Design of Material-integrated Distributed Data Processing Platforms with Mobile Multi-Agent Systems in Heterogeneous Networks

Stefan Bosse

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

Keywords: Multi-Agent Platform, Sensor Network, Mobile Agent, Heterogeneous Networks, Embedded Systems

Abstract: An agent processing platform suitable for distributed computing in sensor networks consisting of low-resource (e.g., material-integrated) nodes is presented, providing a unique distributed programming model and enhanced robustness of the entire heterogeneous environment in the presence of node, sensor, link, data processing, and communication failures. In this work multi-agent systems with mobile activity-based agents are used for sensor data processing in unreliable mesh-like networks of nodes, consisting of a single microchip with limited low computational resources. The agent behaviour, interaction, and mobility (between nodes) can be efficiently integrated on the microchip using a configurable pipelined multi-process architecture based on Petri-Nets. Additionally, software implementations and simulation models with equal functional behaviour can be derived from the same source model. Hardware and software platforms can be directly connected in heterogeneous networks. Agent interaction and communication is provided by a simple tuple-space database and signals providing remote inter-node level communication and interaction. A reconfiguration mechanism of the agent processing system offers activity graph changes at run-time.

1. INTRODUCTION

Trends are recently emerging in engineering and micro-system applications such as the development of sensorial materials (Lang, 2011) show a growing demand for distributed autonomous sensor networks of miniaturized low-power smart sensors embedded in technical structures (Pantke, 2011). These sensor networks are used for sensorial perception or structural health monitoring, employed, for example in Cyber-Physical-Systems (*CPS*), and perform the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource-constrained manner under real-time processing and technical failure constraints.

To reduce the impact of such embedded sensorial systems on mechanical structure properties, single microchip sensor nodes (in mm^3 range) are preferred. Real-time constraints require parallel data processing inadequately provided by software based systems.

Multi-agent systems can be used for a decentralized and self-organizing approach of data processing in a distributed system like a sensor network (Guijarro, 2008), enabling information extraction, for example based on pattern recognition (Zhao, 2008), and by decomposing complex tasks in simpler cooperative agents.

Hardware (microchip level) designs have advantages compared with microcontroller approaches concerning power consumption, performance, and chip resources by exploiting parallel data processing (covered by the agent model) and enhanced resource sharing (Bosse, 2011), which will be applied in this work.

Usually sensor networks are a part of and connected to a larger heterogeneous computational network (Guijarro, 2008). Employing of agents can overcome interface barriers arising between platforms differing considerably in computational and communication capabilities. That's why agent specification models and languages must be independent of the underlying run-time platform. On the other hand, some level of resource and processing control must be available to support the efficient design of hardware platforms.

Hardware implementations of multi-agent systems are still limited to single or a few and immobile agents (Meng, 2005, Naji, 2004), and were originally proposed for low level tasks, for example in (Ebrahimi, 2011) using agents to negotiate network resources. Coarse grained reconfiguration is enabled by using FPGA technologies (Meng, 2005). Most current work uses hardware-software co-design methodologies and code generators, like in (Jamont, 2008). This work provides more fine-grained agent reconfiguration and true agent mobility without relying on a specific technology and employs high-level synthesis to create standalone hardware and software platforms delivering the same functional and reactive behaviour.

There is related work concerning agent programming languages and processing architectures, like *APRIL* (McCabe, 1995) providing tuple-space like agent communication, and widely used *FIPA ACL*, and *KQGML* (Kone, 2000) focusing on high-level knowledge representations and exchange by speech acts, or model-driven engineering (e.g. INGENIAS, Sansores, 2008). But the above required resource and processing control is missing, which is addressed in this work.

There are actually four major issues related to the scaling of traditional software-based multi-agents systems to the hardware level and their design:

- limited static processing, storage, and communication resources, real-time processing,
- unreliable communication,

- suitable simplified agent-oriented programming models and processing architectures qualified for hardware designs with finite state machines (FSM) and resource sharing for parallel agent execution,
- and appropriate high-level design tools.

Traditionally agent programs are interpreted, leading to a significant decrease in performance. In the approach presented here, the agent processing is directly implemented in standalone hardware nodes without intermediate processing levels and without the necessity of an operating system.

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

- One common agent behaviour model, which is implementable on different processing platforms (hardware, software, simulation).
- Agent mobility crossing different 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.
- Support for heterogeneous networks and platforms covered by one design and synthesis flow including functional behavioural simulation.
- A token-based pipelined multi-process agent processing architecture suitable for hardware platforms with Register-Transfer Level Logic offering optimized computational resources and speed.
- A Petri-Net representation is used to derive a specification of the hardware process and communication network, and performing advanced analysis like deadlock detection. Timed Petri-Nets can be used to calculate computational time bounds to support real-time processing.

The next sections introduce the activity based agent processing model, available mobility and interaction, and the proposed agent platform architecture related to the programming model. Finally, a case study shows the suitability of the proposed design approach.

2. STATE-BASED MOBILE AGENTS

The implementation of mobile multi-agent systems for resource constrained embedded systems with a particular focus on microchip level is a complex design challenge. High-level agent programming and behaviour modelling languages can aid to solve this design issue. To carry out multi-agent systems on hardware platforms, the activity-based agent-orientated programming language *AAPL* was designed. Though the imperative programming model is quite simple and closer to a traditional PL it can be used as a common source and intermediate representation for different agent processing platform implementations (hardware, software, simulation) by using a high-level synthesis approach, shown in Figure **1**. Commonly used agent behaviour models based on *PRS*/*BDI* architectures with a declarative paradigm (2APL, AgentSpeak/Jason), communication models (e.g. FIPA *ACL*, *KQML*), and adaptive agent models can be implemented with *AAPL* providing primitives for the representation of beliefs or plans (discussed later). Agent mobility, interaction, and replication including inheritance are central multi-agent-orientated behaviours provided by *AAPL*.

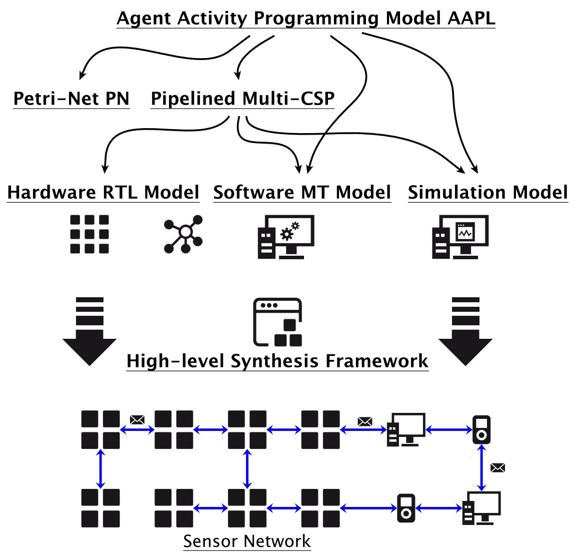
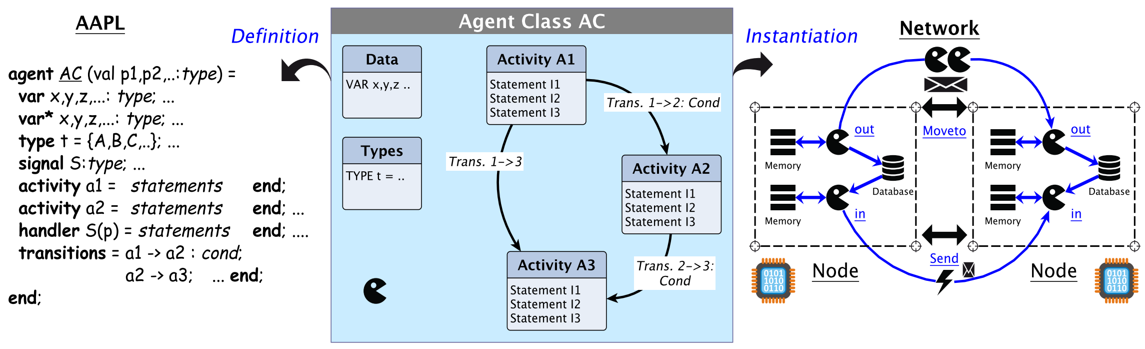


Figure 1. *From AAPL level to heterogeneous distributed networks (RTL: Register-Transfer Level, MT: Multi-Threading, CSP: Communicating Sequential Processes*

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

2.1. AAPL Programming Model

The agent behaviour is partitioned and modelled with an activity graph, with activities representing the control state of the agent reasoning engine, and conditional transitions connecting and enabling activities,. Activities provide a procedural agent processing by sequential execution of imperative data processing and control statements.



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

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

The activity-graph based agent model is attractive due to the proximity to the finite-state machine model, which simplifies the hardware implementation.
An activity is activated by a transition depending on the evaluation of (private) agent data (conditional transition) related to a part of the agents belief in terms of *BDI* architectures, or using unconditional transitions (providing sequential composition), shown in Figure **2**. An agent belongs to a specific parameterizable agent class *AC*, specifying local agent data (only visible for the agent itself), types, signals, activities, signal handlers, and transitions.
Plans are related to *AAPL* activities and transitions close to conditional triggering of plans. Definition **1** summarizes the available language statements.
Instantiation: 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.
Each agent has *private data* (body variables), defined by the var and var* statements. Variables in the latter statement will not be inherited or migrated! Agent body variables in conjunction with transition conditions represent the mobile data part of the agents beliefs database.
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.
Agent interaction and synchronization is provided by a tuple-space database server available on each node (based on McCabe, 1995). 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 as 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 in this way. Simplified the expression of beliefs of agents is strongly based on *AAPL* tuple database model. Tuple values have their origin in environmental perception and processing bound to a specific node location.
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 interaction between agents is provided by signals carrying optional parameters (they can be used locally, too). A signal can be raised by an agent using the send(ID,S,V) statement specifying the ID of the target 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. Signals implement remote procedure calls. Within a signal handler a reply can be sent back to the initial sender by using the reply(S,V) statement.

Timers can be installed for temporal agent control using (private) signal handlers, too. Agent processing can be suspended with the sleep and resumed with the wakeup statements.

Migration of agents (preserving the local data and processing state) to a neighbour node is performed by 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.

Reconfiguration: Agents are capable to *change their transitional network* (initially specified in the transition section) by changing, deleting, or adding (conditional) transitions using the transitionΞ(S1,S2,cond) statements (with Ξ='+':add, '-': remove, and '*': change transition). *This behaviour allows the modification of the activity graph, i. e., based on learning or environmental changes, which can be inherited by child agents.*

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

```
Agent Class Definition
  agent class (arguments) = definitions end;
Activity Definition
  activity name = statements end;
Data Statements
  var x,y,z:type;     var* a,b,c: type;
  x := ε(variable,value,constant);
Conditional Statements
  if cond then statements else statements end;
  case  ε of | v1 -> 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
  out(v1,v2,..);              .. exist?(v1,?,..) ..
  in(v1,x1?,v2,x2?,...);     rd(v1,x1?,v2,x2?,...);
  try_in(timeout,v1,..);     try_rd(timeout,v1,..);
  mark(timeout,v1,v2,..);   rm(v1,?,..);
Signals
  signal S:datatype;
  handler S(x) = statements end;
  send(ID,S,v); reply(S,v);
  broadcast(AC,DX,DY,S,v);
  timer+(timeout,S); timer-(S); sleep; wakeup;
Exceptions
  exception E;                  raise E;
  try statements except E -> statements end;
Mobility, Creation, and Replication
  moveto(direction);
  .. link?(direction) ..
  id := new class (arguments);
  id := fork(arguments);
  kill(id);
Transitional Reconfiguration
  transition+(a1,a2,cond);transition*(a1,a2,cond);
  transition-(a1,a2);
```

2.2. Agent Communication

Agent communication can be achieved basically by three different methods: 1. signal propagation (similar to commitment messages in *AGENT0*, Shoham, 1991), 2. tuple database exchange, and 3. by using agents with a composition of methods 1 & 2. These basic methods can be used to realize common higher-level agent communication languages like *ACL* or *KQML* (tuple patterns correspond to message content entries). Signal propagation implements light-weighted asynchronous peer-to-peer remote-procedure calls, executed on target agents with appropriate signal handlers, which must not necessarily belong to the same agent class, whereas pattern matching based tuple database access can be performed by any group of agents having a common understanding of the meaning of data and which are actually processed on the same platform node.

For example, a simple FIPA *ACL* based request from agent A (initiator) to B (participant), which ask for a database tuple on B can be created with the following *AAPL* code pattern using signals:

```
FIPA ACL: (request :sender IDA
  :receiver IDB :content (p ?) :ontology TS2)
AAPL:
signal REQ1,REQ2,INFR,FAIL;
-- Agent A --
var pv,ps;
handler INFR(v) = pv := v; ps := true; wakeup
..
handler FAIL = ps := false; wakeup ..
function request(AID,p) =
  send(AID,REQ1,p); timer+(100,FAIL);
  sleep;
  if ps then return pv else raise FAILED end ..
-- Agent B --
handler REQ1(arg) =
  var v;
  if exist?(p,?) then
    in(p,?v); reply(INFR,v)
  else reply(FAIL) ...
```

3. AGENT PLATFORM SYNTHESIS

The *AAPL* model is a common source for the implementation of agent processing on hardware, software, and simulation processing platforms. A database driven high-level synthesis approach (Bosse, 2013) is used to map the agent behaviour to these different platforms. The agent processing architecture required on each network node must implement different agent classes and must be scalable to the microchip level to enable material-integrated embedded system design, which represents a central design issue, further focussing on parallel agent processing and optimized resource sharing.

3.1. Hardware Platform

This microchip-level processing platform implements the agent behaviour with *reconfigurable pipelined communicating processes* (*PCSP*) related to the Communicating Sequential Process model (*CSP*) proposed by Hoare (1985). The activities and transitions of the *AAPL* programming model are merged in a first intermediate representation by using state-transition Petri Nets (*PN*), shown in Figure **3**. This *PN* representation allows the following *CSP* derivation specifying the process and communication network, and advanced analysis like deadlock detection. Timed Petri-Nets can be used to calculate computational time bounds to support real-time processing.
Keeping the *PN* representation in mind, the set of activities $\{A_i\}$ is mapped to a set of sequential processes $\{P_i\}$ executed concurrently. Each subset of transitions $\{T_{i,j}\}$ activating one common activity process P_j is mapped to a synchronous N:1 queue Q_j providing inter-activity-process communication, and the computational part for transitions embedded in all contributing processes $\{P_i\}$, shown in Fig. **3**. Changes (reconfiguration) of the transition network at run-time are supported by transition path selectors.
Each sequential process is mapped (by synthesis) to a finite-state machine and a datapath using a register-transfer architecture (RTL) with mutual exclusive guarded access of shared objects, all implemented in hardware.
Agents are represented by tokens (natural numbers equal to the agent identifier, unique at node level), which are transferred by the queues between activity processes depending on the specified transition conditions. This multi-process model is directly mappable to register-transfer level *RTL* hardware. Each process P_i is mapped to a finite state machine FSM_i controlling process execution and a register-transfer data path. Local agent data is stored in a region 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 conditional 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 statements $\{s_{j,i}\}$ of an activity A_i are assigned to separate intermediate IO processes $\{P_{i,j}\}$ handling only IO events or additional post computations, as shown on the bottom of Fig. **3**. Agents in different activity states can be processed concurrently. Thus, activity processes that are shared by several agents may not block. To prevent blocking of *IO* processes, 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.
This pipeline architecture offers advanced resource sharing and parallelized agent processing with only one activity process chain implementation required for each agent class. The hardware resource requirement (digital logic) is divided into a control and a data part. The control part is proportional to the number of supported different agent classes. The data part depends on the maximal number of agents executed by the platform and the storage requirement for each agent.

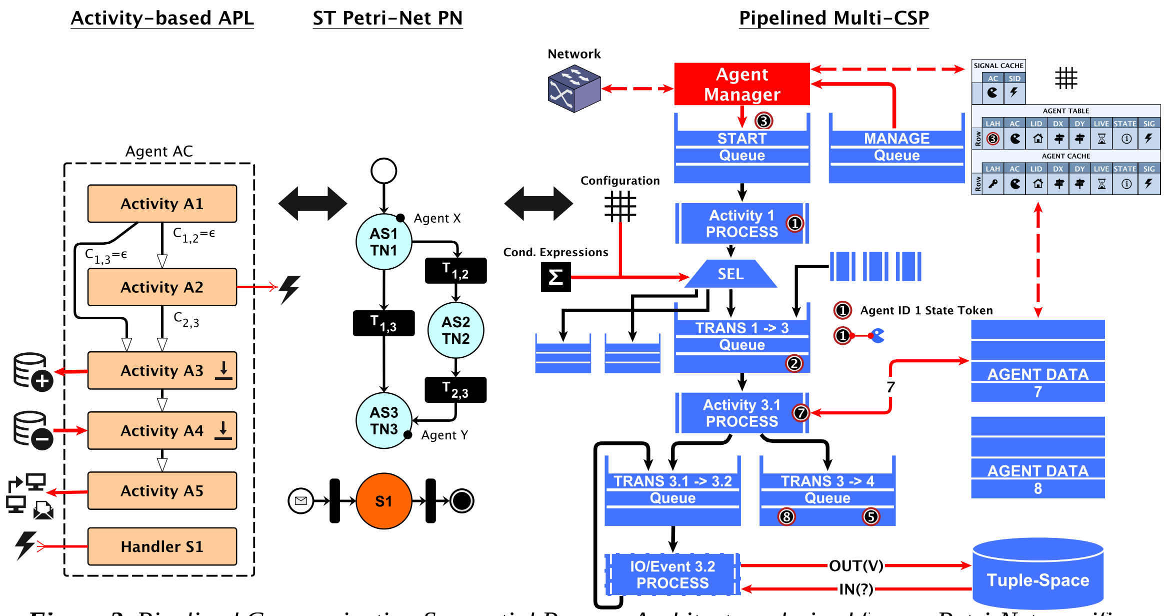


Figure 3. *Pipelined Communicating Sequential Process Architecture derived from a Petri-Net specification and relationship to the activity graph. Signals ar handled asynchronously.*

3.2. Software Platform

There are two different architectures for the implementation of the agent processing on programmable platforms: 1. by using the already introduced *PCSP* architecture, or 2. by using a direct procedural composition of the activity graph and its transitions. In the first case, the activity processes are implemented with light weighted processes (threads) and queues, providing token based agent processing. In the second case, each agent is assigned to and processed by one thread created at run-time. Activities are one-to-one mapped to procedures called by a transition scheduler (as in the simulation platform case). Blocking of agent processing is handled by the thread implementation itself. In any case further software modules implement the agent manager, tuple space databases, and networking. A software platform can be directly connected to hardware nodes and vice versa.

3.3. Simulation Platform

In addition to real hardware and software implemented agent processing platforms there is the capability of the simulation of the agent behaviour, mobility, and interaction on a functional level. The *SeSAm* simulation framework (Klügel, 2009) offers a platform for the modelling, simulation, and visualization of mobile multi-agent systems employed in a two-dimensional world. The behaviours of agents are modelled with activity graphs (specifying the agent reasoning machine) close to the *AAPL* model. Activity transitions depend on the evaluation of conditional expressions using agent variables. Agent variables can have a private or global (shared) scope. Basically *SeSAm* agent interaction is performed by modification and access of shared variables and resources (static agents). In addition to the agent reasoning specification there are global visible feature packages that define variables and function operating on these variables. Features can be added to each agent class. Agents can change their position in the two-dimensional world map enabling mobility, and new agents can be created at run-time by other agents. The *SeSAm* framework was chosen due to the activity-based agent behaviour and the data model which can be immediately synthesized from the common *AAPL* source and can be imported by the simulator from a text based file stored in *XML* format. This model exchange feature allows the tight coupling of the simulator to the synthesis framework.

In principle, *AAPL* activity graphs can be directly mapped on the SeSAm agent reasoning model. But there are limitations which inhibit the direct mapping. First of all, *AAPL* activities (IO/event-based) can block (suspend) the agent processing until an event occurs. Blocking agent behaviour is not provided directly by *SeSAm*. Secondly, the transition network can change during run-time. Finally, the handling of concurrent asynchronous signals used in *AAPL* for inter-agent communication cannot be established with the generic activity processing in SeSAm (the provided exception handling is only used for exceptional termination of agents).

For this reason, the agent activity transitions including the dynamic transition network capability are managed by a special transition scheduler, shown in Figure **4**. This transition scheduler handles signals and timers, too, which are processed prioritized and passed to the signal scheduler. Each agent activity is activated by the transition scheduler. After a specific activity was processed, the transition scheduler is activated and entered again. An *AAPL* activity can be split in computational and IO/event-based sub-activities in the presence of blocking statements (e.g. in and rd tuple space interaction).

There is a special node agent implementing the tuple database with lists (partitioned to different spaces for each dimension), and managing agents and signals actually bound to this particular node. Concurrent manipulation of lists is non-atomic operations in *SeSAm*, and hence requires mutual exclusion.

The *AAPL* mobility, interaction, configuration, and replication statements are implemented by feature packages.

3.4. Synthesis

The database driven synthesis flow (details in Bosse, 2013) consists of an *AAPL* front end, the core compiler, and several backends targeting different platforms. 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 split in 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. Different outputs can be produced: a hardware description enabling *SoC* synthesis using the *ConPro* high-level synthesis framework (details in Bosse, 2011), a software description (*C*) which can be embedded in application programs, and the *SeSAm* simulation model (*XML*). The *ConPro* programming model reflects an extended *CSP* with atomic guarded actions on shared resources. Each process is implemented with an *FSM* and an *RT* datapath.

All implementation models (HW/SW/SIM) provide equal functional behaviour, and only differ in their timing, resource requirements, and execution environments. Some more implementation and synthesis details follow.

Agent Manager

The agent manager provides a node level interface for agents, and it is responsible for the creation, control (including signals, events, and transition network configuration), and migration of agents with network

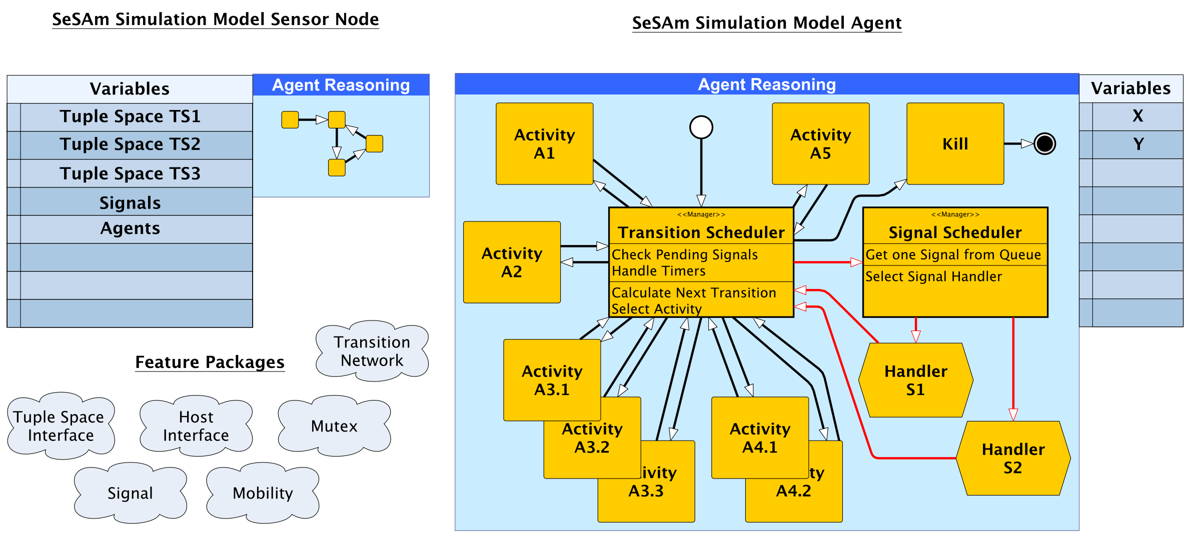


Figure 4. *Simulation Model used in the SeSAm MAS Simulator*

connectivity, implementing a main part of an operating system. The agent manager controls the tuple-space database server and signals events required for IO/event-based activity processes.

The agent manager uses agent tables and caches to store information about created, migrated, and passed through agents (req., for ex., for signal propagation).

Migration

Migration of agents requires the transfer of the agent data and the control state 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.

Transition Network

A switched *transition network* offers support for agent activity graph reconfiguration at run-time. Though the possible reconfiguration and the conditional expressions must be known at compile time (static resource constraints), a reconfiguration can release the use of some activity processes and enhances the utilization for parallel processing of other agents. The transition network is implemented with selector tables in case of the HW implementation, and with dynamic transition lists in case of the SW and SIM implementations.

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 to integer values).

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.

4. CASE STUDY

A small example implementing a distributed feature detection in an incompletely connected and unreliable mesh-like sensor network using mobile agents should demonstrate the suitability of the proposed agent processing and design approach. The sensor network consists of nodes with each node attached to a sensor used, for example, in a structural health monitoring system (e.g. strain-gauge sensors). 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 Figure **5**) is used, derived from the image feature extraction approach (proposed by Liu, 2001). A single sporadic sensor activity not correlated with the surrounding neighbourhood should be distinguished from an extended correlated region, which is the feature to be detected. There are three different agent classes: an exploration, a deliver, and a node agent. A *node agent* is immobile and is primarily responsible for sensor measurement, observation, and creating of exploration agents.

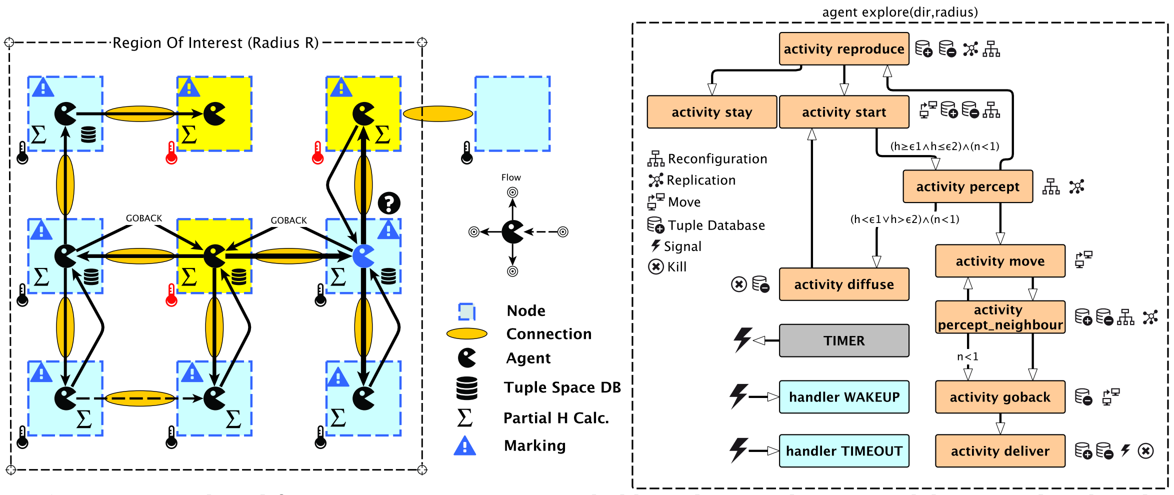


Figure 5. *Distributed feature extraction in an unreliable and incomplete network by using distributed agents with migration and self-organization behaviour (right: AAPL activity graph of the explorer agent)*

The feature detection is performed by the mobile *exploration agent*, which supports two main different behaviours: diffusion and reproduction. The diffusion behaviour is used to move into a region, mainly limited by the lifetime of the agent, and to detect the feature, here the region with increased mechanical distortion (more precisely the edge of such an area). The detection of the feature enables the reproduction behaviour, which induces the agent to stay at the current node, setting a feature marking and sending out more exploration agents in the neighbourhood. The local stimuli $H(i,j)$ for an exploration agent to stay at a specific node with coordinate (i,j) is given by eq. **1**.

$$H(i,j) = \sum_{s=-R}^{R}\sum_{t=-R}^{R}\{\|S(i+s,j+t)-S(i,j)\| \le \delta\} \tag{1}$$

S : Sensor signal strength

R : Rectangular region around (i,j)

The calculation of H at the current location (i,j) of the agent requires the sensor values within the rectangular area (the region of interest ***ROI***) R around this location. If a sensor value $S(i+s,j+t)$ with $i,j \in \{-R,..,R\}$ is similar to the value S at the current position (diff. is smaller than the parameter δ), H is incremented by one.
If the H value is within a parameterized interval $\Delta=[\varepsilon_0,\varepsilon_1]$, the exploration agent has detected the feature and will stay at the current node to reproduce new exploration agents sent to the neighbourhood. If H is outside this interval, the agent will migrate to a neighbour different node and restarts exploration (diffusion).
The calculation of H is performed by a distributed calculation of partial sum terms by sending out child explorer agents to the neighbourhood, which itself can send out more agents until the boundary of the region R is reached. Each child agent returns to its origin node and hands over the partial sum term to his parent agent, shown in Figure **5**. 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. **5**). A *deliver agent*, created by the node agent, finally delivers exploration results to interested nodes by using directed diffusion approaches, not discussed here.

AAPL Specification

Example **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 are moved 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 perceptive activity, which creates child agents (lines 44-46). Forked agents inherit all parent data and the current transition network configuration. Thus, in line 43 the transition percept $\rightarrow$ move is established (and inherited), but after forking reset in lines 47-50 for the parent agent behaviour, which await the return of all child agents and a decision for behaviour selection (reproduce/diffuse).

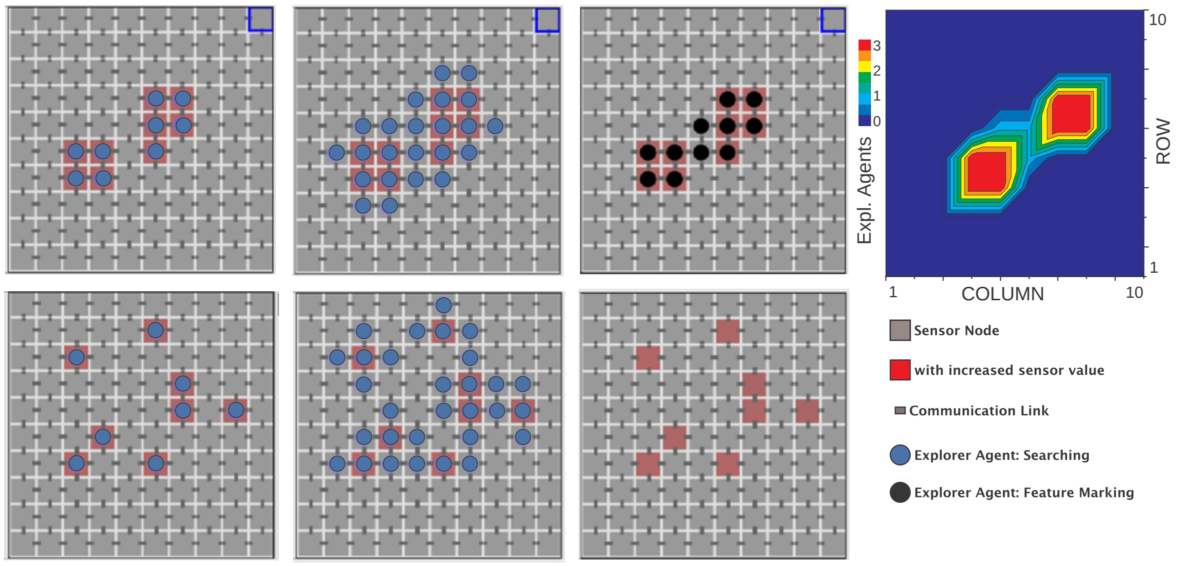


Fig. 6. *Simulation results for two different sensor network situations (left: start, middle: exploration, right: final result situation). Top row: sensor activity within clusters, bottom row: sensor activity scattered over the network.*

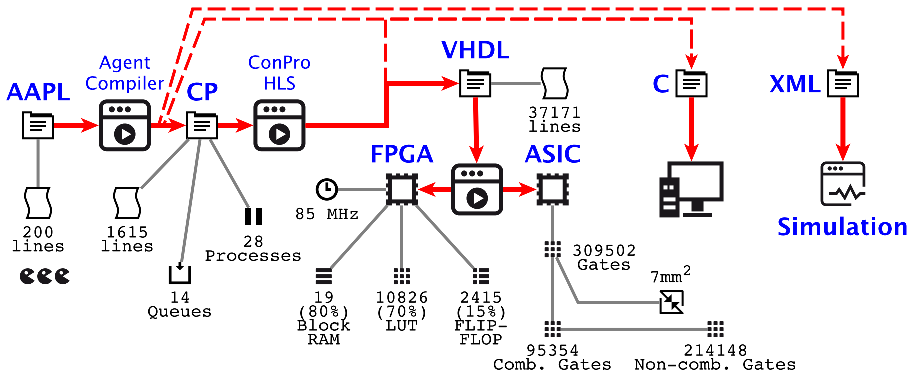
The child agents enter the `move` (lines 20-25) activity after forking and will be migrated in the specific direction to the neighbour node.
Finally, the `percept_neighbour` activity is reached, which performs the local calculation (line 52) if there was no marking found, and stores the partial result 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 of 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 69-72) 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).
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* (Klügel, 2009).

Simulation Results
Simulation results are shown in Figure **6** for two different sensor network situations, each consisting of a network with autonomous sensor nodes arranged in 10 rows and 10 columns. One situation creates significant sensor values arranged in a bounded cluster region, for example, caused by 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 not clustered 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: $\{\delta, \varepsilon_0, \varepsilon_1,$ *lifetime, search radius R*$\}$.

Synthesis Results
The synthesis results of the hardware implementation for one sensor node are shown in Fig. **7**. The *AAPL* specification was compiled to the *ConPro* programming model and synthesized to an *RTL* implementation on *VHDL* level. Two different target technologies were synthesized 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. The processing time of an activity is in the order of 10 µs.
This case study showed 1. the suitability of agent-based approaches for large scale sensor networks, for example used for real-time structural health monitoring, and 2. the suitability of the proposed agent modelling and synthesis approach for single System-on-Chip microchip-level implementations.

Figure 7. *High-level and gate-level synthesis results for one sensor node (FPGA: Xilinx XC3S100, ASIC: LSI10K standard cell library, 180nm technology)*



Example 1. *Shortened and simplified excerpt of the AAPL specification for agent class Explore*

```
type keys = {ADC,FEATURE,H,MARK}; direction = {..}
signal WAKEUP,TIMEOUT; val RADIUS := 4; ...
agent explore(dir: direction,
             radius: integer[1..16]) =
  var dx,dy:integer[-100..100];
      live:integer[0..15];        ......
  var* s: integer[0..1023];   ......
  activity start =
    dx := 0; dy := 0; h:= 0;
    if dir <> ORIGIN then
      moveto(dir);
      case dir of
        | NORTH -> backdir := SOUTH
        | SOUTH -> ......
    else
      live := MAXLIVE; backdir := ORIGIN
    group := random(integer[0..1023]);
    transition*(move,percept_neighbour);
    out(H,id(self),0); rd(ADC,s0?)
  activity move =
    case dir of
      | NORTH -> backdir := SOUTH; incr(dy)
      | SOUTH -> backdir := NORTH; decr(dy)
      | WEST -> ....
    moveto(dir)
  activity diffuse =
    decr(live); rm(H,id(self),?);
    if live > 0 then
      case backdir of
       | NORTH -> dir :=
           random({SOUTH,EAST,WEST})
       | SOUTH -> ....
    else kill(ME)
  activity reproduce =
    var n:integer;
    decr(live);
    if live > 0 then
      for nextdir in direction do
        if nextdir <> backdir and link?(nextdir) then
          fork(nextdir,radius)
    transition*(reproduce,stay)
  activity percept =  --  Master perception --
    enoughinput := 0; transition*(percept,move);
    for nextdir in direction do
      if nextdir <> backdir and link?(nextdir) then
        incr(enoughinput);  fork(nextdir,radius)
    transition*(percept,diffuse, (h<ETAMIN or
       h > ETAMAX) and enoughinput < 1);
    transition+(percept,reproduce, h>=ETAMIN and
       h <= ETAMAX and enoughinput < 1);
    timer+(TMO,TIMEOUT)
  activity percept_neighbour =
    if not exist?(MARK,group) then
      mark(TMO,MARK,group); enoughinput := 0;
      rd(ADC,s?); out(H,id(self), calc());
      transition*(percept_neighbour,move);
      for nextdir in direction do
        if nextdir <> backdir and inbound(nextdir) and
          link?(nextdir) then
          incr(enoughinput); fork(nextdir,radius)
      transition*(percept_neighbour,goback,
                  enoughinput < 1);
      timer+(TMO,TIMEOUT)
    else
      transition*(percept_neighbour,goback)  end
  activity goback =
    h := 0; try_in(0,H,id(self),h?);
    moveto(backdir);
  activity deliver =
    var v:integer;
    in(H,id(parent),v?); out(H,id(parent),h+v);
    send(id(parent),WAKEUP); kill(ME)
  activity stay =
    rm(H,id(self),?);
     n :=0; try_in(0,FEATURE,n?);
    out(FEATURE,n+1)
```

```
handler WAKEUP =
  decr(enoughinput); try_rd(0,H,id(self),h?);
  if enoughinput < 1 then timer-(TIMEOUT) end
handler TIMEOUT =
  enoughinput := 0; again := true
function calc():integer =
  if abs(s-s0) <= DELTA then return 1
  else return 0
function inbound(nextdir:direction):bool =
 case nextdir of
   | NORTH -> return (dy < RADIUS)
   | SOUTH -> ......
transitions =
  start -> percept; percept -> move;
  move -> percept_neighbour;
  .....
```

5. CONCLUSIONS

A novel **design approach** using mobile agents for reliable distributed and parallel data processing in large scale networks of low-resource nodes was introduced. An agent-orientated programming language *AAPL* provides computational statements and statements for agent creation, inheritance, mobility, interaction, reconfiguration, and information exchange, based on agent behaviour partitioning in an activity graph, which can be directly synthesized to the microchip level by using a high-level synthesis approach. The high-level synthesis tool also enables the synthesis of different processing platforms from a common program source, including standalone hardware and software platforms, as well as simulation models offering functional and behavioural testing.

Agents of the same class share one virtual machine consisting of a reconfigurable pipelined multi-process chain based on the *CSP* model implementing the activities and transitions, offering parallelized agent processing with optimized resource sharing. 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). The migration of an agent to a neighbour node takes place by migrating the data and control state of an agent using message transfers. Two different agent interaction primitives are available: signals carrying data and tuple-space database access with pattern templates.

Reconfiguration of the activity transition network offers agent behaviour adaptation (which can be inherited by children) at runtime and improved resource sharing for parallel agent processing. A case study 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.

6. REFERENCES

S. Bosse, *Intelligent Microchip Networks: An Agent-on-Chip Synthesis Framework for the Design of Smart and Robust Sensor Networks*, Proceedings of the SPIE 2013 Microtechnologies Conference

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.

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

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

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

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, Session EMT 102

Kone, M. T., Shimazu, A., & Nakajima, T. (2000). *The State of the Art in Agent Communication Languages*. Knowledge and Information Systems, 2(3), 259–284. doi:10.1007/PL00013712

M. Ebrahimi, M. Daneshtalab, P. Liljeberg, J. Plosila, H. Tenhunen, *Agent-based on-chip network using efficient selection method*, 2011 IEEEIFIP 19th International Conference on VLSI and SystemonChip (pp. 284-289)

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.

F. G. McCabe, K. L. Clark, *APRIL - Agent Process Interaction Language*, 1995, (M. Wooldridge & N. R. Jennings, Eds.) Intelligent Agents Theories Architectures and Languages LNAI volume 890. Springer-Verlag.

W. Lang, F. Jakobs, E. Tolstosheeva, H. Sturm, A. Ibragimov, A. Kesel, D. Lehmhus, U. Dicke, *From embedded sensors to sensorial materials—The road to function scale integration.*, Sensors and Actuators A: Physical, Volume 171, Issue 1, 2011

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

Y. Meng, *An Agent-based Reconfigurable System-on-Chip Architecture for Real-time Systems*, in Proceeding ICESS '05 Proceedings of the Second International Conference on Embedded Software and Systems, 2005, pp. 166–173.

J.-P. Jamont and M. Occello, *A multiagent method to design hardware/software collaborative systems*, 2008 12th International Conference on Computer Supported Cooperative Work in Design, 2008.

H. Naji, "Creating an adaptive embedded system by applying multi-agent techniques to reconfigurable hardware,” Future Generation Computer Systems, vol. 20, no. 6, pp. 1055–1081, 2004.