Chapter 1

Introduction: Outline and Synopsis

Closing the Gap: Robust Distributed Data Processing in Heterogeneous Networked Environments and Sensor Networks with Self-organizing Multi-Agent Systems

epubli, ISBN 9783746752228 (2018)

1.1 Outline and Introduction

The growing complexity of computer networks and their heterogeneous composition with devices ranging from servers with high computational power and high-resource requirements down to low-resource mobile devices with low computational power demands unified and scalable new data processing paradigms and methodologies.

The Internet-of-Things (IoT) is one major example and use-case rising in the past decade, strongly correlated with Cloud Computing and Big Data concepts, and extending the Internet Cloud domain with distributed autonomous sensor networks consisting of miniaturized low-power smart sensors. These smart sensors, for example, embedded in technical structures, are pushed by new trends emerging in engineering and micro-system applications [LAN11]. Sensor nodes equipped with computation and communication capabilities can be scaled down to the cubic millimetre range (e.g., the Smart Dust Project, [WAR01]), leading to loosely coupled networks of thousands up to millions network nodes. These sensor networks are used for sensorial perception or structural monitoring (load- and health monitoring), deployed, for example, in Cyber-Physical-Systems (CPS), shown in Figure *1.1*, 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. Ambient Intelligence using agents demands for such platforms, too. [VIL14]

Distributed material-embedded sensor networks used in technical structures and systems require new data processing and communication paradigms, supporting fundamental different architectures. Traditionally distributed operating systems (DOS), for example, the Amoeba DOS, were used to connect and compose computers in heterogeneous networks to one virtual machine [BOS06A]. Main fields of application of such sensor networks are Load Monitoring (LM), Structural Health Monitoring (SHM), or Tactile Sensing (TS). 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 the limited service capabilities of material-embedded systems after manufacturing.

Smart and distributed sensing systems are one of the technological cornerstones of the Internet-of-Things, wearable electronic devices, future transportation, environmental monitoring and smart cities.

Today, one of the major challenges of using smart wireless sensors in real deployments is related to energy consumption and guaranteeing adequate lifetime.

Technical aspects of Multi-agent Systems (MAS) and the required Agent Processing Platforms (APP) are rarely addressed in the current scientific work.

epubli, ISBN 9783746752228 (2018)

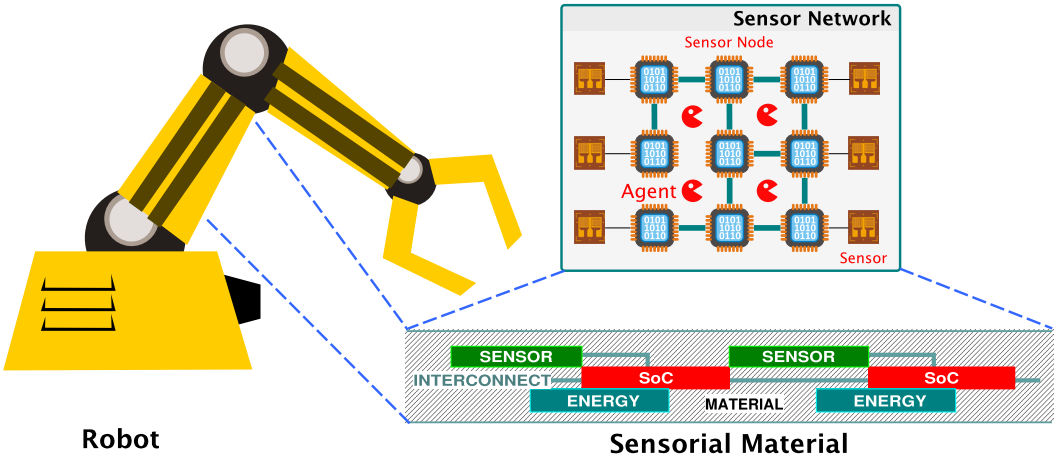

Fig. 1.1 *Sensorial Material: Technical Structure + Material-embedded Sensor Network*

Most agent architectures still assume traditional and powerful underlying data processors and operating systems, which make the processing of mobile agents with mobile processes more difficult.

Mobile Multi-Agent systems represent a well-known parallel and distributed computing paradigm, and can be closely related to the communicating mobile process paradigm [MIL99]. The deployment and programming of MAS means programming of distributed systems, and the programming of distributed systems is a combination of computation and co-ordination. Mobile processes represented by program code and a program state that are capable of migrating in networks between different execution platforms goes back to research work in the 1980s to 1990s and the rising field of Distributed Operating Systems (DOS), for example, considering the prominent Amoeba DOS [MUL90] introducing new communication and process interaction paradigms that consider a heterogeneous computer network environment as one big virtual machine.

Agents are characterised by autonomous and reactive data processing units, which are able to adapt and to be mobile. A MAS can be considered as a society of multiple, coexisting, and interacting agents. Rules are required to co-ordinate and control the behaviour of individual agents, especially concerning interaction. A MAS has a "global" goal that must be achieved by the individual agents by creating organization and societies [FER99], that can address the autonomous and reliable distributed computing that is carried out in Pervasive and Ubiquitous Computing environments.

The deployment of mobile agents in distributed system enables a shift of traditional operating system services like higher-level messaging or resource management to the application level, which can be fully covered by different agents. In pure MAS environments an operating system can be omitted.

Furthermore, the deployment of agents can overcome interface barriers and closes the gap arising between platforms and environments differing considerably in computational and communication capabilities, enabling the integration of sensor networks in large-scale World-Wide-Web (WWW) applications and providing Internet connectivity, shown in Figure *1.2*. This is addressed by using a unified agent-based programming and interaction model, independent of the underlying processing platform. For the following proposed advanced agent processing platform architecture there are suitable hardware, software, and simulation model implementations, which can be interconnected in networks, including a JavaScript implementation with an advanced broker and DOS service suitable for the WEB browser and *node.js* processing. All platform implementations are compatible on the operational and execution level, thus, agents can migrate between these different platform implementations.

The agent mobility crossing different execution platforms, synchronized agent interaction by using tuple-space databases, and global signal propagation aid solving data distribution and synchronisation issues in the design of distributed sensor networks.

In this work Multi-agent systems with state-based mobile agents are used for computing in unreliable networks consisting of generic and embedded computational nodes (e.g., sensor nodes), sometimes consisting only of a single microchip. A novel and unified design approach for reliable distributed and parallel data processing is introduced that can be deployed in embedded systems having static resource constraints and the Internet domain. There is currently still a large gap between agent behaviour models and technological implementations of such resource-constrained processing platforms, which is addressed in this work significantly. Self-organizing Agent Systems (SoS) are one major agent organization structure class, which is considered in this work for solving robustness and distribution problems in general and for solving inherent distributed problems, especially concerning computation with incomplete world models, e.g., mechanical models.

Designing complex distributed systems and processing platforms is a challenge. Therefore, besides unified agent models a unified design framework is required, covering the design of agent processing platforms and agent implementations with different architecture models and different implementation target classes (hardware, software, simulation).

For this purpose, an advanced database driven high-level synthesis approach is used to map the agent behaviour on hardware (Agent-on-Chip processing architecture, AoC), software, and simulation platforms.

The agent behaviour, their interaction, and the mobility behaviour can be fully integrated on the microchip level with a single System-on-Chip (SoC) or System-on-Programmable-Chip (SoPC/FPGA) design.

epubli, ISBN 9783746752228 (2018)

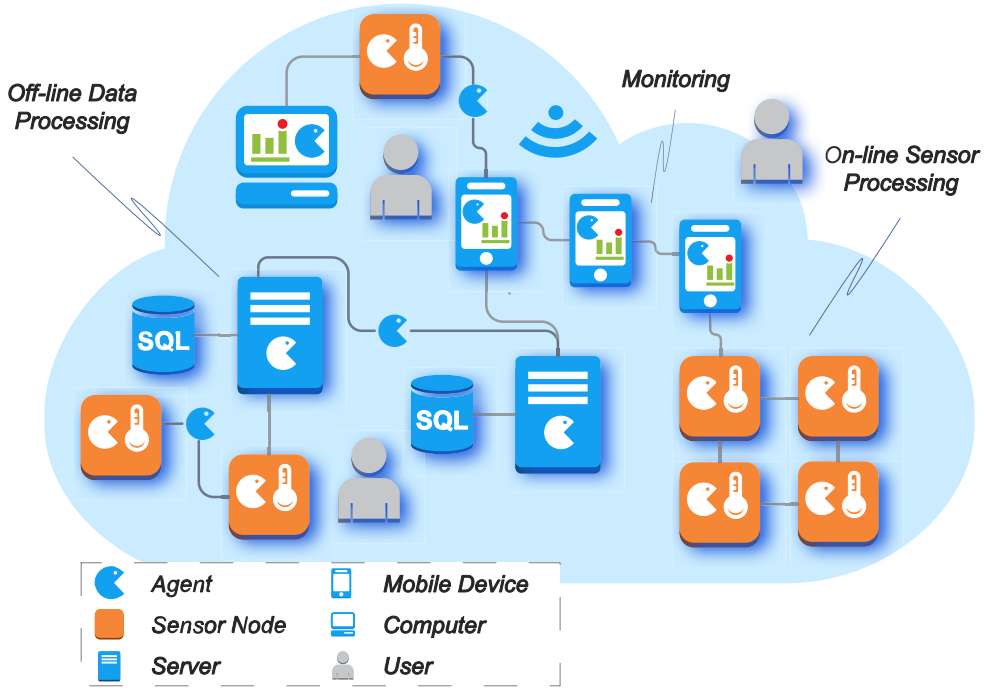

Fig. 1.2 *Deployment of Agents in Sensor Clouds and Internet Applications*

The agent behaviour is modelled with Activity-Transition Graphs (ATG) and a new ATG-based Agent Programming Language (*AAPL*).

The *AAPL*/ATG programming model offers sub- and super agent classification that can be used for run-time adaptation of the agents. The ATG can be modified at run-time by agents, i.e., reconfiguration of transitions and activities composing new ATG behaviour.

The *AAPL* agent behaviour model uses tuple-space and signal communication for the "social" coupling of agents. The replication model bases on forking of loosely bounded parent-child agent groups or instantiation of unbounded MAS.

Three different agent processing platform (APP) classes were investigated and compared differing in resource complexity, computational latency, and flexibility. The first one (*PCSP*) is the *static* pipelined finite-state machine based architecture offering resource and computational latency optimization with predictable real-time capabilities. The second one (*PAVM*) is the *dynamic* program code based architecture offering run-time programmability and a high degree of computational independence of the agents from the platform (that

epubli, ISBN 9783746752228 (2018)

are only loosely coupled). The third one (*JAM*) is an agent platform entirely programmed in JavaScript.

Traditional LM/SHM/TS algorithms like inverse numerical approaches, supervised machine learning, correlation analysis, and pattern recognition, are characterised by a high computational complexity and high memory requirements. Usually these high-level computations are performed off-line (outside the network and not in real-time).

Originally software and multi-agent systems are executed on computers with high computational power and memory capacity, shown in Figure *1.3*. The integration of computing and agents in technical structures or devices requires the downscaling of algorithms and methodologies towards distributed processing networks with low-resource platforms.

The technical miniaturization of data processing nodes leads to a decrease in the computational power that can be compensated only by using efficient parallel and distributed data processing approaches and platforms. One example is the Smart Dust Mote [WAR01], integrating a full sensor node including energy harvesting and optical communication in a cube smaller than $10mm^3$.

It can be shown that agent-based computing can be used to partition these computations in off-line and on-line (in network and real-time) parts resulting in an increased overall system efficiency (performance and energy demands) and a unified programming interface between off- and on-line parts.

The agent model is also capable of providing a programming model for distributed heterogeneous systems crossing different network boundaries. The deployment of MAS in heterogeneous environments is often addressed on the organizational layer, e.g. in [JAY07]. Multi-agent systems are used to enable a paradigm shift from traditionally continuous-data-stream based to event-driven sensor data processing, resulting in increased robustness, performance, and efficiency.

Event-based sensor data processing and self-organizing systems reduce the communication and processing complexity significantly without a loss of Quality-of-Service (QoS), which can be vital in low-resource networks.

Autonomy oriented computation, respectively self-organizing MAS with directed diffusion, replication, exploration, and voting behaviour are used to implement model-free sensor-to-information mapping, suitable for information extraction in sensor networks and LM/SHM/TS applications based on pattern recognition and data-centric algorithms. Smart learning agents based on decision trees are used to distribute and deliver information in unreliable and changing sensor networks.

The run-time behaviour and the requirements of computational, communication, and energy resources in different MAS are analysed using simulation and real-time monitoring techniques in a technical demonstrator.

epubli, ISBN 9783746752228 (2018)

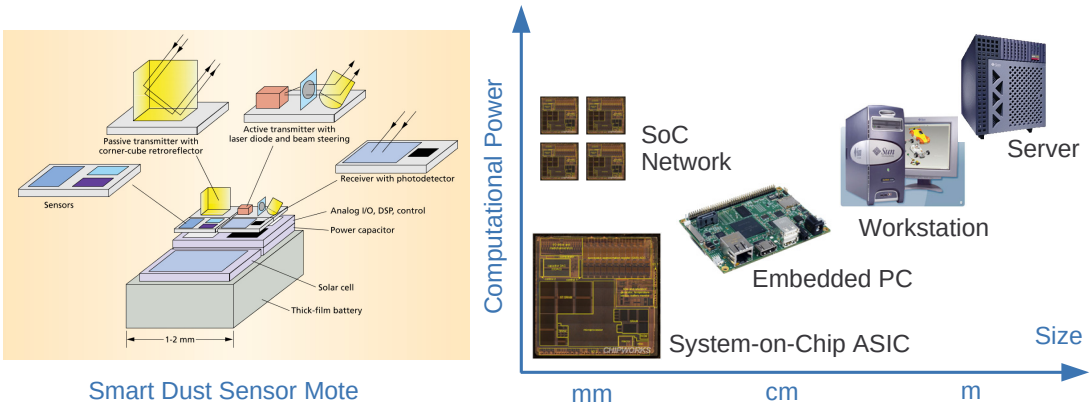

Fig. 1.3 *(Left) Smart Dust Mote* [WAR01] *(Right) Scaling and Miniaturization of Data Processing: from centralised to distributed parallel systems*

Load monitoring based on supervised machine learning and inverse numeric computation are two different use cases for the proposed MAS data processing and off/on-line partitioning approach. The sensor data preprocessing is event-based and uses SoS to detect regions of interest (local significant change of sensors caused by a change of load).

Additionally, negotiated energy management (EM) by using a self-organizing MAS is deployed in self-powering sensor networks. Energy management in autonomous low-power and self- or semi-self-supplying sensor networks is a vital part of robustness, and is the third application case evaluated in this work.

All the different modules and parts considered and handled in this work contributing to the design of intelligent sensing, aggregation, and application systems have a close relationship, as illustrated in Figure *1.4*., that are primarily:

- Distributed Sensor Networks
- Agent Process Platforms
- Agent Programming Languages
- Multi-agent Systems
- Self-organizing and self-adapting systems
- Distributed (agent-based) Machine Learning

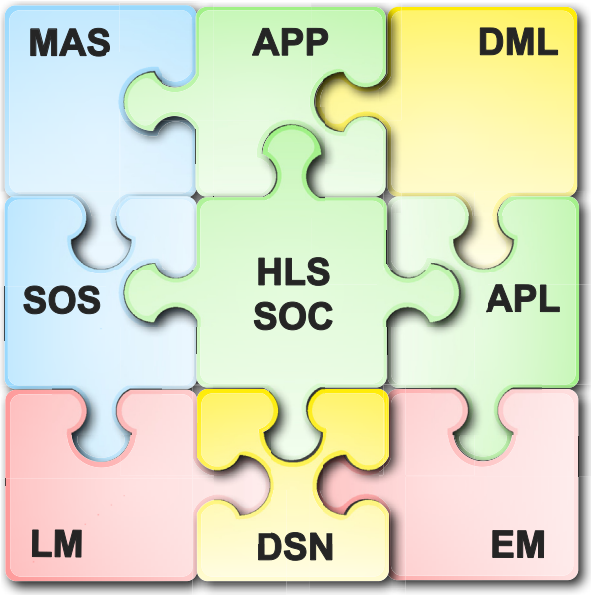

Fig. 1.4 *All modules of this work contributing to the design of intelligent sensing systems: Multi-Agent Systems (MAS), Agent Processing Platforms (APP), System-on-Chip Design (SOC) and High-level Synthesis (HLS), Agent Programming Model and Language (APL), Self-organizing Systems (SOS), Load Monitoring (LM), Energy Management (EM), Distributed Sensor Networks (DSN), Distributed Machine Learning (DML)*

1.2 Data Processing in Sensor Networks with Multi-Agent Systems

1.2.1 Distributed Micro-scale Data Processing in Materials

Guided by the Moore law, the data processing capabilities of single microchips increased dramatically in the last decades based on a transistor density increase, leading to complex System-on-Chip (SoC) designs with more than 100 million transistors. On one hand this leads to a growing gap between transistor density and suitable design tools. On the other hand, such complex SoC circuits enable stand-alone data processing and computing, originally performed by large computer and servers. Therefore, scaling of algorithms to microchip level is one major challenge in the design of future computing environments, including distributed and parallel computation. Data processing migrates from generic computers to technical devices, used in our daily life, i.e., mobile devices. The principal goal of a generic computer is just to perform

data processing only, but a technical device has to perform a specific job, usually under energy and size constraints, and uses data processing only as a tool in the background invisible to the user to achieve the expected goal. New production technologies like printed electronics and the integration of thinned silicon or printed electronics in sheets using system-in-foil processes [VAN14][STE12] introduce a new shift of data processing into a material, e.g., used for perception in robotics, shown already in Figure *1.1*, or structural monitoring. This final step introduces new challenges for the design of suitable data processing architectures satisfying the energy and resource constraints, requires new programming paradigms for distributed programming in environments with a high risk of failure, suitable communication interfaces and paradigms for heterogeneous networks, and finally a rigorous selection and scaling of suitable algorithms for sensor processing applications.

To reduce the impact of material-embedded sensorial systems on mechanical structure properties, single microchip sensor nodes (in mm^3 scale) are preferred [BOS14A]. Real-time constraints require parallel data processing usually not provided by microcontrollers. Hence, with increasing miniaturization and node density, new decentralised network and robust data processing architectures are required, exposed in the next sections.

1.2.2 Multi-Agent Systems

Multi-Agent systems (MAS) can be used for a decentralised and self-organizing approach of data processing in a distributed system like a sensor network [GUI11] enabling the mapping of local sensor data to condensed global information, for example based on pattern recognition [ZHA08][LIU01]. Multi-Agent systems can be used to decompose complex tasks in simpler co-operative agents. MAS-based data processing approaches can aid the material-integration of Structural-Health-Monitoring applications, with agent processing platforms scaled to microchip level that offer material-integrated real-time sensor processing. Agent mobility crossing different execution platforms in mesh-like networks and agent interaction by using, e.g., tuple-space databases and global signal propagation aid solving data distribution and synchronization issues in the design of distributed sensor networks. The deployment of mobile agents enables a shift from traditional operating system services like higher-level messaging or resource management to the application level, which can be fully covered by different agents. Pure MAS environments do not require an operating system, in consequence this can be omitted.

In [GUI11], the agent-based architecture considers sensors as devices used by an upper layer of controller agents. Agents are organised according to roles related to the different aspects to integrate, mainly sensor management, communication and data processing. This organization isolates and uncou-

ples the data management from the changing network, while encouraging a reuse of solutions.

The successful deployment of mobile Multi-Agent Systems in sensor networks was reported in [TYN05] and [MUL08]. But the sensor nodes, called motes and consisting of generic microcontrollers, had still extended physical dimensions beyond the microchip scale (about 50 cm^3 volume), and there was no unified agent processing model supporting heterogeneous networks, i.e., connecting sensor networks with traditional computer networks and the Internet.

There are actually six major issues related to the scaling of traditional software-based multi-agents systems to microchip level implementations, low-power sensor networks, and heterogeneous networks [BOS14B][BOS13A]:

1. Scaling to limited static processing, storage, and communication resources;
2. Real-time processing capabilities;
3. Robustness in the presence of unreliable communication, platform, and processing failures;
4. Suitable simplified agent processing architectures and platforms offering hardware designs with optimized resource sharing and efficient parallel agent execution, also efficient and embedded software platforms, and finally simulation models;
5. Unified agent behaviour and processing models suitable for heterogeneous environments with a high degree of diversity;
6. A unified high-level synthesis design approach covering the design of MAS on specification, programming, communication, and platform level including SoC hardware designs.

1.2.3 Heterogeneous Environments

Usually sensor networks are a part of and connected to a larger heterogeneous computational network [GUI11]. 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. The adaptive and learning behaviour of MAS, central to the agent model, can aid to overcome technical unreliability and limitations [SAN08].

Distributed material-integrated Sensor Networks (DSN) require increased reliability and robustness of the entire heterogeneous environment in the presence of node, sensor, link, data processing, and communication failures. First considerations of heterogeneous network environments and the deployment of MAS were presented in [BOS15A]. It can be shown that the agent

behaviour model and the agent processing platforms investigated in this work are suitable to overcome barriers in heterogeneous networks, offering connectivity of sensor networks with the Internet-of-Things, and finally the WWW.

Current work primarily addresses organizational aspects of MAS in heterogeneous environments using existing agent platforms (e.g., *eHermes* in [JAY07]), rather aspects concerning the platforms that are investigated in this work.

1.3 The Agent Behaviour Model

The agent behaviour model presented and used in this work is associated to the reactive and procedural model class with state-based reactive agents. Such a reactive agent is specified by its state, composed of the values of all data variables and the control state denoting the current activity, and a reasoning engine, implementing the behaviour and actions performed by the agent. Details can be found in Chapter *2* referring to [BOS14B], [BOS14E], and [BOS13A].

Agents record information about an environment state $e \in E$ and history $h: e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow ..$, and so on. Let $I = S \times D$ be the set of all internal states of the agent consisting of the set of control states S related to activities and internal data D. An agent's decision-making process is based on this information. There is a perception function *see* mapping environment states to perceptions, a function *next* mapping internal states and perceptions $p \in Per$ on internal states (state transition), and the action-selection function $action$ that maps internal states on actions $a \in Act$.

Actions performed by agents that are part of their behaviour modify the environment, which is seen by the agent, thus, the agent is part of the environment. Learning agents can improve their performance to solve a given task if they analyse the effect of their action on the environment. After an action was performed the agent gets a feedback in form of a reward $r(t) = r(e_t, a_t)$. There are strategies $\pi: E \rightarrow A$ that map environment states on actions. The goal of learning is to find optimal strategies π^* that is a subset of π. The strategies can be used to modify the agent behaviour. Rewarded behaviour learning was addressed in [JUN12], for example, based on Q-learning.

This reactive behaviour can be summarised with the following operational semantics:

- The programming model is basically related to procedural data processing with activities computing and changing private and global data.
- Transitions between activities represent the progress and the external visible change of the control state of an agent. Transitions can be conditional depending on the evaluation of agent data.

- Body variables of an agent are private data only visible for this specific agent. The data content of body variables is mobile and be inherited by forked child agents.
- There are agent parameters that are initialised during agent creation to distinguish agents instantiated from the same behaviour class.
- Global data is exchanged and coordinated by using a tuple database with synchronised and atomic read, test, remove, and write operations.
- Agents can migrate between different physical and spatially distinguished execution platforms by preserving and transferring the control and data state of the agent.
- The agent behaviour can be either implemented directly by the processing platform (application specific and static platform class), or can be implemented with program code executed by a generic agent processing platform (dynamic platform class).
- Agents can be created at run-time, regardless of the platform class. Agents can inherit the control and data state from parent agents (forking behaviour), enabling bounded parent-child agent groups. The agents can be parametrized during the instantiation, enabling the distinction and variation of agents.
- Agents can synchronise and communicate peer-to-peer by using signals, which can be delivered to remote agent processing nodes, too. This communication feature is primarily used by parent-child agent groups that know from each other.

Definition: There is a Multi-Agent System 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). The node connectivity may be dynamic and changing over time. 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. Each agent class is specified by the tuple* $AC = \langle A,T,F,S,H,V \rangle$*. A is the set of activities (graph nodes), T is the set of transitions connecting activities (relations, graph edges), F is the set of computational functions, S is the set of signals, H is the set of signal handlers, and V is the set of body variables used by the agent class.*

epubli, ISBN 9783746752228 (2018)

Therefore, the agent behaviour and the action on the environment are encapsulated in agent classes, with activities representing the control state of the agent reasoning engine, and conditional transitions connecting and enabling activities. Activities provide a procedural agent processing by a sequential execution of imperative data processing and control statements. The agents can be instantiated from a specific class at run-time. A Multi-Agent System is composed of different agent classes that enables the factorisation of an overall global task in sub-tasks, with the objective of decomposing the resolution of a large problem into agents in that they communicate and cooperate with one other. The ATG agent behaviour model is discussed in Chapter *2*.

1.3.1 Dynamic Activity-Transition Graphs

The behaviour of an activity-based agent is characterised by an agent state, which is changed by activities. Activities perform perception, plan actions, and execute actions modifying the control and data state of the agent. Activities and transitions between activities are represented by an Activity-Transition Graph (ATG). Transitions start activities based on the evaluation of agent data. An ATG relies on an event-based model.

Usually agents are used to decompose complex tasks in simpler ones. Agents can change their behaviour based on learning and environmental changes, or by executing a particular sub-task with only a sub-set of the original agent behaviour. The case studies in Chapter *9* show examples for Self-organizing Multi-Agent Systems (SoMAS) with different agent behaviour and goals forked from one original root agent class. An ATG describes the complete agent behaviour. Any sub-graph and part of the ATG can be assigned to a sub-class behaviour of an agent. Therefore, modifying the set of activities $\boldsymbol{A}$ and transitions $\boldsymbol{T}$ of the original ATG introduces several sub-behaviours implementing algorithms to satisfy a diversity of different goals. The reconfiguration of activities $\boldsymbol{A'}=\{\boldsymbol{A}_1 \subseteq \boldsymbol{A}, \boldsymbol{A}_2 \subseteq \boldsymbol{A}, ..\}$ derived from the original set $\boldsymbol{A}$ and the modification or reconfiguration of transitions $\boldsymbol{T'}=\{\boldsymbol{T}_1,\boldsymbol{T}_2,..\}$ enable dynamic ATGs (DATG) and agent sub-classing at run-time.

Learning agents can improve their performance to solve a given task if they analyse the effect of their action on the environment by getting a feedback in form of a reward. This reward learning approach can be applied to the DATG model modifying the agent behaviour at run-time based on the learning results to find optimal strategies π^* that are subsets of the original strategy set π.

1.3.2 The Agent Interaction with a Tuple Space

A tuple space is basically a shared memory database used for synchronised data exchange among a collection of individual agents, which was already

proposed in [QIN10] as a suitable MAS interaction paradigm. One well-known tuple-space organization and co-ordination paradigm is Linda [GEL85]. The tuple-space organization and access model offers generative communication, i.e., data objects can be stored in a space by processes with a lifetime beyond the end of the generating process. The scope and visibility of a tuple space database can be unlimited and visible and distributed in the whole network, or limited to a local scope, e.g., the network node level. A tuple space provides abstraction from the underlying platform architecture, i.e., it is a *virtualized resource*, and offers a high degree of platform independence, that is vital in a heterogeneous network environment.

In [CHU02] a *Java* based mobile processing and co-ordination platform was introduced, offering agent co-ordination by tuple spaces, too. An *XML* model approach is used to encapsulate and exchange tuples that introduces superfluous communication overhead.

A tuple database stores a set of n-ary (arity of n) data tuples, $t_n=(v_1, v_2,.., v_n)$ that are n-dimensional values. The tuple space is organised and partitioned in sets of n-ary tuple sets $\nabla=\{TS_1,TS_2,..,TS_n\}$. A tuple is identified by its dimension and the data type signature. Commonly the first data element of a tuple is treated as a key. The agents can add new tuples (using the output operation) and read or remove tuples (using the input operations) based on tuple pattern templates and pattern matching, $p_n=(v_1, p_2?, .., v_j,.., p_i?,., v_n)$, a n-dimensional tuple template with actual and formal parameters. Formal parameters are wild-card place-holders, which are replaced with values from a matching tuple. The input operations can suspend the agent processing if there is actually no matching tuple available. After a matching tuple was stored, blocked agents are resumed and can continue processing. Therefore, tuple databases provide inter-agent synchronisation, too. This tuple-space approach can be used to build distributed data structures and the atomicity of tuple operations provides data structure locking. The tuple spaces represent the knowledge of agents. The tuple-space co-ordination and communication model is discussed in Chapter *3*.

1.4 Agent Programming Languages and AAPL

There are multiple existing agent programming languages and processing architectures, like *APRIL* [MCC95] providing tuple-space like agent communication, and widely used FIPA *ACL*, and *KQGML* [KON00] focusing on high-level knowledge representations and exchange by speech acts, or model-driven engineering (e.g. *INGENIAS*, [SAN08]). But required resource and processing control, independence of the processing architecture, and a unified approach for the deployment of MAS in strong heterogeneous networks are missing, which is addressed in this work with *AAPL*, *the Activity-Transition-Graph based Agent Programming Language*. This language enables the design of heteroge-

neous MAS with a unified agent interaction paradigm on different processing platforms. The processing of *AAPL* agents is independent of any particular technology or architecture.

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. Though the imperative programming model of *AAPL* is quite simple and closer to a traditional programming language 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 [BOS14B].

Commonly used agent behaviour models base on the Procedural-Reasoning-System and Belief-Desire-Intention (*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*.

On one hand the *AAPL* approach is simple enough to enable hardware design synthesis, on the other hand powerful enough to model the agent behaviour of complex distributed systems, which is demonstrated in several case studies in Chapters *9* and *14*.

The agent programming model is close to the previously introduced Dynamic Activity-Transition Graphs (DATG). An agent is composed of activities performing actions, e.g., the modification of internal data and the external environment (data). Based on the state of the agent, which is given primarily by the values of the body variables of the agent, there are transitions between activities, representing a change in the control state of the agent. The Activity-based Agent Programming Language *AAPL* offers a textual representation of the DATG model. Agents specified with *AAPL* can be synthesized to and be processed on both the application-specific and the programmable agent processing platforms, which is summarized in Figure *1.5*, having a common source for the development of heterogeneous environments.

The *AAPL programming language* (introduced in [BOS14A], extended in [BOS14B]) offers statements for parametrizable agent instantiation, like the parametrizable creation of new agents and the forking of child agents inheriting the control and data state of the parent agent (creating bounded parent-child agents group).

Multi-Agent and group interaction is offered with synchronised Linda-like tuple database space access and peer-to-peer interaction using signal propagation carrying simple data delivered to and processed by signal handlers of agents. Signals, which can carry additional scalar data values, can be used for

epubli, ISBN 9783746752228 (2018)

local (in terms of node scope) and global (in terms of network scope) agent interaction. In contrast to the anonymous tuple-space interaction, signals are directly addressed to a specific agent or a group of agents.

Agent mobility is offered by a simple move operation that migrates the agent to a node in the neighbourhood, assuming mesh-like networks, not necessarily with static topologies and connectivity.

Agent classes are defined by their parameters, variables, activities, and transition definitions reflecting the ATG model. Optionally an agent class can define additional functions for computation and signal handlers. There are several statements for ATG transformations and composition. Transitions and activities can be added, removed, or changed at run-time.

There are different levels of organization in MAS [FER99], which can be related to the AAPL behaviour and interaction model in the following ways:

1. The micro-social level characterized by a tight bounding of agents, supported by *AAPL* parent-child groups with forking of the control and data state.
2. The group level characterized by a composition of larger structures and organizations, supported by *AAPL* mobility and tuple-space coordination.
3. The global society level characterized by the dynamics of numerous agents with specialisation of some agents, supported by *AAPL* mobility and ATG/class composition.

The *AAPL* agent behaviour model and the programming language are discussed in Chapter *2*. There is a *AAPL* short notation used throughout this book, which is summarized in Appendix *A.3.1*.

The Belief-Desire-Intention (BDI) architecture is a well-known agent behaviour and interaction model capable of rational behaviour and practical reasoning [RAO95] [WOO99], in contrast to, for example, procedural reasoning architectures (like PRS). Although the *AAPL* model is closer to the procedural processing model, it can be related to the BDI architecture, and BDI agents can be implemented with *AAPL*, discussed in Section *2.12*.

The relationship of the *AAPL* model with the mobile process model and process algebra, i.e., the π-Calculus, is discussed in Section *2.8*, using a modified distributed Π-Calculus (based on the original *$\alpha D\Pi$*-Calculus [HEN07]), moving the view of point from spatially located agents in a distributed inter-connected system to one unified concurrent system with dynamic virtual communication channels.

epubli, ISBN 9783746752228 (2018)

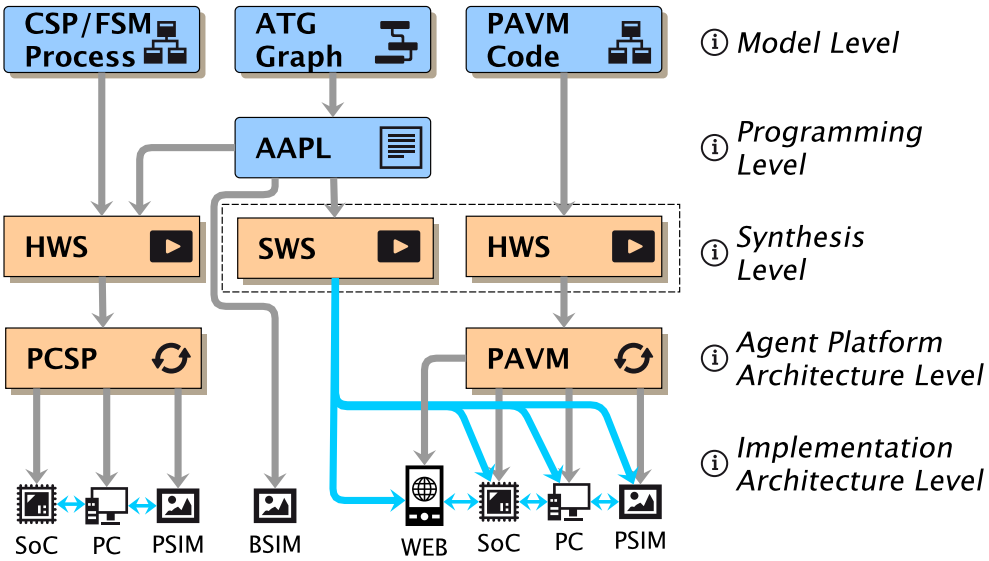

Fig. 1.5 *Different low-resource agent processing platform architectures and implementations, but a common agent behaviour model and programming source. (PCSP: Pipelined Communicating Sequential Processes - static, PAVM: Pipelined Agent Forth Virtual Machine - dynamic, ATG: Activity-Transition Graph, AAPL: Agent Programming Language, HWS: Hardware Synthesis, SWS: Software Synthesis, PSIM: Platform Simulation, BSIM: Behavioural Agent Simulation, WEB: JavaScript, SoC: System-on-Chip)*

1.5 Agent Processing Platforms

Microchip level implementations of Multi-Agent Systems were originally proposed for low level tasks, for example in [EBR11] using agents to negotiate network resources and for high-level tasks using agents to support human beings in ambient-intelligence environments [CAM12]. 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 [MEN05], 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 ([MEN05][NAJ04]). A first attempt can be found in the *PANGEA*

platform [VIL14] that embeds agents in resource limited devices. But it does not provide the concept of mobile processes thoroughly addressed in this work. The *PANGEA* platform addresses different levels of organization aspects of MAS accurately, which is considered here as a higher level task not directly supported by the platform itself, though the investigated new platform architectures are closely related to the *AAPL* behaviour model, and hence provide the necessary atoms for the agent organization. In [ZHO08] the virtual machine *CAVM* is presented, targeting the agent-oriented *CAOPLE* programming language for distributed systems. The *CAVM* is a high-level language virtual machine addressing the caste centric agent programming paradigm based on the encapsulation of state, action, behaviour rules, and environmental description, which is comparable to *AAPL*. One important feature of the *CAVM* is the separation of computation and communication, a requirement for mobile code platforms. This is addressed by the agent platforms investigated in this work, too. But the *CAVM* approach is not scalable to low-.resource embedded platforms and relies on traditional microprocessor architectures; the code is inefficiently encoded with an *XML* textual representation.

Two different microchip-scalable agent processing platforms were investigated and evaluated:

1. A non-programmable (application-specific) [BOS14A][BOS14B][BOS13A];
2. A programmable (application-independent, generic) agent processing architecture [BOS14C][BOS14D][BOS12B].

Both agent processing platform architectures can be implemented in hardware, software, and simulation with a multi-model synthesis, shown in Figure *1.5*. The synthesis process is discussed in Section *1.7* and Chapter *12*.

The non-programmable architecture bases on finite-state machines implementing the agent behaviour purely application-specific with a multi-process token-based pipeline architecture, based on a reconfigurable Petri-Net model. The programmable architecture uses a stack machine supporting a zero-operand instruction set with code-morphing capabilities. They differ in resource complexity (gate count/chip area, memory, and power demands), computational latency (the parallelization degree), and run-time flexibility (optimization contrary to programmability). Both approaches focus on hardware implementations, but offer optimized software implementations and simulation models, too.

All implementations of one platform architecture class can be deployed and connected in a heterogeneous network environment. They are compatible on operational and interface level. That means agents can migrate between different platform implementations and different host environments. Inter-class compatibility can be achieved by using transformation and wrapper modules.

epubli, ISBN 9783746752228 (2018)

1.5.1 The Non-Programmable Application-specific Agent Processing Platform PCSP

The non-programmable approach implements multi-agent systems with mobile activity-based agents capable of 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 can be efficiently integrated on the microchip using a configurable pipelined multi-process architecture with token-based agent processing, offering temporal and spatial fine-grained parallelization and optimized resource sharing. A reconfiguration mechanism of the agent processing system offers activity graph changes at run-time. Agent activities are mapped on communicating sequential processes (finite-state machine model) with one input port and one or several output ports, and transitions are mapped to queues connecting activity processes.

Token-based Processing. Token-based execution of programs was in last decades originally introduced for data-flow computing, e.g., the tagged token-based data-flow processor from MIT [ARV90], offering a MIMD processor class. Here the agent processing architecture is related to Petri Net token processing, with tokens passed between activity processes (states) using queues (representing transitions). A token is associated with a specific agent (i.e., agent identifier). The token based agent processing model enables advanced timed Petri-Net analysis. Mobility of agents between network nodes is provided by transferring the state of an agent (mainly consisting of the private data space and the control state giving an activity entry point after migration) encapsulated in messages.

Beside hardware implementations, efficient software implementations and simulation models with equal functional behaviour can be derived from the same source model. Hardware and software platforms are compatible on execution level and can be directly connected in heterogeneous networks. The agent behaviour, reasoning, interaction, and mobility are modelled and specified with the programming language *AAPL* related to the ATG model. A database driven high-level synthesis approach is used to map the agent behavioural model to multi-agent systems on hardware, software, and simulation platforms. The hardware synthesis is based on the *ConPro* HLS tool [BOS11A][BOS10A], which offers the SoC and SoPC design from programming level with communicating sequential processes and Inter-process communication by using shared synchronization objects like semaphores. Competition and concurrence is resolved by using atomic guarded access of shared resources, resulting in rigorous serialization of parallel access. 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. Access of the tuple-space introduces inter-agent synchronization, which is based on tuple-pattern matching.

Real-time Processing. Full data path parallelism provided by the hardware implementation [BOS11A] and the fine-grained activity-based partitioning of the computation can offer real-time capable low-latency agent data processing within estimated or constrained time bounds by analysing the activities at design time with a closed loop design flow. Though resource sharing always limits a hard real-time constraint satisfaction at run-time, the presented approach can relax time-bounded computation significantly compared with traditional software and operating system controlled systems offering only coarse-grained task scheduling. Agent tokens can be assigned an additional time bound in which the agent must be processed. An activity process transition queue can be extended with a scheduler selecting agents using time line priorities. Timed Petri-Nets can be used to analyse the temporal behaviour of agents and the estimation of time bounds depending on the activity computations [BOS14B] in conjunction with platform simulation [BOS14E].

1.5.2 The Programmable Agent Processing Platform PAVM

There is only few related work regarding programmable agent platforms supporting mobile processes. In [CHU02] a *Java* based VM approach is used to implement and process mobile agents (the *JMAP* platform), with some concepts addressed in this work, too. But *Java* programs and the *Java* VM (though stack based) rely on a random access memory model and memory references, which is not well suited for register-based data processing platforms exploiting parallelism on data and control path level that are applied to microchip level SoC architectures. The *JMAP* platform is implemented on the top of the *Java* VM and incorporates an agent, coordination, and security manager, basically part of both agent processing platforms (application-specific and programmable) that are presented in this work. In the *JMAP* platform the agents are related to threads, in the *PCSP* and *PAVM* platforms with tokens executed by shared processing blocks.

Application-specific platforms offers the best performance and lowest resource requirements, but lack of flexibility. For this reason, a second programmable platform was investigated. The requirements for this agent processing platform can be summarized to:

1. Being suitable for microchip level (SoC) implementations;
2. Supporting a stand-alone platform without any operating system;
3. Performing efficient parallel processing of numerous different agents;

epubli, ISBN 9783746752228 (2018)

4. Being scalable regarding the number of agents processed concurrently;
5. Providing the capability to create, modify, and migrate agents at run-time.

Migration of agents requires the transfer of the data and control state of the agent between different virtual machines (at different node locations). To simplify this operation, the agent behaviour based on the activity-transition graph model is implemented with program code. This embeds the (private) agent data as well as the activities, the transition network, and the current control state [BOS14B][BOS15B], based on early work in [BOS12B] introducing code morphing for agent modification at run-time. The program code can be considered as a self-contained execution unit. The execution of the program by a virtual machine (VM) is handled by a task. The program instruction set consists of zero-operand instructions, mainly operating on the stacks. The VM platform and the supported machine instruction set implement traditional operating system services, too, offering a full operational and autonomous platform with a hybrid RISC and CISC architecture approach. No boot code is required at start-up time. The hardware implementation of the platform is capable to operate after few clock cycles, which can be vital in autonomous sensor nodes with local energy supply from energy harvesting. An ASIC technology platform requires about 500-1000 k gates (16 bit word size), and can be realized with a single SoC design.

The virtual machine executing tasks is based on a traditional *FORTH* processor architecture and an extended zero-operand word instruction set (*αFORTH*), discussed in Chapter 7. Most instructions directly operate on a data and a control stack. A code segment stores the program code with embedded data. There is no separate data segment. Temporary data is stored only on the stacks. The program is mainly organized by a composition of words (functions). A word is executed by transferring the program control to the entry point in the code segment; arguments and computation results are passed only by the stack(s). There are multiple virtual machines with each attached to (private) stack and code segments. There is one global code segment with a word dictionary storing global available functions and code templates that can be accessed by all programs. A dictionary is used to resolve code addresses of global functions and templates. This multi-segment architecture ensures high-speed program execution. The local code segment can be implemented with (asynchronous) dual-port RAM (the other side is accessed by the agent manager, discussed below), the stacks with simple single-port RAM. The global code segment requires a Mutual Exclusion scheduler to resolve competition by different VMs.

The program code frame of an agent consists basically of four parts:

1. A lookup table and embedded agent body variable definitions;
2. Word definitions defining agent activities and signal handlers (procedures without arguments and return values) and generic functions;
3. Bootstrap instructions that are responsible to set up the agent in a new environment (e.g., after migration or at first run);
4. The transition table calling activity words (defined above) and branching to succeeding activity transition rows depending on the evaluation of conditional computations with private data (variables).

The transition table section can be modified by the agent using special instructions.

Commonly the number of agent tasks N_A executed on a node is much larger than the number of available virtual machines N_V. Thus, efficient and well-balanced multi-task scheduling is required to get proper response times of individual agents. To provide fine-grained granularity of task scheduling, a token based pipelined task processing architecture was chosen. A task of an agent program is assigned to a token holding the task identifier of the agent program to be executed. The token is stored in a queue and consumed by the virtual machine from the queue. After a (top-level) word was executed, leaving an empty data and return stack, the token is either passed back to the processing queue or to another queue (e.g., of the agent manager).

The agents can reconfigure at run-time by modifying their program code using code morphing techniques provided by special instruction of the VM. Self-reconfiguration is mainly acting on the transition table of an agent by enabling or disabling of activity transitions. Code morphing is also used to save the state of an agent during process suspending (due blocked IO) or upon migration. Furthermore, recomposing of new agents with existing agent activity words and functions is a suitable tool to create sub-classed or newly composed agents.

1.5.3 JAVM: The JavaScript PAVM

The previous subsection outlined the programmable stand-alone platform that can be implemented in hardware, software, and simulation models. This platform architecture was primarily designed for peer-to-peer networks. Recent work extended the deployment of this platform to the Internet domain by encapsulating the platform VM in a Distributed Co-ordination Layer (*DCL*) supporting capability-based RPC, distributed file and directory services [BOS15A]. The directory service enables the grouping of nodes in domains published in directories. The agent processing platform and the DCL were implemented in *JavaScript* (JS) that can be executed in any WEB browser or a

dedicated *node.js* VM. A broker service is used to enable bidirectional RPC communication for client-side-only applications, i.e., executed by a WEB browser that cannot expose an IP service. This extended JS platform can be connected to sensor networks and is compatible to the stand-alone *PAVM* platform on operational level enabling migration of agents between dedicated sensing networks and the Internet, discussed later in the context of cloud-based manufacturing (see Section *1.12*).

1.5.4 Comparison of the Agent Processing Platforms

Table *1.1* provides a taxonomy and shows the comparison of the characteristics and the advantages/disadvantages for two hardware capable agent processing platform architectures and their different implementations (*PCSP*: Pipelined Communicating Sequential Processes, *PAVM*: Pipelined Agent Forth Virtual Machine).

Both platforms differ with respect to their agent processing approach, but bases basically on the same agent programming model. Among hardware implementations there are distinct software implementations providing operational compatibility.

1.5.5 JAM: The JavaScript Agent Machine

Table *1.2* gives a taxonomy of another pure software-based JavaScript Agent Platform *JAM*, introduced and used in this book, too, which is discussed below.

In contrast to the *JAVM* platform that implements the *PAVM* in *JavaScript* still executing machine *FORTH* code, the recent development of the *JAM* platform enables the execution of mobile agents entirely programmed in *JavaScript*. The *JAM* platform enables the deployment of large-scale MAS in strong heterogeneous environments, primarily targeting the Internet and Clouds. *JAM* is capable of handling thousands of agents per node, supporting virtualization and resource management. Depending on the used underlying *JS* VM, agent processes can be executed with nearly native code speed. *JAM* provides Machine Learning as a service that can be used by agents. Different algorithms can be selected by agents.

JAM can be executed on any *JavaScript* VM engine, including browser engines (Mozilla's *SpiderMonkey*), or from command line using node.js (based on V8) or jxcore (V8 or *SpiderMonkey*), and finally the low-resource engine *JVM* based on *jerryscript*. In contrast to V8-based engines that compile *JS* at runtime to native machine code (Just-in-time compiler), *JVM* is a Bytecode engine that compiles *JS* directly to Bytecode from a parsed AST. *JAM* is available also as an embeddable library (*JAMLIB*) that can be integrated in any JS/HTML application including mobile APPs.

epubli, ISBN 9783746752228 (2018)

	Programmable PAVM	***Non-programmable PCSP***
Approach	• Program code based approach • Zero-operand instruction format • Stack memory centric data processing model • Platform is generic • Code embeds instructions, configuration (control state), and data • Migration: code transfer	• Application-specific approach • Platform is application-specific • Activities of the ATG are mapped to processes • Token-based agent processing • Migration: data and control state transfer
Hardware Implem.	• Optimized Multi Stack Machine • Each stack processor has a local code segment and two stacks shared by all agents. There is no data segment! • Single SoC Design • Multiprocessor architecture with distributed and global shared code memory • Multi-FSM RTL hardware architecture • Automatic Token-based agent process scheduling and processing • Code morphing capability to modify agent behaviour and program code (ATG modification) • Data- and code word sizes can be parametrized	• Pipelined Communicating Processes Architecture composition implementing ATG and token-based agent processing • Single SoC Design • Optimized resource sharing - only one PCSP for each agent class implementation required • Activity process replication for enhanced parallel agent processing • For each agent class there is one PCSP with attached data memory (agent data). • Single SoC Design • LUT configuration matrix approach for ATG reconfiguration

Tab. 1.1 *Comparison of two hardware- and software capable platform architectures and their implementations (PAVM PCSP)*

	Programmable PAVM	***Non-programmable PCSP***
Software Implem.	• Multi-Threading or Multi-Process software architecture • Inter-process-communication: queues • Software model independent from programming language • VM sources for various programming languages: C, ML, JavaScript, ... • Can be embedded in existing software • Platform memory requirements: Code 1MB, Data 0.5-10MB • JavaScript platform offers WEB application integration	• Multi-Threading software architecture • Optimization: Functional composition and implementation of ATG behaviour instead PCSP • Inter-process-communication: queues • Software model independent from programming language • Source code for various programming languages: C, ML, ... • Can be embedded in existing software • Platform memory requirements: Code 1-100kB, Data 10-500kB
Simulation Model	• Agent-based Platform simulation • Generic simulation model - can execute machine code directly • Processor components and managers are simulated with agents	• Agent-based platform simulation • Application-specific simulation model • ATG activity processes are simulated with agents

Tab. 1.1 *Comparison of two hardware- and software capable platform architectures and their implementations (PAVM PCSP)*

The agent behaviour is modelled exactly according to the Activity-Transition Graph (ATG) and *AAPL* model. The ATG is entirely programmed in *JavaScript* (*AgentJS*). *JAM* agents are executed in a sand-boxed environment isolating agents from each other and the computer system by the *JAM* Agent Input/Output System (AIOS). *JAM* agents are mobile, i.e., a snapshot of an agent process containing the entire data and control state including the behaviour program, can migrate to another *JAM* platform. *JAM* provides a broad variety of connectivity, including the Distributed Organization layer (DOS).

epubli, ISBN 9783746752228 (2018)

	Programmable JAM
Approach	• Mobile JavaScript text-code based approach • JAM Platform is generic and can be executed on any JS VM engine, e.g., node.js, jxcore (V8 + Spidermonkey), jxcore+, jvm, WEB Browser • Agent program code embeds instructions (activities), configuration (transitions, control state), and data (body variables) • Migration: text transfer in JSON+ format
Hardware Implem.	• None
Software Implem.	• Single-Threaded software architecture with asynchronous I/O • Advanced agent process scheduling • Software model: JavaScript • Platform: JavaScript • Can be embedded in existing software • JavaScript platform offers WEB (Browser) application integration • Platform memory requirements: JS code: 1MB, Data 5-50MB (depends on JS VM)
Simulation Model	• No platform simulation required by using JAM directly and SeJAM (Simulation and visualization environment on the top of JAM) • Simulation model: JavaScript • Implemented with node-webkit (nw.js) • Hardware-in-the-loop support: SeJAM can be integrated in real world networks

Tab. 1.2 *A pure software-based agent platform: JAM*

The *JAM* platform introduces a new security concept by assigning a role level to agents. A role grants or denies access to *AIOS* operations, e.g., the creation of new agents or the sending of signals. Higher role levels are only granted trustful agents. Agents operate with limited resources (CPU time, memory, ..), which can be negotiated between agents and the platform.

epubli, ISBN 9783746752228 (2018)

1.6 AAPL MAS and Mobile Processes: The Π-Calculus

The *AAPL* mobility, agent instantiation, and the tuple-space agent interaction have a close relation to mobile processes and the Π-Calculus with channel-based communication, which is discussed in Chapter *3*. Furthermore, it can be shown that the unified Bigraph representation [MIL09] is a suitable model to cover heterogeneous network environments populated with a diversity of agents [BOS15A].

The π-Calculus introduced by Milner [MIL99] and the extended asynchronous distributed π-Calculus introduced by Hennessy [HEN07] (aDΠ) are common formal languages for concurrent and distributed systems, suitable for studying the behaviour and reaction of distributed and concurrent systems including dynamic changes caused by mobility. In the following subsection the relationship of the *AAPL*/DATG behaviour and interaction model with the Π-Calculus is pointed out, moving the view of point from spatially located agents in a distributed interconnected system to one unified concurrent system with dynamic virtual communication channels. The Π-Calculus used here extends the π-Calculus with the concept of (structural) domains, locations, resources associated with domains and locations, and migration of processes, close to the MAS paradigm, introduced in Section *2.8*.

1.7 High-Level Synthesis of Agents and Agent Platforms

Designing and implementing MAS for multiple significantly different platforms deployed in heterogeneous network environments is still a superior challenge. A unified High-level synthesis framework should enable the design and simulation of MAS for such a heterogeneous processing environment including and most important hardware SoC designs (discussed in the next section) using application-specific digital logic meeting the goal of miniaturization and material-integration.

Currently the agent architectures are addressed primarily, and rather the synthesis approaches themselves. One example for a high-level agent framework is *SPARK* [MOR04], addressing the need for the synthesis of MAS that can be deployed in real-world applications. This framework relies on the high-level BDI agent model. The framework is limited to software implementations, and primarily falls back on existing *Java* platforms. Another example can be found in [BEL01]. It proposes the today widely used Java Agent Development Environment (*JADE*), based on FIPA *ACL* compliant agent communication and *Java* VM platforms. This agent and synthesis framework is still limited to generic computers connected in IP-based networks. It seems only chemistry deals with agent synthesis and processes [KED12].

In this work, an important key feature is given by the common agent programming model *AAPL* that can be used for the different platform approaches, effecting both the programming and synthesis architecture mod-

els. The central parts of the synthesis framework are the Agent Behaviour and Agent Platform compiler [BOS14A][BOS15B], discussed in Chapter *12* and that are closely coupled.

Application-specific agent synthesis embeds the agent behaviour directly in the platform, therefore the *AAPL* MAS behaviour is entirely synthesized to platform processing blocks, whereas the application-independent agent synthesis flow creates the platform (virtual machine) and the agent behaviour unit (program) separately, similar to a traditional hardware-software co-design.

The Synthesis Development Kit (*SynDK*) was investigated to handle the synthesis of such complex systems and to address different platform architectures and implementations. A graph-based virtual database (*VDB*) driven hardware and software synthesis approach should overcome limitations in traditional compiler designs and enables common synthesis from a set of source programming models and languages to a set of destination models and languages like hardware behaviour models with parsers translating text to graph structured database content and printers creating text from database content, shown in Figure *1.6*. The Agent Behaviour, Agent Platform, and the Hardware Compiler *ConPro* are coupled by the *VDB*. The database is used to store parsed syntax trees (i.e., the input data in AST structure) of various input languages, symbol tables used by various compiler blocks, generated output data (output languages), and compiler block script code that are interpreted by the Virtual Database Programming Language Machine (*VPM*).

The *VDB* organizes database elements (so-called i-nodes) with generic graph-like structures, enabling the mapping of any kind of data structure on the database model.

The database element structure is constrained and defined by Structure Type Definitions *STD* with an extended *DTD* model. This unified database approach eases the design of complex multi-stage and multi-language compilers significantly towards Agent Synthesis.

epubli, ISBN 9783746752228 (2018)

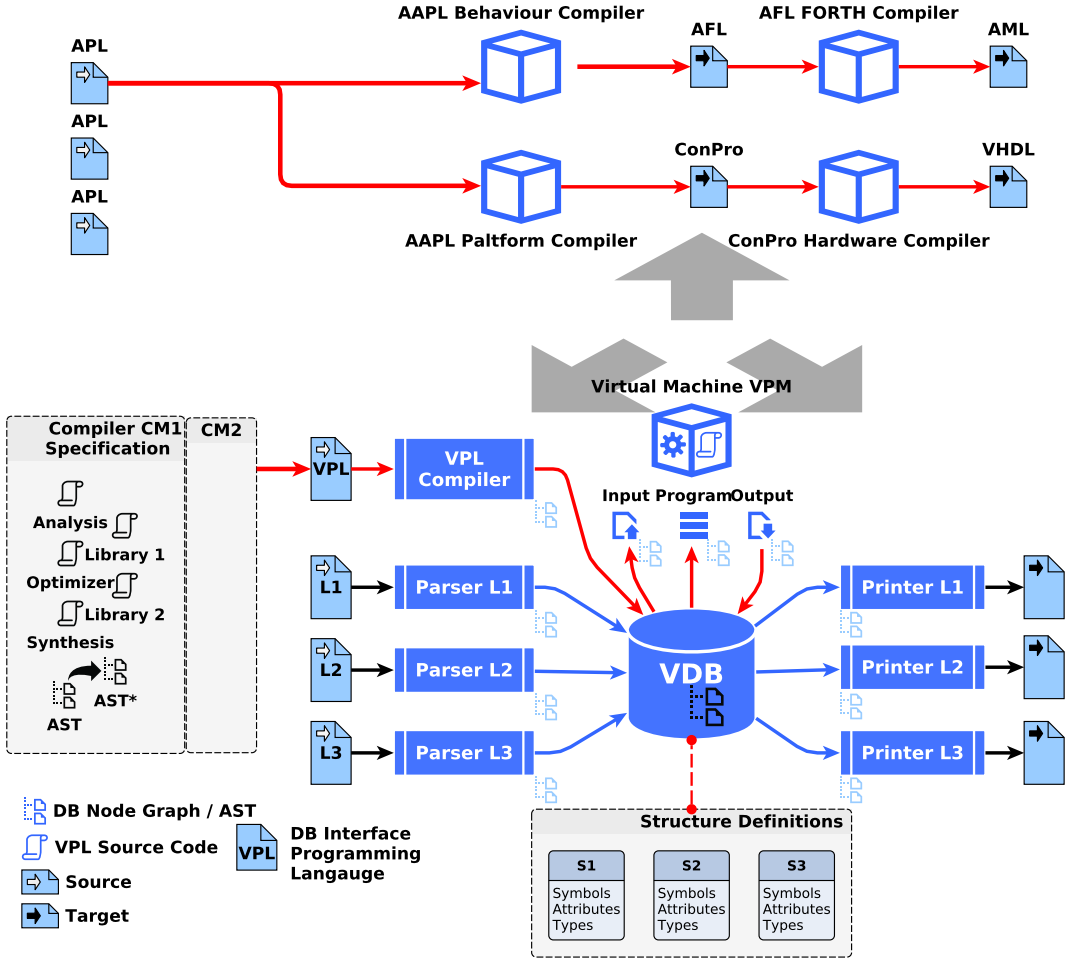

Fig. 1.6 *Heart of the Synthesis Development Kit (SynDK) architecture is the Virtual Database (VDB) used for graph-based structuring of any kind of data involved in compiling and synthesis processes, implementing parser and formatted printer for a set of languages L, and a set of compilers C performing operations on abstract syntax trees AST (analysis, optimization, synthesis). The Agent Behaviour, Agent Platform, and the Hardware Compiler ConPro are coupled by the VDB.*

1.8 High-level Synthesis of SoC Designs

Embedded systems used for control, for example, in Cyber-Physical-Systems, or data processing in sensor networks, perform the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource-constrained manner. System-On-Chip designs are preferred for high miniaturization and low-power applications. Traditionally, program-controlled multi-processor architectures are used to provide the execution platform, but application-specific digital logic circuits gains more importance.

Currently High-level Synthesis (HLS) design flows are closely related to *SystemC* or derivatives of *C* [COU09], well suited for the hardware-software co-design synthesis, but not very well matching the requirements for massive parallel RTL based architectures. Although there are suitable synthesis flows generating application-specific RTL designs, e.g. *Cynthesizer*, they all suffer by the constraints and restrictions of the *C* programming language used to model the behaviour of the data processing system. The *C/SystemC* approach does not provide any higher level of synchronization objects, like mutual exclusion locks, semaphores, or guarded shared objects. Furthermore, the programming paradigm of *C* is memory centric, binding all variables of a program and prevent the extensive exploitation of data path parallelism.

Concurrency has a great impact on system and data processing behaviour concerning latency, data throughput, and power consumption. Stream-based and functional data processing requires fine-grained concurrency (on data path level). However, reactive control systems (for example, communication controller) require coarse-grained concurrency (on control path level).

The ***structural level*** decomposes a SoC into independent sub-modules interacting with each other using centralized or distributed networks and communication protocols, mainly program-controlled multi-processor architectures.

The ***behavioural level*** usually describes the functional behaviour of the full design interacting with the environment. Most applications and data processing are modelled on algorithmic behavioural level using some kind of imperative programming languages.

The *ConPro* High-level Synthesis tool [BOS10A][BOS11A] was designed for the development of complex SoC designs beyond the millions gates, which uses a behavioural multi-process programming language based on the Concurrent Communicating Sequential Processes (CCSP) model, providing an extensive set of inter-process communication and synchronization primitives and guarded atomic actions for shared resource access, with a compiler-based synthesis approach, mapping the algorithmic programming level to Register-Transfer level (concurrent multi-FSM approach) that can be implemented directly with digital logic, shown in Figure *1.7*.

epubli, ISBN 9783746752228 (2018)

Alternatively, software implementations with the same functional and operational behaviour can be synthesized from the same program source, too.

This capability eases the development of heterogeneous data processing environments and networks consisting of single application-specific microchips and generic computers significantly.

The *ConPro* programming model is not fixed on memory-centric data processing architectures, granting a great choice of freedom for the selection of processing architectures and synthesis processes.

The HLS followed by the gate level synthesis targeting different technologies: FPGA and a standard cell library enabling ASIC technology mapping, or alternatively gate-level simulation using an event-based gate-level simulator (based on the *ASIMUT* simulator that is part of the LIP6 Alliance ASIC design tool-kit).

The design of digital circuits with a complexity up to ten million of logic gates can be reached using this advanced CCSP-based programming language model and the HL synthesis tool. Protocol stacks (i.e., [BOS11A]), agent processing platforms [BOS13A][BOS14A], sensors processing nodes [BOS10B], and robot joint controllers [BOS13D] were successfully implemented with this tool.

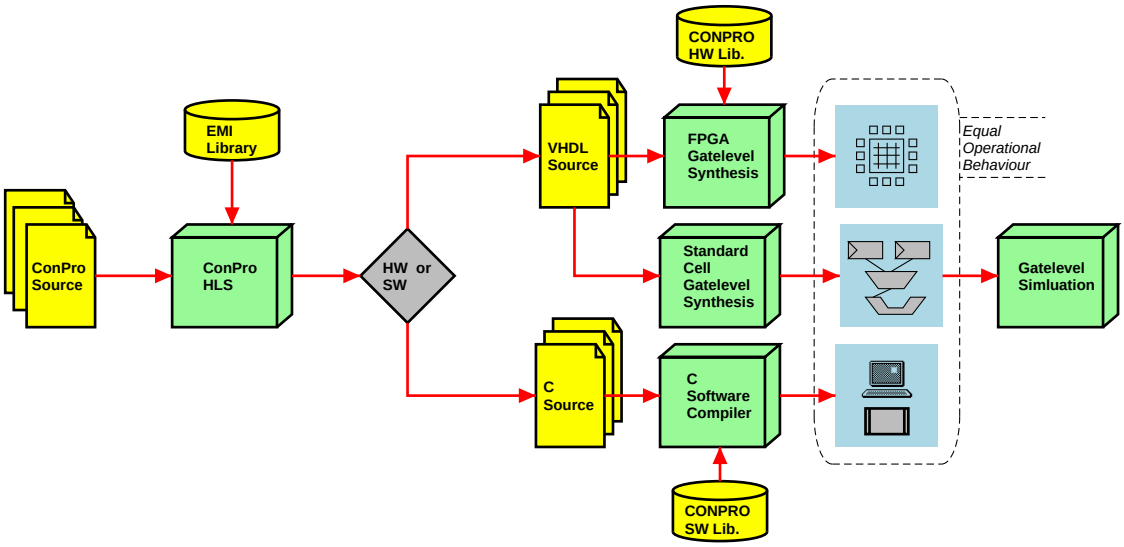

Fig. 1.7 *Complete SoC Design flow using the high-level synthesis framework ConPro that maps the parallel CCSP programming model on SoC-RTL hardware and, alternatively, software targets*

1.9 Simulation Techniques and Framework

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 "Shell for Simulated Agent Systems" (*SeSAm*) simulation framework [KLU09] offers a platform for the modelling, simulation, and visualization of mobile multi-agent systems employed in a two-dimensional world. The behaviour 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).

Simulation aspects of MAS are presented and discussed in Chapter *11*, introducing the textual *SeSAm* Simulation Programming Language *SEM* used to specify the simulation models in textual form (*SeSAm* has only a GUI modeller).

1.9.1 Behavioural Simulation

In behavioural simulation, the *AAPL*/ATG model is mapped basically one-to-one on *SeSAm* agents. But for signal handling a shadow agent is required to handle concurrently incoming signals for the parent agent. Furthermore, the *SeSAM* agent model does not support agent blocking within an activity. Therefore, an *AAPL* activity must be split into computational and IO/event related parts, which may block the agent processing [BOS14B]. Several MAS were simulated, profiled, and evaluated using this simulation level (e.g., event-based and self-organizing sensor processing in [BOS14B][BOS14C]).

1.9.2 Platform Simulation

In contrast to behavioural agent simulation, the platform simulation uses agents to simulate the processing of agents on a fine-grained level on a specific platform architecture, e.g., *PCSP* or *PAVM*. The agent processing platform is simulated with the agent-based *SeSAm* simulation framework. This simulation technique provides the testing and profiling of the proposed processing platform architectures in a distributed network and world environment under different constraints, i.e., resource constraints or connectivity loss. The *PCSP* platform was simulated and evaluated in [BOS14E], and the *PAVM* platform in [BOS15B].

1.9.3 Simulation of Real-world Sensor Networks

The simulation of the operation of entire sensor networks deploying MAS commonly requires real data from the environmental world, which does not exist.

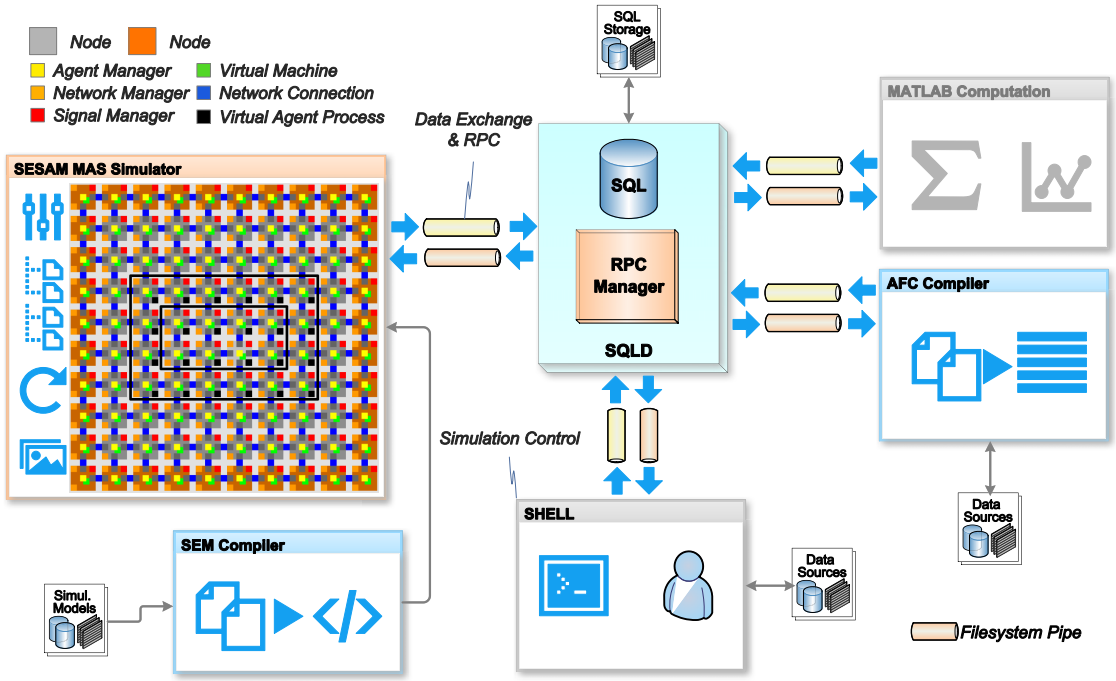

Fig. 1.8 *Simulation Framework with a Database approach (Left: SeSAm, middle: SQL+RPC database server; Right: compiler and additional Numerical/ FEM simulation tools)*

To overcome this limitation, the *SeSAm* agent simulator was embedded in a database centric unified simulation environment. This simulation environment connects the MAS simulator with FEM and numerical computation programs (e.g., *MATLAB*), exchanging data and synchronizing using a SQL database server, which provides an RPC interface for synchronization, too, shown in Figure *1.8*. This approach introduces multi-domain and multi-scale simulation capabilities. Details are discussed in Section *11.3*. Additionally, this approach offers connectivity to real sensor networks.

1.9.4 RTL Simulation

Hardware agent processing platforms based on SoC and RTL designs can be simulated and tested on a behavioural or platform architecture level as explained in the previous sub-sections. A more fine-grained simulation technique for digital logic systems deploys gate-level simulation of pre-synthesized circuits and digital stimuli patterns applied to the circuit for different test cases. The hardware behaviour model (*VHDL*) is synthesized to a standard-cell library target technology resulting in a netlist of standard cell

gates taken from the library, which is simulated with an efficient event-based simulator. The simulator *ASIMUT* from the Alliance VLSI CAD tool framework (details can be found in [ALCAD] and [GRE92]) was extended and used for this purpose. The simulator computes the output signal pattern from an input signal pattern stimuli (commonly generated by a pattern stimuli compiler) based on internal signal changes and gate activity (events). Testing and simulating of complete agent platform hardware implementations, consisting of about 500k-1M equivalent gates [BOS14A], is beyond the capabilities of this approach. Instead, gate-level simulation was used to test components and parts of the circuit, i.e., communication modules or synchronization objects, especially for the development of the *ConPro* HLS framework [BOS11A]. Furthermore, this low-level simulation technique was used to study the power consumption and the algorithmic correlation as an input for AI-based energy management by using an advanced version of *ASIMUT* and the *SiCa* compiler [BOS11B], and further pointed out in Chapter *13*.

1.10 Event-based Sensor Data Processing and Distribution with MAS

Large scale sensor networks with hundreds and thousands of sensor nodes require smart data processing concepts far beyond the traditional centralized approaches. Multi-Agent systems can be used to implement smart and optimized sensor data processing in these distributed sensor networks.

Event-based sensor data distribution and pre-computation with agents reduce communication and overall network activity resulting in reduced energy consumption of single nodes and the entire network.

Different sensor data processing and distribution approaches are used and implemented with agents, leading to a significant decrease of network processing and communication activity and a significant increase of reliability and the Quality of Service (QoS) [BOS14C]:

1. An event-based sensor distribution behaviour is used to deliver sensor information from sensor (source) to computation (sink) nodes.
2. Adaptive path finding (routing) supports agent migration in unreliable networks with missing links or nodes by using a hybrid approach of random and attractive walk behaviour.
3. Self-organizing agent systems with exploration, distribution, replication, and interval voting behaviour based on feature marking are used to identify a region of correlated and stimulated sensors, introduced in the next section. Detected features trigger the events for the above described behaviour.

Simulations of such an event-based sensor processing shows a significant decrease in communication performed by mobile agents that lead to an improved scaling of large networks. The event-triggered agents, carrying the information source, can reach their destinations, the information sinks, with a high success rate in partially destroyed networks due to the adaptive path finding, discussed in Section *9.3*. The event-based adaptive MAS approach introduces robustness and increases the reliability of the entire sensing system [BOS14C], which can be vital and critical in SHM applications.

1.11 Self-organizing Systems and MAS

A common conceptual approach for building adaptive systems involves the design of such systems by using elements that find by themselves the solution to the problem to be solved [GER07]. Mobile Agents that are capable of interacting and of adapting based on perception are well suited for the implementation of Self-organizing Systems (SoS).

Every dynamic and active system can be considered as populated with agents that interact with each other and the agents are characterized by their behaviour and their goals. The behaviour of agents has influence of the future outcome of the behaviour of other agents and their aim to reach their goals or the selection of goals.

An application example implementing a distributed feature detection in an incompletely connected and unreliable mesh-like sensor network using mobile agents demonstrates the suitability of self-organizing MAS for sensor data processing in distributed sensor networks, presented in Section *9.2*., and bases on work published in [BOS13A] and [BOS14C], derived from an original approach proposed by [LIU01] for image processing feature recognition. The goal of the MAS is to find the boundary of extended correlated regions of increased sensor stimuli (compared to the neighbourhood), e.g., in a load monitoring scenario due to mechanical deformation resulting from externally applied load forces. The feature detection is performed by the mobile *exploration agent*, which supports two main different behaviour: diffusion and reproduction. The explorer agent can be composed of the root agent class implementing diffusion and reproduction and an explorer child agent subclass with a reduced behaviour set used for the exploration of the immediate neighbourhood relative to the current position of the explorer agent. The exploration algorithm bases on the divide-and-conquer paradigm with nested parent-child agent groups. 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.

epubli, ISBN 9783746752228 (2018)

1.12 From Embedded Sensing to the Internet-of-Things and Sensor Clouds

Deploying agents in very large scale areas with heterogeneous networks ranging from dedicated sensor networks up to WEB-based applications is a further challenge.

One example for WEB-based sensor processing using agents is the Agent factory micro edition (*AFME*) [MUL07], which is an intelligent agent framework for resource-constrained and mobile devices and is based on a declarative agent programming language, in contrast to the reactive and imperative *AAPL* approach introduced in this work.

Recent work [BOS15A][BOS15C] shows that the *PAVM* agent processing platform is well suited for the implementation in *JavaScript* enabling agent processing in client-side WEB browser applications or by using the *node.js* server-side VM [TIL10]. The *JAVM* implementation is fully operational compatible with the previously described standalone *PAVM* architecture, commonly implemented on microchip level with RTL and SoC architectures.

In the *JAVM* approach network nodes (i.e., programs distributed on the Internet and Intranet) communicate primarily by using the *HTTP* protocol, attractive to be integrated in common computer networks. A broker service is used to establish client-side applications (e.g., WEB browser) as communication endpoints visible in dynamic domains, which enables agent mobility between these applications. Agent code can be executed on any node including WEB browsers. The *JAVM* and broker service approach enables the integration of sensor networks, for example, embedded in technical structures, in generic computer networks. The *JAVM* is encapsulated by a Distributed Co-ordination Layer (*DCL*) that was added to connect application programs on the Internet domain. The *DCL* bases on Object-orientated Remote Procedure Calls derived from the Amoeba DOS [MUL90], and offers distributed file- and directory services. The file service can be used to store agent code, the directory service is used to create and publish virtual domains of nodes, which are required for agent mobility and distribution.

The *JAVM* architecture is discussed in Section *7.8*, and a significant use-case scenario and the deployment in manufacturing processes is discussed in Section *14.5*. This MAS architecture is suitable for additive and adaptive manufacturing based on a closed-loop sensor processing approach with data mining concepts combined with Internet-of-thing architectures.

Agents are already deployed successfully for scheduling tasks in production and manufacturing processes [CAR00B], and newer trends poses the suitability of distributed agent-based systems for the control of manufacturing processes [LEI15], facing not only manufacturing, but maintenance, evolvable assembly systems, quality control, and energy management aspects, finally introducing the paradigm of industrial agents meeting the requirements of

epubli, ISBN 9783746752228 (2018)

modern industrial applications. The MAS paradigm offers a unified data processing and communication model suitable to be employed in the design, the manufacturing, logistics, and the products themselves.

The scalability of complex industrial applications using such large-scale cloud-based and wide-area distributed networks deals with systems deploying thousands up to a million agents. But the majority of current laboratory prototypes of MAS deal with less than 1000 agents [LEI15]. Currently, many traditional processing platforms cannot yet handle big numbers with the robustness and efficiency required by industry [MAR05][PEC08]. In the past decade the capabilities and the scalability of agent-based systems have increased substantially, especially addressing efficient processing of mobile agents. The JAVM platform can contribute to the solving of the scalability problem in such environments. Cloud-based design and manufacturing is composed of knowledge management, collaborative design, and distributed manufacturing, incorporating finally the products in the cloud-based design and manufacturing process.

1.13 Use-Case: Structural Monitoring with MAS

Figure *1.9* poses a conceptual overview for the monitoring of mechanical structures using machine learning and inverse numerical approaches. The MAS is the evident part of the distributed data processing in sensor networks, which are suitable for reliable structural load monitoring in heterogeneous network environments. Different algorithmic LM approaches (with and without mechanical model) are used to derive the load (deformation) information about structures from raw sensor data. The computation is temporally and spatially partitioned into on-line and off-line parts by using one MAS offering preprocessing, pre-computation, and distribution.

Different case studies presented in Chapter *14* address the above introduced sensor and information processing approach, giving examples of the deployment of MAS and Artificial Intelligence methods by providing perception either of the physical load of the environment acting on a technical structure, e.g., a robot manipulator and its interaction with obstacles and objects, or by providing the internal load of a structure, i.e., the deformation of structure caused by loads. It will be demonstrated that Machine Learning do not requiring any mechanical model and inverse numerical algorithms using FEM models are suitable to calculate the load acting on technical structures from noisy and unreliable low-dimensional sensor input.

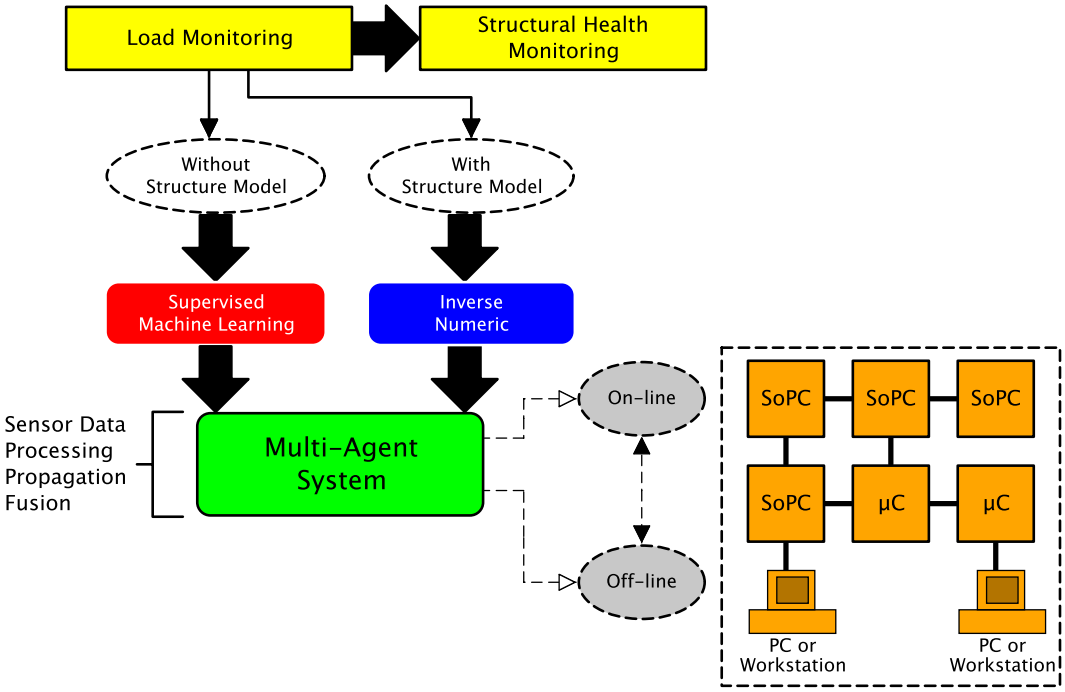

Fig. 1.9 *Structure Monitoring with MAS in heterogeneous environments implementing different algorithms.*

1.13.1 Machine Learning and MAS

Intelligent behaviour of robot manipulators become important in unknown and changing environments. Emergent behaviour of a machine arises intelligence from the interactions of robots with its environment. Sensorial materials equipped with networks of embedded miniaturized smart sensors can support this behaviour.

Environmental perception can be provided by some elastic material covering extended surfaces of robotic structures, e.g., intersection elements or body covers. An integrated autonomous decentralized sensor network can be capable providing perception similar to an electronic skin. Each sensor network is connected to strain gauge sensors mounted on a flexible polymer surface, delivering spatial resolved information about external forces applied to the robot arm, required for example for obstacle avoidance or for manipulation of objects.

The first attempt of a perceptive material was a flat rubber sheet (based on work in [BOS11C]), which was finally bent around a technical structure, presented in Section *14.2*.

Each autonomous sensor node provides communication, data processing, and energy management implemented on microchip level.

Commonly a high number of strain gauge sensors are used to satisfy a high spatial resolution. The chosen approach uses advanced Artificial Intelligence and Machine Learning methods for the mapping of only a few non-calibrated and non-long-term stable noisy strain sensor signals to spatially resolved load information and a decentralized data processing approach to improve robustness. Robustness in the sensor network is provided by

1. Autonomy of sensor nodes;
2. Smart adaptive communication to overcome link failures and to reflect changes in network topology;
3. Intelligent adaptive algorithms.

It is well-known that robust co-operation and distributed data processing is achieved by using Mobile Agent systems [WAN03].

As outlined in this book, the agent behaviour and co-operation can be implemented on microchip level.

The central aim is to derive useful information constrained by limited computational power and noisy sensor signals unable to be captured by a complete system model. Machine Learning (ML) methods are capable of mapping an initially unknown n-dimensional set of input signals on an m-dimensional output set of information like the position and strength of applied forces [MIT97].

Without any interaction and material model Machine Learning requires a training phase. Additional material models and FEM simulation can reduce or avoid the training phase [BOS11C].

The training set contains recorded load positions, masses and classification results for different load cases determined via sensor measurement.

1.13.2 Hybrid approach of MAS and Inverse Numeric Methods

In [BOS14C] an initial guess for a hybrid approach with on-line sensor processing in distributed sensor networks with MAS and off-line inverse numerical methods used for load computation was proposed, profiled, and validated.

The basic sensing environment is shown in Figure *1.10*. One of the major challenges in SHM and LM is the derivation of meaningful information from the sensor input. The sensor output of an SHM or LM system reflects the lowest level of information. Beside technical aspects of sensor integration the main issue in those applications is the derivation of a mapping function $F_m(S)$, which basically maps the raw sensor data input S, a n-dimensional vector consisting of n sensor values, on the desired information I, a m-dimensional result vector.

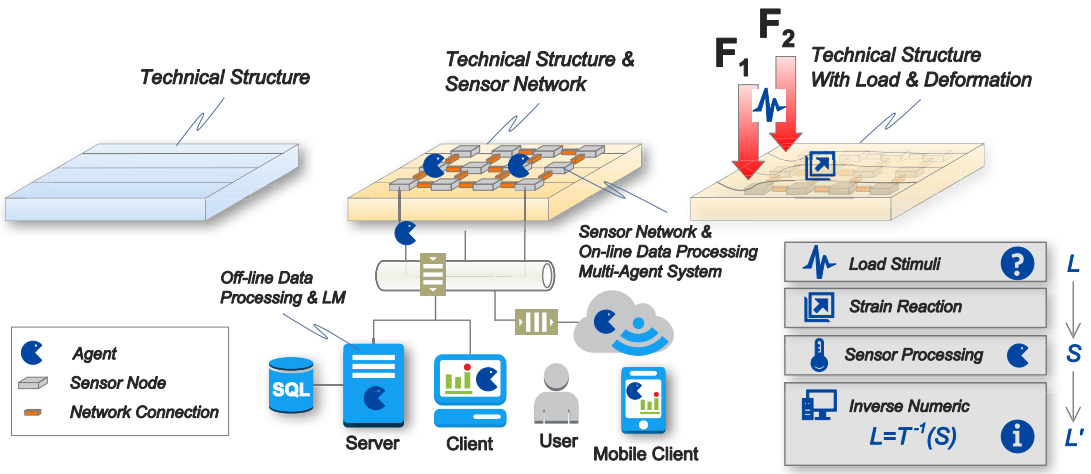

Fig. 1.10 *Initially unknown external forces acting on a mechanical structure lead to a deformation of the material resulting from the internal forces. A material-integrated active sensor network integrating sensors, electronics, data processing, and communication, together with mobile agents can be used to monitor relevant sensor changes with an advanced event-based information delivery behaviour. Inverse numerical methods can compute finally the material response. The unknown system response for externally applied load L is measured by the strain sensor stimuli response S, finally computing an approximation of the response L'.*

It can be shown [BOS14C][BOS14F] that a hybrid data processing approach for material-integrated SHM and LM systems by using self-organizing and event-driven mobile multi-agent system (MAS) is suitable for this sensing system class, with agent processing platforms scaled to microchip level (*PAVM/PCSP*) which offer material-integrated real-time sensor systems, and inverse numerical methods providing the spatially resolved load information from a set of sensors embedded in the technical structure. Inverse numerical approaches usually require a large amount of computational power and storage resources, unsuitable for resource constrained sensor node implementations. Instead, off-line computation is performed, with on-line sensor processing by the agent system. Commonly off-line computation operates on a continuous data stream requested by the off-line processing system delivering sensor data continuously in fixed acquisition intervals, resulting in high communication and computational costs. Here the sensor preprocessing MAS delivers sensor data event-based if a change of the load was detected (feature extraction), reducing network activity and energy consumption of the entire system significantly. The basic SoMAS behaviour is introduced in Chapter 9.

1.14 Use-Case: Smart Energy Management with MAS and AI

Algorithmic Selection. In contrast to various other energy management approaches targeting algorithms and architectures with high computational effort, *smart energy management* can be performed spatially at run-time by applying a dynamic selection from a set of different (implemented) algorithms classified by their demand of computational power, the "accuracy" and Quality of Service (QoS), and temporally by varying data processing rates. The smart energy management can be implemented with decision trees, based on QoS and energy constraints. It can be shown [BOS11B] that the power and energy consumption of an application-specific SoC design strongly depend on the computational complexity of the used algorithms. *Power analysis* using simulation techniques on digital gate-level provides input for the algorithmic selection at run-time of the system leading to a closed-loop design flow. The run-time energy management, which can be based on reinforcement learning, switches between different computational algorithms, varying the power consumption and simultaneous the quality of service. Additionally, the signal-flow approach enables power management by varying the signal flow rate. Details can be found in Section *13.1*.

Energy Distribution with MAS. Self-powered sensor nodes collect energy from local sources, but can be supplied additionally by external energy sources. Nodes in a sensor network can use communication links to transfer energy, for example, optical links are capable of transferring energy using Laser or LE diodes in conjunction with photo diodes on the destination side, with a data signal modulated on an energy supply signal.

A decentralized sensor network architecture is assumed with nodes supplied by:

1. Energy collected from a local source;
2. Energy collected from neighbour nodes using smart energy management (SEM).

Nodes are arranged in a two-dimensional grid with connections to their four direct neighbours. Each node can store collected energy and distribute energy to neighbour nodes, for example, using electrical or optical links [KED06].

Each autonomous node provides communication, data processing, and energy management. There is a focus on single System-On-Chip (SoC) design satisfying low-power and high miniaturization requirements.

Energy management is performed for:

1. The control of local energy consumption;
2. The collection and distribution of energy by using the data links to transfer energy [BOS12E].

Typically, energy management is performed by a central controller in that a program is implemented [LAG10], with limited fault robustness and the requirement of a well-known environment world model for energy sources, sinks, and storage. Energy management in a network involves the transfer of energy. Recent work shows the benefit and suitability of multi-agent systems used for energy management [LAG10].

Having the technical capability to transfer energy between nodes, it is possible to use active messaging for the energy transfer from good nodes having enough energy towards bad nodes, requiring energy [BOS12E][BOS12A]. An agent can be sent by a bad node to explore and exploit the near neighbourhood. The agent examines sensor nodes during path travel or passing a region of interest (perception) and decides to send agents holding additional energy back to the original requesting node (action). Additionally, a sensor node is represented by a node agent, too. The node and the energy management agents must negotiate the energy request.

Help agents with simple exploration and exploitation behaviour are suitable to meet the goal of a regular energy distribution and a significant reduction of bad nodes unable to contribute sensor information, and additional distribute agents can distribute energy proactive. The multi-agent implementation offers a distributed management service rather than a centralized approach commonly used. The simple agent behaviour can be easily implemented in digital logic hardware (using the application-specific platform approach). Details can be found in Section *13.2*.

epubli, ISBN 9783746752228 (2018)

1.15 Novelty and Summary

- Multi-domain *reactivity* in heterogeneous networks provided by mobile state-based agents capable of *reconfiguration* of their behaviour (activity-transition graph modification) for each particular agent at run-time, including the inheritance of (modified) agent behaviour, which increases the *reliability* and *autonomy* of multi-agent systems.
- Agents are mobile by transferring the state only or the program code embedding the state (data+control) of the agents.
- Agents can *reconfigure* their behaviour engine based on learning or sub-classification at run-time. Sub-classification reducing the agent behaviour can save resources at run-time and migration.
- Agent *interaction* and coordination offered by a tuple-space database and global signal propagation aid solving data distribution and synchronization issues in distributed systems design (machine-to-machine communication), whereby tuple spaces represent the knowledge of single and multiple agents (agent belief).
- One common agent behaviour model and *programming language AAPL* suitable for different *processing architectures* and platforms enables the synthesis of stand-alone parallel hardware implementations, alternatively stand-alone software implementations, WEB embedded *JavaScript* applications with an optional Distributed Co-ordination Layer, and behavioural simulation models, enabling the design and test of large-scale heterogeneous systems.
- There are two different *agent processing platform architectures* implementing the agent behaviour either application-specific or programmable with code. With respect to each class, hardware, software, WEB, and simulation platforms are compatible on interface and operational level enabling agents migration between these different platform implementations.
- *AAPL* provides powerful statements for computation, agent control, agent interaction, and mobility with static and limited resources.
- Optimized token-based pipelined agent processing architectures are suitable for parallel hardware platform designs on Register-Transfer Level with a SoC architecture offering optimized computational resources and speed.
- The processing platform is a *stand-alone* unit that does not require any Operating System and boot code for initialization, leading to a low start-up time latency, well suited for self-powered devices. All agent-

epubli, ISBN 9783746752228 (2018)

specific actions like migration or communication are implemented on VM machine level.

- Improved scaling in large network applications compared with full or semi-centralized and pure message based processing architectures.

epubli, ISBN 9783746752228 (2018)

1.16 Structure of the Book

Chap. 2. Agent Behaviour and Programming Model: Modelling and Programming Multi-Agent Systems with *AAPL*

Chap. 3. Agent Communication: Agent-Agent and Agent-World Communication Models and the relationship to *AAPL*

Chap. 4. Distributed Sensor Networks: Deployment of Multi-Agent Systems in Distributed Sensor Networks

Chap. 5. Concurrent Communicating Sequential Processes: The extended Parallel Data Processing Model used to implement the agent processing platforms.

Chap. 6. *PCSP*: The Application-specific Agent Platform. Agent processing platform based on a Parallel Pipelined Communicating Sequential Processes Architecture

Chap. 7. *PAVM*: The Programmable Agent Platform. Agent Processing Platform based on a parallel and token-based Agent *FORTH* Virtual Machine Architecture; *JAVM*: The JavaScript *PAVM* Platform

Chap. 8. JAM: The JavaScript Agent Machine- Agent Processing Platform based on JavaScript

Chap. 9. Self-Organizing Multi-Agent Systems: Event-based Sensor Processing, Feature Recognition, and Energy Management with self-organizing Multi-Agent Systems for Sensor Processing

Chap. 10. Machine Learning: Distributed and incremental techniques with MAS

Chap. 11. Simulation: Simulation of the *AAPL* Agent Behaviour Model, the Agent Processing Platforms, and the Simulation of Sensor Networks

Chap. 12. Synthesis: From Programming Level to Hardware and Software Implementations using a Unified Database driven Synthesis Framework

Chap. 13. Use-Case Energy Management: Low-power Design and Smart Energy Management with MAS

Chap. 14. Use-Cases Environmental Perception, Load Monitoring, and Cloud-based Manufacturing: Load Monitoring with Multi-Agent Systems, Machine Learning, and Inverse Numeric for Structural Monitoring and Environmental Perception, Deployment of agents in Cloud-based environments.

epubli, ISBN 9783746752228 (2018)

Chap. 15. Material-Integrated Sensing Systems: Integration of Sensing and Multi-Agent Systems in Materials and Technical Structures: Technological Aspects and Integration Technologies

epubli, ISBN 9783746752228 (2018)