Design and Simulation of a Low-Resource Processing Platform for Mobile Multi-Agent Systems in Distributed Heterogeneous Networks

Stefan Bosse

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

Abstract: The design and simulation of an agent processing platform suitable for distributed computing in heterogeneous sensor networks consisting of low-resource nodes is presented, providing a unique distributed programming model and enhanced robustness of the entire heterogeneous environment in the presence of node, sensor, link, data processing, and communication failures. In this work multi-agent systems with mobile activity-based agents are used for sensor data processing in unreliable mesh-like networks of nodes, consisting of a single microchip with limited low computational resources, which can be integrated into materials and technical structures. The agent behaviour, based on an activity-transition graph model, the interaction, and mobility can be efficiently integrated on the microchip using a configurable pipelined multi-process architecture based on the Petri-Net model and token-based processing. A new sub-state partitioning of activities simplifies and optimizes the processing platform significantly. Additionally, software implementations and simulation models with equal functional behaviour can be derived from the same program source. Hardware, software, and simulation platforms can be directly connected in heterogeneous networks. Agent interaction and communication is provided by a simple tuple-space database. A reconfiguration mechanism of the agent processing system offers activity graph changes at run-time. The suitability of the agent processing platform in large scale networks is demonstrated by using agent-based simulation of the platform architecture at process level with hundreds of nodes.

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

1. Introduction

Trends are recently emerging in engineering and micro-system applications such as the development of sensorial materials [11] show a growing demand for distributed autonomous sensor networks of miniaturized low-power smart sensors embedded in technical structures [4]. These sensor networks are used for sensorial perception or structural health monitoring, employed, for example in Cyber-Physical-Systems (*CPS*), and perform the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource-constrained manner under real-time processing and technical failure constraints.

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

Multi-agent systems can be used for a decentralized and self-organizing approach of data processing in a distributed system like a sensor network [2], enabling information extraction, for example based on pattern recognition [3], and by decomposing complex tasks in simpler cooperative agents.

Hardware (microchip level) designs have advantages compared with microcontroller approaches concerning power consumption, performance, and chip resources by exploiting parallel data processing (covered by the agent model) and enhanced resource sharing [6], which will be applied in this work.

Usually sensor networks are a part of and connected to a larger heterogeneous computational network [2]. Employing of agents can overcome interface barriers arising between platforms differing considerably in computational and communication capabilities. That's why agent specification models and languages must be independent of the underlying run-time platform. On the other hand, some level of resource and processing control must be available to support the efficient design of hardware platforms.

Hardware implementations of multi-agent systems are still limited to single or a few and non-mobile agents [13][15], and were originally proposed for low level tasks, for example in [8] using agents to negotiate network resources. Coarse grained reconfiguration is enabled by using FPGA technologies [13]. Most current work uses hardware-software co-design methodologies and code generators, like in [14]. This work provides more fine-grained agent reconfiguration and true agent mobility without relying on a specific technology and employs high-level synthesis to create standalone hardware and software platforms delivering the same functional and reactive behaviour.

There is related work concerning agent programming languages and processing architectures, like *APRIL* [10] providing tuple-space like agent communication, and widely used *FIPA ACL*, and *KQGML* [7] focusing on high-level knowledge representations and exchange by speech acts, or model-driven engineering (e.g. INGENIAS, [9]. But the above required resource and processing control is missing, which is addressed in this work.

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

- limited static processing, storage, and communication resources, real-time processing, unreliable communication,
- suitable simulation environments for testing distributed and parallel multi-agent processing on functional and operational level,
- suitable simplified agent-oriented programming models and processing architectures qualified for hardware designs with finite state machines (FSM) and resource sharing for parallel agent execution,
- and appropriate high-level design and simulation tools offering MAS design on programming level.

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

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

- One common agent behaviour model, which is implementable on different processing platforms (hardware, software, simulator).
- Agent mobility crossing different platforms in a mesh-like network and agent interaction by using a tuple-space database and global signal propagation aid solving data distribution and synchronization issues in the design of distributed sensor networks.
- Support for heterogeneous networks and platforms covered by one design and synthesis flow including functional behavioural and architectural simulation.
- A token-based pipelined multi-process agent processing architecture very well suited to hardware platforms with the Register-Transfer Level Logic offering optimized computational resources and speed.
- A Petri-Net representation is used to derive a specification of the hardware process and communication network, and performing advanced analysis like deadlock detection. Timed Petri-Nets can be used to calculate computing time bounds to support real-time processing.
- A fast processing platform simulation on architectural level with large scale networks (with hundreds of nodes) and simulation of the processing of large scale MAS (with hundreds of mobile agents).

The next sections introduce the activity based agent processing model, available mobility and interaction, and the proposed agent platform architecture related to the programming model. Finally, the suitability of the agent processing platform in large scale networks is demonstrated by using a novel agent-based simulation technique for the simulation of the entire platform and network architecture at process level. The simulator can be integrated in an existing sensor network offering a simulation-in-the-loop methodology.

2. Programming State-based Mobile Agents

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

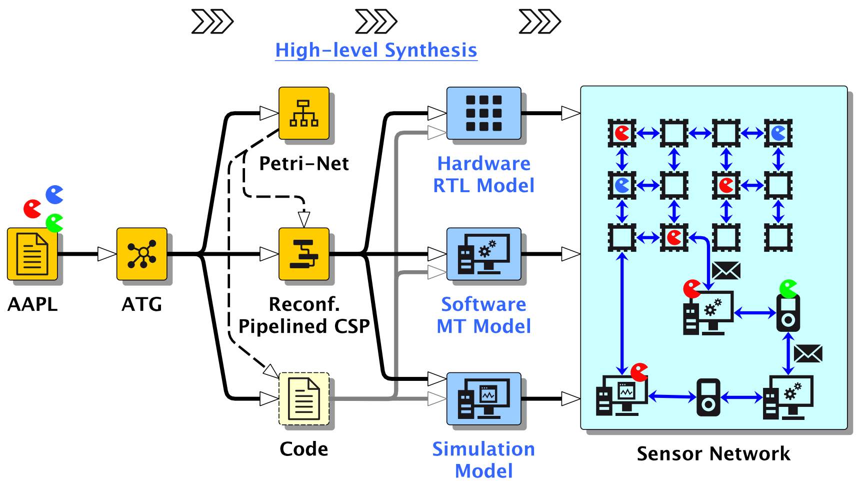

Fig. 1. From one common *AAPL* programming level to heterogeneous distributed networks (*RTL*: Register-Transfer Level, *MT*: Multi-Threading, *CSP*: Communicating Sequential Processes

Definition: *There is a multi-agent system (MAS) consisting of a set of individual agents* $\{A_1,A_2,..\}$. *There is a set of different agent behaviours, called classes* $C=\{AC_1, AC_2,..\}$. *An agent belongs to one class. A super class* AC_i *can be composed of different sub-classes* $\{AC_{i,1},AC_{i,2},..\}$, *sharing activities and transitions of the super 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* $N=\{N_1, N_2,..\}$ *arranged in a mesh-like network with peer-to-peer neighbour connectivity (e.g. two-dimensional grid). Each node is capable to process a number of agents* $n_i(AC_i)$ *belonging to one agent behaviour class* AC_i, *and supporting at least a subset of* $C' \subseteq C$. *An agent (or at least its state) can migrate to a neighbour node where it continues working.*

AAPL Programming Model

The agent behaviour is partitioned and modelled with an activity graph, with activities representing the control state of the agent reasoning engine, and conditional transitions connecting and enabling activities, shown in Fig. **3** (left side). Activities provide a procedural agent processing by sequential execution of imperative data processing and control statements.

The activity-graph based agent model is attractive due to the proximity to the finite-state machine model, which simplifies the hardware implementation.

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

Plans are related to *AAPL* activities and transitions close to conditional triggering of plans. Tables **1** and **2** summarize the available language statements. Their effects on a multi-agent system is shown in Fig. **2**.

Agent Identifiers. Some statements like signal propagation require the identification of agents. In a node-local scope this is the token identifier assigned to the agent (Local *LID*). If an agent migrates, the LID can be extended by the relative displacement (Δ) to a unique identifier in the global network scope (GID). But this enforces the blocking of the token on the root node to avoid duplicates. Random generation of agent identifiers can overcome this issue, which is applied in this work.

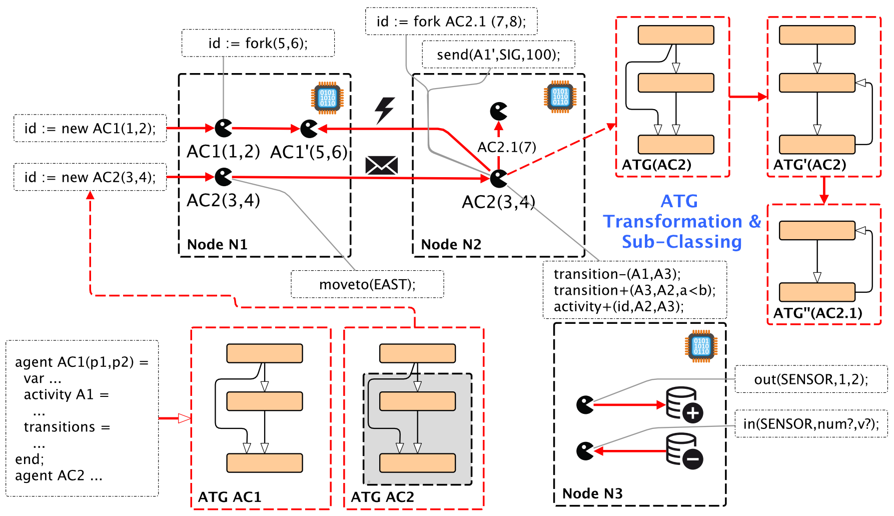

Fig. 2. Effects of AAPL control statements at run-time

Tab. 1. Summary of the AAPL statements used to define the agent behaviour and control

Kind	AAPL Statement	Description
Agent Class Definition	`agent` *AC* (*arguments*) = *definitions* `end;`	Defines a new agent class *AC* with optional arguments. The class body consists of variable, activity, and transition definitions.

Creation and Replication	*id* `:= new` *AC*[.*C*] (*args*);
	 id `:= fork` [*C*] (*args*);
	 `kill(`*id*`);`
		Creates new agents at run-time. They are created from the class template, or forked from the parent agent. A sub-class *C* can be selected, too.

Kind	AAPL Statement	Description

Data	`var x,y,z: datatype;`
	`var* a,b,c: datatype;`
		Defines long- and short term agent body variables. The latter ones are not saved on migration or inherited by children.

Activity	`activity [C.]A =`
	`  statements`
	`end;`
		Defines a new agent activity *A*, which can be bound to a sub-class *C*.

Reconfiguration	`activity+ (id,a1,a2,...);`
	`activity- (id,a1,a2,...);`
		Adds or removes activities at run-time for a specific agent *id*.

Transition	`transitions [C] =`
	`  transitions end;`
	`   ai -> aj: condj; ...`
		Defines transitions at compile time between activities a_i and a_j with predicate $cond_j$. Can be used to define a sub-class *C*, too.

Reconfiguration	`transition+ [C] (a1,a2,c);`
	`transition* [C] (a1,a2,c);`
	`transition- [C] (a1,a2);`
	`                (id,..)`
		Changes transitions at run-time (add, replace all, remove all). Can be applied to a sub-class *C* or for a specific agent *id* only.

Mobility	`moveto(Dir);`
	`.. link?(Dir) ..`
	`Dir={NORTH, SOUTH,`
	`     WEST, EAST}`
		Migrates agent to a neighbour node. The connectivity can be tested by using the `link?` operation.

Instantiation. Parameterized new agents of a specific class *AC* can be created at runtime by agents using the `new AC(v1,v2,..)` statement returning a node unique agent identifier. An agent can create multiple living copies of itself with a fork mechanism, creating child agents of the same class with inherited data and control state but with different parameter initialization, done by using the `fork(v1,v2,..)` statement. Agents can be destroyed by using the `kill(id)` statement. Additionally, sub-classes of an agent super class can be selected by adding the sub-class identifier.

Each agent has *private data* - the body variables -, defined by the `var` and `var*` statements. Variables from the latter definition will not be inherited or migrated! Agent body variables, the current activity, and the transition table represent the mobile data part of the agents beliefs database.

Statements inside an activity are processed sequentially and consist of data assignments (`x := ε`) operating on agent's private data, control flow statements (conditional branches and loops), and special agent control and interaction statements, which can block agent processing until an event has occurred.

Agent interaction and synchronization is provided by a tuple-space database server available on each node (based on [10]). An agent can store an n-dimensional data tuple *(v1,v2,..)* in the database by using the `out(v1,v2,..)` statement (commonly the first value is treated as a key). A data tuple can be removed or read from the database by using the `in(v1,p2?,v3,..)` or `rd(v1,p2?,v3,..)` statements with a pattern template based on a set of formal (variable,?) and actual (constant) parameters. These operations block the agent processing until a matching tuple was found/stored in the database. These simple operations solve the mutual exclusion problem in concurrent systems easily. Only agents processed on the same network node can exchange data in this way. Simplified the expression of beliefs of agents is strongly based on *AAPL* tuple database model. Tuple values have their origin in environmental perception and processing bound to a specific node location.

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

Remote interaction between agents is provided by signals carrying optional parameters (they can be used locally, too). A signal can be raised by an agent using the `send(ID,S,V)` statement specifying the ID of the target agent, the signal name S, and an optional argument value V propagated with the signal. The receiving agent must provide a signal handler (like an activity) to handle signals asynchronously. Alternatively, a signal can be sent to a group of agents belonging to the same class AC within a bounded region using the `broadcast(AC,DX,DY,S,V)` statement. Signals implement remote procedure calls. Within a signal handler a reply can be sent back to the initial sender by using the `reply(S,V)` statement.

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

Migration of agents to a neighbour node (preserving the body variables, the processing, and configuration state) is performed by using the `moveto(DIR)` statement, assuming the arrangement of network nodes in a mesh- or cube-like network. To test if a neighbour node is reachable (testing connection liveliness), the `link?(DIR)` statement returning a Boolean result can be used.

Reconfiguration. Agents are capable to *change their transitional network* (initially specified in the transition section) by changing, deleting, or adding (conditional) transitions using the transition♦(A_i,A_j,cond) statements. *This behaviour allows the modification of the activity graph, i. e., based on learning or environmental changes, which can be inherited by child agents.* The modification can be restricted to a subclass transition set, which is useful for child agent generation. Additionally, the ATG can be transformed by adding or removing activities using the activity♦(A_i,A_j,...) statements, which is only applicable for dynamic code-based agents not considered here.

Tab. 2. Summary of the AAPL statements used for interaction and communication

Kind	AAPL Statement	Description

Signal	`signal` *S*:*datatype*;
	`handler` *S*(*x*) =
	  statements `end`;
	`send(`*id*,*S*,*v*`);`
	`reply(`*S*,v`);`
	`broadcast(`*AC*,*DX*,*DY*,*S*,*v*`);`
		Defines a signal *S* which can be processed by a signal handler similar to an activity. Signals are either send to a specific agent *id* or send to all agents of a specific class within a region.

Kind	AAPL Statement	Description
Tuple Space Database	`out(v1,v2,..);` `.. exist?(v1,?,..) ..` `in(v1,x1?,v2,x2?,...);` `rd(v1,x1?,v2,x2?,...);` `try_in(timout,v1,..);` `try_rd(timeout,v1,..);` `mark(timeout,v1,v2,..);` `rm(v1,?,..);`	Synchronized data exchange by agents using the tuple space operations with tuples and patterns. A marking is a tuple with a limited lifetime. Commonly, the first tuple value is treated as a key, e.g. classifying the tuple.
Timer and Blocking	`timer+(timeout,S);` `timer-(S);` `sleep; wakeup;`	A timer can be used to raise a signal *S*. Agents can be suspended and be woken up.

3. Agent Platform Architecture and Synthesis

The *AAPL* model is a common source for the implementation of agent processing in hardware, software, and simulation processing platforms. A database driven high-level synthesis approach [1] is used to map the agent behaviour to these different platforms. The agent processing architecture required at each network node must implement different agent classes and must be efficiently scalable to the microchip level to enable material-integrated embedded system design, which represents a central design issue, further focussing on parallel agent processing and optimized resource sharing.

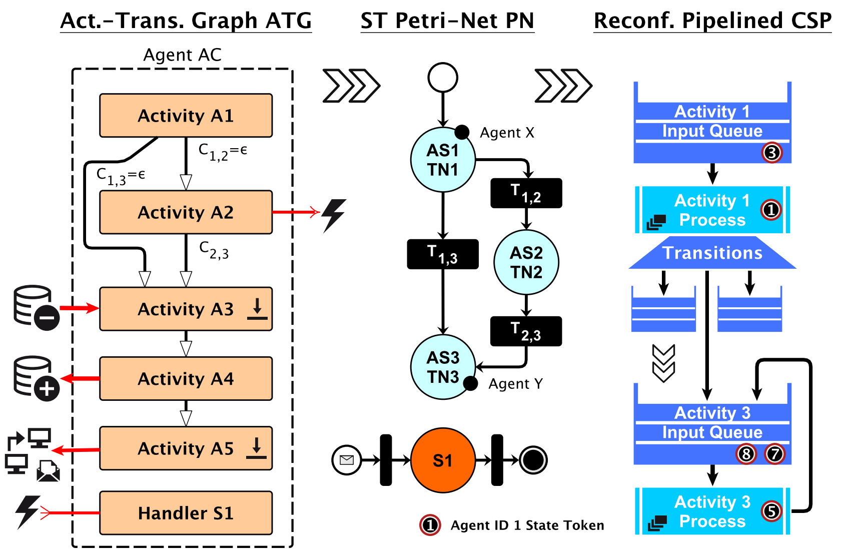

Fig. 3. Pipelined Communicating Sequential Processes Architecture derived from a Petri-Net specification and relationship to the activity-transition graph. Signals are handled asynchronously and independently from the activity processing.

3.1. The RPCSP Agent Platform

This processing platform - very well matching microchip-level designs - implements the agent behaviour with *reconfigurable pipelined communicating processes* (*RPCSP*) related to the Communicating Sequential Process model (*CSP*) proposed by Hoare (1985). The activities and transitions of the *AAPL* programming model are merged in a first intermediate representation by using state-transition Petri Nets (*PN*), shown in Fig. **3**. This *PN* representation allows the following *CSP* derivation specifying the process and communication network, and advanced analysis like deadlock detection. Timed Petri-Nets can be used to calculate computing time bounds to support real-time processing.

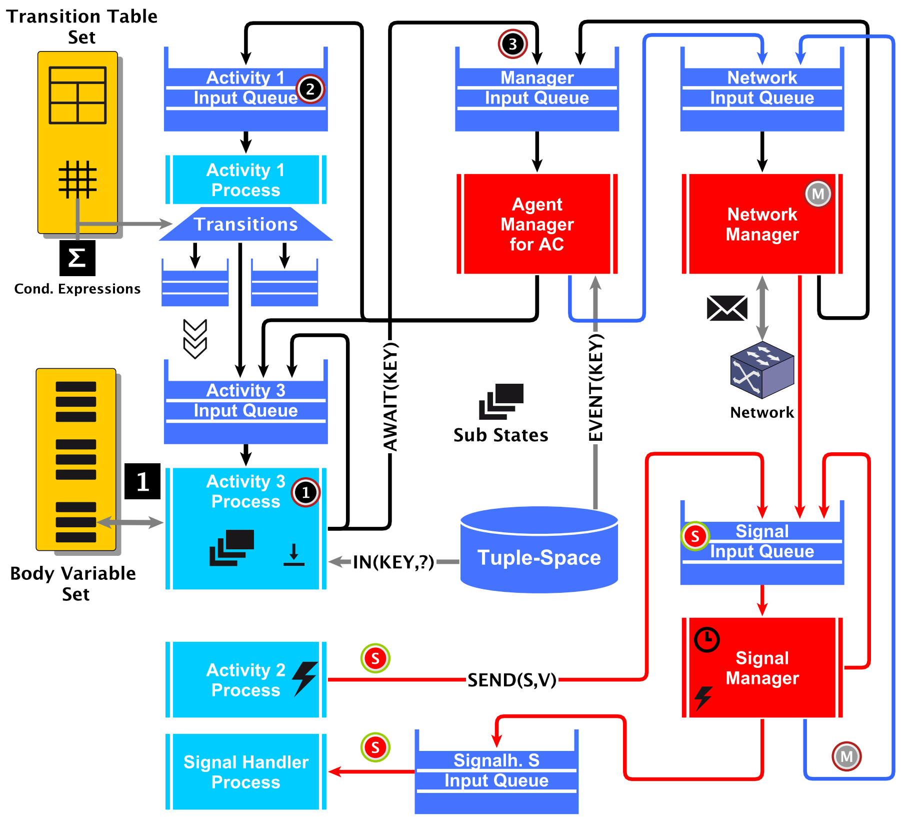

Fig. 4. Interaction of the agent, signal, and network manager with activity processes

Keeping the *PN* representation in mind, the set of activities $\{A_i\}$ is mapped on a set of sequential processes $\{P_i\}$ executed concurrently. Each subset of transitions $\{T_{i,j}\}$ activating one common activity process P_j is mapped on a synchronous n:1 queue Q_j providing inter-activity-process communication, and the computational part for transitions embedded in all contributing processes $\{P_i\}$, shown in Fig. **3**. Changes (reconfiguration) of the transition network at run-time are supported by transition tables, shown in

Fig. **4**. Body variables of agents are stored in an indexed table set. Activity processes are partitioned into sub-states, at least one computational and one transitional state, discussed below.

Each sequential process is mapped (by synthesis) on a finite-state machine and a data path using a register-transfer architecture (RTL) with mutual exclusive guarded access of shared objects, all implemented in hardware.

This pipeline architecture offers advanced resource sharing and parallel agent processing with only one activity process chain implementation required for each agent class. The hardware resource requirement (digital logic) is divided into a control and a data part. The control part is proportional to the number of supported different agent classes. The data part depends on the maximal number of agents executed by the platform and the storage requirement for each agent.

Token-based Processing. Agents are represented by tokens (natural numbers equal to the agent identifier, unique at node level), which are transferred by the queues between activity processes depending on the specified transition conditions and the enabling of transitions. This multi-process model is directly mappable to Register-Transfer Level (*RTL*) hardware architectures. Each process P_i is mapped on a finite state machine FSM_i controlling process execution and a register-transfer data path. Local agent data are stored in a region of a memory module assigned to each individual agent. There is only one incoming transition queue for each process consuming tokens, performing processing, and finally passing tokens to outgoing queues, which can depend on conditional expressions and body variables. There are computational and IO/event-based activity statements. The latter ones can block the agent processing until an event occurs (for example, the availability of a data tuple in the database).

Agents in different activity states can be processed concurrently. Thus, activity processes which are shared by several agents may not block. To prevent blocking of IO processes, not-ready processes pass the waiting agent to the agent manager.

Activity Sub-State Partitioning and Event-based Processing. To handle I/O-event and migration related blocking of statements, activity processes executing these statements are partitioned into sub-states $A_i \Rightarrow \{a_{i,1}, a_{i,2}, \ldots, a_{i,TRANS}\}$ and a sub-state-machine decomposing the process in computational, I/O statement, and transitional parts, which can be executed sequentially by back passing the agent token to the input queue of the process (sub-state loop iteration). The control state of an agent consists therefore of the actual/next activity A_i/A_{i+n} and the activity sub-state $a_j(A_i)$ to be executed. Agents which wait for the occurrence of an event are passed to the agent manager queue releasing the activity process. After the event occurred, the agent token is passed back to the activity process continuing the execution, shown in Fig. **4**. Usually I/O events are related to tuple-space database (*TSDB*) access (*in*-operation is blocked until a matching *out*-operation is performed). For this reason the *TSDB* module is directly connected to the agent manager which is notified about the keys of new tuples stored in the database releasing waiting consumer agents. The following annotated code snippet shows the sub-state partitioning and sub-state transitions ($\rightarrow$: immediate, $\perp$: blocked and passed to the agent manager).

```
activity init =
init_1: dx := 0; dy := 0; h := 0; → init_2
init_2: if dir <> ORIGIN then
          moveto(dir); ⊥ init_3
          init_3: case dir of
                      | NORTH => backdir:=SOUTH;
                      | SOUTH => backdir:=NORTH;
                      | WEST =>  backdir:=EAST;
                      | EAST =>  backdir:=WEST;
                    end; → init_4
        else
          live:=MAXLIVE; backdir:=ORIGIN; → init_4
        end;
init_4: group := Random(integer[0..1023]);
        out(H,id(SELF),0); → init_5
init_5: rd(SENSORVALUE,s0?);  ⊥ init_TRAN
init_TRAN: Transition Computation
```

If there are conditional outgoing transitions which cannot actually be satisfied, the activity process can be suspended (by using the *AAPL* `sleep` statement) by transferring the agent token to the agent manager. A signal handler of the agent can be used to wake up the agent again (by using the `wakeup` statement).

Agent and Network Managers. The agent manager is connected with all input queues of the activity processes and with the network managers handling remote agent migration and signal propagation. Agents are associated with control state structures. Agent tokens are injected by the agent manager after agent creation, migration, or resumption.

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

Transitions and Reconfiguration. Each activity process has a final transition sub-state $a_{i,TRAN}$ which tests for enabled transitions in the current context. All possible (enabled and disabled) transitions outgoing from an activity are processed in the transition sub-state of each activity process. If a condition of an enabled transition is true, the agent token is passed to the respective destination queue.

Configuration of the transition network at run-time modifies transition tables, storing the state of each transition {enabled, disabled}. There is one table set for each individual agent which can be divided further in the super class and possible sub-classes.

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

Reconfiguration can aid to increase and optimize utilization of the activity process network populated by different sub-classed agents using only a sub-set of the activities.

Migration *of agents* requires the transfer of the agent data and the control state of the agent together with a unique global agent identifier (extending the local *id* with the agent class and the relative displacement of its root node) encapsulated in messages.

Messages carrying the state of agents consisting of the body variables (only the long-term part) and the control structure with the current activity, the sub-state which is entered after migration, and agent identifiers (id, Δ). Furthermore messages are used to carry signals. The network managers (input & output) perform message encoding, decoding, and delivery. Migration requires at least one more activity sub-state. After migration, the next sub-state of the last activity is executed.

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

Signal Handling. Signals are handled asynchronously by activating signal handlers, implemented by a process and a signal handler input queue. The signal manager is responsible for the creation and propagation of signals, shown in the bottom of Fig. **4**. Signal tokens represent tuple values *(signal, argument, dst-id, src-id, Δ)*. Remote signals are processed by the signal and network managers, which encapsulate signals in messages sent to the appropriate target node and agent.

Replication of activity processes sharing the same input queue offers advanced parallel processing of multiple agents for activities with high computing times reducing the mean computational latency.

3.2. Software Platform

The already introduced *RPCSP* architecture can be implemented in software, too. In this case the activity processes are implemented with light weighted processes (threads) communicating through queues, providing token based agent processing, too. The software platform includes the agent and signal managers, tuple space databases, and networking. Software platforms can be directly connected to hardware platforms and vice versa. They are compatible at the interface (message) and agent behaviour level.

Implementing the *RPCSP* architecture in software has the advantage of low-resource requirements and the exploitation of parallelism by multi-processor or multi-core architectures including advanced hyper-threading techniques. The number of threads and resources are known and allocated in advance, which can be mandatory for hard real-time processing systems.

3.3. Simulation Platform

In addition to real hardware and software implemented agent processing platforms there is the capability of the simulation of the agent behaviour, mobility, and interaction on a functional level. The *SeSAm* simulation framework ([5]) offers a platform for the modelling, simulation, and visualization of mobile multi-agent systems employed in a two-dimensional world. The behaviours of agents are modeled 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 of complex MAS on behavioural level and the methodology using the *SeSAm* simulator was already demonstrated in [16], mapping *AAPL* agents of the MAS one-to-one to *SeSAm* agents. In this work instead the *RPCSP* agent processing platform is simulated with the agent-based *SeSAm* simulation framework, discussed in detail in the following section. This simulation provides the testing and profiling of the proposed processing platform architecture in a distributed network world.

The simulator is also fully compatible with the software and hardware platforms on behavioural and interface level and can be integrated in an existing real-world network, offering simulation-in-the-loop capabilities.

3.4. Synthesis

The database driven synthesis flow (details in [1]) is shown in Fig. **5** and consists of an *AAPL* front end, the core compiler, and several backends targeting different platforms. The *AAPL* program is parsed and mapped on an abstract syntax tree (*AST*). The first compiler stage analyses, checks, and optimizes the agent specification *AST*. The second stage is split in three parts: an activity to process-queue pair mapper with sub-state expansion, a transition network builder, manager generators, and a message generator supporting agent and signal migration. Different outputs can be produced: a hardware description enabling *SoC* synthesis using the *ConPro SoC* high-level synthesis framework (details inside [6]), a software description (*C*) which can be embedded in application programs, and the *SeSAm* simulation model (*XML*). The *ConPro* programming model reflects an extended *CSP* with atomic guarded actions on shared global resources. Each process is implemented with a *FSM* and a *RT* data path. The simulation design flow includes an intermediate representation using the *SEM* programming language, providing a textual representation of the entire *SeSAm* simulation model, which can be used independently, too.

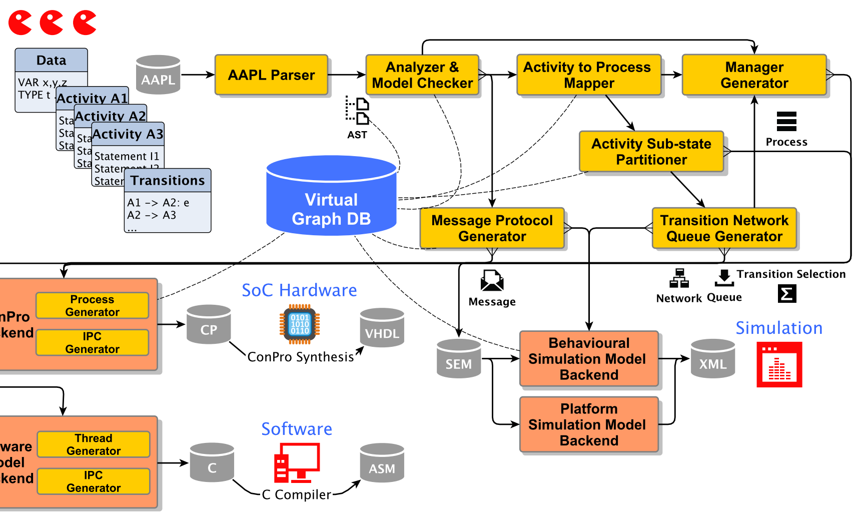

Fig. 5. Simplified overview of the high-level synthesis flow architecture

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

4. Platform Simulation

This section will demonstrate that agent-based simulation is suitable to for the simulation of the *RPCSP* agent processing platform itself and large scale distributed networks, e.g., sensor networks, using the agent-based *SeSAm* simulator [5]. Simulation and analysis of parallel and distributed systems are a challenge. Performance profiling and the detection of race conditions or deadlocks are essential in the design of such systems, where the agent processing platform is a central part. Furthermore, platform simulation allows the estimation and optimization of static resources like agent tables or queues, completed with the ability to study the temporal behaviour of the entire network including communication treated as a distributed virtual machine, e.g., identifying bottlenecks for specific task situations, hard to monitor in technical systems.

Behavioural simulation [1][16] maps agents of the MAS to be tested directly and isomorphic on agent objects of the simulation model. Platform simulation uses agents to implement architectural blocks like the agent manager or activity processes. Hence, agents of the MAS are virtually represented by the data space of the simulator, and not by the agent objects themselves.

The simulation of the processing platform with large scale networks processing large scale MAS aid to modify and refine the *RCSP* architecture, and to tune the static resource parameters like token pool and queue sizes or activity process replication to optimize timing. The platform simulation allows a fine grained estimation of the required resources.

The networks to be simulated (aka. the simulated world) consist of nodes arranged in a two-dimensional mesh grid, with each node connected to his four neighbour nodes, shown in Fig. **6** for a 10 by 10 sensor network with dedicated computational nodes at the outsides of the network. The entire platform and network system is partitioned into different non-mobile agent and resource classes (a resource is a passive agent with a data state only):

World Agent. The world agent creates all node agents and provides some network wide services. The world agent implements a reduced physical environment, e.g., by creating sensor signals or by disabling (destroying) connections between network nodes. Connections are represented by resources (passive agents providing only a geometric shape and body variables).

Node Agent. Each node is represented by a node agent, basically providing a common interface to data structures and tables required by the node managers and the activity processes. The node agent creates all the platform agents at the beginning of the simulation run.

Manager Agent. There is one "agent manager" agent for each agent class which is supported on the network node platform.

Network Manager Agents. There are two network manager agents. One input network manager agent handling incoming messages from neighbour nodes, decoding messages, creating agent or signal tokens, and finally passing the tokens to the agent or signal manager. The second output network manager agent is responsible for encoding and sending of messages carrying agent states or signals.

Activity Process Agents. For each agent class and each activity process of an agent class there is one activity processing agent performing token-based agent processing. The sub-states of an activity process are implemented by a simple sub-state selector and token loop-backing providing a sub-state *FSM*. Each activity process agent has local storage and an global visible token input queue.

Monitor Agent. There is one monitor agent per world collecting temporal resolved statistical data, finally writing the results to a CSV data file.

Token and Queues. The agent token queues are implemented with lists in the body variable space of each node agent. The size of the list can be monitored at run-time to detect resource underflow. Mutex guarded operations (*inq*,*outq*) allow concurrent access to the queues by different agents (manager, activity processes,..). Tokens are record structures with additional descriptive entries like the current queue they are stored in.

Virtual Agent. For visualization and debugging there is a mobile virtual agent resource representing an agent to be processed by a specific agent node platform. The virtual agent references the data and control state of an agent.

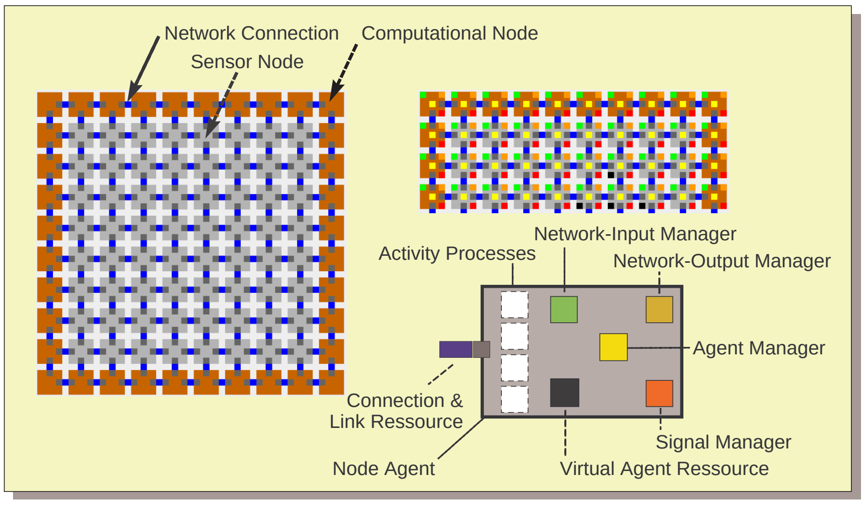

Fig. 6. Simulation world of a sensor network (left) consisting of 10x10 nodes and the network populated with non-mobile platform and virtual mobile agents (right)

The new platform simulation is compared with the behavioural simulation from previous work in Tab. **3** for the simulation of a self-organizing MAS used for feature extraction in a sensor network. The behaviour model of the MAS is described in detail in [16]. It bases on a distributed divide-and-conquer approach. The number of (non-mobile) agents implementing the processing platform depends mainly on the number of activities decomposing the agent behaviour and the number of agent classes to be supported on the platform. For this example, the platform simulation model requires five times more agents and twenty times more computing time than the behavioural model. But the required resources and computing time for the fine-grained platform simulation is still reasonable and can be handled well with low end computers.

The analysis of a simulation run is shown in Fig. 7. It shows the temporal resolved analysis of the population of explorer agents of the MAS and the utilization of the *PC-SP* network for nodes processing actually agents. There are nodes capable to process up to four agents simultaneous (speedup 4, in different activity states and processes). The mean speedup factor is about 1.5 for all nodes actually processing agents.

Both platform and behavioural simulation deliver the same computational results of the distributed MAS.

Tab. 3. Comparison of behavioural and platform simulation of the same MAS [16] using the SeSAm simulator

	Behavioural Simulation	**Platform Simulation**
Number of Agents and Resources (dynamic=mobile)	static:300 agents, 700 res.; dynamic: 130 explorer agents	static: 1600 agents, 700 res.; dynamic: 130 virtual agent resources
Simulation time including setup of simulation, with a correlated cluster scenario of 8 nodes, until MAS has finished work.	60 simulation steps in 5 s (on 1.2 GHz Intel U9300, 3GB)	280 simulation steps in 60 s (on 1.2 GHz Intel U9300, 3GB)

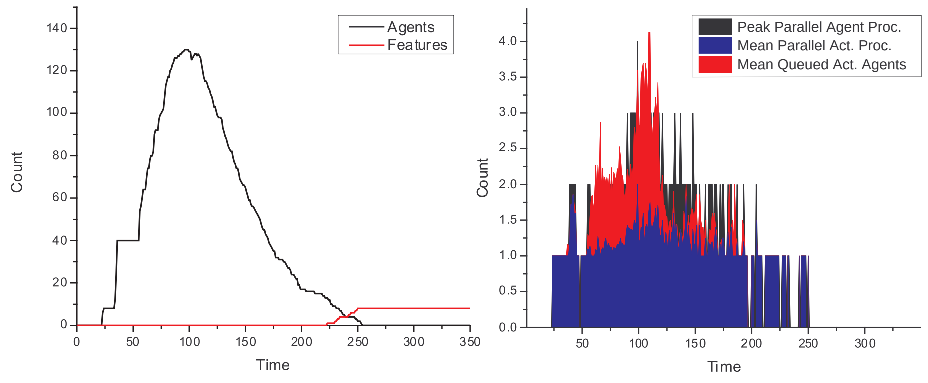

Fig. 7. Analysis of the MAS simulation: Left plot shows the temporal development of the agent population (explorer agent) and the rise of found features in the sensor network, right plot shows the utilization of the platform processes (peak parallel agent processing on one node, mean parallel active processes per node, and mean agent tokens queued per active node).

5. Conclusions

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

Agents of the same class share one virtual machine consisting of a reconfigurable pipelined multi-process chain based on the *CSP* model implementing the activities and transitions, offering parallel agent processing with optimized resource sharing. Unique identification of agents does not require unique absolute node identifiers or network addresses, a prerequisite for loosely coupled and dynamic networks (due to failures, reconfiguration, or expansion). The migration of an agent to a neighbour node takes place by migrating the data and control state of an agent using message transfers. Two different agent interaction primitives are available: signals carrying data and tuple-space database access with pattern templates.

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

A novel agent-based simulation of the agent processing platform and large-scale networks with a MAS case study demonstrated the suitability of the proposed programming model, processing architecture, and synthesis approach. The platform simulation offers advanced study and visualization of the platform behaviour, performance, and synchronisation issues in a distributed system under real world conditions with respect to the executed MAS. The platform simulation was compared with earlier behavioural agent simulations using the same MAS. Though there is a significant increase of the required data resources and computation time, this simulation approach is well suited for large-scale MAS simulation.

6. References

1. S. Bosse, *Distributed Agent-based Computing in Material-Embedded Sensor Network Systems with the Agent-on-Chip Architecture*, IEEE Sensors Journal, Special Issue MIS, 2014, DOI:10.1109/JSEN.2014.2301938.
2. M. Guijarro, R. Fuentes-fernández, and G. Pajares, *A Multi-Agent System Architecture for Sensor Networks*, Multi-Agent Systems - Modeling, Control, Programming, Simulations and Applications, 2008.
3. X. Zhao, S. Yuan, Z. Yu, W. Ye, J. Cao. (2008), *Designing strategy for multi-agent system based large structural health monitoring*, Expert Systems with Applications, 34(2), 1154–

1168. doi:10.1016/j.eswa.2006.12.022

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

5. F. Klügel, *SeSAm: Visual Programming and Participatory Simulation for Agent-Based Models*, In: Multi-Agent Systems - Simulation and Applications, A. M. Uhrmacher, D. Weyns (ed.), CRC Press, 2009

6. S. Bosse, *Hardware-Software-Co-Design of Parallel and Distributed Systems Using a unique Behavioural Programming and Multi-Process Model with High-Level Synthesis*, Proceedings of the SPIE Microtechnologies 2011 Conference, Session EMT 102, DOI:10.1117/12.888122

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

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

9. C. Sansores and J. Pavón, “An Adaptive Agent Model for Self-Organizing MAS ” in Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), May, 12-16., 2008, Estoril, Portugal, 2008, pp. 1639–1642.

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

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

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

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

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

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

16. S. Bosse, *Design of Material-integrated Distributed Data Processing Platforms with Mobile Multi-Agent Systems in Heterogeneous Networks*, Proc. of the 6-th International Conference on Agents and Artificial Intelligence ICAART 2014, 2014, DOI:10.5220/00048175 00690080.