Intelligent Microchip Networks: An Agent-on-Chip Synthesis Framework for the Design of Smart and Robust Sensor Networks

Stefan Bosse[1,2]
[1]University of Bremen, Department Computer Science,
Workgroup Robotics, Germany
[2]ISIS Sensorial Materials Scientific Centre, Univ. of Bremen, Germany

Abstract

Sensorial materials consisting of high-density, miniaturized, and embedded sensor networks require new robust and reliable data processing and communication approaches. Structural health monitoring is one major field of application for sensorial materials.

Each sensor node provides some kind of sensor, electronics, data processing, and communication with a strong focus on microchip-level implementation to meet the goals of miniaturization and low-power energy environments, a prerequisite for autonomous behaviour and operation. Reliability requires robustness of the entire system in the presence of node, link, data processing, and communication failures.

Interaction between nodes is required to manage and distribute information. One common interaction model is the mobile agent. An agent approach provides stronger autonomy than a traditional object or remote-procedure-call based approach. Agents can decide for themselves, which actions are performed, and they are capable of flexible behaviour, reacting on the environment and other agents, providing some degree of robustness.

Traditionally multi-agent systems are abstract programming models which are implemented in software and executed on program controlled computer architectures. This approach does not well scale to microchip level and requires full equipped computers and communication structures, and the hardware architecture does not consider and reflect the requirements for agent processing and interaction.

We propose and demonstrate a novel design paradigm for reliable distributed data processing systems and a synthesis methodology and framework for multi-agent systems implementable entirely on microchip-level with resource and power constrained digital logic supporting Agent-On-Chip architectures (*AoC*). The agent behaviour and mobility is fully integrated on the micro-chip using pipelined communicating processes implemented with finite-state machines and register-transfer logic.

The agent behaviour, interaction (communication), and mobility features are modelled and specified on a machine-independent abstract programming level using a state-based agent behaviour language (*APL*). With this *APL* a high-level agent compiler is able to synthesize a hardware model (*RTL*, *VHDL*), a software model (*C*, *ML*), or a simulation model (*XML*) suitable to simulate a multi-agent system using the *SeSAm* simulator framework.

Agent communication is provided by a simple tuple-space database implemented on node level providing fault tolerant access of global data.

A novel synthesis development kit (*SynDK*) based on a graph-structured database approach is introduced to support the rapid development of compilers and synthesis tools, used for example for the design and implementation of the *APL* compiler.

Keywords: Agent-on-Chip, Embedded Systems, System-on-Chip design, High-Level Synthesis, Parallel systems, Parallel computing, Distributed Systems, Multi-Agent Systems, Artificial Intelligence, *ASIC* and *FPGA* technology, Communication

1. Introduction and Overview

Embedded systems used for control, for example in Cyber-Physical-Systems (*CPS*), perform the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource-constrained manner.

Trends recently emerging in engineering and micro-system applications such as the development of sensorial materials show a growing demand for autonomous networks of miniaturized smart sensors and actuators embedded in technical structures **[6]**. With increasing miniaturization and sensor-actuator density, decentralized network and data processing architectures are preferred or required. A multi-agent system can be used for a decentralized and self-organizing approach of data processing in a distributed system like a sensor network, enabling the mapping of distributed data sets to related information, for example, required for object manipulation with a robot manipulator **[2][7]**.
Simplification and reduction of synchronization constraints owing to the autonomy of agents is provided by the distributed programming model of mobile agents **[5]**.
Traditionally, mobile agents are processed on generic program-controlled computer architectures **[7][8]**, which usually cannot easily be reduced to single microchip level like they are required, e.g., in sensorial materials with high sensor node densities. Furthermore, agents are treated with abstract heavy-weighted knowledge-based models, not suitable for today distributed data processing for example in sensor networks.
In the following sections, an agent data processing and communication architecture (Sec. **5**) together with a simple easy to learn but powerful agent programming language *APL* (Sec. **3** & **4**) is introduced, suitable for the programming and implementation of mobile state-based agents in distributed networks consisting of single micro-chip low-resource nodes. Traditionally agents programs are interpreted, with an interpreter implemented using a common imperative programming language like *JAVA* **[13]**, leading to a significant decrease in performance. In the approach presented here the agent processing can directly be implemented in hardware without intermediate processing levels. The novel behavioural agent programming language *APL* is introduced, a prerequisite for the technology- and implementation independent design of agents from programming level, though the language core is based on Modula-3, extended with agent specific language features implementable on micro-chip level. Agents implemented on micro-chip level are reported in **[1]**.
There is related work concerning agent programming languages and processing architectures, like *APRIL* **[14]** providing tuple-space like agent communication, extended *Erlang* derivation **[13]** using an existing language, and widely used *FIPA*, *ACL*, and *KQGML* **[11]** focusing on high-level knowledge representation and exchange. All those approaches represent communication and information on complex and abstract level not well suited for low-resource data processing in distributed loosely coupled networks, especially in sensor networks.
APL provides inter-agent communication through a tuple-space database server implemented on node level. Data is stored in form of tuples in the database and can be extracted by using patterns. This approach decreases the computational dependency of agents from the node data processing systems, thus agent and node must not complain about data in advance, improving reliable run-time behaviour in the case of (minor) data type mismatch. Furthermore *APL* provides global agent communication by using signals which can be send to a specific agent or broadcast to a group of agents.
System-On-Chip (*SoC*) designs are preferred for high miniaturization and low-power applications. The high-level synthesis *SoC* flow, discussed in Section **6**, part of *AoC* synthesis enables the design of application-specific constrained power- and resource-aware embedded systems on Register-Transfer-Level efficiently mapped to *FPGA* and *ASIC* technologies. A case study showing finally the suitability of the proposed agent synthesis approach in Section **9**.
What is novel compared with other data processing approaches?

- Reliability and robustness provided by smart autonomy of mobile state-based agents.
- Agent mobility and interaction solves data distribution and synchronizations issues in distributed systems design.
- One common agent programming language and processing architecture enables hardware and software synthesis.

- *APL* provides powerful statements for computation and agent control.

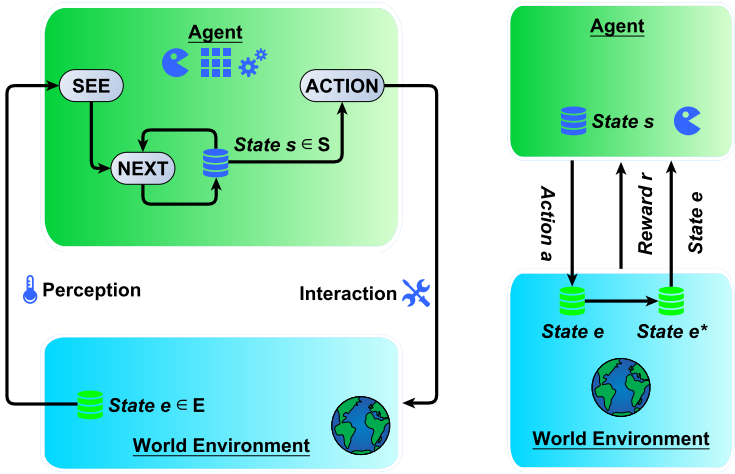
2. Distributed Data Processing with State-based Agents

Initially, a sensor network is a collection of independent computing nodes. Interaction between nodes is required to manage and distribute data and computed information. One common interaction model is the mobile agent. An agent is capable of autonomous action in an environment with the goal to meet its delegated objectives. An agent is a data processing system, a program executed on a computer system, that is situated in this environment **[5]**. A multi-agent system is a collection of loosely coupled autonomous agents migrating through the network. Agents can be used in sensor networks for

- local and global sensor data processing and extraction,
- sensor data fusion, filtering, and reduction of sensor data to information in a region of interest,
- sensor data and information distribution and transport in terms of distributed data processing,
- global energy management, exploration and negotiation

Agents can operate state-based. Such an agent consists of a state, holding data variables and the control state, and a reasoning engine, implementing behaviours and actions. In this proposed data processing and communication architecture, the state of an agent is completely kept in messages transferred in the network providing agent mobility. The functional behaviour of an agent can be easily implemented statically with a finite-state machine part of the local data processing system on register-transfer level (*RTL*), or dynamically by using a programmable code approach.

Figure 1. *State-based agents and interaction with the environment.*



Agents record information about an environment state $e \in E$ and history $h: e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots$. Let $I = S \times D$ be the set of all internal states of the agent consisting of the set of control states S related with activities and internal data D. An agent's decision-making process is based on this information. The perception function *see* maps environment states to perceptions, function *next* maps an internal state and perception $p \in Per$ to an internal state, the action-selection function *action* maps internal states to actions $a \in Act$ (see also Fig. **1**):

$$see : E \rightarrow Per$$
$$next : I \times Per \rightarrow I$$
$$action : I \rightarrow Act$$

Actions of agent 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 analyze the effect of their

action on the environment. After a performed action the agent gets a feedback in form of a reward $r(t)=r(e_t,a_t)$. There are strategies $\pi: E \rightarrow A$ which map environment states to actions. The goal of learning is to find optimal strategies π^*.

3. APL: The Agent Programming Language

The reliable implementation of mobile multi-agent systems on resource constrained embedded systems focussing on micro-chip level is a complex design challenge. High-level agent programming languages can aid to solve this design issue. There are already exist several agent modelling, interaction, and communication languages. But they target to software implementations and they are not suitable to implement multi-agent systems on micro-chip level directly. For this purpose, the Agent Programming Language ***APL*** was designed with strong focus on micro-chip scaled processing enabling direct high-level synthesis with multiple output targets. This language consists of generic imperative statements and a type system derived from a subset of the *Modula-3* language, and agent specific statements to specify the behaviour, mobility, and interaction of agents. This programming model is synthesizable directly to hardware (on Register-Transfer level ***RTL***, explained in Section **6**), software (*C*, *ML*), and simulation model targets (explained in Section **7**).
The agents behaviour is modeled with activities (representing the processing state of the agent) and transitions between activities. Activities provide sequential execution of data processing statements. An activity is activated by a transition depending on the evaluation of agent data (conditional transition), though unconditional transitions are supported, too. An agent belongs to a specific agent class, specifying local agent data (only visible to a specific agent itself), activities, and transitions. Additionally an agent class definition allows the definition of user defined types (enumeration, records, sub-ranges), signals (explained later), exceptions, and functions. Global types, signals, exceptions, and functions can be defined in modules, instead. The left side of Table **1** gives an overview of the agent class definition statements provided by ***APL***.
Agent creation and mobility is provided by the statements explained on the right side of Table **1**. An agent can be created dynamically at runtime by using the `new` statement with optional agent parameter arguments. An agent can move to a neighbour network node by applying the `moveto` statement expecting a migration direction argument. Migration is performed from inside some activity A_m. After migration an unconditional or conditional transition $A_m \rightarrow A_i$ is performed. To check the connection status to a specific neighbour node the `link?` statement can be used. The `iam?` statement checks both the connection fitness and the ability of a neighbour node to process this agent (class). Agents can be destroyed (terminated) by using the `kill` statement, and copied (with a copy of state and data) by using the `fork` statement.
There are pure computational activities and event or input-output (*IO*) based activities which can block the agent processing (e.g. moveto, agent interaction, discussed in Section **4**). The consideration of these different activity classes are important for the implementation and synthesis of agents, discussed in Section **5**.

Table 1. *Agent class definition, agent creation, destruction, and mobility **APL** programming language statements*

Agent Specification	Description	Creation & Mobility	Description
agent *AC*(p1,p2,..) = *variables* *signals* *exceptions* *types* *activities* *transitions* *functions* end;	Definition of an agent class with optional (constant) parameter list assigned with value arguments on agent creation. Data (variables), types, signals, exceptions, and functions are only visible locally (private).	ID := **new** *AC*(v1,..); eval(**new** *AC*(v1,..));	Create an agent of specified class with optional arguments. Returns local unique agent identifier `ID`.

Agent Specification	Description	Creation & Mobility	Description
var x,y,z : DT; **type** T = { .. }; **type** T = record .. end; **type** T = array ..; **type** T = list ..;	Definition of (private) data storage objects of data type DT, symbolic enumeration, record structure , and array types.	**kill**(ME); **kill**(ID);	Kill this agent or another agent identified by his agent identifier ID.
activity A = *statements* end;	Definition of an agent activity representing a state of the agent. Statements of the activity are executed in sequential order.	ID := **fork**(v1,..);	Create a living child copy from the forking parent agent with same state with optional arguments. Returns local unique agent identifier(s).
transitions = A1 -> A2 : *cond*; A2 -> A3; ... AS: signal; AE: exception; end;	Definition of agent state transitions (conditional depending on evaluation of a boolean expression and unconditional). Special activities can be reached by raised signals and exceptions.	type DIR = (North, South, West, East, Up, Down); **moveto**(DIR);	Mobility direction type and operation to move the agent to the neighbour node in specified direction. An exception NOCONN is raised if there is no connection to this particular node.
function F(p1,p2,..) : T = *statements* return *expr*; end; **procedure** P(p1,p2,..) = ...	Definition of a (private) function (with return type T) and procedure (without a return value).	**link?**(DIR) **iam?**(DIR)	Check connection to neighbour node in specified direction (boolean result) and check possibility to execute agent on neighbour node in specified direction (boolean result)

4. Agent Interaction and Communication

Reliable and type-safe interaction between agents providing data exchange is required for a multi-agent system performing distributed data processing in a network. There are two different interaction methods available and supported by ***APL***.

Locally there is a tuple-space dictionary which can be used to exchange data between agents operating on the same network node. There are agents producing data and agents consuming data. For example a sensor node provides sensor signal information by a local and non-migrating sensor processing agent storing sensor data in the dictionary by using the `in` operation, explained on the left side of Table **2**. Mobile agents visiting this node can collect the sensor data by removing or reading the data from the dictionary by using the `out` or `read` operation, respectively. Data in the database is found by matching regular expressions. For example the search pattern `("adc0",x?,y?)` matches value `("adc0",1,2)` - but not `("adc0",2)` - and assigns values 1 and 2 to x and y, respectively. These operation block the processing of the agent until a matching data tuple is found. To avoid blocking, the `try_**` operations with optional time-out can be used, returning a boolean status value.

Globally (and locally) signals can be used to notify specific agents or a group of agents by using the `send` or `broadcast` operation, respectively. Signals are delivered globally by using the agent trace stored in the

agent node cache, discussed in Section **5**.
APL uses a simple message passing mechanism for communicating between nodes performed by an agent manager avoiding additional communication overheads (below 5% compared with payload).

Table 2. APL *statements for tuple-space dictionary access and signal interaction*

Tuple-Space	Description	Signals	Description
in (v1,v2,..);	Store data tuple in database.	**signal** S; **send** (ID,S);	Send a signal S to agent specified by agent handler identifier ID.

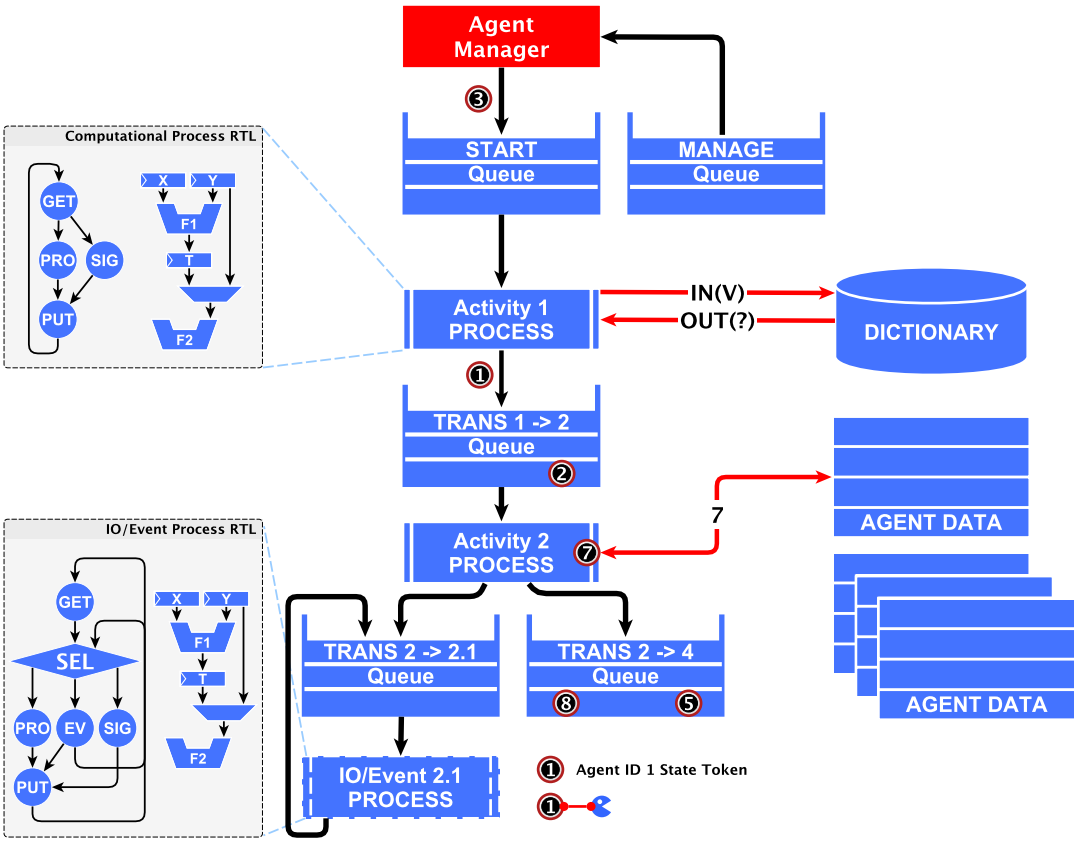
out (p1,v?,p2,..);
 	
read (p1,v?,p2,..);
 	
b := **try_out** (tmo, p1,v?,p2,..);
b := **try_read** (tmo, p1, v?,p2,..);
	Remove or read data tuple from database matching pattern (p1,*,p2,..). First two statements suspends agent processing until a matching tuple was found, the try_ statements return immediately (or after time-out) with a status.	**broadcast** (AC,DX,DY, S);	Send a signal S to all agents specified by agent class AC in the bounded region with extension of |DX, DY| relative to this node.

5. Computational Independency and Agent Processing Architecture

The agent behaviour (actions and transitions) is modelled on programming level with ***APL*** agent classes. The agent processing architecture required on each node must be scalable to micro-chip level to enable material-integrated embedded system design, and represent a central design issue for new the Agent-on-Chip data processing paradigm.
Different agent processing architectures were proposed in the past **[4]** with a non-programmable pure ***RTL/ FSM*** based and a programmable code-morphing based approach with multi-stack Forth processors.
In the first *non-programmable* approach the agent behaviour was mapped to a finite-state machine, with each state representing an activity (application specific virtual machine ***VM***). Only one agent can use one virtual machine at the same time, no agent sharing or multi-threading were possible. Agent data was stored in local ***VM*** registers. This approach requires $N_{VM}=N_{AC}*N_{agents/AC}$ virtual machines for a set of agent classes ***AC*** and a number of agents for each agent class determined in advance, not suitable for the implementation of a large set of agent classes and agents processing at runtime on the same node. Agent interaction is performed by using shared data structures specified at compile time (of the complete system). Agent migration transfers the control and data state of an agent with messages between network nodes. Advantage of this approach is the ease of implementation and the low amount of digital resources.
The second *programmable* approach using Forth interpreters (virtual machines) supporting code morphing allows the implementation and execution of more complex algorithms with limited support of concurrent multi-processing. Agent interaction is performed by using a word-data dictionary, allowing flexible behaviour at runtime with a higher degree of computational independency than by using predefined data structures at compile time. Agent migration is performed by transferring the program code or parts of the code between nodes, again increasing computational independency. Due to the stack-based Forth programming model with zero operand instructions and high-level instructions like loops, the code size can be kept compact, and hence message size.
But to decrease resource demands for concurrent agent processing of agents belonging to the same class and agents belonging to different classes *resource sharing* is necessary.
Thus the solution introduced in this section implements agent behaviour with a *pipelined communicating process model* (similar to ***CSP*** / Hoare **[3]**) mapping actions to sequential processes and transitions to queues

providing inter-activity-process communication, shown in Fig. **2**. This multi-process model is directly mappable to software and hardware implementations.
Individual agents are represented by tokens (natural numbers) which are transferred with queues from an outgoing activity to another in-coming activity depending on the activity transitions specified in the ***APL*** model. An activity is implemented with a finite-state machine reading an agent token from the input queue, processing the activity statements, and finally passing the token to an outgoing queue. There is only one incoming transition queue for each process. There are two different kinds of activities: pure computational non-blocking activities, and event based activities which can block the agent processing until an event happened, for example the availability of external data or a time delay. These two activity processing kinds require different implementations.

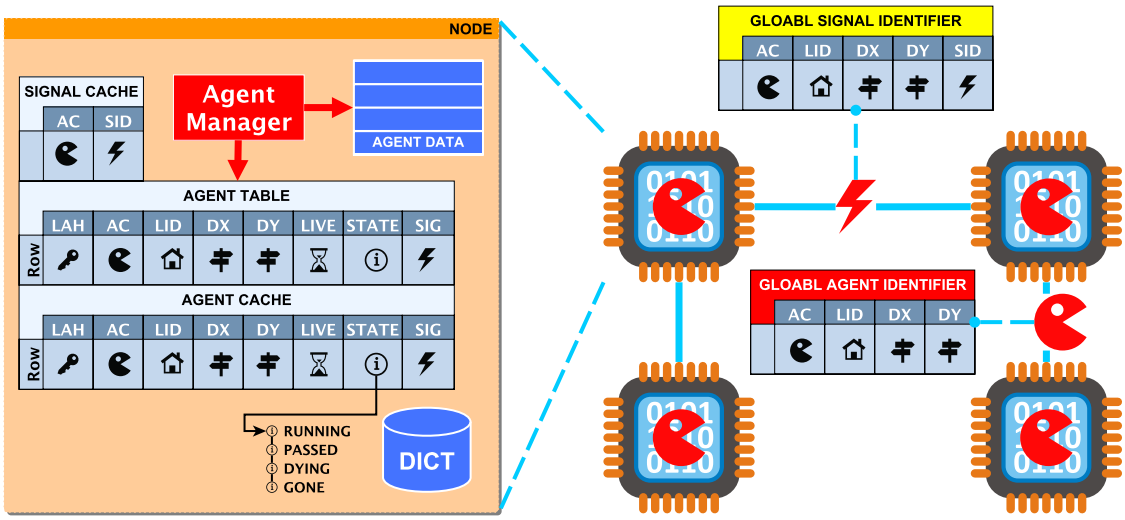
Figure 2. *Agent implementation with a pipelined multi-process architecture allowing operational unit sharing. Sequential processes are assigned to agent activities (representing states), queues implement state transitions. Inter-agent communication is performed by using a dictionary. Local agent data is stored in a region of a memory module assigned to each agent.*



Agent interaction is provided (locally on network node level) by a tuple-space dictionary (look-up table) and globally by using signals. Pending agent signals are checked each time an agent token was removed from a queue. The dictionary operations (introduced in Section **4**) are event-based statements which can block the agent processing. Thus *IO*/event-based processes remove an agent token from the input queue, check the availability of requested data or the happening of an event, and in the case the request can not be satisfied, the agent token is returned to the input queue, and the next agent token (if any) can be processed.
Local agent data is stored in a region of a memory module assigned to each agent and addressed by the agent token number. Each agent class is assigned to its own memory module.
An agent is started by an agent manager which transfers the agent token to the start queue or in the case of a

migrated agent in a different queue respective to the last agent control state. Terminating or migrating agents are handled by a manager queue.

Figure 3. *Agent-on-Chip network consisting of autonomous nodes with agent processing, communication, and agent administration, connected in mesh network with serial point-to-point links. Global agent- and signal identifiers are used to identify agents uniquely without the necessity of global unique node identifiers (*`LAH`*: Local Agent Handler,* `LID`*: Local Agent Identifier,* `AC`*: Agent Class,* `LIVE`*: agent live,* `STATE`*: agent state,* `DX` *and* `DY`*: spatial displacement vector,* `SID`*: Signal Identifier,* `SIG`*: pending signal* `ID`*).*



Agents are *identified* node-locally by an unique local agent handler (***LAH***), which is a slot number in the local *agent table* (see Fig. **3**) keeping information about all locally created agents. The ***LAH*** is equal to the local agent identifier (***LID***) in the case of a locally created agent, but can differ in the case of a remotely created agent now migrated to this node. All agents migrated to the local node or passing this node are stored in a following agent cache. Both the agent table and cache - part of the agent manager - store processing information about agents like agent class, control state, and pending signals (***SIG***). The agent cache holds additional information about the origin of the agent or the routing direction of an agent recently passed this node. Agent live times are used in conjunction with a garbage collector.

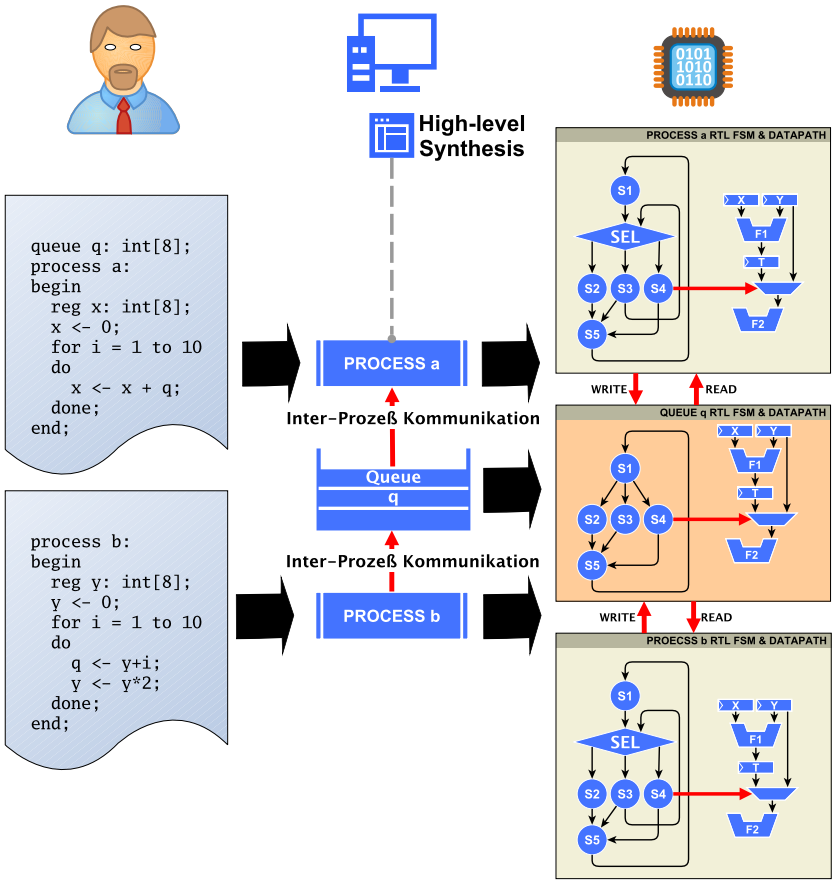
Agent *migration* requires a global agent identifier (***GID***) encapsulated in a network message holding information about the agent class ***AC***, the original ***LID***, and the displacement vector (***DX***,***DY***) relative to the origin of the agent. The ***GID*** is followed by the actual state of the agent consisting of local agent data and the current control state. *Signals* can be propagated encapsulated in a message from a source node to a destination node actually processing a specific agent by using a global signal identifier (***SID***), shown on the right side of Fig. **3**. Signals are not reliable because they depend on routing information stored in the agent cache. Due to limited cache size older entries can be removed. Missing routing information or changes in the network topology (link failures or reconfiguration) prevent signal delivery.

Summary: This new agent processing architecture requires only one virtual machine consisting of pipelined communicating processes for one agent class which should be supported on a particular network node. The application-specific optimized processes with low resource demand can be directly implemented in hardware on microchip level and alternatively in software without any modification of the processing architecture and the programming model. Unique identification of agents do not require unique absolute node identifiers or network addresses, a prerequisite for loosely coupled and changing networks (due to failures, reconfiguration, or expansion).

6. High-Level SoC Synthesis: from Programming level to Hardware Implementation

The *ConPro* high-level synthesis of *SoC* designs **[10]** uses a behavioural imperative programming language with a compiler-based synthesis approach from algorithmic programming level to register-transfer level mappable directly to digital logic. The *ConPro* compiler is integrated in the Synthesis Development Kit, discussed in Section **8**.

Figure 4. *Mapping of the ConPro communicating multi-process programming model to the multi-FSM RTL architecture using high-level synthesis.*



Concurrency is modelled explicitly on control path level with processes executing a set of instructions sequentially, initially independent of any other process. Inter-process communication (*IPC*) provides synchronization with different objects (mutex, semaphore, event, timer) and data exchange between processes using queues or channels, based on the *Communicating Sequential Processes* (*CSP*, Hoare 1985, **[3]**) model. There are local and global resources (storage, *IPC*) , accessed by one process and several processes, respectively. Concurrent access of global resources is automatically guarded by a mutex scheduler, serializing access, and providing atomic access without conflicts. There are process and top-level instructions. Top-level instructions are evaluated during synthesis (configuration). Process instructions are transformed and mapped to states of a clock-synchronous finite-state-machine (*FSM*) controlling the process *RTL* data path temporally and spatially.

More fine-grained concurrency is provided on data path level using bounded blocks executing several instructions (only data path, e.g., data assignments) in one time unit.

The *ConPro* programming language supports traditional imperative statements like loops, branches, and functions. Different types of storage objects (single registers and variables bound in shared *RAM* blocks, true bit-scaled), different aggregation types (array, structure), and abstract objects are supported. Programming statements can modify data (expressions, assignments) or have impact on the control flow (conditional and

counting loops, conditional branches, concurrent multi-value selection).
The complete synthesis process can be fine-grained parameterized on programming block level, for example selection of different expression models (allocation) or activation of specific schedulers and optimizers.
Hardware blocks, modelled on hardware level (*VHDL*), can be accessed from the programming level using an *object-orientated programming approach with methods*. All hardware blocks, including *IPC*, are treated like abstract data type objects (*ADTO*) with a defined set of methods accessible on process level and top level (only applicable with configuration methods, for example setting the time interval of a timer). The bridge between the hardware and software model is provided by the External Module Interface (*EMI*).
The relationship of the programming, execution, and *RT* level model is illustrated in figure **4**.
Algorithm **1** shows *ConPro* program code patterns providing module generators for the implementation of agent activity processes and transition queues. Signal handling and blocking *IO* or event-based statements create some overhead and special treatment in the processes. Pure computational agent activities are implemented with the process `p_comp` pattern. A infinite service loop (line 11) reads agent tokens from an in-coming queue (there exist only one for each activity process) and performs computations. Finally the transition to the next activity process is computed and the token passed to the appropriate outgoing queue (line 17-18). Pending agent signals are checked before computation is performed (line 14). If there is a signal, the control flow of the process is passed to line 21 by raising an exception. The agent token is passed to a signal handler (if there is one, else the signal is not checked and will be ignored). Event based activities which can block agent processing require additional treatment (`p_ev_io`, `p_ev_timer`). Because activity processes are shared by different agents, the process may not block. Thus after an agent token was read form the in-coming queue and the signal check has passed, the availability of data (line 43) is checked by using a non-blocking dictionary operation (`try_out`). If the requested data is available (matching tuple found in database), post activity computations can be performed (line 45) and finally the activity transition can be processed (lines 47-48). If the requested data is not available, the agent token is passed back in the in-coming queue (line 54) and the next agent token can be processed until there are no further unprocessed agents. In this case the control flow branches by exception to line 62 and waits for an *IO* event (raised by the agent manager if new data was stored in the dictionary).

Algorithm 1. *ConPro behavioural implementation patterns of agent activity processes (computational and event based) with sequential executing processes communicating with queues (incoming transition with queue q_{in}, outgoing transitions with queues q_{out1} and q_{out2} depending on conditional evaluation, q_s is signal handler process serving queue).*

```
1  open Event;
2  -- Pipelined processes model patterns --
3  exception Signal,Await;
4  queue qin,qout1,qout2,qs: int[8];
5  -- Gobal agent event signaled by agent manager
6  event agent with latch;
7
8  -- Pure computational activity process --
9  process p_comp: begin
10   reg t: int[8];
11   always do begin
12     try begin
13       t <- qin;
14       if check_signal(t) = true then raise Signal;
15       .. computations ..
16       -- state transitions --
17       if cond1 then qout1 <- t;
18       else if cond2 then qout2 <- t;
19     end
20     with begin
21       when Signal : qs <- t;
22     end;
23   end;
24 end;
25
26 -- IO events --
27 process p_ev_io: begin
28   reg first: bool;
29   reg t,t0: int[8];
30   select <- 1:
31   always do begin
32     try begin
33       t0 <- (-1), first <- true;
34       always do begin
35         t <- qin;
36         if check_signal(t) = true then raise Signal;
37         if t = t0 then begin
38           -- at beginning of queue; wait for IO event
39           qin <- t;
40           raise Await;
```

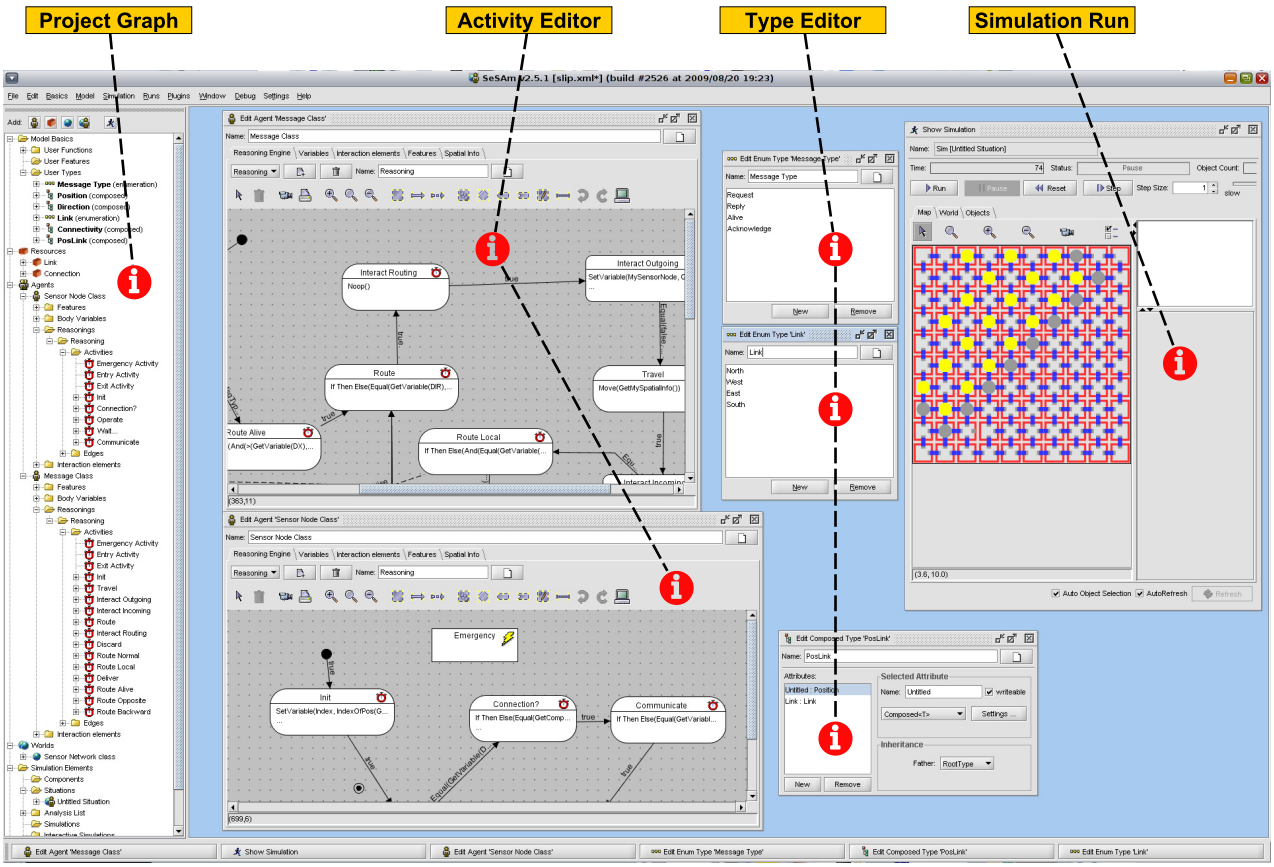
```
      end;
      {x,b} <- try_out(PATTERN);
      if b = true then begin
        .. post computations ..
        -- state transitions --
        if cond1 then q_out1 <- t;
        else if cond2 then q_out2 <- t;
      end
      else begin
        if first = true then begin
          t0 <- t,  first <- false; end;
        q_in <- t;
      end;
    end;
  end;
  with begin
    when Signal : q_s <- t;
    when Await : begin
      agent.await ();
      -- to avoid race conditions:
      -- signal event again for other waiters!
      agent.wakeup(); end;
  end;
 end;
end;

-- Timer interval delays (at least DELAY) --
process p_ev_timer: begin
 reg first: bool;
 reg t,tn: int[8];
 always do begin
  try begin
    always do begin
      t <- q_in;
      first <- true,t0 <- t,tn <- 0;
      -- count all agents actually waiting
      -- and perform pre computations
      while t0 <> t or first = true do begin
        if check_signal(t) = true then raise Signal;
        tn <- tn+1; first <- false;
        .. pre computations ..
        q_in <- t;
        t <- q_in;
      end;
      q_in <- t;
      wait for DELAY;
      for i = 1 to tn do begin
        t <- q_in;
        if check_signal(t) = true then raise Signal;
        .. post computations ..
        -- state transitions --
        if cond1 then q_out1 <- t;
        else if cond2 then q_out2 <- t;
      end;
    end;
  end;
  with begin
      when Signal : q_s <- t;
    end;
  end;
end;

process manager: begin
  -- start agent t in state x --
  q_x <- t;
end;
```

7. Multi-Agent Simulation for Functional Testing

Figure 5. *Simulation environment* ***SeSAm*** *for time-discrete behavioural simulation of state-based multi agent systems.*



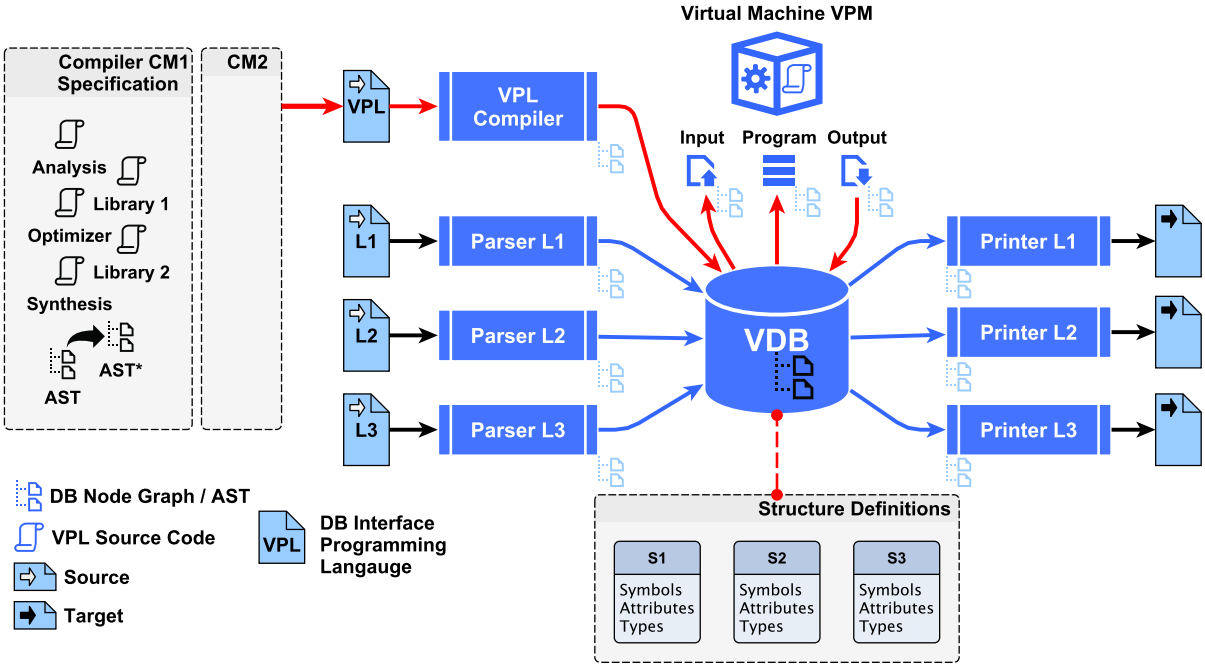
Simulation of multi-agent systems can support rapid testing of multi-agent systems on functional behaviour level, especially concerning concurrent agent processing issues like race conditions and deadlocks. The ***APL*** compiler part of the ***SynDK*** environment is able to synthesize behavioural simulation models (***XML***) which can be imported in the multi-agent simulation framework ***SeSAm*** **[9]**, shown in Fig. **5** with a multi-agent simulation from the case study discussed in Section **9**.

8. SynDK: The Synthesis Development Kit

In general a synthesis compiler transforms a set of source input languages ***IPL*** into a set of target output languages ***OPL***. The development process of compilers and synthesis tools involves always the same design steps: definition of languages, specification and implementation of lexers and parsers (input), formatted printers (output), internal structure and data representations (syntax trees, intermediate representations,..), code analyzers, code transformers, and finally code synthesizer. The design of compilers for high-level languages is complex and error prone. Using traditional monolithic compiler approaches makes debugging of the compiler operation very difficult.

A graph-based virtual database driven hardware and software synthesis approach (***VDB***) should overcome limitations in traditional compiler design and enables common synthesis from a set of source programming models and languages ***IPL*** to a set of destination models and languages ***OPL*** like hardware behaviour models with parsers translating text to graph structured database content and printers printing text from database content.

Figure 6. *System architecture of the virtual database (**VDB**) driven synthesis development kit **SynDK** 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 Virtual Data Base framework, shown in Fig. **6**, is used to manage large structured data sets in ***CAD*** (synthesis) systems, like abstract syntax trees (***AST***). The basic database element is an i-node (internal node) which can be linked with other nodes providing a graph structure, like in hierarchical filesystems. There are different representation levels of content structures like filesystem-like path and i-node level. The ***VDB*** i-node level maps the elements of graph structured content to an i-node based filesystem with paths (assuming i-nodes have names). Each node of an ***AST*** is mapped to an i-node database element. I-nodes can be searched and retrieved either by their unique i-node number or hierarchically by using paths with regular expressions. An i-node can represent any kind of structure type encapsulating the structure elements and directories with arbitrary typed child content.

An i-node consists of an unique identifier (`Node #`), an unique structure type (`Type`), an optional name (`NAME`), an attribute table (`ATTR`) holding attributes of this database element, and a row table (`ROWS`) containing the i-node content holding data and links to the childs of this database element. Linking with child elements create hierarchical orders of i-nodes and the mapping of i-nodes to the graph structure, shown in Fig. **7**.

Though a database element is generic, typing and structure is constrained by using a structure type definition (***STD***), defining the types of database content elements and the order they may appear in the database. A structure type element can contain child elements and attributes related with this element. These childs can be specified with repeating lists, (ordered or unordered) sub-structures (product types), and disjunction lists (sum types). A simple ***STD*** for expressions is shown in Ex. **1**. There are types (commonly i-node types), attributes (commonly appearing in attribute tables) and symbols (symbolic values of attributes).

Figure 7. *Database node element (i-node) structure and linking of database elements to a directed graph by using node links (directed red node edges).*

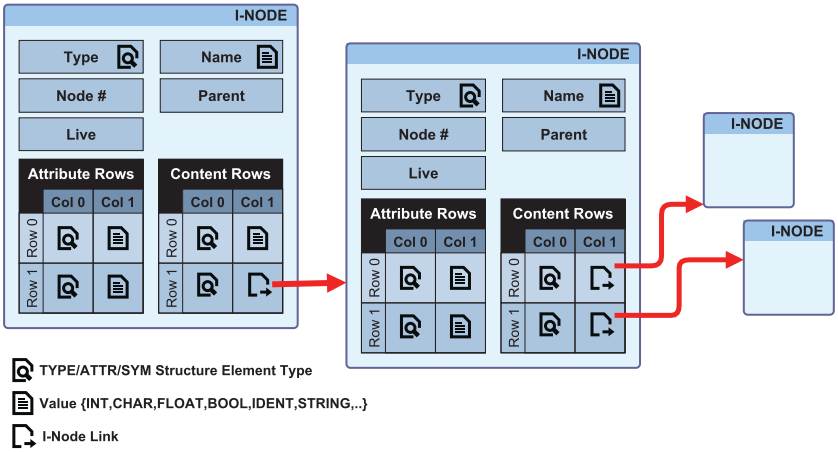
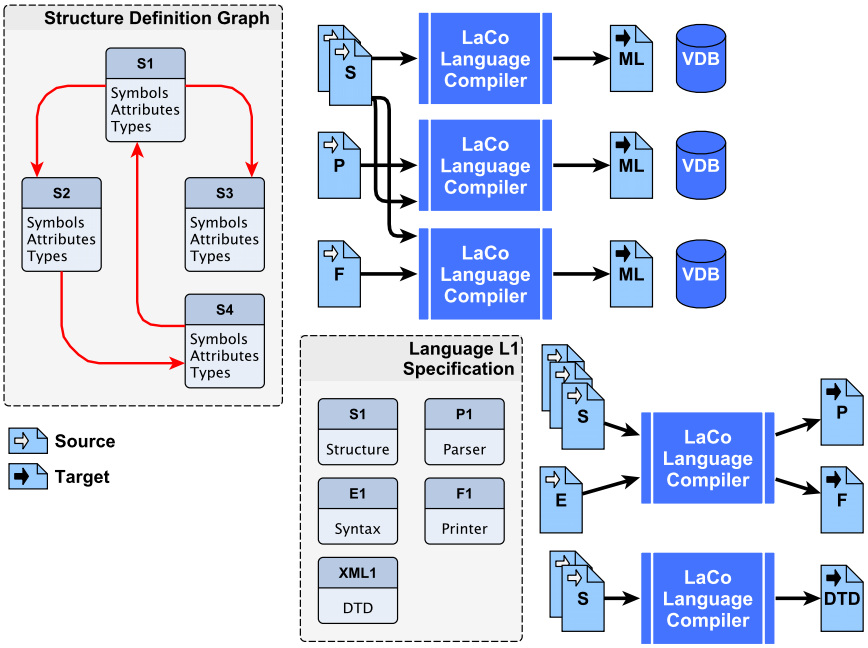


Figure 8. *Language Compiler* ***LaCo*** *transforming language descriptions into parser, lexer, and formatted printer code (ML).*



Example 1. *Structure Type Definition for simple expressions using regular expressions (shortened S language, see below)*

```
1  STRUCTURE EXP;
2  TYPES:  element := IDENT,(INT|FLOAT)?;
3          expr [operator] := (element|expr|value)+;
4          value :=  INT|FLOAT;
5  ATTRIBUTES: operator := add|sub|mul|div;
6  SYMBOLS: add; sub; mul; div;
```

The virtual database (stored in memory) can be saved to and loaded from a file. It is a complete and generic interchange format which can be used by different programs at different runtime states. It can represent the

complete state of a *CAD* program.
There are different specification languages used to define the structure type definition *STD* and for generating different compiler software modules linked to the *VDB* core system like parsers and lexers (formatted input) or printers (formatted output) handling content. There is a structure type definition language *S*, a parser and lexer generator language *P*, and a formatted printer language *F*. They are all part of the language compiler system *LaCo*, shown in Fig. **8**, transforming languages and creating programcode.
The structure type definition graph can be constructed from different sub-structures, shown in Fig. **8**. For example there are structures common to all languages like expressions (*EXP*), symbols (symbol tables, *SYM*), instructions (*INSTR*) like branches or loops, and many more.
A language is defined by a syntax (*EBNF*, *E* language) mapped by parsers to this structure type definition *STD*. Each language can be imported from or exported to a plain text file by using specified lexers, parsers, and printers. Alternatively, database content can be imported or exported using universal *XML* representations only requiring the structure type definition *STD*.
This database driven approach enables *Syntax Directed Synthesis* (*SDS*) providing strong correlation of source code with synthesized hardware components and software instructions on different fine grained levels. *SDS* enables enhanced debugging features and a considerable speed-up of the design process of the compilers under development. In general *SDS* translates an input *AST* to several intermediate *AST*s and finally to a destination *AST*, e.g. *SDS:* $AST_1 \rightarrow AST_2 \rightarrow \ldots \rightarrow AST_n$. Transformation results can be linked directly to the original *AST* nodes generated by the parser. For example an expression `(x+1)*2` represented by a sub-graph in the *AST* is synthesized to a hardware block on *RTL*. This synthesized *RTL* can be linked to the original expression node correlating input and output of the compilation.
There is an *object-orientated database interface programming language VPL* enabling the implementation of all parts of a compiler (except parsers and printers, which are part of the core *VDB* system for performance reasons). In addition to objects *VPL* supports variables and traditional imperative statements like assignments, branches, loops, exceptions, and functions. *VPL* provides direct access to the database by using paths - used for query and modification - and constructors used for creating database content. *VPL* source code is parsed and stored directly in the database in an abstract syntax tree preserving the program structure. The *VPL* compiler operates directly on the *AST*, transforms the source tree to a compiled tree *AST** (for example by adding symbol tables, linking of elements with symbols), executed finally directly by a virtual machine *VPM*, shown in Fig. **6**. The data space of a *VPL* program including stacks is stored in the database, too. Usage of row, attribute, and i-node constructors and paths for accessing the database content is shown in Ex. **2**.

Example 2. *VPL program example fragment demonstrating the usage of paths and constructors providing a programming interface for VDB access and manipulation (assuming EXP structure type definition, see Ex. above)*

```
1  var i,j,a,res; --variables of gen. type
2  a := [EXP.operator:EXP.add]; -- row list constr.
3  i := EXP.element "x" [] [Int 2]; --  node constr.
4  j := EXP.expr [a] [i;EXP.value [] [Int 1]];
5  append "/" j;  -- add node j to root directory
6  if j->ATTR->EXP.op = EXP.add then
7    res :=  j->ROWS.[1]->VALUE+
8            j->ROWS.[2]->VALUE;  -- arith. comp.
9  end;
10 for n in j->EXP.element do -- all rows match. elem.
11   if n->NAME = "x" then
12      n->NAME := "y";
13   end;
14 end;
15 print res;
```

9. Case Study: Smart Communication Agent

The main problem in message-based communication is routing and thus addressing of nodes. Absolute and unique addressing of nodes in a high-density sensor network is not suitable. An alternative routing strategy is delta-distance routing, used in the message deliver agent those ***APL*** specification is shown in Alg. **2**. First the normal XY routing is tried (activity `route_normal`, lines 29-55), where the packet is routed in each direction one after another with the goal to minimize the delta count `{dx,dy}` of each particular direction. If this is not possible (due to missing connectivity), the packet is tried to send to the opposite direction (activity `route_opposite`, lines 57-71), marked in the gamma entry part of the agent data. Opposite routing is used to escape small area traps, backward routing is used to escape large area traps or to send the packet back to the source node (message not deliverable, lines 73-95). The routing decision is based on the actual agent data `{dx,dy,gamma_x/y,dir}` and achieves adaptive routing reflecting the actual network topology and the path the message already had travelled, including back-end traps, resulting in alternative paths by choosing different routing directions.

This agent behaviour specification was synthesized to the ***ConPro*** multi-process model (intermediate representation) and finally to a hardware behaviour model (***RTL*** architecture, ***VHDL***). The design includes activities, transition computations, an agent manager, and communication including four serial links connecting to other nodes in the network. On ***ConPro*** level the design is partitioned into 27 sequential processes, 6 functions, 8 queues, and 17 RAM blocks. Gate-level FPGA synthesis (Xilinx ISE 9.2i) with a Xilinx Spartan 3 Target (XC3S1000) requires 7279 4-input LUTs (47% device utilization), 1909 FLIP-FLOPs (12%), and 18 block RAMs (75%). The estimated maximal clock frequency of the design is about 90 MHz. This agent implementation and the processing architecture fit well in common FPGA or ASIC architectures.

Algorithm 2. APL *agent class* `SmartMessage` *implementing a message transfer agent with smart routing rules providing robustness and a high degree of reliable communication in a mesh-like network with possible link failures, missing connections or nodes (based on* ***SLIP*** *protocol router algorithm).*

```
1  const ASC := 16;
2  const DSC := 8;
3  type DIMENSIONS = {X,Y};
4  type message = record
5    dx0, dy0: integer [-ASC/2..ASC/2-1];
6    len: integer [-DSC/2..DSC/2-1];
7    data: text;
8  end;
9
10 agent SmartMessage(m:message) =
11   var dx,dy: integer [-ASC/2..ASC/2-1];
12       gamma_x,gamma_y: integer[-1..1];
13       dir: DIR;
14       processed, reverse: bool;
15
16   activity start =
17     dx := m.dx0; dy := m.dy0;
18     gamma_x := 0; gamma_y := 0; dir := 0;
19     processed := false;
20   end;
21
22   activity deliver =
23     if dx = 0 and dy = 0 then
24       in(m.text);
25       processed := true;
26     end;
27   end;
28
29 activity route_normal =
30   for dim in {X,Y} do
31     case dim of
32     | X =>
33       reverse := gamma_x <> 0 and dx <> m.dx;
34       if dx < 0 and dir <> WEST and
35          not reverse and link?(EAST) then
36          dx := dx +1;  dir := EAST; gamma_y := 0;
37          processed <- true;
38       elsif dx > 0 and dir <> WEST and gamma_x = 0
39          and link?(WEST) then
40          dx := dx - 1; dir := WEST; gamma_y := 0;
41          processed := true;
42       end;
43     | Y =>
44       reverse := gamma_y <> 0 and dy <> m.dy;
45       if dy < 0 and dir <> NORTH and
46          not reverse and link?(NORTH) then
47          dy := dy +1;  dir := NORTH; gamma_x := 0;
48          processed := true;
49       elsif dy > 0 and dir <> SOUTH and
50          gamma_y = 0 and link?(SOUTH) then
51          dy := dy - 1; dir := SOUTH; gamma_x := 0;
52          processed := true;
53       end;
54   end;
```

```
55  end;
56
57  activity route_opposite =
58    for dim in {X,Y}  do
59      case dim of
60      | X =>
61       if gamma_x = 0 then
62          if  dir <> EAST  and link?(EAST) then
63            dx := dx +1;  dir := EAST; gamma_x := -1;
64            processed := true;
65          elsif dir <> WEST and link?(WEST) then
66            dx := dx - 1; dir := WEST; gamma_x := 1;
67            processed := true; end;
68       end;
69      | Y => ...
70    end;
71  end;
72
73  activity route_backward =
74    for dim in {X,Y} do
75      case dim of
76      | X =>
77       if gamma_x = 0 then
78          if  (dir = EAST or dir = NORTH)
79               and link?(EAST) then
80            dx := dx +1;  dir := EAST; gamma_x := -1;
81            processed := true;
82          elsif link?(WEST) then
83            dx := dx - 1; dir := WEST; gamma_x := 1;
84            processed := true; end;
85       else
86         if gamma_x = -1 and link?(EAST) then
87            dx := dx +1;  dir := EAST;
88            processed := true;
89         elsif gamma_x = 1 and link?(WEST) then
90            dx := dx - 1; dir := WEST;
91            processed := true;
92       end;
93      | Y => ...
94    end;
95  end;
96
97  activity migrate =
98    moveto(dir);
99  end;
100
101 activity discard =
102   kill(ME);
103 end;
104
105 transitions =
106   start -> deliver;
107   deliver -> discard: processed;
108   deliver -> route_normal: not processed;
109   route_normal -> route_opposite:
110                                 not processed;
111   route_opposite -> route_backward:
112                                 not processed;
113   route_backward -> discard:
114                                 not processed;
115   route_normal -> migrate: processed;
116   route_opposite -> migrate: processed;
117   route_backward -> migrate: processed;
118   migrate -> deliver;
119 end;
120 end;
```

10. Summary

A novel design paradigm using mobile agents for reliable distributed data processing in networks with nodes constrained by low computational resources was introduced. An agent programming language ***APL*** provides computational statements and simple statements for agent mobility, interaction, and information exchange. This programming model ease the implementation of common signal processing algorithms and can be directly synthesized to micro-chip level by using an database driven high-level synthesis approach and a pipelined process model enabling parallel agent processing with resource sharing. A case study demonstrated the suitability of the proposed programming, processing architecture, and synthesis approach. Migration of agents require the transfer of the data space of an agent with messages. The high-level synthesis tool enables the the synthesis of different output models from a common programming source, including hardware, software, and simulation models providing advanced design features for functional testing.

11. References

[1] 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.

[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] C. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985

[4] S. Bosse, F. Pantke, *Distributed computing and reliable communication in sensor networks using multi-agent systems*, Journal of Production Engineering, Research and Development, Springer, 2012, ISSN 0944-6524, Prod. Eng. Res.

Devel., DOI 10.1007/s11740-012-0420-8

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

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

[7] H. Peine and T. Stolpmann, *The Architecture of the Ara Platform for Mobile Agents*, MA '97 Proceedings of the First International Workshop on Mobile Agents,Springer-Verlag London, 1997

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

[9] F. Klügel, *The Multi-Agent Simulation Environment SeSAm*, In: H. Kleine Büning (Ed.): Proceedings of Workshop "Simulation in Knowledge-based Systems", Paderborn, April 1998

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

[11] S. Napagao, B. Auffarth, N. Ramirez, *Agent Language Analysis: 3-APL*, 2007, (pp. 1-14), retrieved from http://www-lehre.inf.uos.de/~bauffart/mas_3apl.pdf

[12] 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). IEEE. doi:10.1109/VLSISoC.2011.6081593

[13] A. D. Stefano, C. Santoro, *Using the Erlang language for multi-agent systems implementation*, 2005, IEEE-WICACM International Conference on Intelligent Agent Technology (pp. 679-685). doi:10.1109/IAT.2005.141

[14] 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.