Distributed Agent-based Computing in Material-Embedded Sensor Network Systems with the Agent-on-Chip Architecture

Stefan Bosse

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

Abstract

Distributed material-embedded systems like sensor networks integrated in sensorial materials require new data processing and communication architectures. Reliability and robustness of the entire heterogeneous environment in the presence of node, sensor, link, data processing, and communication failures must be offered, especially concerning limited service of material-embedded systems after manufacturing. In this work multi-agent systems with state-based mobile agents are used for computing in unreliable mesh-like networks of nodes, usually consisting of a single microchip, introducing a novel design approach for reliable distributed and parallel data processing on embedded systems with static resources. An advanced high-level synthesis approach is used to map the agent behaviour to multi-agent systems implementable entirely on microchip-level supporting Agent-On-Chip processing architectures (AoC). The agent behaviour, interaction, and mobility are fully integrated on the microchip using a reconfigurable pipelined communicating process architecture implemented with finite-state machines and register-transfer logic. The agent processing architecture is related to Petri Net token processing. A reconfiguration mechanism of the agent processing system achieves some degree of agent adaptation and algorithmic selection. The agent behaviour, interaction, and mobility features are modelled and specified with an activity-based agent behaviour programming language (AAPL). Agent interaction and communication is provided by a simple tuple-space database implemented on node level and signals providing remote inter-node level communication and interaction.

I. INTRODUCTION AND OVERVIEW

Embedded systems required for sensorial perception and structural monitoring (perceptive networks), used, for example in Cyber-Physical-Systems (CPS) and Structural Health Monitoring (SHM) [6], perform the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource-constrained manner and with real-time processing constraints. Trends emerging recently in engineering and micro-system applications such as the development of sensorial materials [16] show a growing demand for autonomous networks of miniaturized smart sensors and actuators embedded in technical structures [6] (see Fig. 1). To reduce the impact of such embedded sensorial systems on mechanical structure properties, single microchip sensor nodes (in mm² range) are preferred. Real-time constraints require parallel data processing usually not provided by microcontrollers. Hence with increasing miniaturization and node density, new decentralized network and data processing architectures are required. Multi-agent systems (MAS) can be used for a decentralized and self-organizing approach of data processing in a distributed system like a sensor network [2], enabling the mapping of distributed raw sensor data to condensed information, for example based on pattern recognition [5]. In [2], the agent-based architecture considers sensors as devices used by an upper layer of controller agents. Agents are organized according to roles related to the different aspects to integrate, mainly sensor management, communication and data processing. This organization isolates largely and decouples the data management from the changing network, while encouraging reuse of solutions. Multi-agent system-based structural health monitoring technologies are used to deal with high-density and different kinds of sensors in reliable monitoring of large scale engineering structures [5]. In [19] and [20], agents are deployed for distributed sensing and power management in wireless sensor networks, but still using embedded system nodes not suitable for material integration.

Material-embedded data processing systems usually consist of single microchip nodes connected either wired in mesh-like networks [6] or wireless using ad-hoc networks [8] with limited energy supply and processing resources. But traditionally, mobile agents are processed on generic program-controlled computer architectures using virtual machines [7][8][19][20], which usually cannot easily be reduced to single microchip level like they are required in sensorial materials. Furthermore, agents are treated with abstract heavy-weighted knowledge-based models, not entirely matching distributed data processing in sensor networks. In [3], a multi-agent system is used for advanced image processing making profit from the inherent parallel execution model of agents.

Application specific digital logic hardware design has advantages compared to program controlled microcontroller approaches concerning power consumption, performance, and chip resources by exploiting parallel data processing (covered by the agent model) with lower clock frequencies and enhanced performance [10]. There are actually four major issues related to the scaling of traditional software-based multi-agents systems to microchip level and their design:

- limited static processing, storage, and communication re-
sources, real-time processing,
• unreliable communication,
• suitable simplified programming models and processing architectures offering hardware designs with finite state machines (FSM) and resource sharing for parallel agent execution, and
• a generic high-level synthesis design approach.

Microchip level implementations of multi-agent systems were originally proposed for low level tasks, for example in [12] using agents to negotiate network resources and for high-level tasks using agents to support human beings in ambient-intelligence environments [15]. The first work implements the agent behaviour directly in hardware, the second uses still a (configurable) microcontroller approach with optimized parallel computational blocks providing instruction set extension. A more general and reconfigurable implementation of agents on microchip level is reported in [1], providing a closed-loop design flow especially focussing on communication and interaction, though still assuming and applying to program controlled data processing machines and architectures. Hardware implementations of multi-agent systems are still limited to single or a few and non-mobile agents ([1][21])

In this work, an advanced high-level synthesis approach is introduced to map the agent behaviour of multi-agent systems on microchip-level with an Agent-On-Chip processing architecture (AoC). The agent behaviour, interaction, and mobility are fully integrated on the microchip using a reconfigurable pipelined communicating process architecture implemented with finite-state machines and register-transfer logic. This architecture supports parallel agent execution with a resource shared pipeline approach. In this approach, the agent processing is comparable to Petri Net token processing. A reconfiguration mechanism of the agent processing system achieves some kind of agent adaptation and algorithmic section based on environmental changes like partial hardware or inter-connect failures or based on learning and improved knowledge base.

The agent behaviour, interaction, and mobility features are modelled and specified with an activity-based agent behaviour programming language (AAPL). The activity-graph based agent model is attractive due to the proximity to the finite-state machine model, which simplifies the hardware implementation. With this AAPL a high-level agent compiler is able to synthesize a hardware model on Register-Transfer Level (RTL, VHDL), alternatively a software model (C, ML), or a simulation model (XML) suitable to simulate a multi-agent system using the SeSAm simulator framework [9].

Agent interaction and communication are provided firstly by a simple tuple-space database implemented on each node providing access and sharing of local data, and secondly by signals able to propagate in the network (like messages) preferred for fast and light-weighted remote inter-node level communication and interaction. To enable dynamic adaptation of the agent behaviour at run-time, the agent processing architecture implementing the agent behaviour can be (re)configured by agents by modifying the transitional network.

Traditionally agent programs are interpreted, leading to a significant decrease in performance. In the approach presented here the agent processing platform can directly be implemented in standalone hardware nodes without intermediate processing levels and without the necessity of an operating system, but still enabling software implementations that can be embedded in applications.

There is related work concerning agent programming languages and processing architectures, like APRIL [14] providing tuple-space like agent communication, and widely used FIPA, ACL, and KQGML [11] focusing on high-level knowledge representation and exchange. All those approaches represent communication and information on complex and abstract level not fully suited for the synthesis of low-resource data processing systems in distributed loosely coupled networks, especially in sensor networks, in contrast to the proposed AAPL approach, simple enough to enable hardware design synthesis, but powerful enough to model the agent behaviour of complex distributed systems, which is demonstrated in the following case study.

Fig. 1. Sensorial Materials embedded in robots providing perception information of external applied load forces or internal structure load.
Though the imperative programming model is quite simple and closer to a traditional PL it can be used as a common source and intermediate representation for different agent processing platform implementations: hardware (HW), software (SW), and simulation (SIM).

What is novel compared to other approaches?

- **Reliability and reactivity** provided by the autonomy of mobile state-based agents and reconfiguration.
- Agent mobility and **interaction** by using tuple-space databases and global signal propagation aid solving data distribution and synchronization issues in distributed systems design, and tuple spaces represent agent belief.
- One common agent **programming language** and **processing architecture** enables the synthesis of standalone parallel hardware implementations, alternatively standalone software implementations, and behavioural simulation models, enabling the design and test of large-scale heterogeneous systems.
- **AAPL** provides powerful statements for computation and agent control with static resources.
- A token-based pipelined multi-process agent processing architecture suitable for hardware platforms with Register-Transfer Level Logic offering optimized computational resources and speed.
- Improved scaling in large network applications compared with full or semi centralized and pure message based processing architectures.

II. **FIELDS OF APPLICATION: SENSORIAL MATERIALS**

Sensorial Materials are equipped with material-embedded high miniaturized distributed sensor networks performing load monitoring or environmental perception [16], shown in principle in Fig. 1. These embedded sensor networks consist of nodes equipped with sensor signal electronics and digital logic performing computation and communication. Option-ally there are material-embedded energy sources (energy harvester) supplying the nodes locally.

Fig. 2 shows a prototype of a Sensorial Material using intelligent sensor networks. Each autonomous network node is connected with up to four neighbours and strain-gauge sensors mounted on a rubber sheet, altogether equipped with nine bi-axial strain-gauge sensors placed at a distance of 70 mm. The Sensorial Material was used to retrieve load information about the sheet (applied by external forces) by using advanced machine learning methods from a small set of un-calibrated sensors with unknown electro-mechanical model still providing high spatial resolution compared with the distance of the sensors to each other.

Fig. 3 shows the usage of such material in an intersection element of a robot arm manipulator providing external environmental perception required for robot control. The proposed robot manipulator [17] consists of actors (joint drives) and intersection elements with integrated smart sensor networks. Distributed data processing is provided by mobile agents. The agent behaviour is implemented with SoC designs on hardware-level. The intersection element connects two joint actors with a rigid double-pipe construction, which is surrounded by two opposite placed load sensitive skins (bent rubber plate), equipped each with four strain-gauge sensor pairs (bi-axially aligned). Each sensor pair is connected to a sensor node providing parallel data processing, agent behaviour implementation, and communication/networking. All sensor nodes are arranged in a mesh-like network connected with serial point-to-point links. Communication is established by a smart and robust routing protocol.

III. **STATE-BASED AGENTS AND THE AGENT PROGRAMMING LANGUAGE AAPL**

Initially, a sensor network is a collection of independent computing nodes. Interaction between nodes is required to manage and distribute data and to compute information. One common interaction model is the state-based mobile agent.
Fig. 3. Robot arm manipulator intersection element equipped with smart sensor networks providing perception information of external applied load forces based on preliminary work with flat rubber plate.

The behaviour of a state-based agent can be easily modelled with a finite-state machine completely implementable with register-transfer logic (RTL) on microchip level easing high-level synthesis and the exploitation of concurrency required for material-embedded real-time data processing. The implementation of mobile multi-agent systems for resource constrained embedded systems with a focus on microchip level is a complex design challenge. High-level agent programming languages can aid to solve this design issue. Though there are already several agent modelling, interaction, and communication languages, they are not fully suitable to carry out multi-agent systems on microchip level. For this purpose, the Agent Programming Language AAPL was designed to enable the optimized design of state-based agents and microchip scaled processing dealing with limited static resources. This language consists of generic imperative and computational statements and a type system derived from a subset of the Modula-3 language, allowing subrange types required for hardware synthesis, and agent specific statements to specifying the behaviour, mobility, and interaction of agents. This technology-independent programming model can be directly synthesized to hardware, alternatively to software and simulation model targets without modification.

The agent behaviour is partitioned and modelled with an activity graph, with activities (representing the control state of the agent) and conditional transitions enabling activities. Activities provide sequential execution of procedural data processing statements. An activity is activated by a conditional transition depending on the evaluation of agent data (conditional transition), or using unconditional transitions. An agent belongs to a specific parameterized agent class AC, specifying local agent data (only visible for the agent itself), types, signals, activities, signal handlers, and transitions, shown in principle in Fig. 4.

New agents of a specific class can be created at runtime by agents using the new AC(v1,v2,..) statement returning a node unique agent identifier. An agent can create multiple living copies of itself with a fork mechanism, creating child agents of the same class with inherited data and control state but with different parameter initialization, done by using the fork(v1,v2,..) statement. Agents can be destroyed by using the kill(ID) statement.

Statements inside an activity are processed sequentially and consist of data assignments (x := ε) operating on agent’s private data, control flow statements (conditional branches and loops), and special agent control and interaction statements, summarized in Def. 1.

Agent interaction and synchronization is provided by a tuple-space database server available on each node. An agent can store an n-dimensional data tuple (v1,v2,..) in the database by using the out(v1,v2,..) statement (commonly the first value is treated like a key). A data tuple can be removed or read from the database by using the in(v1,p2?,v3,..) or rd(v1,p2?,v3,..) statements with a pattern template based on a set of formal (variable,?) and actual (constant) parameters. These operations block the agent processing until a matching tuple was found/stored in the database. These simple operations solve the mutual exclusion problem in concurrent systems easily. Only agents processed on the same network node can exchange data this way.

The existence of a tuple can be checked by using the exist? function or with atomic test-and-read behaviour using the try_in/rd functions. A tuple with a limited lifetime (a marking) can be stored in the database by using the mark statement. Tuples with exhausted lifetime are removed automatically by a garbage collector. Tuples matching a specific pattern can be removed with the rm statement.

Remote light-weighted interaction between agents is provided by signals with optional parameters, implementing a remote-procedure call interface.
A signal can be raised by an agent using the send(ID,S,V) statement specifying the ID of the target agent (which must be created by the sending agent), the signal name S, and an optional argument value V propagated with the signal. The receiving agent must provide a signal handler (like an activity) to handle signals (asynchronously). Alternatively, a signal can be sent to a group of agents belonging to the same class AC within a bounded region using the broadcast(AC,DX,DY,S,V) statement.

Migration of agents to a neighbour node (by preserving the local data and processing state) is performed by an agent using the moveto(DIR) statement, assuming the arrangement of network nodes in a mesh- or cube-like network. To test if a neighbour node is reachable (testing connection liveliness), the link?(DIR) statement returning a boolean result can be used. Within activities agents can change the transitional network (initially specified in the transition section) by changing, deleting, or adding (conditional) transitions using the transition+() and transition()-() statement. Alternatively, the reconfiguration step can be supported by changing the transitional network in an activity, assuming the arrangement of network nodes in a mesh- or cube-like network. To test if a neighbour node is reachable (testing connection liveliness), the link?(DIR) statement returning a boolean result can be used.

The usage of the programming language is illustrated in more detail in the following case study.

**Def. 1. Summary of the AAPL Language** (.. x .. means x is part of an expression ε, and ε; terminates procedural statements)

**Agent Class Definition**

agent class (arguments) = definitions end;

**Activity Definition**

activity name = statements end;

**Data Statements**

var x,y,z: type;

x := ε(variable, value, constant);

**Conditional Statements**

if cond then statements else statements end;

case ε of { ε1 | ε2 | ... } statements | .. end;

**Loop Statements**

for i := range do statements end;

while cond do statements end;

**Transition Network Definition**

transitions = transitions end;

a1 -> a2: cond;

**Tuple Database Statements**
pending on the specified transition conditions. This multi-
process model is directly mappably to RTL hardware and
software implementations. Each process $P_i$ is mapped to a fi-
nite state machine $FSM_i$ controlling process execution and a
register-transfer data path. Local agent data is stored in a re-
gion of a memory module assigned to each individual agent.
There is only one incoming transition queue for each process
consuming tokens, performing processing, and finally passing
tokens to outgoing queues, which can depend on condi-
tional expressions. There are computational and IO/event
based activity statements. The latter ones can block the
agent processing until an event occurs (for example, the
availability of a data tuple in the database). Blocking state-
ments $S_{i,j}$ of an activity $A_i$ are assigned to separate inter-
mediate IO processes $P_{i,j}$ handling IO events or additional post computations, as shown on the bottom of
Fig. 5.

Agents in different activity states can be processed concur-
rently. Thus, activity processes that are shared by several
agents may not block. To prevent blocking of IO-event
based processes (for example waiting for data), not-ready processes pass the agent token back to the input queue. An
IO process either processes unprocessed agent tokens or
waits for the happening of events, controlled by the agent
manager.

The pipeline architecture offers advanced resource sharing and concurrent processing of agents in different activity
states. Only one activity process chain implementation for
each agent class is required on each node, in contrast to pre-
vious programmable architectures [4] providing only limited concurrency and resource sharing.

Resources
A rough estimation of the resource requirements $R$ for the
hardware implementation of the agent processing architec-
ture supporting a set of $N$ different agent classes $\{AC_i\}$ is
shown in Eq. 1, with each class having $M_j$ activities, $T_j$ transi-
tions, $D_j$ data cells with a resource weight $w_{data}$, and $w_{act,i,j}$
each for activity $i$, and a maximal number of managed agents
for each class $N_{agents,i}$. The tuple space database requires
$w_{tsp,k} \times S_j$ resources for each supported n-dimensional space. The $C_j$ values are control parts independent of the above values.

$$R = \left( w_{data} \sum_{i \in AC} N_{agents,i} D_i \right) +$$

$$+ C_{sched} + w_{sched} \left( \sum_{i \in AC} M_i \right) + w_{sched} \max(N_{agents,i}) +$$

$$+ C_{comm} + (w_{queue} + w_{cond}) \left( \sum_{i \in AC} T_i \right) + \left( \sum_{i \in AC} \sum_{j \in AJ} w_{act,i,j} \right) +$$

$$+ C_{ts} + \left( \sum_{i \in DS} w_{ts} S_j \right)$$

(Eq. 1)

For example, assuming simplified four agent classes with $N$
=16 agents for each class, each class requires $D=512$ bit
memory, $M=10$ ($w_{act}=500$), $T=16$, three tuple spaces (1,2,3)
with $S=32$ (and $w_{j}=32$, $w_{2}=64$, $w_{3}=128$) entries each, and

$w_{data}=4$, $w_{queue}=150$, $w_{act}=60$, $w_{cond}=50$, $w_{act}=500$,
$C_{sched}=5000$, $C_{comm}=10000$, $C_{ts}=1000$ (based on experi-
mental experiences, all $w$ and $C$ values in eq. gates units),
which results in 189400 eq. gates for the HW implementa-
tion.

Power/Efficiency
Agents are often heavy-weighted processing entities inter-
preted by software-based virtual machines. In contrast, in
the proposed RTL architecture the agent behaviour is mapped
on finite state machines and a data path with data
word length scaling, offering minimized power- and re-
source requirements, both in the control and data path. Most
activity statements are executed by the platform in one or
two clock cycles! All commonly administrative parts like
the agent manager, communication protocols, and the tuple-
space database commonly part of an operating system are
implemented in hardware, offering advanced computational
power enabling low-frequency and low-power designs, well
suited for energy-autonomous systems. Transition network
changes can be performed within a few clock cycles.

Agent Manager
The agent manager provides a node level interface for
agents, and it is responsible for the creation, control (includ-
ing signals, events, and transition network configuration),
migration of agents with network connectivity, implement-
ing a main part of an operating system. The agent man-
ger controls the tuple-space database server and signal
events required for IO/event based activity processes.

The agent manager uses agent tables and caches to store in-
formation about created, migrated, and passed through
agents (req., for ex., for signal propagation), see Fig. 6.

Migration
Migration of agents between nodes incorporates only the
transfer of the agent state consisting of data (the content of
body variables) and the control state (a pointer to the next
activity to be executed after migration and the transition
configuration) of the agent together with a unique global
agent identifier (extending the local ID with the agent class
and the relative displacement of its root node) encapsulated
in messages with low overhead, shown in Fig. 6. This ap-
proach minimizes network load and energy consumption
significantly. Migration of simple agents results in a mes-
sage size between 100-1000 bits. The agent start-up time af-
after the data transfer is low (about some hundred clock
cycles).

Transition Network
A switched transition network allows the reconfiguration of
the activity transitions at runtime. Though the possible re-
configuration and the conditional expressions must be
known at compile time (static resource constraint!), a recon-
figuration can release the use of some activity processes and
enhances the utilization for concurrent processing of other
agents of the same class. The transition network is imple-
mented with tables in case of the HW implementation, and
with dynamic lists in case of the SW and SIM implementa-
tions. Agent activity transition configurations can be inher-
ited by child agents.
Fig. 5. Mapping of the agent behaviour programming level to the agent processing architecture with pipelined communicating sequential processes and the final mapping on RT level. Agent tokens are passed by queues and conditional selectors from an outgoing to an incoming activity process.

Fig. 6. Agent migration and signal propagation using message transfers (LAH: Local Agent Handler, LID: Local Agent Identifier, AC: Agent Class, LIVE: agent life, STATE: agent state, DX and DY: spatial displacement vector, SID: Signal Identifier, SIG: pending signal ID)

Tuple-Space Database
Each n-dimensional tuple-space $TS^n$ (storing n-ary tuples) is implemented with fixed size tables in case of the hardware implementation, and with dynamic lists in the case of the software and simulation model implementations. The access of each tuple-space is handled independently. Concurrent access of agents is mutually exclusive. The HW implementation implicates further type constraints, which must be known at design time (e.g. limitation of integer ranges) provided by sub-range-typing in the AAPL specification.

Signals
Signals must be processed asynchronously. Therefore, agent signal handlers are implemented with a separate activity process pipeline, one for each signal handler. For each pending agent signal, the agent manager injects an agent token in the respective handler process pipeline independent of the processing state of the agent. Remote signals are processed by the agent manager, which encapsulate signals in messages sent to the appropriate target node and agent, shown in Fig. 6.

Synthesis
The database driven synthesis flow is illustrated in Fig 7. The AAPL program is parsed and mapped to an abstract syntax tree (AST). The first compiler stage analyzes, checks, and optimizes the agent specification AST. The second stage is divided into three parts: an activity to process mapper, a transition to queue mapper, a transition (pipelined processing architecture) network builder, and a message generator supporting agent and signal migration. There are different supported backends (HW/SW/SIM). The high-level hardware description enables the SoC synthesis using the ConPro high-level synthesis framework [10], which maps activity processes on finite state machines and the RT datapath level.
The ConPro programming model reflects an extended CSP, which provides atomic guarded actions on shared resources access. Each process is implemented with a FSM and a RT datapath. Furthermore, a software description (C), which can be embedded in application programs, and a simulation model usable for MAS simulation using the SeSAM simulator [9] can be derived. All implementation models (HW/SW/SIM) provide equal functional behaviour, and only differ in their timing, resource requirements, and execution environments.

Simulation

In addition to real hardware-implemented agent processing platforms there is the capability of the simulation of the agent behaviour, mobility, and interaction on a functional level using the SeSAM simulation framework [9], which offers a platform for the modelling, simulation, and visualization of mobile multi-agent systems employed in a two-dimensional world. The behaviour of agents is modelled with activity graphs (specifying the agent reasoning machine) close to the APL model. But some special transformations must be applied to enable the simulation: 1. APL activities (IO/event-based) can block the agent processing until an event occurs. Blocking agent behaviour is not provided directly by SeSAM. ⇒ activity decomposition 2. The transition network can change during run-time ⇒ use of a transition scheduler 3. The handling of concurrent asynchronous signals used in APL for inter-agent communication cannot be established with the generic activity processing in SeSAM ⇒ use of a signal scheduler.

V. CASE STUDY: STRUCTURAL HEALTH MONITORING

A small example implementing a distributed feature detection in an incomplete and unreliable mesh-like sensor network using mobile agents should demonstrate the suitability of the proposed agent processing approach. The sensor network consists of nodes with each node attached to a sensor (e.g. Strain-gauge). The nodes can be embedded in a mechanical structure, for example, used in a robot arm. The goal of the MAS is to find extended correlated regions of increased sensor intensity (compared to the neighbourhood) due to mechanical distortion resulting from externally applied load forces. A distributed directed diffusion behaviour and self-organization (see Fig. 8) is used, derived from the image feature extraction approach proposed in [18]. Single sporadic sensor activities not correlated with the surrounding neighbourhood should be distinguished from an extended correlated region, which is the feature to be detected.

There are three different agent classes: an exploration, a node agent, and a deliver agent. A node agent is immobile and is primarily responsible for sensor measurement and observation. The feature detection is performed by the mobile exploration agent that supports two main different behaviours: diffusion and reproduction. The diffusion behaviour is used to move within 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 that 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 $H_{ij}$ for an exploration agent to stay at a specific node with the coordinates $(i,j)$ is given by eq. 2.

$$H(i, j) = \sum_{s} \sum_{t} \left| S(i+s, j+t) - S(i, j) \right| \leq \delta$$

where $S$ : Sensor Signal Strength

$R$ : Square Region around $(i,j)$

The calculation of $H$ at the current location $(i,j)$ of the agent requires the sensor values within the square area (the region of interest ROI) $R$ around this location. If a sensor value $S(i+s, j+t)$ with $i,j \in [-R...R]$ is similar to the value $S$ at the current position (diff. is smaller than the parameter $\delta$), $H$ is incremented by one. If the $H$ value is within a parameterized interval $\Delta=[ε_0, ε_1]$, the exploration agent has detected the feature and will stay at the current node to reproduce new exploration agents send to the neighbourhood. If $H$ is outside this interval, the agent will migrate to a neighbour different node and restarts exploration (diffusion).

The calculation of $H$ is performed by a distributed calculation of partial sum terms by sending out child explorer agents to the neighbourhood, which itself can send out more agents until the boundary of the region $R$ is reached. Each child agent returns to its origin node and hand over the partial sum term to his parent agent, shown in Fig. 8. Because a
node in the region $R$ can be visited by more than one child agent, the first agent reaching a node sets a marking $MARK$. If another agent finds this marking, it will immediately return to the parent. This multi-path visiting has the advantage of an increased probability of reaching nodes with missing (non operating) communication links (see Fig. 8). A deliver agent, created by the node agent, finally delivers exploration results to interested nodes by using directed diffusion approaches, not discussed here.

Ex. 1 shows the $AAPL$ behaviour specification for the exploration agent. The agent behaviour is partitioned in nine activities and two signal handlers. If a sensor node agent observes an increased sensor value, it creates a new explorer agent that enters the start activity (lines 8-19). Each explorer agent is initialized on creation with two parameter arguments: a direction and a radius value. The first agent created by the sensor node has no specific direction. Child agents with a specific direction move to the respective node (line 11). In line 18, the transition $move \rightarrow percept_{neighbour}$ is created (all existing transitions starting from activity $move$ are deleted first). The start activity transitions to the percept activity, which creates child agents (lines 44-46). Forked agents inherit all parent data and the current transition network configuration. For this, in line 56 the transition $percept \rightarrow move$ is established (and inherited), but after forking reseted in lines 61-62 for the parent agent behaviour, which await the return of all child agents and a decision for behaviour selection (reproduce/diffuse).

The child agents enter the $move$ (lines 20-25) activity after forking and will migrate in the specific direction to the neighbour node. Finally, the $percept_{neighbour}$ activity is reached, which performs the local calculation (line 55) if there was no marking found, and finally stores the partial result in the tuple database. Further child agents are sent out if the boundary of the $ROI$ is still not reached.

Otherwise the agent goes back to his origin (parent) by entering the $goback$ activity performing the migration (lines 66-68), previously updating its $h$ value from the tuple database. If the returning agent has arrived, it will deliver its $h$ value by adding it to the local $H$ value stored in the database (lines 71-72) and raising the $WAKEUP$ signal to notify the parent, which causes the entering of the parent’s signal handler (lines 77-79).

If there is enough input and all child agents had returned (or a time-out has occurred handled by the signal handler $TIMEOUT$, lines 80-81), the exploration agent either enters the $diffuse$ or $reproduce$ activity. Diffusion and reproduction is limited by a lifetime (decreased each time an explorer agent is replicated or on migration, lines 27 & 36).

Synthesis and Simulation
The agent behaviour specification was synthesized to a digital logic hardware implementation (single $SoC$) and a simulation model with equal functional behaviour suitable for the $MAS$ simulator environment $SeSaM$ [9]. The suitability of the self-organizing approach for feature detection was justified by simulation results shown in Fig. 9 for two different sensor network situations, each consisting of a 10 by 10 network with autonomous sensor nodes. Each node is connected with up to four neighbours. One situation creates significant sensor values arranged in a bounded cluster region, for example, caused my mechanical forces applied to the structure, and the other situation creates significant sensor values scattered around the network without any correlation, for example, caused by noisy or damaged sensors.

In the first clustered situation, the explorer agents are capable to detect the bounded region feature for the two separated regions (indicated by the change of the agent colour to black). Due to the reproduction-behaviour there are several agents at one location, shown in the right agent density contour plot. In the second unclustered situation, the explorer agents did not find the feature and vanish due to their limited lifetime behaviour.

The feature-search is controlled by a set of parameters: $\{\delta, \varepsilon_0, \varepsilon_1, \text{lifetime}, \text{search radius} \, R\}$.

**Fig. 8.** Distributed feature extraction in an unreliable and incomplete mesh network (with missing links) by using distributed agents with directed diffusion migration and self-organization behaviour.
The synthesis results of the hardware implementation for one sensor node are shown in Tab. 1, which are in accordance with the resource estimation from Sec. IV. The AAPL specification was compiled to the ConPro programming model and synthesized to an RTL implementation creating VHDL models. Two different target technologies were synthesized by using gate-level synthesis: 1. FPGA, Xilinx XC3S1000 device target using Xilinx ISE 9.2 software, 2. ASIC standard cell LIS10K library using the Synopsys Design Compiler software. The agent processing architecture consisted of the activity process chain for the explorer and node agent, the agent manager, the tuple-space database (supporting two- and three-dimensional tuples with integer type values), and the communication unit.

This case study showed firstly the suitability of the multi-agent-based approach for feature detection in large scale sensor networks, for example used in real-time structural health monitoring for sensor data filtering, and secondly the suitability of the proposed agent modelling and synthesis approach for single System-on-Chip microchip-level implementations.

**Fig. 9. Simulation results for two different sensor network situations (left: start, middle: exploration, right: final result situation). Top row: sensor activity within clusters, bottom row: sensor activity scattered over the network.**

**Table 1. High-level and gate-level synthesis results for one sensor node**

<table>
<thead>
<tr>
<th>Description</th>
<th>FPGA/XC3S1000 Synthesis</th>
<th>ASIC LSI10K Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAPL Source: 200 lines</td>
<td>LUTs (4-input): 10826 (70 %)</td>
<td>Eq. NAND Gates: 309502</td>
</tr>
<tr>
<td>CP Source: 1615 lines</td>
<td>FLIP-FLOPs: 2415 (15 %)</td>
<td>Comb. Gates: 95354</td>
</tr>
<tr>
<td>VHDL Source: 37171 lines</td>
<td>BLOCK RAMs: 19 (80 %)</td>
<td>Non-comb. Gates: 214148</td>
</tr>
<tr>
<td>CP Processes: 28</td>
<td>Max. Clock: 85 MHz</td>
<td>Chip area (180 nm): 7 mm²</td>
</tr>
<tr>
<td>CP Queues: 14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Ex. 1. Excerpt of the AAPL specification for agent class Explore implementing a feature extraction agent with distributed directed diffusion and self-organizing behaviour:**

```apl
1 type keys = {ADC,FEATURE,H,MARK}; direction = {.} 
2 signal WAKEUP,TIMEOUT; val RADIUS ::= 4; ... 
3 agent Explore(dir: direction, 
4 radius: integer[1..16]) = 
5 var dx,dy:integer[-100..100]; 
6 live:integer[0..15]; ......
7 var* s: integer[0..1023]; ......
8 activity start = 
9 dx := 0; dy := 0; h := 0; 
10 if dir <> ORIGIN then 
11 moveto(dir); 
12 case dir of 
13 | NORTH -> backdir := SOUTH 
14 | SOUTH -> ...... 
15 else 
16 live := MAXLIVE; backdir := ORIGIN 
17 group := random(integer[0..1023]); 
18 transition*(move.percept_neighbour); 
```
VI. CONCLUSION

A novel design approach using mobile agents for reliable distributed and parallel data processing in low-resource networks with embedded hardware nodes was introduced. A multi-agent programming language AAPL provides computational statements and statements for agent creation, inheritance, mobility, interaction, reconfiguration, and information exchange, based on the agent behaviour partitioning in an activity graph, which can be directly synthesized to the microchip level by using a high-level synthesis approach and finite state machines on RT level.

Agent interaction is delivered by a simple but powerful tuple database approach. The tuple-space is a central part of the agent’s belief, and contributes to the decision making process of agents. Agents can be created dynamically at runtime by other agents.

This proposedagent processing architecture implements a resource and speed optimized virtual machine consisting of a reconfigurable pipelined communicating process chain. Only one virtual machine is required for each agent class, which should be supported on a particular network node. The pipeline approach enables concurrent agent processing and advanced resource sharing. Replication of activity processes can increase the computational performance significantly, for example, based on timed Petri-Net analysis.

Unique identification of agents does not require unique absolute node identifiers or network addresses, a prerequisite for loosely coupled and dynamic networks (due to failures, reconfiguration, or expansion).

Reconfiguration of the activity transition network supported on programming level offers agent behaviour adaptation at runtime based on the data state of the agent resulting from environmental changes like partial hardware or interconnect failures or based on learning and improved knowledge base. The transitional configuration can be inherited by child agents. Finally, improved resource sharing for parallel processing is offered.

A case study implementing a self-organizing multi-agent system in a sensor network demonstrated the suitability of the proposed programming model, processing architecture, and synthesis approach. Migration of agents requires only the transfer of the control and data space of an agent using messages. The agent behaviour is fixed and bound to each node. The high-level synthesis tool enables the synthesis of different output models from a common programming source, including hardware, software, and simulation models delivering an advanced design methodology for functional testing.

VII. REFERENCES


