Distributed Computing in Sensor Networks Using Multi-Agent Systems and Code Morphing

Stefan Bosse[1,3], Florian Pantke[2,3], Frank Kirchner[1,3]

University of Bremen, Department of Computer Science, Workgroup Robotics, Germany[1], TZI Centre for Computing and Communication Technologies,[2], University of Bremen, ISIS Sensorial Materials Scientific Centre, Germany[3]

Abstract

There is a growing demand for distributed computing and systems in sensor networks. We propose and show a parallel and distributed runtime environment for multi-agent systems that provides spatial agent migration ability by employing code morphing. The focus of the application scenario lies on sensor networks and low-power, resource-aware single System-On-Chip designs, used in sensor-equipped technical structures and materials. An agent approach provides stronger autonomy than a traditional object or remote-procedure-call based approach. Agents can decide for themselves which actions are performed, and they are capable of reacting on the environment and other agents with flexible behaviour. Data processing nodes exchange code rather than data to transfer information. A part of the state of an agent is preserved within its own program code, which also implements the agent's migration functionality. The practicability of the approach is shown using a simple distributed Sobel filter as an example.

1. Introduction and Overview

Recently emerging trends in engineering and microsystem applications such as the development of sensorial materials show a growing demand for autonomous networks of miniaturized smart sensors and actuators embedded in technical structures **[8]**. With increasing miniaturization and sensor-actuator density, decentralized network and data processing architectures are preferred or required.

We propose and show a spatial distributed and execution-parallel runtime environment for multi-agent systems providing migration mobility using a

code morphing approach (Distributed-Parallel Code-Morphing Runtime Environment: abbrev. DPCM-RE) in which computing nodes exchange code rather than data to transfer information, basically similar to work discussed in **[2]**. The advantage of this distributed computation model is the computational independence of each node and the eliminated necessity for nodes to comply with previously defined common data types and structures as well as message formats. Computing nodes perform local computations by executing code and cooperate by distributing modified code to execute a global task. Multi-agent systems providing migration mobility using code morphing can help to reduce the communication cost in a distributed system **[5]**. The distributed programming model of mobile agents has the advantage of simplification and reduction of synchronization constraints owing to the autonomy of agents.

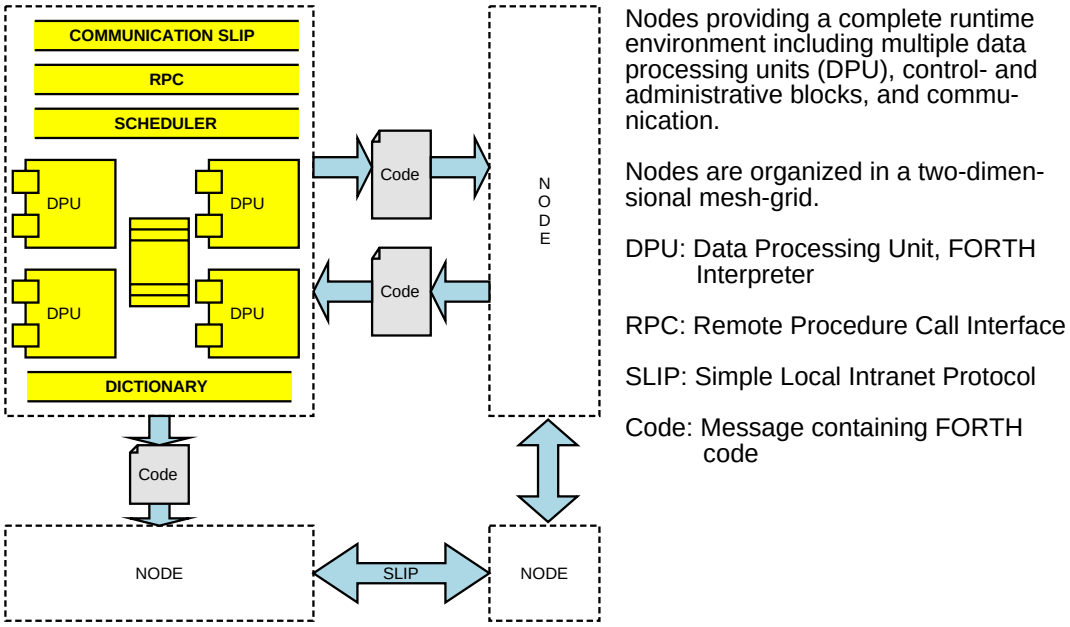
Traditionally, mobile agents are executed on generic computer architectures **[6][7]**, which usually cannot easily be reduced to single-chip systems as they are required, for example, in sensorial materials with high sensor node densities.

We present a hardware architecture for mesh networks of data processing nodes, which can be organized, for example, in a two-dimensional grid topology with each node having connections to its up to four direct neighbours. An example of such a network with dimensions 2 x 2 is illustrated in figure **1**. A fault-tolerant message-based communication system SLIP is used to transfer messages (containing code) between nodes using smart delta-distance-vector routing **[4]**. The network topology can be irregular, for example, depending on design or owing to temporary failures or permanent physical defects of the existing communication links. Smart routing is used to deliver messages on alternative routes around partially connected or defective areas.

A central issue was the design of the data processing architecture used for the execution of code, and especially regarding support of code morphing. A multi-parallel stack-based multi-stack, zero-operand FORTH machine was chosen leading to small programs, low system complexity, and high system performance **[9]**. The architecture design focuses on low-power and resource-aware single System-On-Chip (SoC) design on RT level, though both hardware and software implementations were created. Though the FORTH programming language is high level, it can be directly mapped

to and executed on the machine level.

Fig. 1. Distributed data processing framework with four nodes



A complete runtime unit consists of a communication system with a smart routing protocol stack, one or more FORTH processing units with a code morphing engine, resource management, code relocation and dictionary management, and a scheduler managing program execution and distribution, which are normally part of an operating system which does not exist here.

2. Implementing Migrating Agents with FORTH Using Code Morphing

FORTH is an interpreted language whose source code is extremely compact. Furthermore, FORTH is extensible, that is new language constructs can be defined on the fly by its users **[3]**.

A FORTH program can be sent to and executed on any node in the network. A FORTH program contains built-in core instructions directly executed by the FORTH processing unit and user-defined high-level word and object definitions that are added to and looked up from a dictionary data structure. This dictionary play a central role in the implementation of distributed systems and mobile agents. Words can be added, updated, and removed (forgotten), controlled by the FORTH program itself (already

considered in **[3]**). User-defined words are composed of a sequence of words.

FORTH maintains two push-down stacks, providing communication between FORTH words. Most instructions interact directly with the data stack SS, the second stack is known as the return stack RS and is used to hold return addresses enabling nesting. Literal values are treated as special words pushing the respective value on the data stack. Due to the LIFO nature of the stacks, FORTH instructions use a postfix notation (reverse polish notation, RPN). FORTH provides common arithmetic data manipulation instructions and high-level control constructs like loops and branches.

For mobile agents, not only code may migrate from node to node but also state information of the agent, at least a subset of the process state has to be transferred with the agent. On the one hand, the process state of a stack-based FORTH program and execution environment consists of the data values stored on the data stack (and in an additional random-access data segment), on the other hand, of the control state defined by the program counter and the values on the return stack. A program capable of modifying its own code can store a subset of its process state by modifying code, applied to both data and control instructions.

A program can fork a modified (or unmodified) replica of itself for execution on a different processing unit (locally parallel or globally distributed). This feature enables migration of dynamic agents holding locally processed information and a subset of execution state in their code.

The simple FORTH instruction format is an appropriate starting point for code morphing, i.e., the ability of a program to modify itself or make a modified copy, mostly as a result of a previously performed computation. Calculation results and a subset of the processing state can be stored directly in the program code which changes the program behaviour. The standard FORTH core instruction set was extended and adapted for the implementation of agent migration in mesh networks with two-dimensional grid topology.

Table **1** gives a summary of the new words provided for code morphing. These instructions can be used to modify the program behaviour and enable the preservation of the current program execution state. In our system, a FORTH program is contained in a contiguous memory fragment, called a frame. A frame can be transferred to and executed on remote nodes and

processing units. Modification of the program code is always performed in a shadow frame environment, which can be identical with the execution frame. This is the default case used for in-place code modification. One or more different frames can be allocated and used for out-of-place modification, required if the execution frame is used beyond code morphing.
All code morphing instructions operate on the shadow frame. Both the execution and the shadow frames have their own code pointer.

Tab. 1. FORTH extensions providing program code morphing.

Word	Stack	Description
`c!`	`( frame -- )` `( RESET -- )`	`SETC`: Sets frame of shadow environment for code morphing. `RESET` sets code pointer of shadow frame to the beginning of shadow frame.
`>>c`	`( m1 m2 -- )`	`COPYC`: Switches to morphing state: Transfers code from program frame between markers `m1` and `m2` into shadow frame (including markers). Only marker and `STOC` commands are interpreted.
`>c`	`( -- )`	`TOC`: Copies next word from program frame into shadow frame.
`s>c`	`( n -- )`	`STOC`: Pops n data value(s) from stack and stores values as word literals in shadow frame.
`<m>`	`( -- )`	`MARKER`: Sets a marker position anywhere in a program frame.
`<m>@`	`( -- marker )`	`GETMARKER`: Gets a marker (maps symbolic names to unique numbers).
`<m>!`	`( -- )`	`SETMARKER`: Sets shadow code pointer after marker in shadow frame. Marker is searched in shadow frame, thus either in-place of execution frame or in a new created/copied shadow frame (containing already code and marker). Can be used to edit a partial range of shadow frame code using `STOC` and `TOC` instructions.

The `STOC` command is used to store the data that is part of an agent's process state for migration. The `TOC` and `COPYC` instructions can be used to indirectly save the control state of the agent as they enable reassembly and modification of code fragments depending on the current data and control state. Alternatively, the process control state can be saved by implementing a Finite-State Machine (FSM) in FORTH, for instance, using a switch branching statement, and saving the state variable in code before the migration step with `STOC`.
Table **2** explains several FORTH extensions which can be used for the modification of the dictionary and for the creation of objects. These instruc-

tions allow mobil agents to create (allocate) and import memory, word, and inter-process communication (IPC) objects. Finally, FORTH instructions required for program frame execution and distribution are shown. The contents of a shadow frame are always sent to and executed on a different or remote processor.

Tab. 2. Some FORTH extensions providing 1. dictionary modification and object creation, and 2. multi-processing support and frame distribution.

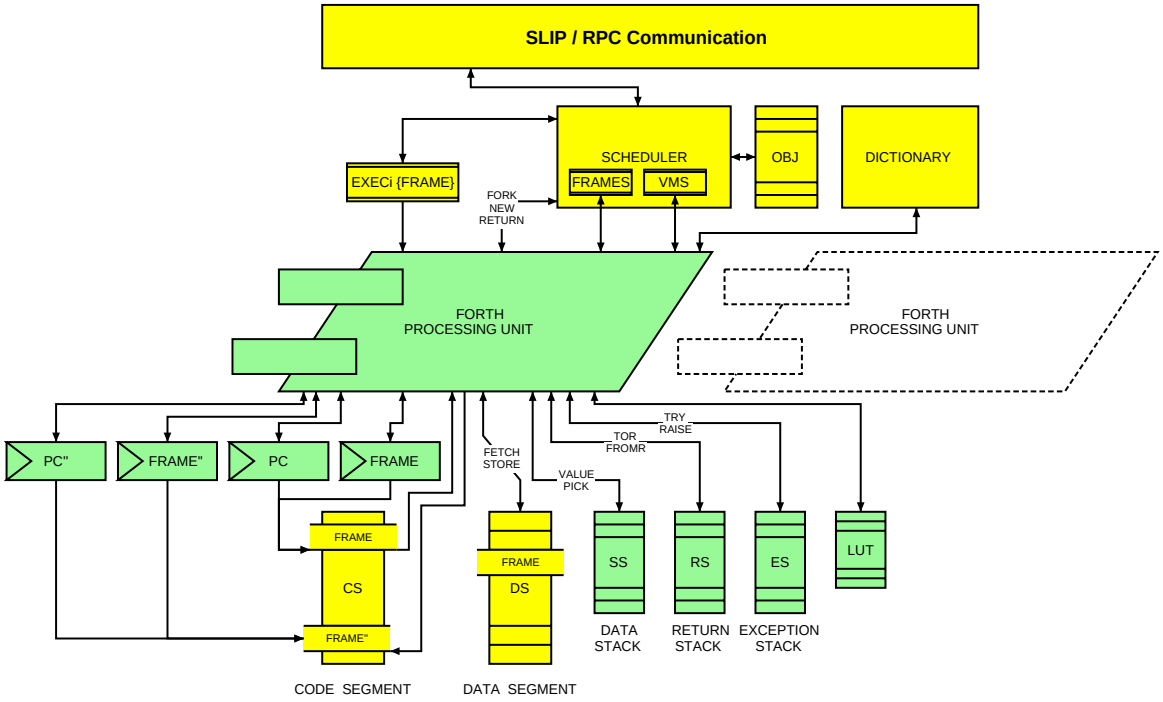
Instruction	Description
`VARIABLE` *x* `ARRAY [n,m]` *x* `VARIABLE*` *x* `ARRAY* [n,m]` *x*	Creates a new variable or array and allocates memory. The first two definitions create public objects and they are added to the dictionary. The star definitions create private objects.
`OBJECT MUTEX` *x* `OBJECT FRAME` *f*	Creates (allocates) a new IPC or frame object. The object is added to the dictionary. Other supported IPC object types: `SEMA`, `EVENT`, `TIMER`.
`IMPORT VARIABLE` *x* `IMPORT OBJECT` *x*	Imports a variable or object from the dictionary. If not found, then the program execution terminates (return status 0).
dx dy `fork`	Sends contents of shadow frame for execution to node relative to actual node. If dx=0 and dy=0, then the shadow frame is executed locally and concurrently on a different FORTH processing unit. The fork instruction returns the frame sequence or processing unit number.
id `join`	Waits for termination of a forked frame or the reception and execution of a reply program frame.
status `return`	Finishes execution of a program. If status is zero, no reply is generated. If status is equal to -1, an empty reply is generated. Finally, if status is equal 1, the contents of the shadow frame are sent back to the original sender of the execution frame.

3. Runtime Environment and Data Processing Architecture: DPCM-FORTH

The principal system architecture of one **FORTH processing unit** (PU) part of the DPCM-FORTH runtime environment is shown in figure **2**. A FORTH processing unit executes instructions from a node-shared code segment `CS`. The code segment is partitioned into frames. The next instruction to be executed is pointed by a program counter `PC`. A FORTH program containing top-level instructions and word definitions belongs to one particular frame, thus the code segment can hold various programs (and word definitions). The actually executed program is referenced by a frame point-

er FRAME. In addition to the frame to be executed (the execution frame) there is a shadow frame environment with its own set of program and frame pointers, PC" and FRAME". This shadow frame, which is initially identical to the execution frame, is used for code morphing. Local data manipulation performed by the program uses a data stack SS and return stack RS, known from traditional FORTH architectures. Data manipulation with random-access behaviour is possible and operates on a separate data segment DS shared by all PUs of the same network node. There is a third stack ES used for exception handling.

Fig. 2. **Runtime architecture consisting of FORTH data processing units, shared memory and objects, dictionary, scheduler, and communication.**



A FORTH processing unit initially waits for a frame to be executed. During program execution, the FORTH processing unit interacts with the scheduler to perform program forking, frame propagation, program termination, object creation (allocation), and object modification. The set of objects consists of the inter-process communication objects (IPC: mutex, semaphore, event, timer) and frames. There are private and public (node-visible) variables and arrays. All program frames have access to public variables by looking up references stored in the dictionary. Program word, memory variable, and object relocation are carried out by using a frame-bounded **lookup table** LUT.

The **scheduler** is the bridge between a set of locally parallel executing FORTH processing units, and the communication system, a remote procedure call (RPC) interface layered above SLIP, building a DPCM RE. At least two processing units are required to perform synchronous remote code execution (one for the actual program execution performing the request, and one for the reply).

The RPC processing unit receives messages (packets) from the protocol stack and transforms them into program frames, finally passed to the scheduler. The scheduler takes a free FORTH processor from the processor pool (queue) `VMS` and schedules execution of the frame. During program execution, the scheduler can be used to send a program frame to a different node, passed to the RPC processing unit.

All program processing units share a common dictionary, code, and data segment. There is a pool of objects `OBJ` (memory, IPC, frames), managed by the scheduler and a garbage collector.

The FORTH processing unit is a CISC (complex instruction set) architecture in terms of the number of instructions and the expressiveness of their operationality. The motivation for a CISC approach was the minimization of code size and hence communication cost of transferring/propagating of program code. The machine instruction format is fixed and independent of the particular instruction (though there are different classes of commands with different sub-structure).

3.1. From High-Level Modelling to a Hard- and Software Implementation of a Node

The runtime environment is modelled on behavioural level using a high-level multi-process programming language with atomic-guarded actions and inter-process communication (communicating sequential processes) **[1]**. Mostly, processes communicate with each other by using queues, for example, the FORTH processor or the RPC and SLIP implementation processes. The architecture and implementation model can be matched to different word and data sizes and sizes of code and data segments. The number of FORTH processors included in one node can be chosen in the range from one to eight. The communication system is also scalable and adaptable to different environments. Because the implementation of the FORTH runtime system is static, a pool of objects (memory, IPC, frames)

is created, and during runtime those objects are allocated from and returned to the pool. The entire design is partitioned into 43 concurrently executed processes, communicating by using 24 queues, 13 mutex, 8 semaphores, 52 RAM blocks, 59 shared registers, and 11 timers.
All architecture parts of the DPCM-FORTH node, including communication, FORTH processing units, scheduler, dictionary and relocation support, are mapped entirely to **hardware** multi-RT level and a single SoC design using the ConPro compiler **[1]**. The resource demand depends on the choice of design parameters and is between 1M - 3M equivalent gates (in terms of FPGA architectures).
The same multi-process programming model and source code used for the synthesis of the hardware implementation can also be compiled into **software** with the ConPro compiler. Multi-processing is simulated with light-weight processes. The software model has the same function as the hardware model (though with different latency and data through-put). A DPCM-FORTH compiler transforms source code into machine instructions.

4. Distributed Sobel Filter: an Example

After the main part of this paper dealt with the details of the hardware architecture and FORTH extensions that we introduced for the implementation of mobile agents by means of code morphing, this section gives an example of how the described runtime environment can be used. We proof the approach with a FORTH implementation of a distributed Sobel operator. Originally applied in image processing and computer vision, this edge detection filter can also be of use for crack detection in the aforementioned application scenario of sensorial materials. It is assumed that each network node can read data from one local sensor, that is, an accumulated central view of the structural state does not exist.
A Sobel kernel *S* is used for a neighbourhood operation on the original image composed of sensor data, for example, a 4x4 matrix *A*, and each matrix entry represents a node in the sensor network. There are two different operators *S*, each for a different direction sensitivity (x/y), shown in eq. **1**. The output image *G* results from a convolution operation.
The FORTH program implementing an agent moving and migrating through the area of interest is shown in Ex. **1**. Initially, a master agent is sent to the

upper left corner node, sampling data and performing a partial image convolution. Each node calculation carries out sum terms of $g_{i,j}$ elements containing only the local sensor data $a_{x,y}$, updating $g_{i,j}$ with pseudo-code shown in Eq. **2** (assuming array index numbers within range 1...N).

Eq. 1. **Sobel operator definition and image convolution**

$$S^x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, S^y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \begin{aligned} G^x &= S^x \bullet A \\ G^y &= S^y \bullet A \end{aligned}$$

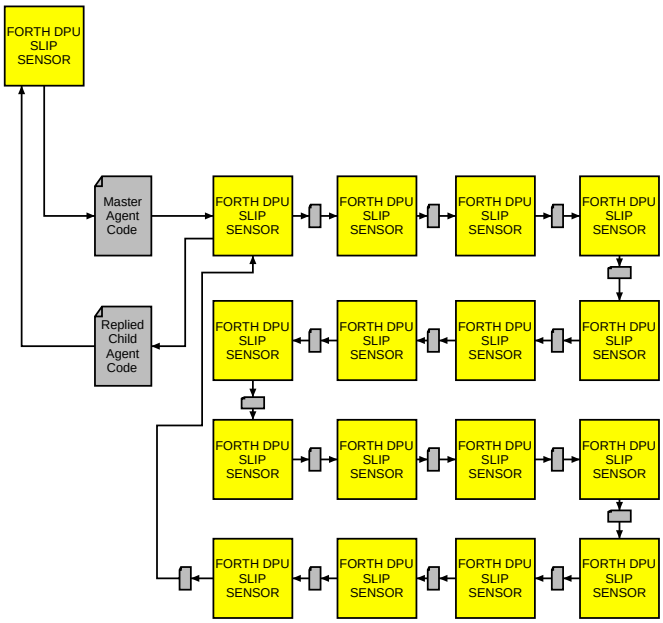
Eq. 2. **Partial calculation of G-elements containing actual image value $a_{x,y}$ at node (x,y)**

$$\forall (i,j) \in \{x-1, x, x+1\} \times \{y-1, y, y+1\} \text{ DO}$$

$$g_{i,j} \leftarrow g_{i,j} + s_{2-i+x,\,2-j+y} \cdot a_{x,y}$$

The results are stored in the agent program code using code morphing and finally the agent travels to the next node, and so forth. If the last node has been visited, then the agent is sent to the first initial node and initiates a reply to be sent to the original node requesting the filtered image data. The migration path of the agent is shown in figure **3**.

Fig. 3. **Travelling agent performing a distributed implementation of the Sobel image filter applied to an area of a sensor network.**



Ex. 1. **FORTH implementation of the Sobel filter agent. Note: only the x-sensitive Sobel operator is shown here.**

```
1   VARIABLE* x
2   VARIABLE* y
3   VARIABLE* dir
4   VARIABLE* data      4 CONSTANT N
5   <x> 1 x !
6   <y> 1 y !
7   <dir> 1 dir !
8   IMPORT WORD getdata
9   x @ 1 = y @ 1 = and if
10    ARRAY [N,N] a  ( original sensor data  )
11    ARRAY [N,N] g  ( convoluted data       )
12  else
13    ARRAY* [N,N] a  ( original sensor data  )
14    ARRAY* [N,N] g  ( convoluted data       )
15  then
16  ARRAY* [3,3] s    ( sobel operator        )
17  1 0 -1 -2 0 -2 1 0 -1   s >>[]
18  ( a11,a12,a13,a14,a21,...,a44 )
19  <matrix_start>
20    <matrix_a>  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   a >>[]
21    <matrix_g>  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   g >>[]
22  <matrix_end>
23
24  :* sobel              ( note: lowest array index is 0 )
25    x @ x @ 2 - do      ( pos. x-1..x+1 => array x-2..x )
26      I a!              ( save index i in register a    )
27      y @ y @ 2 - do    ( pos. y-1..y+1 => array y-2..y )
28        I b!            ( save index j in register b    )
29        a@ 0 >= b@ 0 >= and a@ < N and b@ < N and if
30          a@ b@ [g] @       ( fetch g[i,j]    )
31          1 a@ - x @ +      ( 2-i+x  => 1-i+x )
32          1 b@ - y @ +      ( 2-j+y  => 1-j+y )
33          [s] @ data @ *    ( s[i',j']*a[x,y] )
34          + a@ b@ [g] !     ( store g[i,j]    )
35        then
36      loop
37    loop
38  ;
39  :* sample
40    getdata  dup
41    x @ 1 - y @ 1 - [a] !
42    data ! ( save sampled data for local computation )
43  ;
44  :* update
45    (
46      transfer data of array to stack and then
47      convert values to word literals in program code
48    )
49    N 1- 0 do
50      I a!
51      N 1- 0 do
52        I b!
53        a@ b@ [g] @
54      loop
55    loop
56    <matrix_g>! N N * s>c
57    N 1- 0 do
58      I a!
59      N 1- 0 do
60        I b!
61        a@ b@ [a] @
62      loop
63    loop
64    <matrix_a>! N N * s>c
65  ;
```

```
66  :* reply
67    RESET c!
68    <reply_header_start>@ <reply_header_end>@ >>c
69    <matrix_start>@ <matrix_end>@ >>c
70    <reply_start>@ <reply_end>@ >>c
71  ;
72  :* migrate
73    (
74      migrate to next node depending on {x,y,dir} settings
75    )
76    dir @ 1 = if
77      x @ 4 = if
78        y @ 1 + <y>! 1 s>c     ( update y counter              )
79        <dir>! -1 1 s>c        ( revert propagation direction  )
80        0 1 fork
81      else
82        x @ 1 + <x>! 1 s>c     ( update x counter, goto right  )
83        1 0 fork
84      then
85    else
86      x @ 1 = y @ N <> and if
87        y @ 1 + <y>! 1 s>c     ( update y counter              )
88        <dir>! 1 1 s>c         ( revert propagation direction  )
89        0 1 fork
90      else
91        y @ N = if
92          ( create reply and go back to origin )
93          reply
94          0 -4 fork
95        else
96          x @ 1 - <x>! 1 s>c   ( update x counter, goto left   )
97          -1 0 fork
98        then
99      then
100   then
101     x @ 1 = y @ 1 = and if
102     ( master agent, wait for reply )
103     OBJECT EVENT sobel_await
104     sobel_await #await
105     ( propagate reply to original sender of agent )
106     update reply
107     forget a forget g        ( cleanup dictionary          )
108     1 return
109   else
110     0 return
111   then
112 ;
113 sample
114 sobel
115 update
116 migrate
117
118 ( not reached )
119 <reply_header_start>
120     IMPORT ARRAY [N,N] a
121     IMPORT ARRAY [N,N] g
122     IMPORT OBJECT sobel_await
123 <reply_header_end>
124 <reply_start>
125     sobel_await #wakeup
126     0 return
127 <reply_end>
```

The FORTH program consists of five words (private, indicated by the star after the definition command). In lines 9 to 15, arrays `a` and `g` are defined, either private or public depending on the location of the agent. Public arrays are required for the processing of the final result and creation of a reply agent performed by the master agent at the origin node (1,1). In line 16, the Sobel `s` matrix is defined and initialized with (constant) values (line 17). The `>>[]` operator copies values taken from the stack to the respective array. The main word execution sequence is defined in lines 113-116.

First, a new data value is sampled from the node's AD converter by calling the word `sample`. The value is saved in the image array `a` and variable `data`. The `[a]` operator calculates the memory address required for the matrix access. After data sampling, the Sobel computation is performed by

calling the word `sobel`. Two nested loops (lines 25 to 37) compute sum terms of elements of array `g` containing only the actual sampled image value `a[x,y]`. The x and y positions are stored in their respective variables.
The `migrate` code distinguishes different cases regarding the agent's current location. When code morphing is done, the modified program frame is dispatched to the next node. Once all nodes have been visited, the agent sends back a reply agent to the requesting node. This transmits the final result of the distributed computation (line 94).
The program is compiled to a machine program consisting of 599 words. The final reply code requires only 103 words.
The size of the program code (determining the communication cost) of the migrating agent performing the computations can be reduced by using a two-level agent system. The arrays `a`, `g`, and `s` (with initialization), and the definitions for the words `sample`, `sobel`, and `update`, which remain untouched by code morphing the entire time, are distributed and permanently stored using a distribution agent before the computation agent is started. In this case, the words will be stored in the public dictionary of each node. The program frame of the distribution agent is held permanently until the words are removed from the dictionary, cleaned up finally by the garbage collector.

5. Summary, Conclusion, and Outlook

This paper introduced a hardware architecture and runtime environment specifically designed towards the implementation of mobile agents by using dynamic code morphing under the constraints of low-power consumption and high component miniaturization. It uses a modified and extended version of FORTH as the programming language for agent programs. The runtime environment is modelled on the behavioural level using a multi-process-oriented programming language and can be embedded in a single-SoC hardware design. A functional equivalent piece of software can be synthesized and executed on a generic desktop computer. To show the viability of the presented distributed and parallel computing approach, a filtering algorithm was borrowed from the field of image processing and applied in the application scenario of sensorial materials. In the given example, multiple mobile agents move through a network of sensor nodes, jointly execut-

ing a spatially distributed data processing task. Calculation results and a subset of the agents execution state are preserved within the agent's program code during migration to different network nodes. The size of the migrating code can be significantly reduced in size by decoupling functions that remain unaffected by code morphing during operation from the migrating agent program and distributing them in the data processing network beforehand.

Synchronous inter-agent communication can be carried out by using reply agents send back to a parent agent waiting for the reception and execution of the reply.

Future work will be the development and practical evaluation of sophisticated distributed load and defect detection algorithms on this architecture for use in sensorial materials.

10. Bibliography

[1] S. Bosse, *Hardware-Software-Co-Design of Parallel and Distributed Systems Using a unique Behavioural Programming and Multi-Process Model with High-Level Synthesis*, Proceedings of the SPIE Microtechnologies 2011 Conference, 18.4.-20.4.2011, Prague, Session EMT 102 VLSI Circuits and Systems

[2] L. Iftode, C. Borcea, and P. Kang, *Cooperative Computing in Sensor Networks*, In: Ilyas, M. (ed.) Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems. CRC Press, Boca Raton (2004)

[3] E. D. Rather, D. R. Colburn, C. H. Moore, *The evolution of Forth*, Proceedings SIGPLAN Not. 28, 3 (March 1993)

[4] S. Bosse, D. Lehmhus, *Smart Communication in a Wired Sensor- and Actuator-Network of a Modular Robot Actuator System using a Hop-Protocol with Delta-Routing*, Proceedings of Smart Systems Integration conference, Como, Italy, 23-24.3.2010 (2010)

[5] A. Kent, J. G. Williams (Eds.), *Mobile Agents*, Encyclopedia for Computer Science and Technology, New York: M. Dekker Inc., 1998

[6] H. Peine, T. Stolpmann, *The Architecture of the Ara Platform for Mobile Agents*, MA '97 Proceedings of the First International Work-

shop on Mobile Agents,Springer-Verlag London, 1997

[7] A.I. Wang, C.F. Sørensen, and E. Indal., *A Mobile Agent Architecture for Heterogeneous Devices*, Wireless and Optical Communications, 2003

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

[9] P. Koopmann, *Stack Computers: the new wave*, 1989