itself using dedicated *AAPL* transformation statements [18]. By using program code that is executed on a Virtual Machine (VM) this is performed by code morphing techniques provided by the VM. Agents carrying the code, data, and already applied modifications, are capable to migrate in the network between nodes [18]. Both processing platform architectures use token-based and pipelined agent processing for optimized resource sharing and parallel execution.

The programmable agent processing platform used for the execution of agents is a pipelined stack-based virtual machine, with support for code morphing and code migration. This VM approach offers small sized agent program code, low system complexity, high data processing performance, and enables the support of heterogeneous network ranging from microchips to the internet. 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. For material-integration, there is an application specific agent processing platform that implements the agent behaviour on-chip, offering the lowest resource and chip area requirements.

The mechanical model of the structure under investigation allows in particular the pre-computation of a sufficiently accurate discretization of the forward mapping 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.

As discussed above, Tikhonov regularization is a feasible regularization scheme, computing an approximation to the true load l as solution to the linear system

$$(T^*T + \alpha)l = T^*s + \alpha l_0, \quad (3)$$

where T^* denotes the transpose of T .

The solution to this system is hence computed rapidly with low cost if one is able to pre-compute a 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 $T^*Tl = T^*g$ 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 [9][19].

Combining Self-organizing Multi-Agent Systems (SoMAS) and event-based sensor data distribution 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 and previously published work [16][18][17][20]:

- Complete multi-domain simulation of large-scale multi-agent systems, sensor networks, and numerical computation with a unified database centric simulation environment.
- Sensor signal preprocessing at run-time inside the sensor network by using multi-agent systems. Event-based sensor data distribution and pre-computation with agents reduces communication and overall network activity resulting in reduced energy consumption.
- 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 connected to generic computer networks and the Internet.
- Enhanced inverse numeric improving stability and accuracy is used to compute the load for a structure from noisy and incomplete sensor data.
- An advanced on- and off-line processing with off-line inverse numerical computation of mechanical loads from on-line preprocessed sensor data allows the determination of the system response. FEM simulation is used to 1. Retrieve the inverse numeric for real-world sensing system; and 2. For the creation of synthetic sensor data of the technical structure in simulation.
- Rigorous analysis and evaluation of the load computation for a structure with temporal and spatial disturbed or incomplete sensor input data. Temporal disturbance is caused by data distribution latencies, and spatial disturbance is caused by technical failures in the sensor network (e.g., communication failures).

The next sections summarize the activity-based agent processing model, available mobility and interaction, and the used agent processing platform architecture related to the programming model. A summarized description of the sensor signal processing algorithms and a rigorous description of the proposed inverse numerical methods for load computations are following, which are profiled and validated with an extensive simulation framework.

2. Agent Behaviour Model

The agent behaviour is composed and modelled with an activity graph, with activities representing the control state of the agent reasoning engine, and conditional transitions connecting and enabling activities, shown in Fig. 2 (detailed description in [16][20]). Activities provide a procedural agent processing by sequential execution of imperative data processing and control statements.

The activity-graph based agent model is attractive due to the proximity to the finite-state machine model, which simplifies the hardware implementation, but still enabling efficient software implementations.

An activity is activated by a transition which can depend on a predicate as a result of the evaluation of (private) agent data related to a part of the agents belief in terms of Belief-Desire-Intention (BDI) architecture. An agent belongs to a specific parameterizable agent class *AC*, specifying the behaviour, local agent data (only visible for the agent itself), types, signals, activities, signal handlers, and transitions.

The class *AC* can be composed of sub-classes, which can be independently selected. Plans are related to *AAPL* activities and transitions close to conditional triggering of plans.

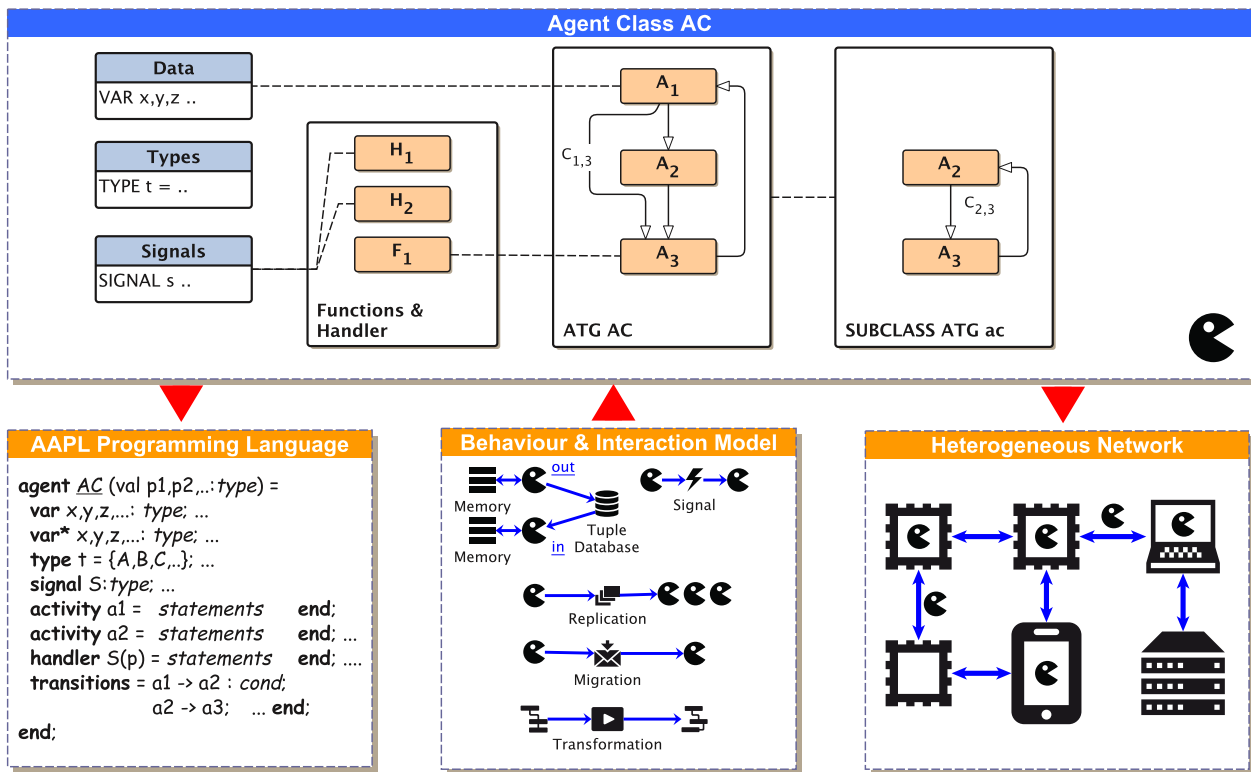


Figure 2. Agent behaviour programming level with activities and transitions (*AAPL*, left); agent class model and activity-transition graphs (top); agent instantiation, processing, and agent interaction on the network node level (right) [17].

This reactive behaviour can be summarized to the following operational semantic:

- Procedural data processing takes place in activities computing and changing private and global data.
- Transitions between activities represent the progress and the external visible change of the control state of an agent. Transitions can be conditional depending on the evaluation of agent data.
- Body variables of an agent are private data only visible to the specific agent. The data content of body variables are mobile and can be inherited by forked child agents.
- Each agent owns a public set of agent parameters initialized on agent creation.
- Global data is exchanged by using a tuple database and synchronized and atomic read, test, remove, and write operations.
- Agents can migrate between different physical and spatially distinguished execution platforms by preserving and transferring the control and data state of the agent.
- The agent behaviour can be either implemented directly by the processing platform (application specific and static platform class) or can be implemented with program code executed by a generic agent processing platform (dynamic platform class).
- Agents can be created at run-time, regardless of the platform class. Agents can inherit the control and data state from parent agents (forking behaviour).
- Agents can communicate and synchronize peer-to-peer by using signals, which can be delivered to remote execution nodes, too.

A short notation of *AAPL* used to specify the agent behaviour in this work is presented in App. A.

3. Agent Processing Platform

There are two different agent processing platforms capable of processing of the mobile agents used in this work: a non-programmable and a programmable platform architecture. The non-programmable platform implements the agent behaviour for different agent classes directly, whereas the programmable platform provides generic agent processing of program code supplied externally which implements the agent behaviour of a particular agent class. Both platforms can be implemented entirely on hardware (SoC), or software, or simulation model level. All implementations of each platform class can be deployed and connected in a heterogeneous network environment, shown in Fig. 3. They are compatible on operational and interface level. That means agents can migrate between different platform implementations and different host environments. The simulation of MAS can be based on a pure behavioural model (BSIM) by simulating the agents itself or based on an architectural model by simulating the agent processing platform (PSIM) using agents.

Tab. 1 compares the characteristics and the advantages/disadvantages of both platform architectures. Fig. 3 shows the modelling and synthesis of agent platforms from a common model and programming sources

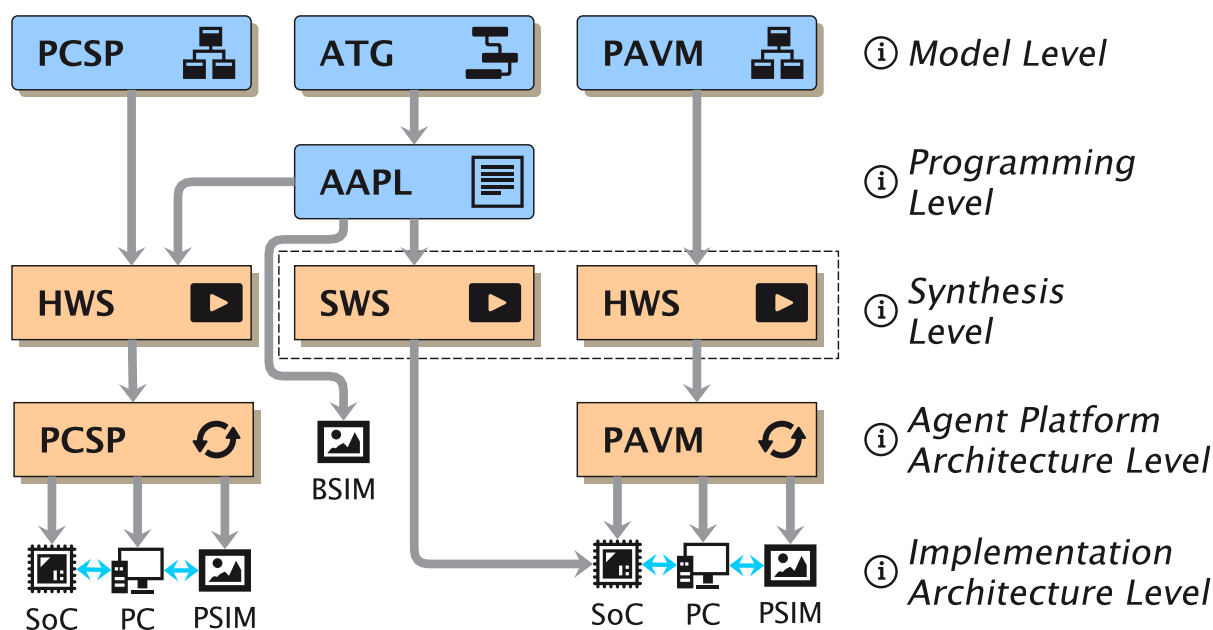


Figure 3. Different agent processing platform architectures and implementations, but a common agent behaviour model and programming sources. (*PCSP*: Pipelined Communicating Sequential Processes, *PAVM*: Pipelined Agent Forth Virtual Machine, *ATG*: Activity-Transition Graph, *AAPL*: Agent Programming Language, *HWS*: Hardware Synthesis, *SWS*: Software Synthesis, *PSIM*: Platform Simulation, *BSIM*: Behavioural Agent Simulation)

	Programmable PAVM	Non-programmable PCSP
Approach	<ul style="list-style-type: none"> • Program code based approach • Data is embedded in code • Zero-operand instruction format • Stack memory centric data processing model • Platform is generic • Code embeds instructions, configuration (control state), and data • Migration: code transfer 	<ul style="list-style-type: none"> • Application-specific approach • Platform is application-specific • Activities of the ATG are mapped to processes • Token-based agent processing • ATG reconfiguration by switching token channel paths • Migration: data and control state transfer (including configuration table)
Hardware	<ul style="list-style-type: none"> • Optimized Multi Stack Machine • Each stack processor is attached to a local code segment and two stacks shared by all agents. There is no data segment! • Single SoC Design • Multiprocessor architecture with distributed and global shared code memory • Multi-FSM RTL hardware architecture • Automatic Token-based agent process scheduling and processing • Code morphing capability to modify agent behaviour and program code (ATG modification) • Data- and code word sizes can be parameterized 	<ul style="list-style-type: none"> • Pipelined Communicating Processes Architecture composition implementing ATG and token-based agent processing • Single SoC Design • Optimized resource sharing - only one <i>PCSP</i> for each agent class implementation required • Activity process replication for enhanced parallel agent processing • For each agent class there is one <i>PCSP</i> with attached data memory (agent data). • Single SoC Design • LUT configuration matrix approach for ATG reconfiguration
Software	<ul style="list-style-type: none"> • Multi-Threading or Multi-Process software architecture • Inter-process communication: queues • Software model independent from programming language • VM sources for various programming languages: <i>C</i>, <i>ML</i>, <i>Javascript</i>, ... • Can be embedded in existing software 	<ul style="list-style-type: none"> • Multi-Threading software architecture • Optimization: Functional composition and implementation of ATG behaviour instead <i>PCSP</i> • Interprocess-communication: queues • Software model independent from programming language • Source code for various programming languages: <i>C</i>, <i>ML</i>, ... • Can be embedded in existing software
Simulation	<ul style="list-style-type: none"> • Agent-based Platform simulation • Generic simulation model - can execute machine code directly • Processor components and managers are simulated with agents 	<ul style="list-style-type: none"> • Agent-based platform simulation • Application-specific simulation model • ATG activity processes are simulated with agents

Table 1. Comparison of both agent processing platform architectures and their implementations (*PCSP*: Pipelined Communicating Sequential Processes, *PAVM*: Pipelined Agent Forth Virtual Machine)

4. Sensing with Multi-Agent Systems

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.

In this work, three different 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.

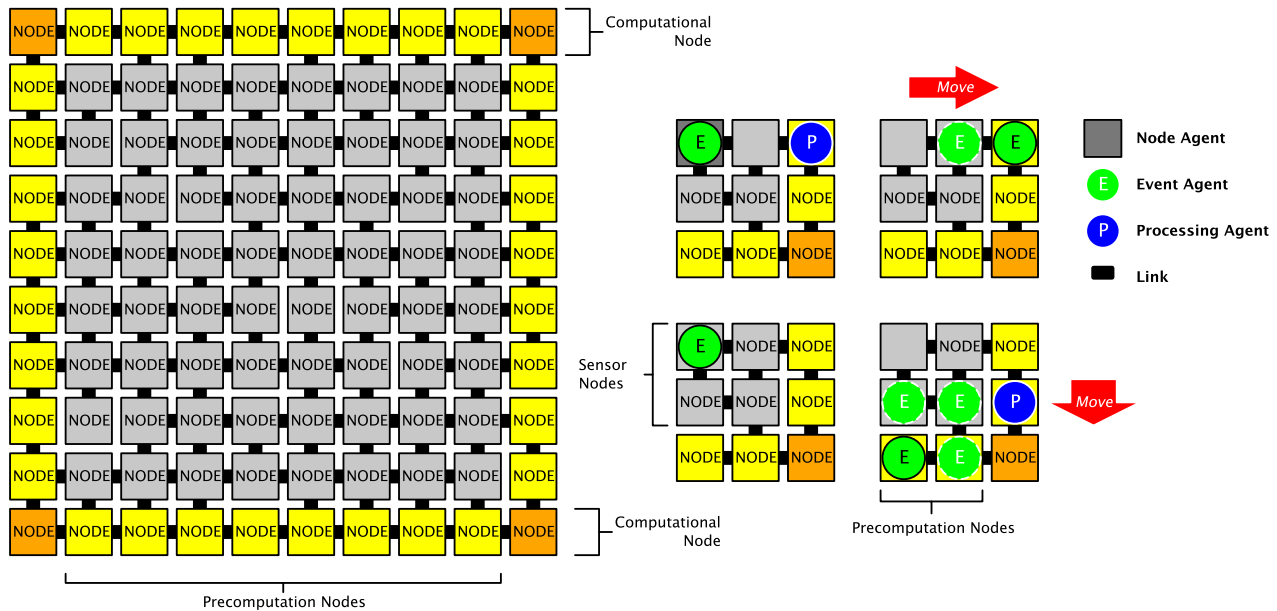


Figure 4. The logical view of a sensor network with a two-dimensional mesh-grid topology (left) and examples of the population with different mobile and immobile agents (right): event deliver, node, and computational processing agents. The sensor network can contain missing or broken links between neighbour nodes.

It is assumed that sensor nodes arranged in a two-dimensional grid network (as shown in Fig. 4) providing spatially resolved and distributed sensing information of the 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. 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 use mobile agents to deliver their sensor data to dedicated computational nodes located at the edges of the sensor network, shown in Fig. 4, 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.

4.1. Event-based Sensor Data Processing with Agents

The computation of the system response information requires basically the complete sensor signal matrix \mathcal{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 6.

The sensor processing uses both stationary (non-mobile) and mobile agents carrying data, illustrated in Fig. 5 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 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. The agent behaviour is specified in Alg. 2 in App. A., and an overview of the agent behaviour and the ATG can be found in Fig. 5 on the right side.

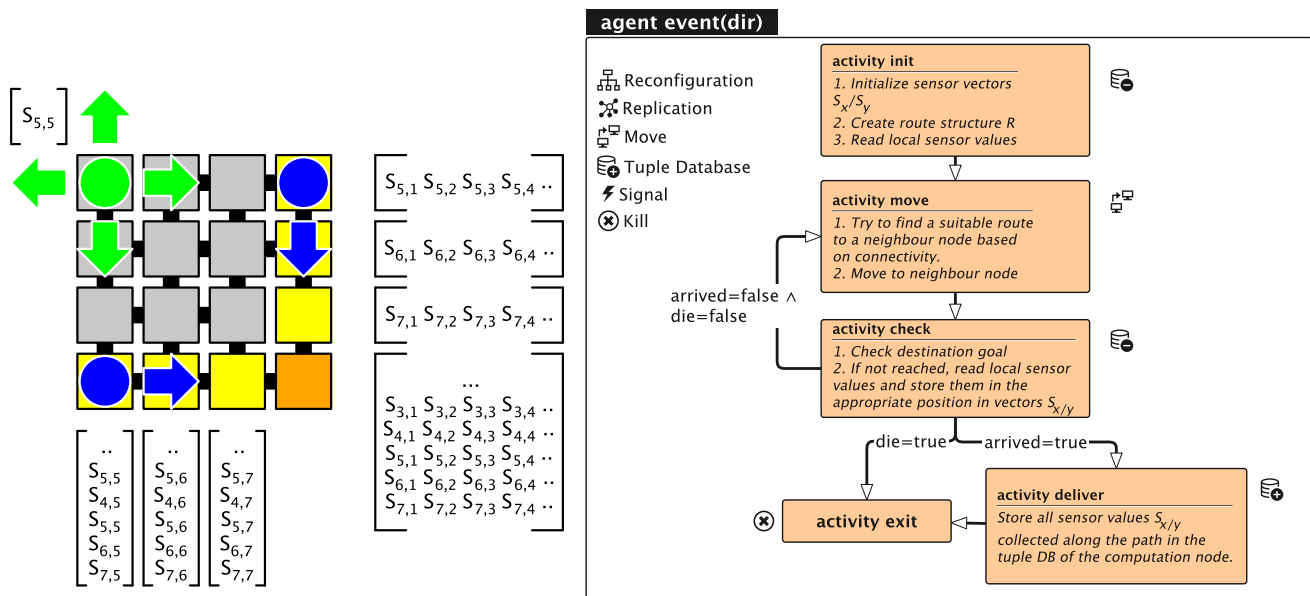


Figure 5. Left: Sensor data distribution with event (green) and pre-processing agents (blue): A sensor node which detected a significant change of the sensor values creates event agents which are sent in all four directions to the network boundary (pre-computation nodes). Right: ATG behaviour model of the event agent.

An event agent has a pre-defined path in the direction *dir* which 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 not possible to migrate in the pre-defined direction, an alternative path is chosen by using the `route_opposite` routing strategy, which chooses a path away from the original destination to bypass not connected 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 columns of the sensor matrix S . If their data changes, the pre-computation nodes will sent 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.

4.2. Self-organizing MAS and Feature Recognition

The event-based sensor data distribution relies on well operating sensors and trustful sensor values. Faulty or noisy sensors can disturb the further data processing algorithms (inverse numeric) significantly and should not delivered to the computational nodes. Usually sensor values are correlated within a spatially bounded region. The goal of the following MAS is to find the outline of extended correlated regions of increased sensor stimuli which can be distinguished from the neighbourhood. For example, strain gauge sensors which deliver information about mechanical distortion of a material resulting from externally applied load forces. A single stimulated sensor cannot be a result from a mechanical load. To find such a correlated stimulated sensor region, a distributed directed diffusion behaviour and self-organization (see Fig. 6) are used, derived from the image feature extraction approach (proposed by [15]). A single sporadic sensor activity not correlated with the surrounding neighbourhood should be distinguished from an extended correlated region by marking nodes nearby the boundary of the region, which is the feature to be detected (edge detector).

Sensor nodes detecting a significant change of their sensor values send out explorer agents which perform the feature detection and marking. Sensors nodes sending out the feature recognition agents but get no markings can decrease a trust probability for their sensor values for further assessment and activity planning.

The feature detection is performed by the mobile *exploration agent*, which supports three main different behaviours: exploration, 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 mark a specific node with the coordinate (i,j) is given by Eq. 4.

$$H(i, j) = \sum_{u=-R}^R \sum_{v=-R}^R \{ |s(i+u, j+v) - s(i, j)| \leq \delta \} \quad (4)$$

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 $s(i+u, j+v)$ with $i, j \in \{-R, \dots, R\}$ 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 $\partial\epsilon = [\epsilon_0, \epsilon_1]$, the exploration agent has detected the feature and will stay at the current node to reproduce new exploration agents sent to the neighbourhood.

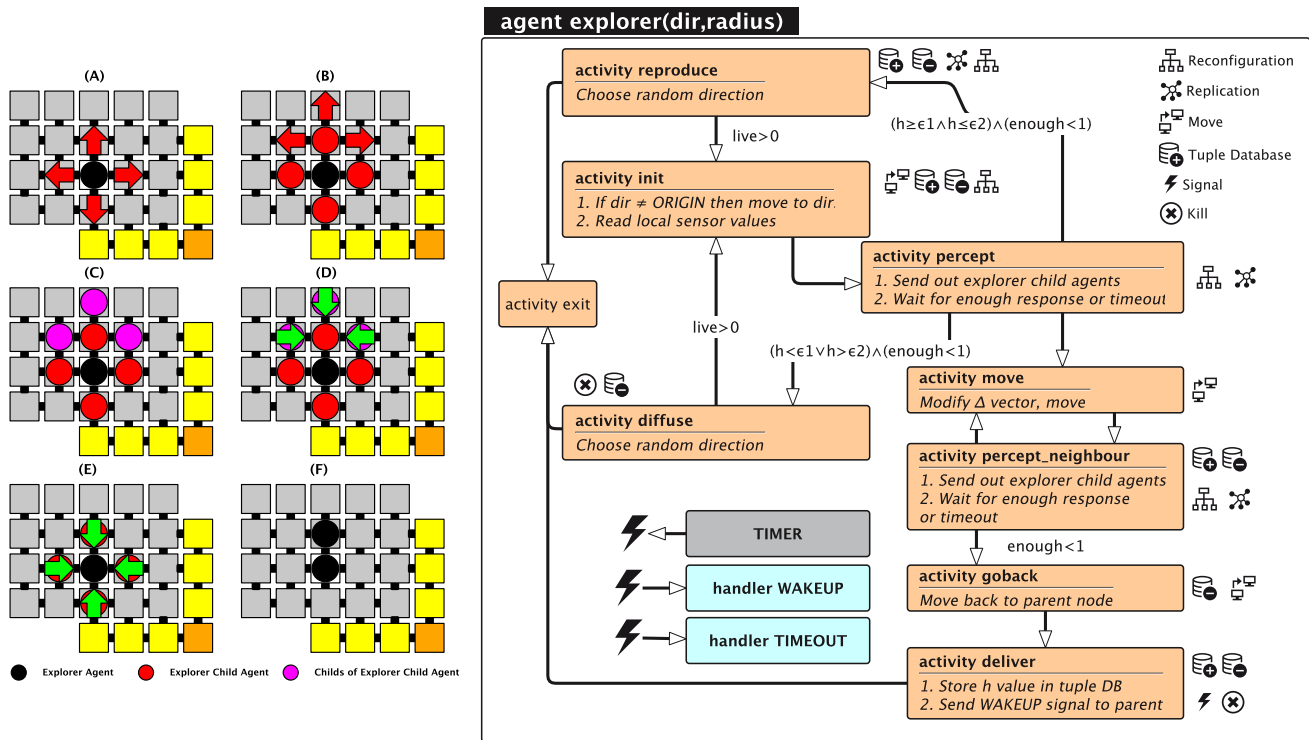


Figure 6. Left: Feature marking of stimulated sensor clusters by using explorer agents investigating the neighbourhood of their current position using forked child agents. On successful feature detection explorer agents are replicated, otherwise they diffuse to the neighbourhood to find features. Right: ATG behaviour model of the explorer agent and the explorer child agent class branching from the percept activity.

If H is outside this interval, the agent will migrate to a neighbour different node and restarts exploration (diffusion).

The agent behaviour is specified in Alg. 4 in App. A., and an overview of the agent behaviour and the ATG can be found in Fig. 6 on the right side. An initial root explorer agent is instantiated by the node agent with a direction argument ORIGIN. This explorer agent will read the local sensor values from the tuple database. In this work there are two strain-gauge sensors connected to each node (sensor data s_x/s_y). The root agent will send out explorer child agents to all connected neighbour nodes (in activity *percept*). These child agents compute a partial term of the H calculation by sending out additional explorer child agents (in activity *percept_neighbour*) until the boundary of the ROI is reached. To avoid multiple visiting of a node by different child agents of the same exploration group, a marking is set on each visited node (a tuple with a limited lifetime removed by a garbage collector). If there is already a marking, an explorer child agent will go back immediately to its parent agent node location and delivers the computed partial term h of $H(i,j)$. An explorer or explorer child agent which sent out additional child agents will wait (sleep) until all child agents had returned their computation results or a time-out occurs. Data is exchanged between child and parent agents by using the tuple space database and synchronization (wake-up) is handled by using signals.

4.3. SoMAS Simulation and Evaluation

To evaluate the capabilities of the feature marking SoMAS introduced in the previous section, the simulation environment described in Sec. 5. 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).

Fig. 7 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 decision interval $\partial\varepsilon=[\varepsilon_0, \varepsilon_1]$. In all simulations a setting of $\partial\varepsilon=[3,6]$ was used.

With a parameter set $\{R=1, L=1\}$ the sharp boundary of the sensor stimuli is detected reliably 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 $CL=8$. 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.

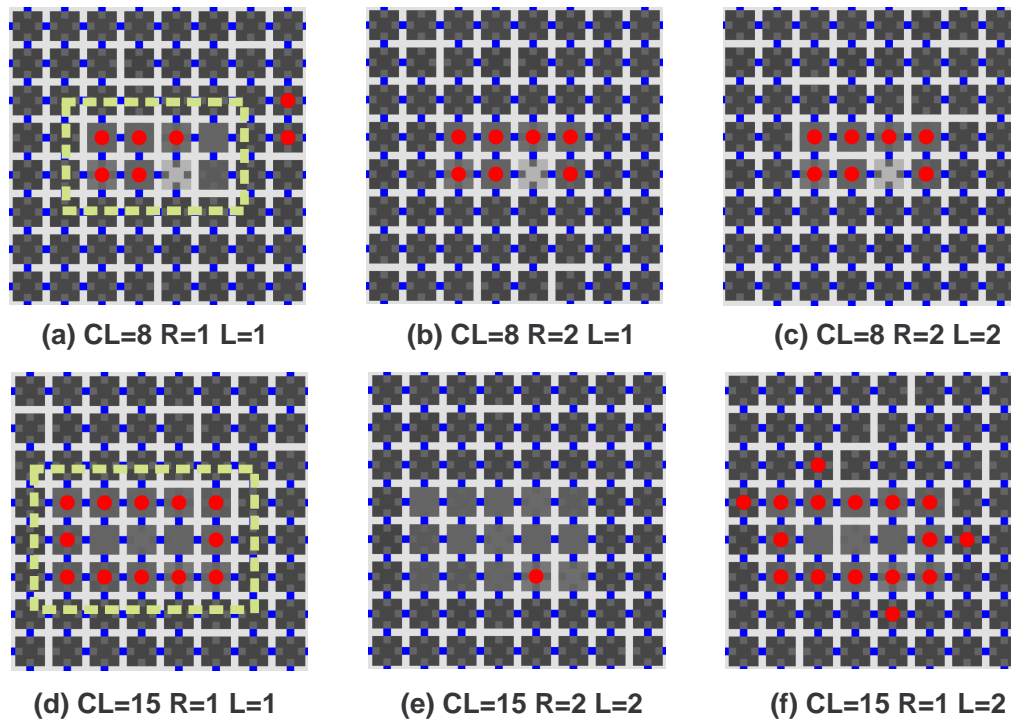


Figure 7. SoMAS Feature Marking (red circles) with a localized rectangular sensor stimuli region having a sharp boundary (yellow dotted line). CL : Cluster size, R : Exploration radius, L : Explorer lifetime, Network connectivity $CN=70\%$

In Fig. 8 the feature detection is applied to data sets retrieved from load and strain simulations of a steel plate using FEM simulation described in Sec. 6., which leads to a more continuously sensor stimuli distribution without having a sharp boundary.

The first data set related with a specific load case has a significant increase of sensor values at the east side of the network. The boundary feature detection SoMAS reliably 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 $\partial\epsilon$. The $R=1$ setting finds again the west side of the sensor stimuli related with 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 Sec. 4.1. The quality of the feature detection depends on the parameter set $\{R, \partial\epsilon\}$, which can be adjusted at run-time by using reinforcement learning performed by the agents based on a quality feedback from the computational nodes.

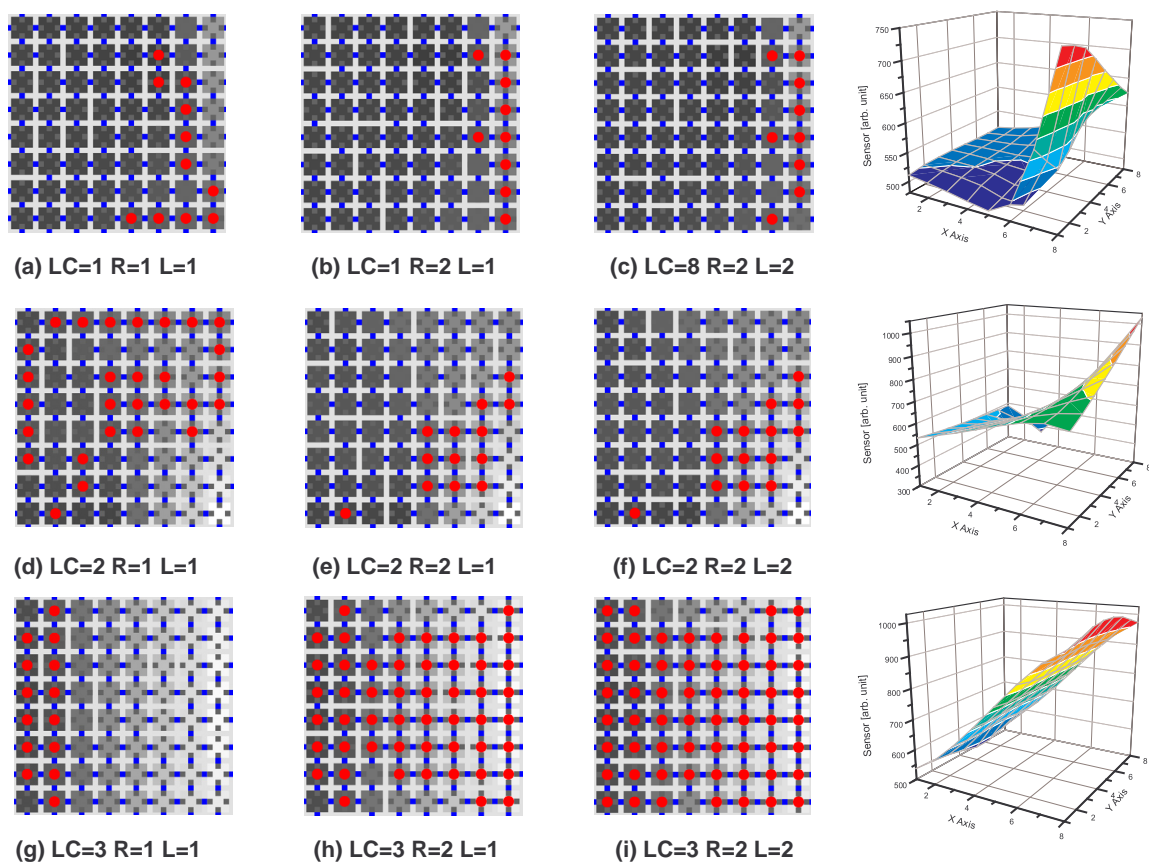


Figure 8. SoMAS Feature Marking (red circles) with a large area sensor stimuli region having no clearly defined boundary (continuous change). *LC*: Load case, *R*: Exploration radius, *L*: Explorer lifetime, Network connectivity $CN=70\%$

5. Simulation

In this work a multi-domain simulation technique is used to study the effects of technical failures, sensor noise, and computational dependencies on the computation of structure loads from disturbed and incomplete sensor data, both in temporal and spatial dimensions. Simulation domains are related here to different levels of the sensing environment, basically the communication network, the agent processing, the sensing process, computational mechanics, and FEM simulation used for the computation of T and delivering synthetic sensor data.

The simulation of the data processing and the physical simulation using FEM and numerical computation approaches are combined in one unique database centric and agent-based simulation environment.

The outcome of numerical computations depends on the sensorial input data, which can be disturbed by the physical sensing process itself and by technical features and failures, for example, caused by communication (temporal latency, lost of data). Commonly Monte Carlo experiments are used in simulation experiments for computing some general time-dependent system function $F(s, E, t)$ of an initial synthetic sensor data input $s = \{s_1, s_2, \dots, s_n\}$ by adding noise δ (e.g., common distributed) to the original input data, i.e., $s' = \{s_1 + \delta_1, s_2 + \delta_2, \dots, s_n + \delta_n\}$, creating multiple simulation experiments with sets s'_1, s'_2, \dots of different artificially pertubated input data with variances of the original input data set and the outcome of F :

$$\begin{aligned} I : F(s, E, t) &\xrightarrow[\text{Technical-Failure}]{\text{Monte-Carlo}} \{I_1 : F(s + \delta_1, E_1), I_2 : F(s + \delta_2, E_2), \dots\} \\ E &\mapsto \{\text{Communication, Processing, Sensing}\} \end{aligned} \quad (5)$$

The outcome of the sensor processing (the desired information) is the mechanical load I that depends additionally on a particular environmental setting E , i.e., network connectivity, communication error rates, data distribution and processing algorithms, and the processing platform lifeliness, disturbed by probalistic processes, too. I.e., the system function F includes the inverse computation, but additionally technical failures arising in the sensor network, e.g., communication failures between nodes, missing or faulty sensor data, temporal effects (collected sensor data is updated with a latency creating incomplete input data sets at a specific time), and effects of distributed computing algorithms on the sensor data (rounding errors, missing data, old data,...) are considered in the simulation resulting in a more realistic simulation of a technical sensing system. This is covered by different experiments with different environmental settings E_i that parameterizes a particular simulation run. In the best case, the output of the computation function F should asymptotical converge to the expected information output, in the worst case F delivers completely different results, analysed in the next sections.

The complete simulation framework is shown in Fig. 9. The central part of the simulation framework is a SQL database server enabling the data exchange and synchronization between different programs, mainly the Multi-Agent Simulator *SeSam* [21] and the *MATLAB* program used for the inverse numerical computation. A Remote Procedure Call (RPC) interface provides synchronization between the programs of the framework. All programs are communicating with the database server by using named file system pipes. This approach has the advantage to require only a native file system interface to connect a heterogeneous program environment, supported basically by all programs. No program modification and no special database or inter-process communication modules are required.

The behavioural simulation of the agents based on the *AAPL* model using the *SeSam* agent behaviour model [21] - which is only a partial sub-set of the *AAPL* model - requires the application of some transformation rules:

- a. Each *AAPL* agent class AC_i is implemented with a *SeSam* agent class SC_i .
- b. Each *AAPL* sub-class $AC_{i,j}$ is implemented with a *SeSam* agent class SC_j . At run-time different agents exist derived from each sub-class.
- c. Functions and procedures of an *AAPL* agent class AC_i must be implemented with *SeSam* feature class FC_i .

- d. *AAPL* signal handlers must be implemented with a separated *SeSAM* agent class $SC_{i,sig}$. At simulation time each *AAPL* agent having signal handlers is associated with a shadow agent of the class $SC_{i,sig}$.
- e. Signals ξ are passed by synchronized queues.
- f. *AAPL* activities a_i which contain blocking statements (tuple space access and waiting for timeouts) require a split into a set of computational and blocking *SeSAM* activities $a_i \Rightarrow \{a'_{i,1}, a'_{i,2}, a'_{i,3}, \dots\}$
- g. Migration of agents is only virtual by changing the position of a *SeSAM* agent and connecting the agent to the new node agent infrastructure, i.e., changing the data scope. Migration of agents requires the migration of the shadow agents (signal handler agents), too.

The sensor network simulation and the SoMAS- and event-based sensor data distribution is used for the proof and the profiling of the inverse numerical methods, discussed in Sec. 6. Fig. 10 shows the temporal resolved agent population (in simulation time steps) for the experiments performed for Sec. 6. and the feature recognition marking resulting from the SoMAS feature recognition (explorer and explorer child agents), triggering the sensor data distribution by the event and distributor agents.

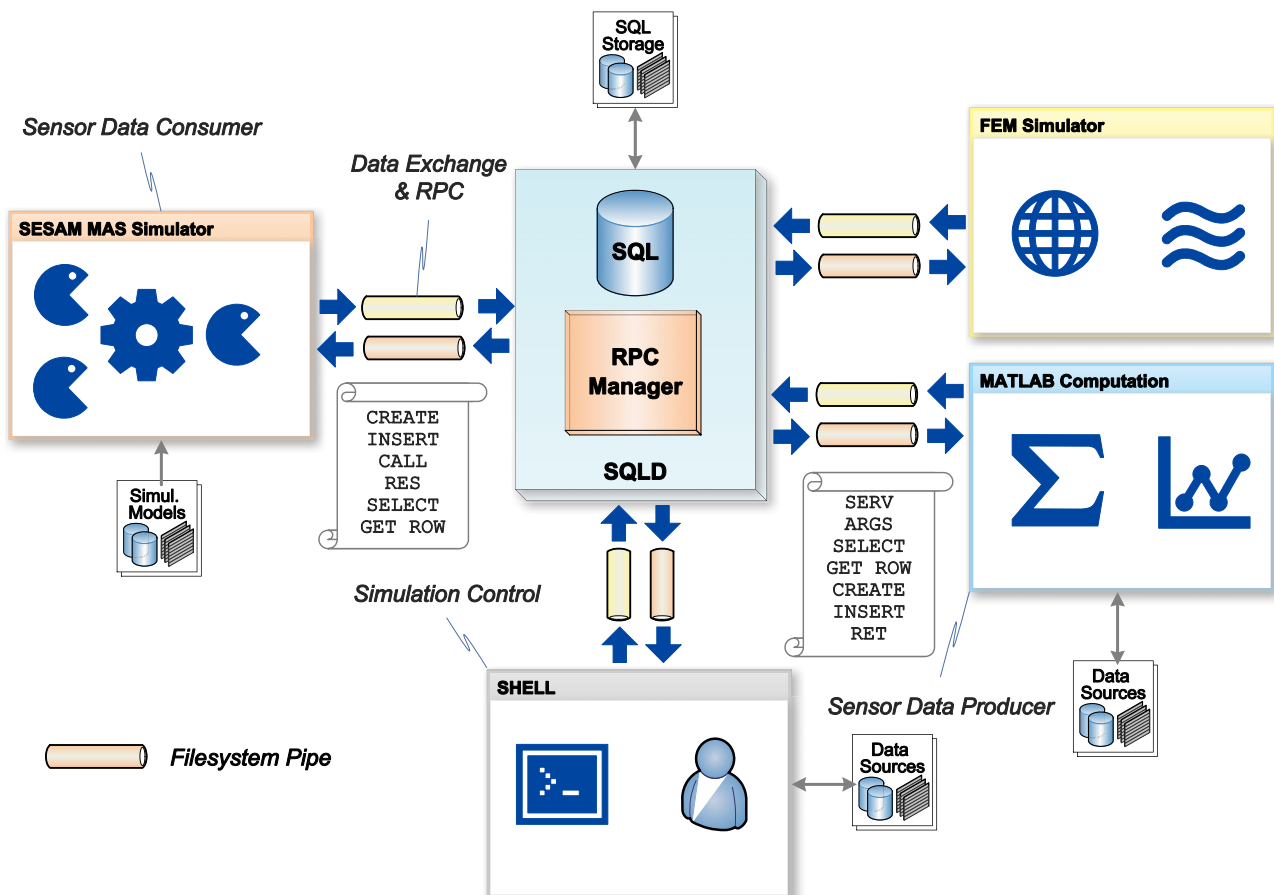


Figure 9. Simulation Framework with a database approach: Multi-Agent Simulator *SeSAM*, *MATLAB*, and other utility programs are exchanging data and synchronizing using a SQL database server, which provides a RPC interface for synchronization, too.

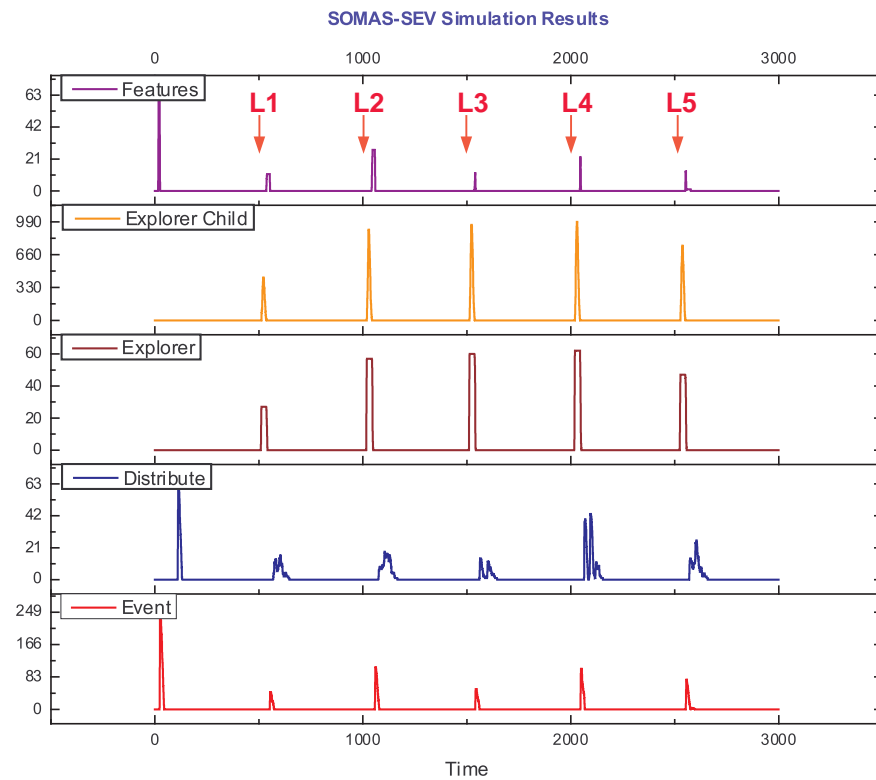


Figure 10. Agent population of the SoMAS- and event-based sensor data distribution in the sensor network used in the following Sec. 6.3. (L_i : i -th load case data set).

6. Inverse Methods for Structural Load Monitoring

Inversion algorithms for the computation of loads from measured surface strains are based on a mechanical model of the considered structure and require stabilization due to intrinsic ill-posedness such that they typically rely on regularization methods [9][19]. If such a model is discretized and if a loading on the structure is given, numerically solving the discretized system allows to approximate the response of the structure to the load. Thus, a mechanical model first allows to simulate signals measured by sensors on or in the structure that are due to loads. This yields a mapping \mathbf{T} such that for given load \mathbf{l} , the sensor signals equal $\mathbf{T}\mathbf{l}$. The mechanical model allows in a second step to deduce loads from sensor signals \mathbf{s} by solving the equation $\mathbf{T}\mathbf{l}=\mathbf{s}$ for the load \mathbf{l} .

Our goal is to demonstrate that the latter task can be tackled by an embedded sensor network relying on inverse methods based on models from linear elasticity.

Obviously, numerical simulations always involve several errors, most obviously the discretization error and the modelling error due to the choice of the model, but also errors due to imprecise knowledge of the plate's geometry or the sensor positions. Finally, any experimentally measured sensor signal is affected by measurement errors, too, due to limited digital precision leading to round-off errors or incorrect data transmission. Numerical precision limited to few digits is a crucial restriction of accuracy in particular for embedded and autonomous sensor networks.

6.1. Computational Setting and Pre-computations

We consider load monitoring assuming a sensor network that is embedded in a thin plate of size 0.5m x 0.5m x 0.02m consisting of construction steel. This plate is fixed at one of its four vertical sides. We

limit ourselves to the case of loads that are so small that the equations of linear elasticity provide an accurate physical model, i.e., stresses caused by such loads are below the yield strength.

We moreover assume in our numerical experiments that the loaded structure is isotropic and homogeneous, such that the material behaviour can be described by its elastic modulus E and its Poisson ration ν . Moreover, any (spatially variable) load on the structure is assumed act constantly in time during a certain time interval, such that the deformation field $\mathbf{u} = (u_1, u_2, u_3)^\top$ is static during this time interval and can be described by the equations of static linear elasticity,

$$(\lambda + \mu)\nabla \operatorname{div} \mathbf{u} + \mu \Delta \mathbf{u} = 0 \quad \text{in } \Omega, \quad (6)$$

where λ and μ are Lamé parameters defined by $\lambda = \nu E / [(1+\nu)(1-2\nu)]$ and $\mu = E / (2(1+\nu))$. The deformation field vanishes at the boundary where the plate is fixed and satisfies traction and free boundary conditions on the top and remaining sides, respectively. The (linearized) strain tensor then equals $\varepsilon(\mathbf{u}) = 1/2 (\nabla \mathbf{u} + (\nabla \mathbf{u})^\top)$, which allows to compute surface strains once the deformation field has been approximated numerically.

The structure's response to a load hence is elastic deformation. Note that loads are always placed on the upper horizontal surface of the plate, such that the associated forces act in normal direction to the surface.

We assume that the sensors embedded in the material measure surface strain on the lower horizontal surface of the plate at $M = M_x M_y$ sensor positions in a grid of sensors of size $M_x \times M_y$. More precisely, these sensors measure surface strain on the lower surface of the plate at the sensor position, both in the x - and in the y -direction.

Using a finite element discretization we simulate a finite number of loads caused by cylindrical weights with (cross-section) radius of 0.0125m placed at $N = N_x N_y$ equidistant grid points

$$\mathbf{w}_{i,j} = \begin{pmatrix} (0.25m + (i-1)0.5m) / N_x \\ (0.25m + (j-1)0.5m) / N_y \\ 0.02m \end{pmatrix}, \quad i = 1, \dots, N_x, j = 1, \dots, N_y, \quad (7)$$

that form a grid of $N_x \times N_y$ points on the upper side of the plate. Simulating the deformation field caused by these weights allows to compute the resulting surface strain at sensor points on the lower side of the plate.

The corresponding equidistant sensor positions are

$$\mathbf{m}_{i,j} = \begin{pmatrix} (0.25m + (i-1)0.5m) / M_x \\ (0.25m + (j-1)0.5m) / M_y \\ 0.00m \end{pmatrix}, \quad i = 1, \dots, M_x, j = 1, \dots, M_y. \quad (8)$$

This way of simulating the reaction of the plate under loading is motivated by the experimental set up presented in [1].

We emphasize that our finite element discretization does neither model the embedded sensors nor any communication infrastructures in between these sensors. Thus, the resulting simulated sensor value are merely accurate for sensors of small size that do not interact strongly with the plate; arguably, this

inaccuracy is negligible to test the ability of a sensor network in connection with a mobile agent platform to monitor loads.

Note that the grid of weight points $w_{i,j}$ and of sensor points $m_{i,j}$ yield a natural surface quadrangulation of the upper and lower surface of the plate, respectively, that we will use later on to image applied and reconstructed loads.

Due to the chosen model of linear elasticity, the relation between applied load and simulated strain is linear and hence gives rise to a linear function mapping loads to surface strain. Moreover, due to discretization, this mapping is finite dimensional and can hence be represented by a real-valued matrix T with $2M$ rows and N columns. (Since we assume that sensors measure surface strain in x - and y -direction, the dimension of the image space equals $2M$, i.e., twice the number of sensors.)

This so-called load-strain matrix T maps vectors modelling discretized loads to surface strains at the sensor positions. Since we will later on monitor loads by stably inverting this matrix, this procedure of assembling the load-strain matrix obviously introduces a further discretization error for any load that cannot be represented as a linear combination of the above-mentioned cylindrical weights.

If $l \in \mathbb{R}^N$ is a vector modeling loads applied to the upper surface, the resulting strain $s \in \mathbb{R}^{2M}$ hence equals the matrix-vector product

$$s = Tl. \quad (9)$$

Thus, in contrast to the preceding sections where loads or strains were considered as matrices l or $s_{x,y}$, we start from now on to consider them as vectors to be able to use the latter matrix operation in between them. Both points of view are of course equivalent if we agree on the following one-to-one correspondence: The entry $l(i,j)$ of a load matrix l corresponding to the weight point $w_{i,j}$ is the $(N_y-1)i+j$ th entry of the vector l . Further, the entry $s_x(i,j)$ of a strain matrix s_x is the $(M_y-1)i+j$ th entry of the vector s and the entry $s_y(i,j)$ of s_y is the $M+(M_y-1)i+j$ th entry of the vector s .

To compute the load-strain matrix T we used the C++-based and open-source finite-element software *FreeFem++* [23]. In detail, we set up a uniform and regular tetrahedral volume mesh of the plate with a mesh width of 0.11 mm and used globally continuous and piece wise quadratic basis functions ($P2$ elements) for each of the three components of the deformation field. The dimension of the resulting linear system for each of the M loads is about 11.5 million. Note that the system matrix needs to be set up merely once for all forward computations as the columns of T correspond to different loadings due to the N weights at the points $w_{i,j}$ from Eq. (7), i.e., to different right-hand sides of the finite-element system. Using a parallel solver on a workstation with eight cores and 32 GB RAM it took about two days to do all simulations.

Note that the standard theory of finite element methods implies that the approximate solutions converge to the exact in the energy norm as the mesh width tends to zero. When the approximation error is measured in the quadratic mean (i.e., in the L^2 -norm), then a convergence rate in between one and two is achieved, and if the exact solution is sufficiently smooth (which is not always the case in our examples), then the convergence rate of the deformation field is higher than quadratic.

However, we are ultimately interested in the surface strains, that is, in derivatives of the deformation field, which converge in the quadratic mean with a rate that is one order smaller than the one of the deformation field itself.

As the solution of the N linear systems to compute the load-strain matrix \mathbf{T} is part of the pre-computations and does influence the inversion technique merely by the accuracy of the simulated strains, we did not opt to speed up this process by, e.g., resorting to simpler two-dimensional models for the deformation field or the strain tensor (as, e.g., Kirchhoff–Love plate theory) but simply tackled the full three-dimensional problem directly and ensured that the results are sufficiently accurate by taking a small mesh width. (The pre-computations took about two days on a modern workstation, using parallel computations.) Note further that we compared the computed surface strains against corresponding results for smaller mesh widths and obtained in all experiments presented below a relative error in the strain data of less than one percent.

The plot of the entries of such a load-strain-matrix on the right of Fig.11 for $N=400$ weights and $M=64$ sensors shows in each column the (absolute values of 128 surface strains computed from one finite-element computation; strains in x and y -direction are contained in the upper and lower half of the plotted matrix, respectively. As the plate is fixed at one side parallel to the x -axis, strain in x -directions is generally smaller (due to darker colours in the plot) than strain in y -direction. The singular values of that load-strain matrix are plotted on the left of Fig. 11 and indicate a relative noise level of about 0.002.

After computing the load-strain matrix \mathbf{T} the inversion task required for load monitoring is to stably compute load vectors \mathbf{l} that satisfy the equation $\mathbf{T}\mathbf{l}=\mathbf{s}_m$ for given inaccurate strain measurements \mathbf{s}_m . This task is intrinsically difficult since the matrix \mathbf{T} is ill-conditioned:

As Fig. 11 shows, the singular values of \mathbf{T} for $M_x=M_y=8$ and $N_x=N_y=20$ decrease rapidly, which implies that small errors in the data make an accurate load reconstruction impossible. Already a relative error in the data of 1% implies that in general merely less than twenty singular values will remain accurate; for relative noise levels of more than 10%, less than five singular values remain accurate.

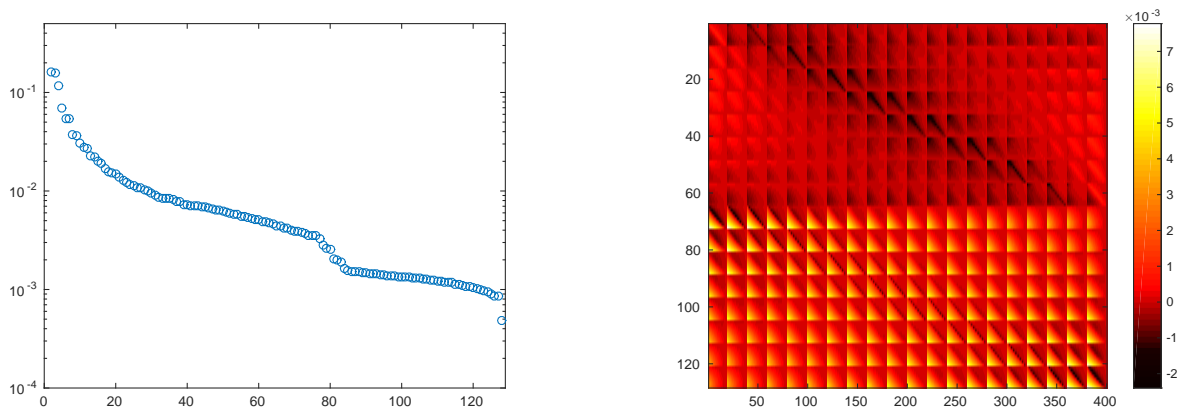


Figure 11. On the left: 128 non-zero singular values of load-strain matrix $\mathbf{T} \in \mathbb{R}^{2M \times N}$ for $M=64$ simulated sensors $M_x=M_y=8$ and $N=400$ simulated weights ($N_x=N_y=20$). On the right: Absolute values of the entries of the load-strain matrix \mathbf{T} . Since the steel plate is fixed at one vertical side parallel to the x -axis, strain in x -direction is generally smaller than strain in y -direction.

Since the number of columns of \mathbf{T} equals the number of weight positions N that is in general different from the number of strain measurements, the load-strain matrix \mathbf{T} is in general not square and cannot be inverted -- even if the matrix was square, one should refrain from inverting it due to ill-conditioning.

A possible remedy is to consider the following least-squares problem for $\mathbf{l} \in \mathbb{R}^N$,

$$\mathbf{T}^* \mathbf{T} \mathbf{l} = \mathbf{T}^* \mathbf{s}_m, \quad (10)$$

that is derived from the linear system $\mathbf{T}\mathbf{l} = \mathbf{s}_m$. (As above, \mathbf{T}^* is the transpose matrix of \mathbf{T})

One can indeed show that a vector \mathbf{l} solving Eq. (10) minimizes the functional $\mathbf{l} \rightarrow \|\mathbf{T}\mathbf{l} - \mathbf{s}_m\|_2$, where we recall that $\|\cdot\|_2$ denotes the 2-norm (or the Euclidean norm), see Eq. (2). The linear system in Eq. (10) features a quadratic matrix that is however even more ill-conditioned than before and, for this reason, should not be inverted directly, i.e., the system is under determined. Instead, we advertise to stabilize the computation of an approximate load vector \mathbf{l} using so-called inverse (aka. regularization or inversion) methods. We refer to [9] and [19] for an introduction to such methods.

6.2. Inverse Methods

In this section, we will introduce two inversion methods that we afterwards compare in view of their applicability in the framework of load monitoring using multi-agent systems in an embedded sensor network as introduced in Sec. 4. As discussed in the first part of this paper, propagating sensor measurements by a Self-organizing Multi-Agent system to the four computation nodes (see Fig. 4) introduces significant errors in the measured data arriving at the computation node due to different arrival times or broken network connections.

Thus, the strain vector \mathbf{s}_m available for computations is in general incomplete and contains a very high level of noise. This fact causes severe difficulties to any inverse method; nevertheless, even for such a difficult setting with incomplete data and a drastically under-determined inversion problem, we will see later on that such methods are still able to provide some meaningful information.

Note that the proceedings paper [18] contains several numerical example of loads reconstructed from sensor network data where the computation node is aware of all strain measurements without data corruption due to data propagation. In such an idealized setting, one can study the dependence of the reconstruction quality on the number of sensors and the noise level in the data after adding artificial noise to the simulated load-strain matrix \mathbf{T} and the simulated strain vectors \mathbf{s}_m . We refer to [18] for results in this direction.

As a first technique, we consider the classical Tikhonov regularization, where one stabilizes the inversion by requiring that \mathbf{l} does not minimize the 2-norm of the residual $\mathbf{T}\mathbf{l} - \mathbf{s}_m$ as in Eq. (10) but the stabilized functional

$$\mathbf{l} \mapsto \|\mathbf{T}\mathbf{l} - \mathbf{s}_m\|_2^2 + \alpha \|\mathbf{l}\|_2^2, \quad (11)$$

where $\|\cdot\|_2$ denotes the 2-norm (or Euclidean norm) of a vector. The positive and small parameter α should to be chosen in dependence of the (expected) noise level in the measured data and the modelling and computation error in the load-strain matrix \mathbf{T} , see, e.g., [9][19].

It is not difficult to compute that the minimizer of this quadratic minimization problem solves the linear equation

$$(\mathbf{T}^*\mathbf{T} + \alpha)\mathbf{l} = \mathbf{T}^*\mathbf{s}_m. \quad (12)$$

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 \mathbf{T} must anyway be pre-computed before the monitoring device is launched, and hence one can directly pre-compute a singular value decomposition of \mathbf{T} , boiling down the solution of the linear system to matrix multiplications:

Assume that (U, D, V) is a singular value decomposition of the matrix T , such that the three matrices U , D , and V allow to decompose T as

$$T = UDV^*. \quad (13)$$

Moreover, D is a diagonal matrix of size $M \times N$, U and V are orthogonal matrices of size $M \times M$ and $N \times N$, respectively (i.e., $U^*U = I_M$ and $V^*V = I_N$ where I_N is the unit matrix of size $N \times N$). We denote the diagonal elements of the matrix D by d_i , $i = 1, \dots, \min(N, M)$, and can hence write $D = \text{diag}(d_i)$. The d_i are the so-called singular values of T . Together with the orthogonality of the matrices U and V , they allow to write to the Tikhonov regularization l , solution to Eq. (12) as

$$l = V \text{diag}(d_i / (d_i^2 + \alpha)) U^* s_m. \quad (14)$$

This inversion method hence multiplies the k -th strain value (i.e., the k -th entry of s_m) with the k -th column of the matrix $K = V \text{diag}(d_i / (d_i^2 + \alpha)) U^*$ and sums up all resulting column vectors to obtain l . Note that the matrix-vector multiplication in Eq. (14) is the only higher-order operation that has to be executed by the computation node to compute l . Since the matrix K is of size $N \times M$, this requires M scalar products of the rows of K with s_m and hence MN scalar multiplications and $(N-1)M$ additions of binary numbers with a fixed number of digits.

A disadvantage of Tikhonov regularization is the smoothing property of the scheme. For accurate data, i.e., at noise levels below 1%, 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 (see [18]). This makes a precise location of the support of a load difficult, in particular when several loads act simultaneously on the structure. However, the advantage of the Tikhonov regularization scheme described in Eq. (14) is the stability of the technique for high noise level and even incomplete data.

The second technique we consider for load monitoring is the conjugate gradient (cg) iteration applied to the normal equation $T^*Tl = T^*s$, an iterative technique where the k -th iterate l_k minimizes the discrepancy $\|Tl - s\|_2$ in the so-called Krylov subspace spanned by the vectors

$$T^*s, (T^*T)T^*s, (T^*T)^2T^*s, \dots, (T^*T)^{k-1}T^*s. \quad (15)$$

Alg. 1. The cg-algorithm applied to the linear system $Tl = s$ with starting vector l_0 (pseudo-notation)

Input: Matrix T , right-hand side s , starting vector l_0 ,
expected relative noise level $\delta > 0$

```

 $r_0 := s - T \cdot l_0$ ; // residuals
 $d_0 := T^* \cdot r_0$ ;
 $p_1 := T^* \cdot r_0$ ;
 $j := 1$ ;
while  $d_{j-1} \neq 0 \wedge \|r_{j-1}\|_2 > \delta \|s\|_2$  do
     $q_j := T \cdot p_j$ ;
     $a_j := \|d_{j-1}\|_2^2 / \|q_j\|_2^2$ ;
     $l_j := l_{j-1} + a_j \cdot p_j$ ;
     $r_j := r_{j-1} - a_j \cdot q_j$ ;
     $d_j := T^* \cdot r_j$ ;
     $\beta_j := \|d_j\|_2^2 / \|d_{j-1}\|_2^2$ ;
     $p_j := d_j + \beta_j \cdot p_{j-1}$ ;

```

$j := j + 1;$

Due to the ill-conditioning of the load-strain matrix \mathbf{T} , the cg-iteration applied to noisy data \mathbf{s}_m with relative noise level $\delta = \|\mathbf{s}_m - \mathbf{s}_{exact}\|_2 / \|\mathbf{s}_{exact}\|$ has to be stopped whenever the discrepancy of the actual iterate reaches the noise level δ of the sensor signals. Of course, for measured data, this noise level can only be guessed, since it is composed of measurement errors, modelling errors, discretization errors and data propagation errors. A standard way how to proceed with measured data is to test the cg-iteration for several measurements and several guesses of ε when the true load is known and to choose a guess for ε in dependence of the quality of the results. For small noise level, i.e., for accurate data, the experiments in [18] show that the conjugate gradient iteration usually outperforms Tikhonov regularization when reconstruction loads with small support. The advantage of this method is its speed as it is for instance known to compute iterates reaching a given discrepancy earliest among all Krylov subspace methods. We will however note in the experiments presented in the next section that a disadvantage of the cg-iteration is its sensitivity to lacking strain data. Further, the method shows less stability than Tikhonov regularization at high noise levels in the particular context of our setting.

Estimating the required work load for the cg-iteration is more difficult than the Tikhonov regularization, since the number of times the while-loop in Alg. 1 is repeated is variable. Generally, more accurate data implies a higher number of repetitions. In numerical experiments with 20% relative synthetic white noise added to the data, the loop was repeated up to nine times and about 7.5 times in average. In each loop, two matrix-vector multiplications must be computed, once with \mathbf{T} and once with \mathbf{T}^* ; these require NM scalar multiplications each, plus $(N-1)M$ and $(M-1)N$ additions for \mathbf{T} and \mathbf{T}^* , respectively. This shows a disadvantage of the cg-method for load monitoring in a sensor network: The resulting vector \mathbf{l} cannot be expressed via one matrix-vector product and pre-computation strategies for this method are not obvious.

6.3. Numerical examples

The simulated experiments we present in the rest of this section follow the setting we already outlined in Sec. 6.1.: To pre-compute the load-strain matrix \mathbf{T} , we use a conforming finite element method with $P2$ elements and simulate static loading of a 0.02m thick quadratic building steel plate Ω with side lengths of 0.5m that is fixed at one of its four vertical sides. As material parameters of the homogeneous and isotropic plate we fix the elastic modulus E of 210 kN/mm² and the Poisson's ration ν of 0.3. Further, we use the equations of static linear elasticity as a model for the deformation field $\mathbf{u} = (u_1, u_2, u_3)^T$ (see Eq. (6)). As mentioned above, the deformation field \mathbf{u} is governed by Eq. (6) and the strain measurements are simulated as point values of the derivatives of u_1 and u_2 at the sensor nodes.

When pre-computing the load-strain matrix the simulated loads are cylindrical weights placed at $N=400$ weight positions from an equidistant rectangular grid (the grid points are defined in Eq. (7) for $N_x=N_y=20$). The force on the upper surface of the upper horizontal plate due to the loading hence vanishes outside the circle covered by the weight; inside this circle the force points in direction $-z$ and equals 1 N/cm². As discussed in Sec. 6.1., after computing the deformation field, we compute the surface strain in x and y direction at the sensor points given in Eq. (8) where $M=64$ (since $M_x=M_y=8$). By computing the deformation field and the extracted surface strain for a sequence of refined meshes, we checked that the relative error of the strain data used in the examples below is below one percent.

Apart from the cylindrical loads to compute the load-strain matrix T we also simulated five pairwise different loads $I^{(1)}, \dots, I^{(5)}$ with different characteristics and acting on different parts of the steel plate, see Fig. 12.

The simulated strain values are then converted into integer values in between 1 and 1024, a zero signal corresponding to the value 512; if ε_i denotes a simulated strain value, this conversion is done using the formula $s_i = \lfloor 512 + 10000 \cdot \varepsilon_i \rfloor$, $1 \leq i \leq 2M$. ($\lfloor a \rfloor$ denotes the largest integer smaller than or equal to $a \in \mathbb{R}$.) Thus, five strain measurement vectors $s^{(1)}, \dots, s^{(5)}$ are computed. We use these five data sets and feed them consecutively as sensor values into the simulation framework for the sensor network shown in Fig. 4: Choosing a fixed intermediate time interval of 500 simulation steps, we start by stimulating the sensor network using the sensor data s_1 ; during the next 500 simulation steps, the network identifies the load and propagates the sensor data to the computational nodes. After 500 time steps we stimulate the network by inserting the sensor data s_2 , and wait again 500 time steps until we stimulate again relying on s_3 , and so on. In an intact network, the multi-agent system requires about 100 time steps to identify an activated region and to send out event agents to the computational nodes. In a defective network where some connections between sensor nodes are broken, this procedure takes more time; the more connections are defect, the more time is required to propagate information through the network.

The identification of a loading event and the distribution of sensor data works as explained in Sec. 4.: If a sensor node notes a significant change of one of its sensor values (i.e., strain in x - or y -direction), he sends out explorer agents.

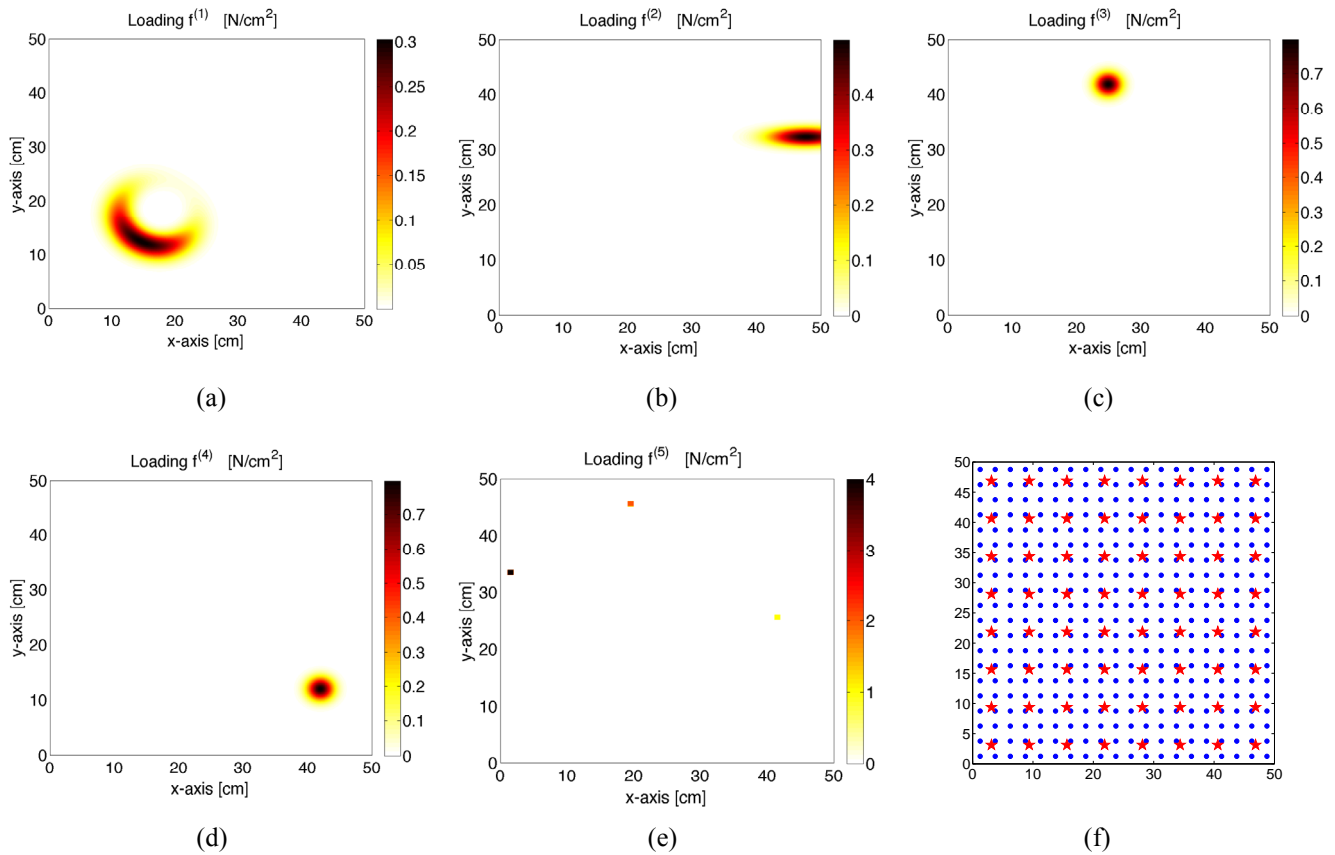


Figure 12. From (a) to (e): The five loads $I^{(1)}, \dots, I^{(5)}$. (f) The positions of the 400 weight points $w_{\{i,j\}}$ are indicated using blue dots; red crosses indicate the positions of the 64 sensors.

In case that the agent system detects an activated region (see Sec. 4.2.), each sensor node in the boundary of this region sends out four event agents in the four directions of the network. Each event agent tries to propagate the sensor values of its creating node and of all nodes passed on its way through the network to one of the four computation nodes in the four corners of the network. For simplicity, we name these four computation nodes according to their position in the network (see Fig. 4): The computation node in the upper left and right corner is called NW (north west) and NE (north east), respectively, and the one in the lower left corner and lower right corner is called SW (south west) and SE (south east), respectively. Note that event agents coming from different nodes will arrive at different times at the computational nodes, in particular, if some of the network connections are broken. The computation nodes collect the incoming sensor values and back-transform them into floating point number via $s_{m,i} = (s_i - 512)/10000$. If no event agents arrive at a computation node, the node computes a reconstruction of the load causing the event agents using either Tikhonov regularization (see Eq. (14)) or the cg-method (see Alg. 1). This implies that possibly not all sensor values from the individual sensors are propagated, that not all sensor nodes know all values from the complete strain data vector s_m , that several values might arrive too late at a computation node and either erroneously cause a further load computation or perturb the computation of the next load.

Whenever a computation node starts a computation with incomplete sensor data (which is rather common than an exception in our experimental setting), then the missing values in s_m are set to 0. It is at this point obvious that the noise level in s_m is usually enormous, typically way beyond 15%. Due to possibly late arrival of sensor data, the noise level of the data moreover becomes time-dependent and the entire monitoring process also becomes a time-dependent inversion problem.

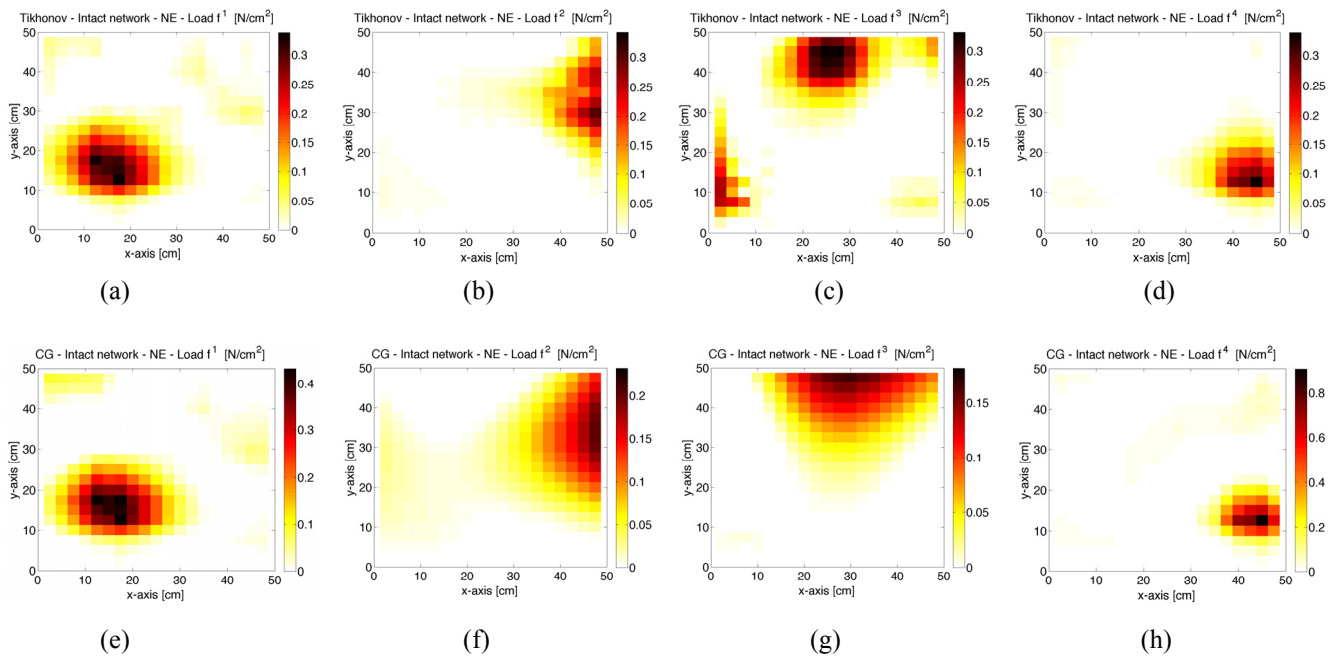


Figure 13. Loads inverted by the NE computation node of the self-organizing sensor network based on mobile agents. All connections in the network are intact. (a)-(d): Loads $l^{(1)}, \dots, l^{(4)}$, Tikhonov regularization Eq. (14), $\alpha = 6 \cdot 10^{-5}$. (e)-(h): Loads $l^{(1)}, \dots, l^{(4)}$, cg method (Alg. 1), $\delta = 0.2$.

Fig. 13 shows the successively monitored loads of the computation node NE during the simulation time of an intact sensor network as sketched in Fig. 4, i.e., all connections in the network are faultless. Both methods generally detect the correct position of a load, however, a precise shape detection is

impossible given the quality of the data. The third loading $I^{(3)}$ poses problems to both algorithms; the Tikhonov regularization detects a load on the lower left side of the plate while the cg-method over-estimates the magnitude of the load. (The latter behaviour is a typical feature of this method in our setting.)

The fifth load $I^{(5)}$ consists of three point-like loadings with relatively high magnitude (see Fig. 12 (e)). Both inverse methods have significant difficulties to cope with this situation: Fig. 14 shows reconstructions using both methods by two different computation nodes (NE, SW). While NE is via Tikhonov regularization able to detect the three loads, the cg-method merely indicates the presence of the highest load at the left side of plate (both estimated magnitude are wrong). The data arriving at SW is not sufficient to reasonably determine any feature of the load. However, Fig. 14 shows another typical feature of the cg-method: If there is not enough sensor data available to reasonably reconstruct at least parts of the load, then the method produces highly oscillating results with very high amplitudes that are obviously wrong. We will see later that such output of the method can reliably be filtered by rejecting monitored loads with high amplitudes.

Since the sensor network is self-organizing via the multi-agent system, the strain data are not arriving at the same time at the computation nodes. As discussed above, it might hence happen that a computation node reconstructs a load before all relevant strain data has arrived. Independently of the chosen inverse method, an updated computation is in this case often significantly better than the result gained from incomplete data, as Fig. 15 indicates.

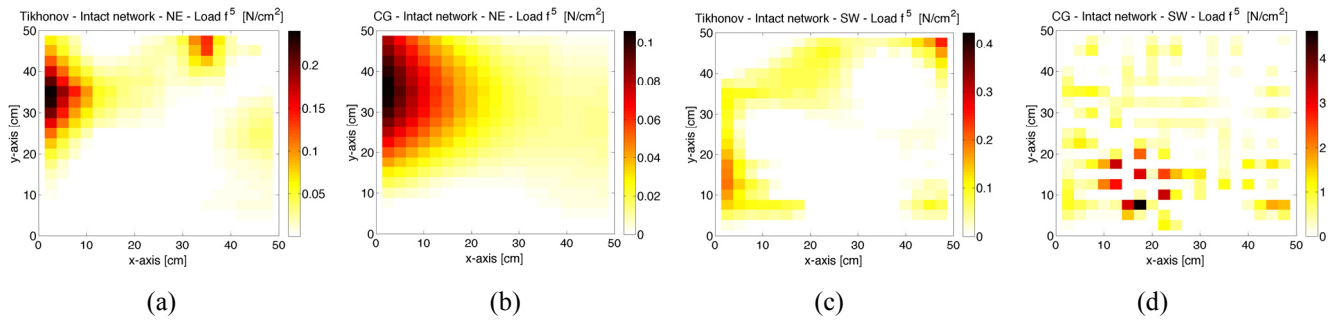


Figure 14. Load $I^{(5)}$ inverted by the NE and SW computation nodes of the self-organizing sensor network based on mobile agents. All connections in the network are intact. (a) NE node, Tikhonov regularization Eq. (14), $\alpha = 6 \cdot 10^{-5}$. (b) NE node, cg method (Alg. 1), $\delta = 0.2$. (c) SW node, Tikhonov regularization Eq. (14), $\alpha = 6 \cdot 10^{-5}$. (d) SW node, cg-method (Alg. 1), $\delta = 0.2$.

A typical phenomenon of the sensing system appears in Fig. 15 (e) and (g): The reconstruction of $I^{(5)}$ in Fig. 15 (e) by the north-western computation node NW using Tikhonov regularization is perturbed by data stemming from the strain data s_4 corresponding to $I^{(4)}$. The analogous observation holds for the cg-method applied to the same data sets (see Fig. 15 (e)-(h)).

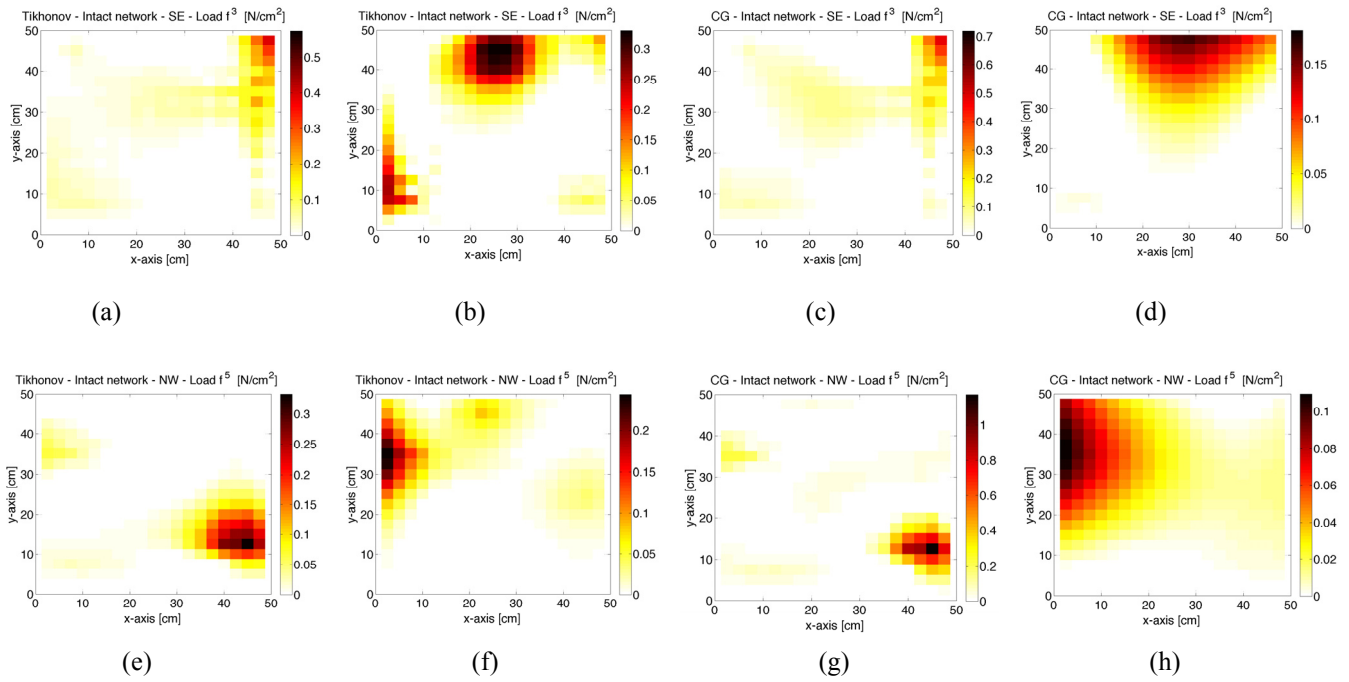


Figure 15. When sensor data arrives at a computation node after a load has already monitored, the node re-computes the load. All indicated pairs of plots show two consecutive reconstructions of a load due to late sensor data arrival; the network is intact. (a)-(b) Two reconstructions of $l^{(3)}$ by SE node; Tikhonov regularization Eq. (14), $\alpha = 6 \cdot 10^{-5}$. (c)-(d) Two reconstructions of $l^{(3)}$ by SE node; cg method (Alg. 1), $\delta = 0.2$. (e)-(f) Two reconstructions of $l^{(5)}$ by NW node; Tikhonov regularization Eq. (14), $\alpha = 6 \cdot 10^{-5}$. (g)-(h) Two reconstructions of $l^{(5)}$ by NW node; cg-method (Alg. 1), $\delta = 0.2$.

We already noted above that the magnitudes of the reconstructed loads are in some cases roughly speaking more sensitive than the overall position of the loads. To this end, we plot in Fig. 16 (a) the maximal force of a reconstructed load for 52 computations (26 Tikhonov and 26 cg reconstructions) by the four computation nodes during the above-described test of the sensing system. In 10 cases, the maximal loading of a cg reconstruction is significantly larger than the corresponding result of the Tikhonov regularization; in all these cases, the corresponding monitored load is unusable and should be rejected. We emphasize that this is a good feature of the method since it allows to easily identify incorrectly reconstructed loads. Note that Fig. 16 (a) further indicates that the magnitudes of the loads computed by the Tikhonov regularization are rather good compared to the exact magnitude (given the accuracy of the data) and, except for the point-like load $l^{(5)}$, rather robust against noise. This is obviously different for the cg method that does not show the same stability in this setting. Fig. 16 (b) moreover indicates that checking the normalized cross-correlation between the two reconstructions given by the two inverse methods can be used as an indicator for their reconstruction quality: If cross correlation factor above 0.8 is in all examples a good indicator for a reasonable reconstruction quality.

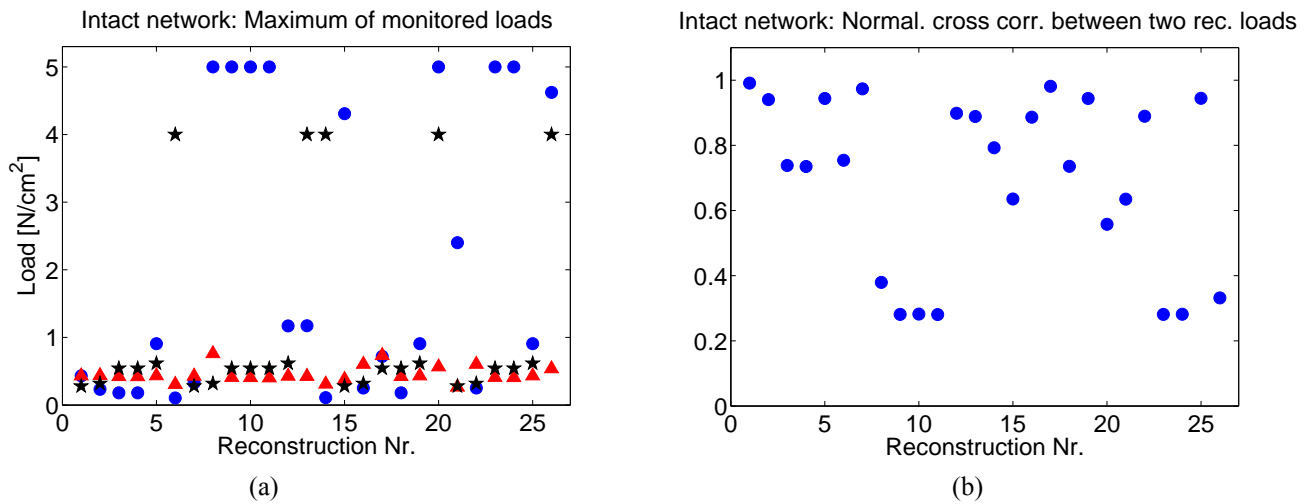


Figure 16. (a) Maximum of the loads by an intact sensor network for 54 reconstructions computed by the four computation nodes NE, NW, SE, SW during the evolution of the simulated system. If the maximum of a monitored load is larger than 5 N/cm² we plot the value 5. (Red triangles: Tikhonov regularization; blue dots: cg-method; black stars: maximum of the exact load.) (b) Normalized cross correlation between the loads reconstructed by Tikhonov regularization and the cg-method for each of the 26 reconstruction events.

In the last part of this section we consider the same test simulation for the sensor network as above, but perturb 30% of all network connections. The agents of the multi-agent system can hence no longer propagate between all neighbouring nodes (and they no a-priori knowledge which connections are defect). Apart from this change of the network topology, the setting of the simulation is precisely the same as above. Thus, compared to the experiments discussed above, sensor data is in this new setting likely to arrive later at computation nodes than before and might, in some cases, not arrive at all. The data quality hence is worse than before. Nevertheless, the monitored loads shown in Fig. 17 are (in the visual norm) of a similar same quality as the corresponding ones from Fig. 13. At several of the plots, the incorrectly appearing yellow regions are somewhat extended.

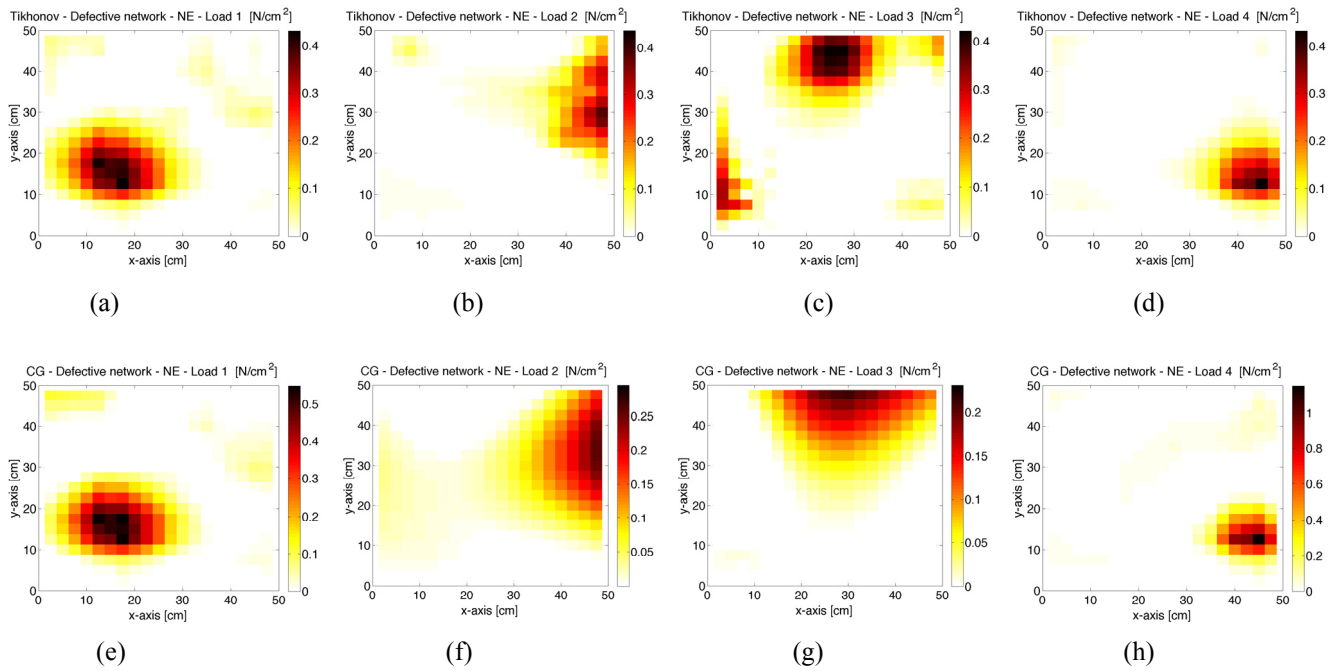


Figure 17. Loads inverted in a defective network by the NE computation node. 30% of all network connections are defective. Parameters for the reconstruction algorithms are the same as for the intact network. (a)-(d): Loads $I^{(1)}, \dots, I^{(4)}$, Tikhonov regularization Eq. (14), $\alpha = 6 \cdot 10^{-5}$. (e)-(h): Loads $I^{(1)}, \dots, I^{(4)}$, cg-method (Alg. 1), $\delta = 0.2$.

Fig. 18 indicates that the phenomenon of sensor data arriving at a computation node after the node already computed a first reconstruction becomes stronger:

In Fig. 18 (a)-(d) one notes data from the the surface strain s_1 corresponding to $I^{(1)}$ appears in the reconstruction of $I^{(2)}$; thus, data from s_1 and s_2 is arriving at about the same type and both loads show up in the same reconstruction.

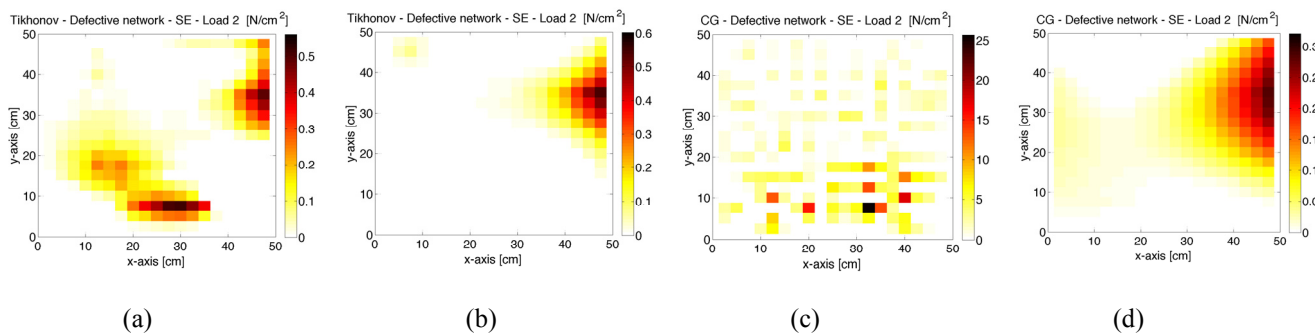


Figure 18. Changing reconstructions of loads in a defective sensor network due to late arrival of data. 30% of all network connections are defective. The plots (a)-(b) and (b)-(c) show two consecutive reconstructions of $I^{(2)}$ by SE node, the update is due to additionally arriving sensor data. (a)-(b) Tikhonov regularization Eq. (14), $\alpha = 6 \cdot 10^{-5}$. (c)-(d) cg method (Alg. 1), $\delta = 0.2$.

Finally checking for the magnitudes of the reconstructed loads and the cross-correlation between the results of the two different inverse methods shown in Fig. 19 allows to draw similar conclusions as above: The perturbed network reduces the data quality and the cg-method accordingly yields results

with too high amplitude more frequently (13 out of 28 cases). However, the results gained by Tikhonov regularization remain more stable, which indicates another time the potential of this technique (and its refinements) for load monitoring in a substantially involved setting.

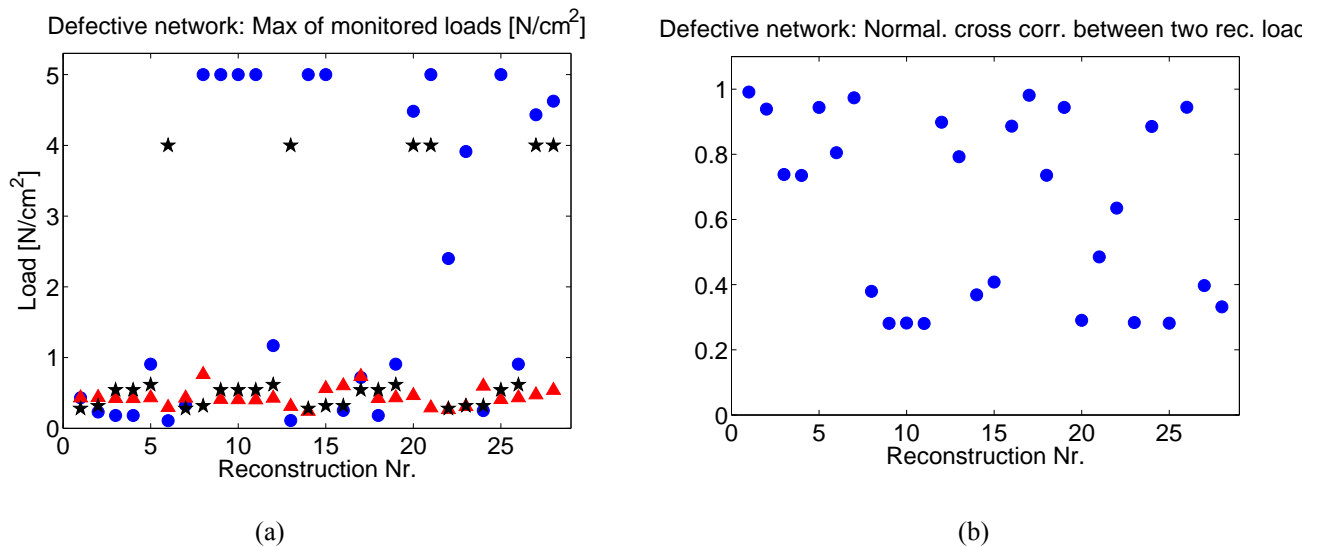


Figure 19. Left: Maximum of the loads monitored by a defective sensor network; 30% of all network connections are defective. The 28 reconstructions are computed by the four computation nodes NE, NW, SE, SW during the evolution of the sensing system. If the maximum of a monitored load is larger than 5 N/cm^2 we plot the value 5. (Red triangles: Tikhonov regularization; blue dots: cg-method; black stars: maximum of the exact load.) Right: Normalized cross correlation between the loads reconstructed by Tikhonov regularization and the cg-method for each of the 28 reconstructions.

7. Conclusions and Outlook

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 and investigated, 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 unknown system response (i.e., the load) based on data from prior FEM simulations of the technical structure.

Throughout this work, a multi-domain simulation was used capable to simulate multi-agents systems deployed in large-scale networks and numerical computation with a unified database centric simulation environment.

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.

The self-organizing load monitoring system presented in this paper shows a partly surprising robustness against missing or incorrect data present at the computation nodes resulting, mainly a result of 1. missing or defective communication links prohibiting sensor data distribution and delivery to the computational nodes and 2. the event-based partial delivery of sensor data. The incomplete sensor data sets resulting from the event-based data distribution and the SoMAS feature detection bases on two effects: 1. not fully covering the stimulated region and 2. recognizing the wrong region, especially in the case of a spatially smooth sensor stimuli distribution. The mismatch of sensor distribution can be partially covered by the four redundant but spatially distributed edge computational nodes at the outside of the network, which performs the final load computations.

As an outlook into future work, we mention two points: It becomes apparent in the second part of the numerical experiments that time-stamping sensor data would make the entire load monitoring system more stable and accurate. This direction of research will be investigated in future projects. Further, the detection of point-like loading (as $l^{(5)}$) is currently insufficient. We will further investigate the possibility to improve the detection of point-like loads and the correct estimation of their correct magnitudes, which is clearly a weak point of the monitoring system presented in this paper. The second work concerns the classification of the four mostly different load computations of the edge computers using machine learning or fusion concepts to filter out the load computation with the highest trust.

The multi-agent behaviour is composed of different agent classes and specified using the agent-orientated programming language *AAPL* based on the Dynamic ATG model, 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. There are programmable and application-specific agent processing platforms suitable for embedded sensor networks, and the programmable platform can be deployed in the Internet domain, too. In case of the programmable platform, 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. Furthermore, the high-level synthesis tool enables the synthesis of different processing platforms implementations from a common specification (programmable platform) or the *AAPL* sources (application-specific platform), including standalone hardware and software platforms, as well as simulation models offering functional and behavioural testing. All platform implementations are compatible on operational and communication level. 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 (programmable platform). 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 child agents) at run-time and relax communication and storage requirements. Additionally, (re-) composition allows the derivation of agent sub-classes from a super class, matching the requirements in SoMAS.

Appendix A. Algorithms

Alg. 2. Agent behaviour of the Event agent class offering a robust event-based and path tracking sensor data distribution

```

1   $\kappa$ : { SENSORVALUE,DISTRIBUTER }           set of key symbols
2   $\delta$ : { NORTH,SOUTH, WEST, EAST, ORIGIN } set of directions
3  MAXFAILED = 4                               some constant parameters
4
5  type Route = (dir =  $\delta$ , lastdir =  $\delta$ , delta =  $\Delta$ , gamma =  $\Delta$ , routed = boolean);
6
7   $\Psi$  Event: (dir)  $\rightarrow$  {
8    Body Variables
9     $\Sigma$ : { route, arrived, failed, die, SX=[0.. $\Delta$ IMX-1], SY=[0.. $\Delta$ IMY-1] } global persistent variables
10    $\sigma$ : { vx, vy, index, found, row, col, rown, coln } local temporary variables
11
12   Activities
13    $\alpha$  init: {
14     arrived  $\leftarrow$  false;
15      $\forall \{i \mid 0 \dots \Delta$ IMX-1  $\}$  do SX[i]  $\leftarrow$  -1;
16      $\forall \{i \mid 0 \dots \Delta$ IMY-1  $\}$  do SY[i]  $\leftarrow$  -1;
17     route  $\leftarrow$  Route(dir,ORIGIN,(0,0),(0,0),false);
18     found  $\leftarrow$   $\nabla^?(0, \text{SENSORVALUE}, vx?, vy?)$ ;
19     if found then SX[0]  $\leftarrow$  vx; SY[0]  $\leftarrow$  vy;
20   }
21    $\alpha$  move: {
22     route.dir  $\leftarrow$  dir; Try different routing strategies to reach the destination
23     route  $\leftarrow$  route_relax(route);
24     if not route.routed then route  $\leftarrow$  route_normal(route);
25     if not route.routed then route  $\leftarrow$  route_opposite(route);
26     if route.routed then  $\Leftrightarrow$ (route.dir) else failed++;
27   }
28    $\alpha$  check: {
29     found  $\leftarrow$   $\nabla^?$ (DISTRIBUTER);
30     if found  $\wedge$  route.gamma=(0,0) then arrived  $\leftarrow$  true
31     else if route.gamma=(0,0) then
32       found  $\leftarrow$   $\nabla^?(0, \text{SENSORVALUE}, vx?, vy?)$ ; Collect all sensor values along delivery path
33       if found then
34         case dir of
35           | NORTH  $\Rightarrow$  index  $\leftarrow$  -route.delta.Y
36           | SOUTH  $\Rightarrow$  index  $\leftarrow$  route.delta.Y
37           | WEST  $\Rightarrow$  index  $\leftarrow$  -route.delta.X
38           | SOUTH  $\Rightarrow$  index  $\leftarrow$  route.delta.X
39         SX[index]  $\leftarrow$  vx; SY[index]  $\leftarrow$  vy;
40     if failed > MAXFAILED then die  $\leftarrow$  true;
41   }
42    $\alpha$  deliver: {
43      $\nabla^?$ (MATRIXDIM,row?,col?,rown?,coln?);
44     index  $\leftarrow$  0;
45     case dir of Deliver all sensor values collected along delivery path
46       | NORTH  $\Rightarrow$ 
47          $\forall$  row  $\in$  { -route.delta.Y-1 .. 0 } do
48            $\nabla^+$ (SENSORVALUE,row,col,SX.[index],SY.[index]); index++;
49       | SOUTH  $\Rightarrow$ 
50          $\forall$  row  $\in$  { rown-route.delta.Y .. rown-1 } do
51            $\nabla^+$ (SENSORVALUE,row,col,SX.[index],SY.[index]); index++;
52       | WEST  $\Rightarrow$ 
53          $\forall$  col  $\in$  { -route.delta.X-1 .. 0 } do
54            $\nabla^+$ (SENSORVALUE,row,col,SX.[index],SY.[index]); index++;
55       | EAST  $\Rightarrow$ 
56          $\forall$  col  $\in$  { coln-route.delta.X .. coln-1 } do
57            $\nabla^+$ (SENSORVALUE,row,col,SX.[index],SY.[index]); index++;
58   }
59    $\alpha$  exit: {  $\otimes$ ($self) }
60   Main Transitions
61    $\Pi$ : {

```

```

62   entry → init
63   init → move
64   move → check
65   check → deliver | arrived = true
66   check → move    | arrived = false ∧ die = false
67   check → exit    | die = true
68   deliver → exit
69 }
70 }

```

Alg. 3. Routing functions

```

1   $\delta$ : { NORTH, SOUTH, WEST, EAST, ORIGIN } set of directions
2  type Route = (dir =  $\delta$ , lastdir =  $\delta$ , delta =  $\Delta$ , gamma =  $\Delta$ , routed = boolean);
3
4  route_normal: (route) → {
5    if ? $\Lambda$ (dir) ∧ route.lastdir ≠  $\varpi$ (dir) then
6      route.routed ← true; route.lastdir ← dir;
7      route.delta ← route.delta +  $\partial$ (route.dir);
8      case route.dir of
9        | NORTH ⇒
10         if route.gamma.Y ≠ 0 then route.gamma ← route.gamma +  $\partial$ (dir);
11        | SOUTH ⇒
12         route.routed ← true; route.lastdir ← NORTH;
13         if route.gamma.Y ≠ 0 then route.gamma ← route.gamma +  $\partial$ (dir);
14        | WEST ⇒
15         if route.gamma.X ≠ 0 then route.gamma ← route.gamma +  $\partial$ (dir);
16        | EAST ⇒
17         if route.gamma.X ≠ 0 then route.gamma ← route.gamma +  $\partial$ (dir);
18      ↑route
19 }
20
21 route_opposite: (route) → {
22   routes ← {d ∈  $\delta$  | ? $\Lambda$ (d) ∧ route.lastdir ≠  $\varpi$ (d) };
23   if routes ≠  $\emptyset$  then
24     route.routed ← true;
25     route.dir ←  $\Re$ (routes);
26     route.lastdir ← route.dir;
27     route.delta ← route.delta +  $\partial$ (route.dir);
28     route.gamma ← route.gamma +  $\partial$ (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 ∧ ? $\Lambda$ (EAST) ∧ route.lastdir ≠  $\varpi$ (EAST) then nextdir ← EAST;
36     if route.gamma.X > 0 ∧ ? $\Lambda$ (WEST) ∧ route.lastdir ≠  $\varpi$ (WEST) then nextdir ← WEST;
37     if route.gamma.Y < 0 ∧ ? $\Lambda$ (SOUTH) ∧ route.lastdir ≠  $\varpi$ (SOUTH) then nextdir ← SOUTH;
38     if route.gamma.Y > 0 ∧ ? $\Lambda$ (NORTH) ∧ route.lastdir ≠  $\varpi$ (NORTH) then nextdir ← NORTH;
39     if nextdir ≠ ORIGIN then
40       route.dir ← nextdir;
41       route.routed ← true; route.lastdir ← dir;
42       route.delta ← route.delta +  $\partial$ (route.dir);
43   ↑route
44 }
45
46  $\partial$ : (dir) → {
47   case dir of
48     | NORTH ⇒ (0, -1)
49     | SOUTH ⇒ (0, +1)
50     | WEST  ⇒ (-1, 0)
51     | EAST  ⇒ (+1, 0)
52 }
53

```

Alg. 4. Agent behaviour of the Explorer agent class using a SoS based feature recognition detecting the boundary of a spatially correlated sensor stimuli region.

```

1  κ: { SENSORVALUE, FEATURE, H, MARK }      set of key symbols
2  ξ: { TIMEOUT, WAKEUP }                   set of signals
3  δ: { NORTH, SOUTH, WEST, EAST, ORIGIN }  set of directions
4  ε1 = 3; ε2 = 6; MAXLIVE = 1; DELTA = 50; some constant parameters
5
6  Ψ Explorer: (dir, radius) → {
7    Body Variables
8    Σ: { dx, dy, live, h, sx0, sy0, backdir, group } global persistent variables
9    σ: { enoughinput, again, die, back, sx, sy, 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 ← ⌘(dir)
16    else
17      live ← MAXLIVE; backdir ← ORIGIN
18      ∇+(H, $self, 0);
19      found ← ∇?(0, SENSORVALUE, sx0?, sy0?)
20  }
21  α percept: {
22    enoughinput ← 0;
23    ∀{nextdir ∈ δ | nextdir ≠ backdir ∧ ?Λ(nextdir)} do
24      enoughinput++;
25      Θ→Explorer.child(nextdir, radius)
26      τ+(ATMO, TIMEOUT)
27  }
28  α reproduce: {
29    live--;
30    ∇×(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 ∈ δ | nextdir ≠ backdir ∧ ?Λ(nextdir)} do
36        Θ→(nextdir, radius)
37      π*(reproduce → exit)
38  }
39  α diffuse: {
40    live--;
41    ∇×(H, $self, ?);
42    if live > 0 then
43      dir ← ℝ{nextdir ∈ δ | nextdir ≠ backdir ∧ ?Λ(nextdir)}
44    else
45      die ← true
46  }
47  α exit: { ⊗($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  $\tau^-(\text{TIMEOUT})$ ;
65 }
66
67 Main Transitions
68  $\Pi$ : {
69   entry  $\rightarrow$  init
70   init  $\rightarrow$  percept | found
71   init  $\rightarrow$  exit |  $\neg$ found
72   percept  $\rightarrow$  reproduce |  $(h \geq \varepsilon_1 \wedge h \leq \varepsilon_2) \wedge (\text{enoughinput} < 1)$ 
73   percept  $\rightarrow$  diffuse |  $(h < \varepsilon_1 \vee h > \varepsilon_2) \wedge (\text{enoughinput} < 1)$ 
74   reproduce  $\rightarrow$  exit
75   diffuse  $\rightarrow$  init | die = false
76   diffuse  $\rightarrow$  exit | die = true
77 }
78 Explorer child sub-class
79  $\varphi$  child: {
80    $\alpha$  exit imported from root class
81    $\xi$  TIMEOUT
82    $\xi$  WAKEUP
83    $\alpha$  percept_neighbour {
84     found  $\leftarrow \nabla^?(\emptyset, \text{SENSORVALUE}, \text{sx?}, \text{sy?})$ ;
85     if found  $\wedge$  not  $\nabla(\text{MARK}, \text{group})$  then
86       back  $\leftarrow$  false; enoughinput  $\leftarrow$  0;
87        $\nabla^-(\text{MTMO}, \text{MARK}, \text{group})$ ;
88       h  $\leftarrow$  (if  $|\text{sx}-\text{sx}_0| > \text{DELTA}$  or  $|\text{sy}-\text{sy}_0| > \text{DELTA}$  then 0 else 1);
89        $\nabla^+(\text{H}, \$\text{self}, h)$ ;
90        $\pi^*(\text{percept\_neighbour} \rightarrow \text{move})$ 
91        $\forall \{\text{nextdir} \in \delta \mid \text{nextdir} \neq \text{backdir} \wedge \nabla^+(\text{nextdir}) \wedge \text{inbound}(\text{nextdir})\}$  do
92          $\Theta^+(\text{nextdir}, \text{radius})$ 
93        $\pi^*(\text{percept\_neighbour} \rightarrow \text{goback} \mid \text{enoughinput} < 1)$ 
94        $\tau^+(\text{ATMO}, \text{TIMEOUT})$ 
95     else back  $\leftarrow$  true
96   }
97    $\alpha$  move: {
98     backdir  $\leftarrow \varpi(\text{dir})$ ;  $(\text{dx}, \text{dy}) \leftarrow (\text{dx}, \text{dy}) + \partial(\text{dir})$ ;
99      $\Leftrightarrow \text{dir}$ ;
100  }
101    $\alpha$  goback: {
102     if  $\nabla(\text{H}, \$\text{self}, ?)$  then  $\nabla^-(\text{MARK}, \$\text{self}, h?)$  else h  $\leftarrow$  0;
103      $\Leftrightarrow \text{backdir}$ ;
104   }
105    $\alpha$  deliver: {
106      $\nabla^-(\text{H}, \$\text{parent}, v?)$ ;  $\nabla^+(\text{H}, \$\text{parent}, v+h)$ ;
107      $\xi \text{WAKEUP} \Rightarrow \$\text{parent}$ ;
108   }
109    $\pi$ : {
110     entry  $\rightarrow$  move
111     move  $\rightarrow$  percept_neighbour
112     percept_neighbour  $\rightarrow$  goback |  $(\text{enoughinput} < 1) \vee \text{back}$ 
113     goback  $\rightarrow$  deliver
114     deliver  $\rightarrow$  exit
115   }
116 }
117 }

```

Appendix B. Notation












<p><u>Tuple Space</u></p> <p>$\nabla^+(TP)$: Add tuple TP to database $\nabla^-(TP)$ $\nabla^{?-(TMO, TP)}$: Read and remove (or try) tuple TP from database $\nabla\%(TP)$ $\nabla^{? \%(TP)}$: Read tuple (or try) TP from database $\nabla^X(TP)$: Remove tuple TP from database $\nabla^T(TMO, TP)$: Add marking tuple TP to database with lifetime TMO $? \nabla(TP)$: Test for existence of tuple TP $x?$: Formal Parameter</p> <hr/> <p><u>Mobility</u></p> <p>$? \wedge(\Delta)$ $? \wedge(\delta)$: Test for connection link in direction Δ δ: Set of directions {NORTH, SOUTH, ...} Δ: Relative position vector and hop vector relative to root node $\partial(\delta)$: Direction position difference vector ϖ: Opposite direction $\Leftrightarrow(\delta)$ $\Leftrightarrow(\Delta)$: Migrate to direction δ / Δ</p> <hr/> <p><u>Agents</u></p> <p>ψ AC: $(x, y, \dots) \rightarrow \{ \dots \}$: Define an agent class AC with parameters x, y, ... φ ac: $\{ \dots \}$: Define a sub-class ac Σ: $\{ \dots \}$: Define body variables, σ: $\{ \dots \}$: none-persistent α A: $\{ \dots \}$: Define activity A F: $(x, y, \dots) \rightarrow \{ \dots \}$: Define a function F with parameters x, y, ... ζ S: $(P) \rightarrow \{ \dots \}$: Define signal handler S with parameter P Π: $\{ \dots \}$: Define transitions, π: $\{ \dots \}$: Define sub-class transitions $\theta^+(v1, v2, \dots)$: Fork agent with arguments $\theta^X(A)$: Destroy agent A $\theta^+AC(v1, v2, \dots)$: Create new agent from agent class AC with arguments</p> <hr/> <p><u>Timer</u></p> <p>$\tau^+(TMO, S)$: Add timer with timeout TMO and signal S $\tau^-(S)$: Remove timer for signal S</p> <hr/> <p><u>Reconfiguration</u></p> <p>$\pi^+(T_1, T_2, C)$: Add transition $T_1 \rightarrow T_2$ with condition C $\pi^*(T_1, T_2, C)$: Update transitions $T_1 \rightarrow T_2$ with condition C $\pi^-(T_1, T_2)$: Remove transitions $T_1 \rightarrow T_2$ $\alpha^+(A)$: Add activity A $\alpha^-(A)$: Remove activity A</p> <hr/> <p><u>Signals</u></p> <p>ζ S(V) \Rightarrow A: Send a signal S with argument V to agent A</p> <hr/> <p><u>Values</u></p> <p>$\\$V$: Agent reference variable (V=self, parent, ...) $\mathfrak{R}\{SET\}$: Random element selection from a set or in the range {min .. max} $x \leftarrow e$: Change value of variable x</p>	<p><u>Symbols</u></p> <p> Timer  Reconfiguration  Replication  Move  Tuple Database  Database  Signal  Kill  Blocking Process  Static Transition  Dynamic Transition</p> <hr/> <p><u>Set Iteration</u></p> <p>$\forall \{ x \in X \mid c(x) \} \text{ do } I$ $\forall \{ x \mid c(x) \} \text{ do } I$</p> <p>Repeat the following statement $I(x)$ (using x) for each element x of the set X for which the condition $c(x)$ is satisfied.</p> <hr/> <p><u>Interval set Iteration</u></p> <p>$\forall x \in \{ a \dots b \} \text{ do } I$ $\forall \{ x \mid x \in \{ a \dots b \} \} \text{ do } I$</p> <p>Repeat the following statement $I(x)$ (using x) for each element x of the interval set $\{ a \dots b \}$</p>
--	---

Figure 20. Short notation of the *AAPL* agent behaviour programming language and some symbols

References

1. F. Pantke, S. Bosse, D. Lehmhus, and M. Lawo, *An Artificial Intelligence Approach Towards Sensorial Materials*, Future Computing Conference, 2011
2. C. R. Farrar and K. Worden, *Structural Health Monitoring: A Machine Learning Perspective*. Wiley-Interscience, 2013.
3. K. WORDEN and G. MANSON, *The application of machine learning to structural health monitoring*, Phil. Trans. R. Soc. A, vol. 365, pp. 515–537, 2007.
4. M. Friswell, *Damage identification using inverse methods*, Phil. Trans. R. Soc. A, 365, 393–410,

2007.

5. 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
6. 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
7. 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
8. Giurgiutiu, V. (2008) *Structural Health Monitoring with Piezoelectric Wafer Active Sensors*, Elsevier, 2008
9. H. W. Engl, M. Hanke, A. Neubauer, *Regularization of inverse problems*, Kluwer Acad. Publ., Dordrecht, Netherlands, 1996
10. Daubechies, I., Defrise, M., De Mol, C. *An iterative thresholding algorithm for linear inverse problems with a sparsity constraint*. *Comm. Pure Appl. Math.* 57 (11) 1413-1457, 2004.
11. 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.
12. 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
13. 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.
14. S. Yuan, X. Lai, X. Zhao, X. Xu, and L. Zhang, *Distributed structural health monitoring system based on smart wireless sensor and multi-agent technology*, *Smart Materials and Structures*, vol. 15, no. 1, pp. 1–8, Feb. 2006.
15. J. Liu, *Autonomous Agents and Multi-Agent Systems*, World Scientific Publishing, 2001 (ISBN 981-02-4282-4)
16. S. Bosse, *Distributed Agent-based Computing in Material-Embedded Sensor Network Systems with the Agent-on-Chip Architecture*, *IEEE Sensors Journal*, Special Issue on Material-integrated Sensing, DOI 10.1109/JSEN.2014.2301938
17. S. Bosse, *Processing of mobile Multi-Agent Systems with a Code-based Agent Platform in Material-integrated Distributed Sensor Networks*, in *Proceedings of the 1st Int. Electron. Conf. Sens. Appl.*, 1 - 16 June 2014, 2014, pp. 1–7, DOI 10.3390/ecsa-1-d010.
18. S. Bosse, A. Lechleiter, *Structural Health and Load Monitoring with Material-embedded Sensor Networks and Self-organizing Multi-agent Systems*, *Procedia Technology*, vol. 15, no. 0, pp. 669–691, 2014, DOI 10.1016/j.protcy.2014.09.039
19. A. Kirsch, *An introduction to the mathematical theory of inverse problems*, Springer, 1996
20. 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
21. 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
22. 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
23. F. Hecht, *New development in FreeFem++*, *J. Numer. Math.*, (2012), no. 3-4, 251-265. 65Y15