Chapter 6

PCSP: The Reconfigurable Application-specific Agent Platform

Agent Processing Platform based on a Parallel Pipelined Communicating Sequential Processes Architecture

In this Chapter an application-specific and reconfigurable agent platform is introduced that is capable of executing *AAPL* based agents. The platform architecture relies on a pipelined processes model and token-based agent processing. Various implementations of the platform are presented and discussed, supporting agent processing and mobility in heterogeneous networks environments.

6.1 Pipelined Processes

The Communicating Sequential Processes (CSP) Model was originally proposed by C. Hoare (1985) and is synchronized parallel processing model based on interactions and fundamental algebraic laws. The CSP model can be directly mapped on hardware implementations using a Multi Finite-State Machine and Register-Transfer Level architecture (FSM-RTL).

In the CSP model there is a set of processes P={p_1, p_2, ..} executing statements sequentially. The set of processes is executed in parallel. Processes can be started and stopped any time at run-time (by other processes). Inter-process communication (IPC) is performed by using queues. The original CSP model was extended with concurrent access of global shared objects (queues).

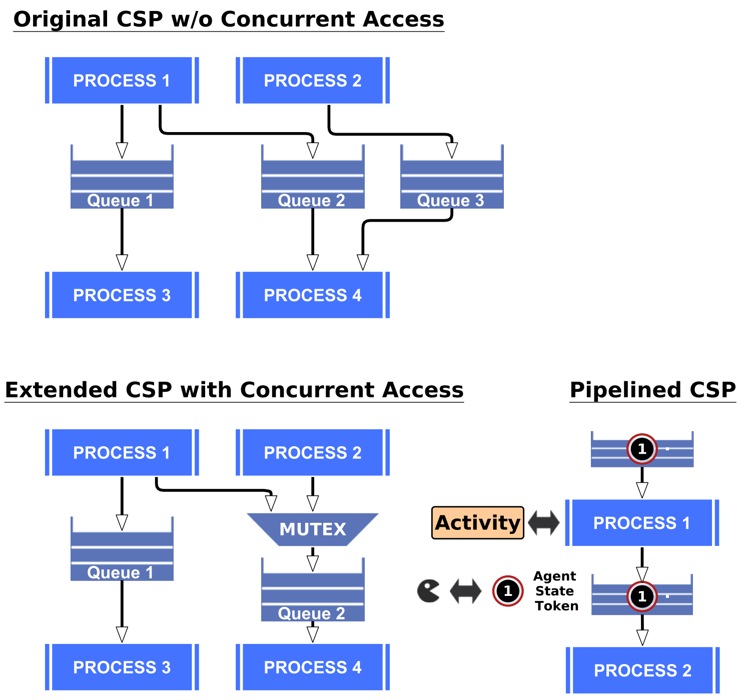

Fig. 6.1 *(Top) CSP Model (Bottom) CCSP Model, IPC Architecture and pipelined CSP with token-based Agents*

Competition is resolved by a mutual exclusion (Mutex) scheduler (atomic guarded actions), discussed in Section 5. A set of multiple processes can be connected by using queues, transferring execution unit tokens (related to data of operational processes created dynamically at run-time), introducing a pipelined token-based processing architecture, shown in Figure 6.1. Each queue has exactly one output port that is connected to a process, and an arbitrary number of input ports coming from other processes. The order of the execution of the tokens is arbitrary and may not affect the result of the parallel data processing.

6.2 Agent Platform Architecture

The *AAPL* model is a common source for the implementation of agent processing on hardware, software, and simulation processing platforms. With the database driven high-level synthesis approach introduced in Section 7 it is possible to map the agent behaviour to these different platforms. The agent processing architecture required on each network node must implement different agent classes and must be 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.

6.2.1 Token-based Agent Processing and Petri-Nets

Towards the implementation of the (hardware) agent processing platform and for the ATG analysis the ATG is transformed in a State-Transition (ST) ***Petri-Net*** (PN), shown in Figure 6.2.

Activities are mapped on ***states*** of the PN, conditional transition expressions and transition scheduling are merged with the activity states!

Agents are represented by tokens passed by transitions between states of the PN. The token-based approach originates in the CSP model, discussed in Section 5.5, pipe-lining processes by channels. *Only one token can be consumed by an activity state at any time.* There exist exactly one token for each agent, either moving within the agent processing Petri network, or temporarily consumed by an external agent manager. Signal handler are related to their own and independent Petri-Net.

The only existing connection between the agent activity processing and the signal handler Petri-nets are the body variables of an agent, shared by the Petri-Nets (see Figure 6.2, lower right corner), introducing concurrency. This requires atomic operations applied to the shared variables. But those shared variables should usually only read in transitions, and modified by signal handlers (except the initialization of these variables in activities), resolving any competition and race conditions.

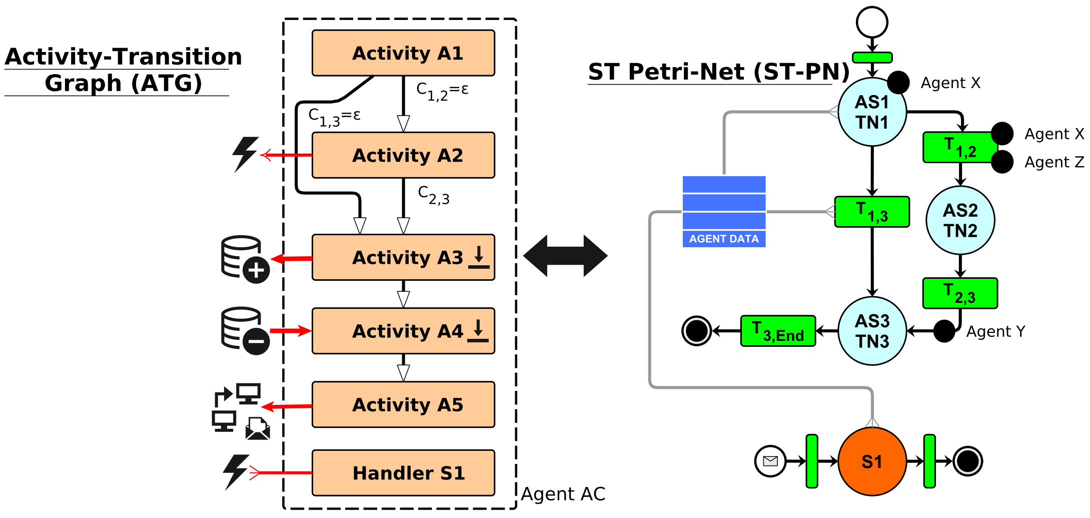

Fig. 6.2 *Transformation from the agent ATG behaviour to ST-PN model*

An activity is treated here as a sequential execution unit performing computation and interaction with the environment. The execution of an activity is started by a transition and an agent token, which is bound the currently processed activity. There can be multiple different agents consumed by a satisfied transition, acting as a queue.

6.2.2 The (R)PCSP Agent Platform

This processing platform - very well matching microchip-level designs - implements the agent behaviour with *reconfigurable pipelined communicating processes* (*RPCSP*), in short form abbreviated with *PCSP*, related to original the Communicating Sequential Process model (CSP) proposed by Hoare (1985). The activities and transitions of the *AAPL* programming model are merged in a first intermediate representation by using state-transition Petri Nets (PN), shown in Figure 6.3.

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 class.

This *PN* representation allows the following *CSP* derivation specifying the process and communication network, and advanced analysis like deadlock detection. Timed Petri-Nets can be used to calculate computational time bounds to support real-time processing.

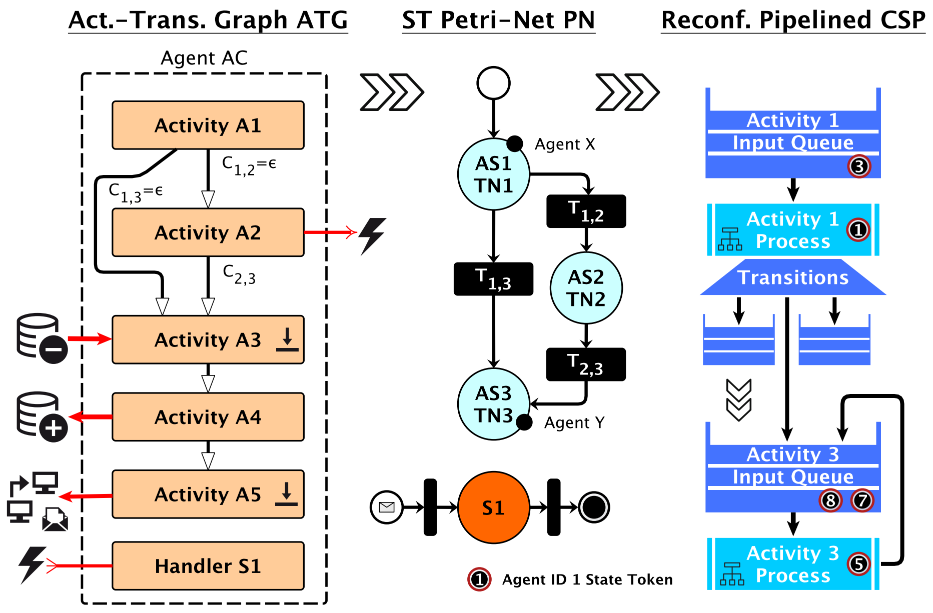

Fig. 6.3 *Pipelined Communicating Sequential Processes Architecture derived from a Petri-Net specification and relationship to the activity-transition graph. Signals are handled asynchronously and independently of the activity processing.*

Keeping the PN representation in mind, the set of activities {A_1, A_2, A_3, ..} is mapped on a set of sequential processes {P_1, P_2, P_3, ..} executed concurrently. Each subset of transitions {$T_{a,j}$, $T_{b,j}$, $T_{c,j}$,..} activating one common activity process P_j is mapped on a synchronous and multiplexed n:1 queue Q_j providing inter-activity-process communication, and the computational part of the transitions are embedded in all contributing processes {P_a, P_b, ..}, shown in Figure 6.3. Changes (i.e., reconfiguration) of the transition network at run-time are supported by transition tables, shown in Figure 6.4. Body variables of agents are stored in an indexed table set. Activity processes are partitioned in 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 parallelized agent processing with only one activity process chain implementation required for each agent class. The hardware resource requirement (digital logic) is divided into a control and a data part.

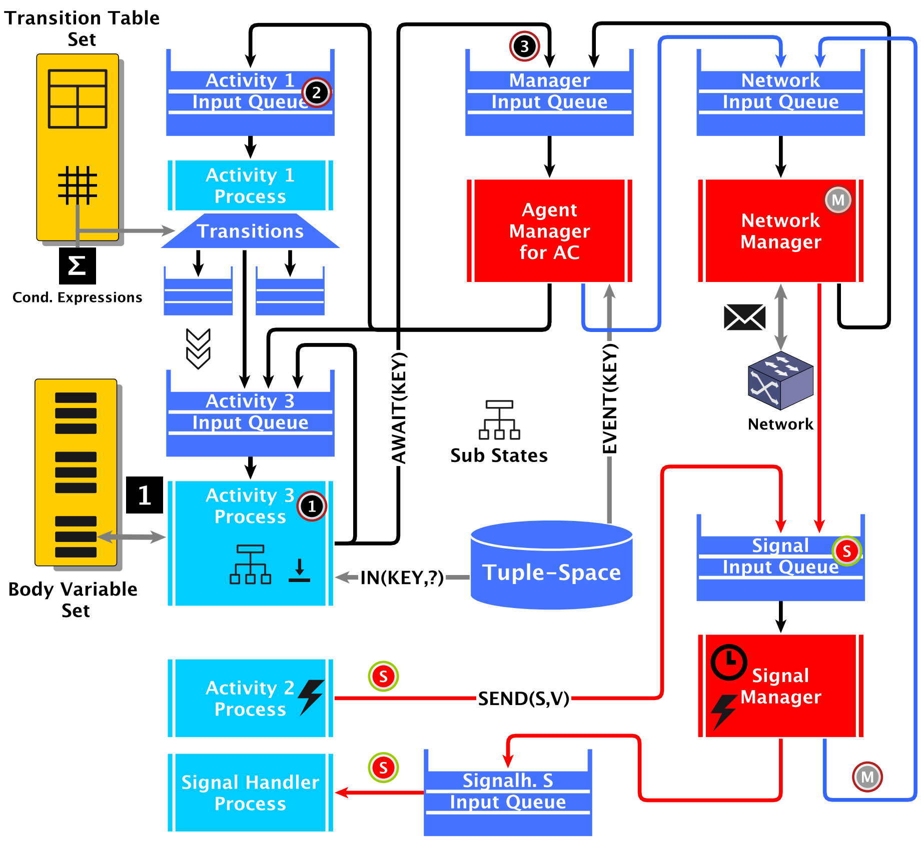

Fig. 6.4 *Interaction of the agent, signal, and network manager with activity processes*

Token-based Processing

As already introduced, agents are represented by tokens (natural numbers associated with the agent identifier that is unique on node level), which are transferred by the queues between activity processes depending on the specified transition conditions and the enabling of transition by the transition tables, shown in Figure 6.5.

This multiprocess model can be directly mapped on 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 is stored in a region of a memory module assigned to each individual agent.

There is only one incoming transition queue for each process consuming tokens, performing processing, and finally passing tokens to outgoing queues, which can depend on conditional expressions and body variables.

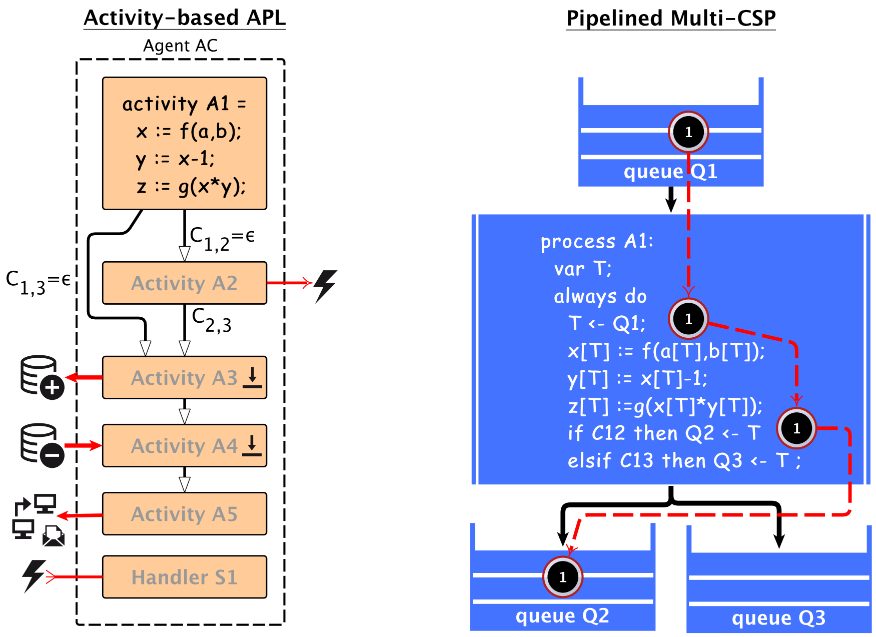

Fig. 6.5 *Mapping of pure computational agent activities to sequential processes*

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 that 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 I/O event-based Processing

To handle I/O-event and migration related blocking of statements, activity processes executing these statements are partitioned in 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 that 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 processing, shown in Figure 6.6.

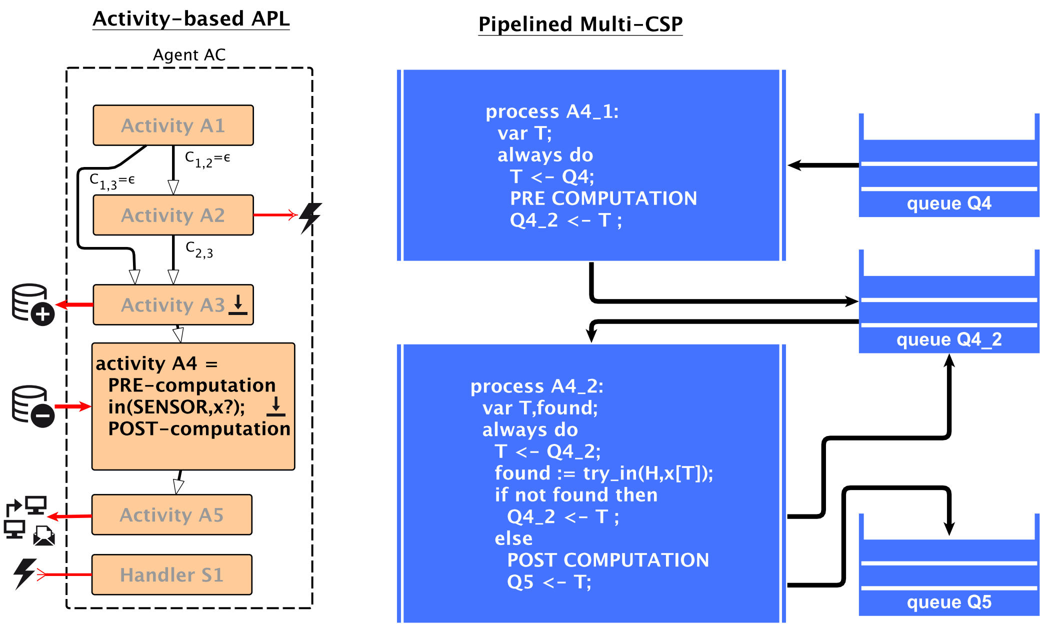

Fig. 6.6 *Mapping of mixed IO/computational agent activities on sequential processes*

Usually I/O events are related to a 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).

An example for sub-state partitioning of an agent ATG is shown in Example 6.1.

Ex. 6.1 *Sub-state partitioning of an agent ATG behaviour description that decomposes the activities in computational and event parts that can block the agent process.*

```
activity init =
  init₁: dx := 0; dy := 0; h := 0; → init₂
  init₂: if dir <> ORIGIN then
           moveto(dir); ⊥ init₃
           init₃: 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
```

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 (AM) provides a node level interface for agents, and it is responsible for the creation, control (including signals, events, and transition network configuration), and migration of agents with network connectivity, implementing a main part of an operating system. The agent manager controls the tuple-space database server and signal events required for IO/event based activity processes.

The agent manager uses agent tables and caches to store information about created, migrated, and passed through agents (required, for example, for signal propagation), see Figure 6.7.

Agents are *identified* node-locally by a unique local agent handler (*LAH*), which is a slot number in the local *agent table* (see Figure 6.7) 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.

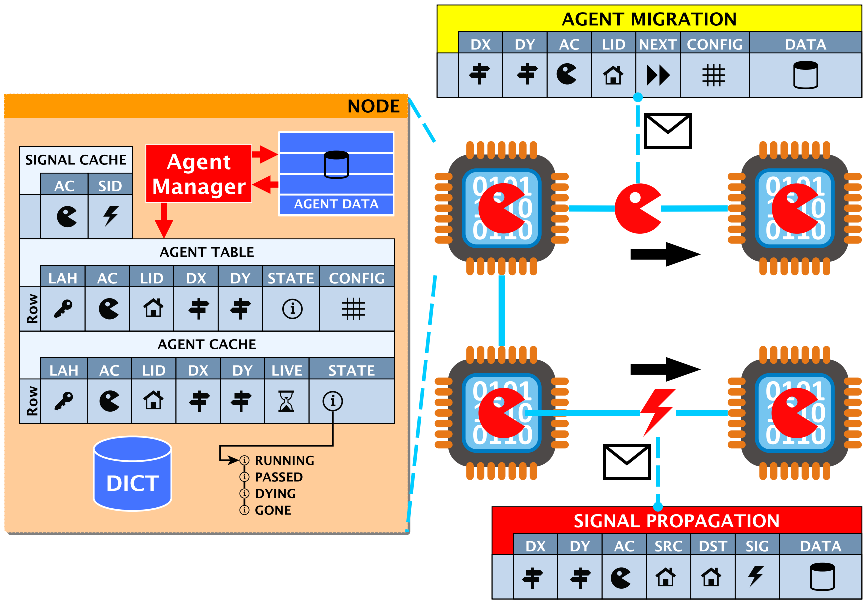

Fig. 6.7 *Each node has an agent manager for agent control, migration, and signal handling. Global agent- and signal identifiers are used to identify agents uniquely without the necessity of global unique node identifiers (LAH: Local Agent Handler, LID: Local Agent Identifier, AC: Agent Class, LIVE: agent live, STATE: agent state, DX and DY: spatial displacement vector, SID: Signal Identifier, SIG: pending signal ID)*

Transitions and Reconfiguration

Each activity process has a final transition sub-state, which tests for enabled transitions in the current context. If an enabled transition condition is true, the agent token is passed to the respective destination queue.

Modification of the transition network 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 subclasses.

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

Migration

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 that 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.

Signal Processing

Signals are handled asynchronously by activating signal handlers, implemented by a process and a signal handler queue. The signal manager is responsible for the creation and propagation of signals, shown in the bottom of Figure 6.4. Signal tokens are tuple values *(dst-id, src-id, signal, argument)*. *Signals* can be propagated encapsulated in a message from a source node to a destination node actually processing a specific agent by using the global signal identifier (*SID*), shown on the right side of Figure 6.7. Signals are unreliable because they depend on routing information stored in the agent cache. Due to the limited cache size older entries can be removed. Missing routing information or changes in the network topology (link failures or reconfiguration) prevent signal delivery.

Replication

Replication of activity processes sharing the same input queue offers advanced parallel processing of multiple agents for activities with high computation times.

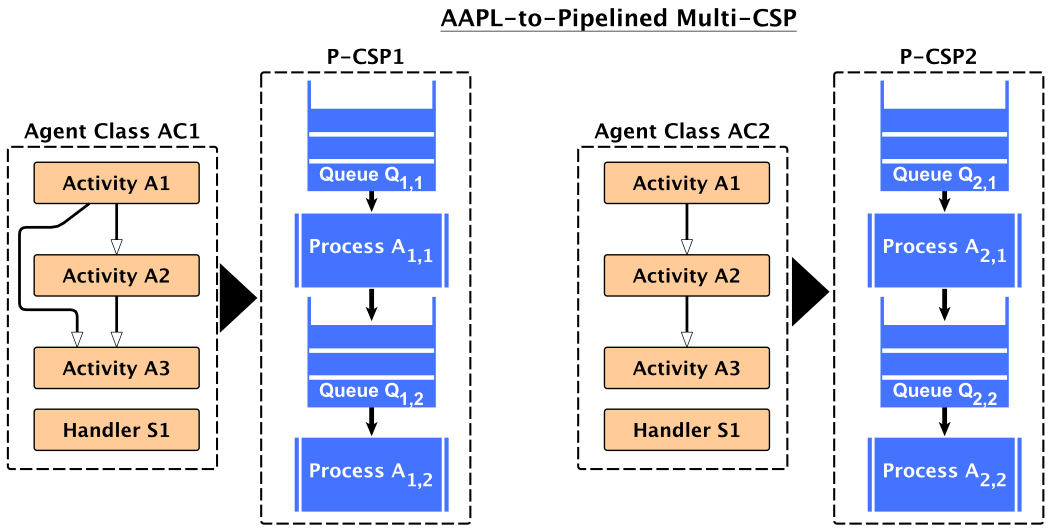

Fig. 6.8 *Each agent class is mapped on one PCSP*

Multiple Agent Behaviour Classes

Each Agent Class is mapped on one Pipelined-CSP, shown in Figure 6.8. Each *PCSP* is shared by a set of agents belonging to the same class.

Reconfiguration of the agent behaviour ATG only affects and modifies a transition table, with a set of possible transitions finally embedded in the activity transition selector.

6.2.3 Replication and Factoring

Timed Petri-Net analysis can be used to optimise the execution of multiple agents in the same activity state and belonging to one agent class *AC*. Processes of computational activities with high computation time can be

1. Factored and split into smaller units and processes with intermediate queues; and/or
2. Replicated to enable parallel agent processing;

to improve the overall pipeline throughput, shown in Figure 6.9.

6.2.4 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 on 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 multiprocessor 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.

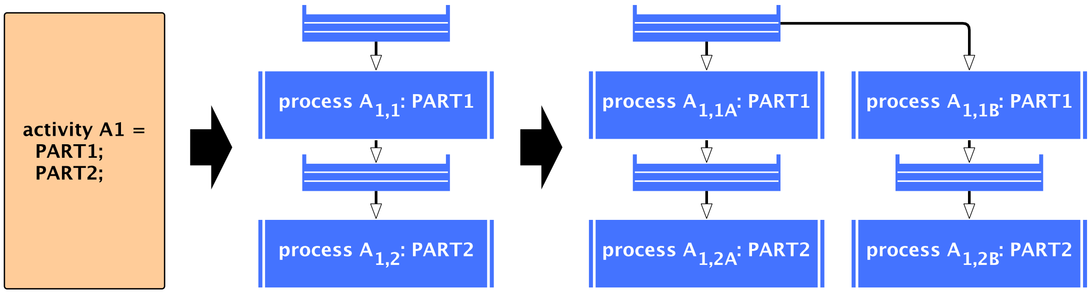
activity A1 = PART1; PART2;
process $A_{1,1}$: PART1
process $A_{1,2}$: PART2
process $A_{1,1A}$: PART1
process $A_{1,1B}$: PART1
process $A_{1,2A}$: PART2
process $A_{1,2B}$: PART2

Fig. 6.9 *Transformation of computational activity processes*

6.2.5 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 [KLU09] offers a platform for the modelling, simulation, and visualization of mobile multi-agent systems employed in a two-dimensional world. The behaviours of agents are modelled with activity graphs (specifying the agent reasoning machine) close to the *AAPL* model. Activity transitions depend on the evaluation of conditional expressions using agent variables. Agent variables can have a private or global (shared) scope. Basically *SeSAm* agent interaction is performed by modification and access of shared variables and resources (static agents).

Simulation of complex MAS on behavioural level and the methodology using the *SeSAm* simulator was already demonstrated in [BOS14B], mapping *AAPL* agents of the MAS one-to-one on *SeSAm* agents. The *RPCSP* agent processing platform simulation with the agent-based *SeSAm* simulation framework is discussed in detail in Section *11.4*. This simulation provides the testing and profiling of the proposed processing platform architecture in a distributed network world.

The simulator is also fully compatible to 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.

6.3 Agent Platform and Hardware Synthesis

The database driven synthesis flow consists of an *AAPL* front end, the core compiler, and several back-ends 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* high-level synthesis framework (details in Section 7), a software description (*C*) which can be embedded in application programs, and the *SeSAm* simulation model (*XML*). The *ConPro* programming model reflects an extended CSP with atomic guarded actions on shared resources. Each process is implemented with an FSM and an RT 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.

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.

RTL Architecture

The set of activities $\{A_i\}$ is mapped on a set of sequential processes $\{P_i\}$ executed concurrently. The set of transitions $\{T_i\}$ is mapped on a set of synchronous queues $\{Q_i\}$ and transition selectors $\{S_i\}$ embedded at end of the activity processes, providing the inter-activity-process communication. The PCSP multiprocess model is directly mappable on RTL hardware. Each process P_i is synthesized to a finite state machine FSM_i controlling the process execution and a register-transfer data path. Local agent data is stored in a region of a memory module assigned to each individual agent. There is only one incoming transition queue for each process consuming tokens, performing processing, and finally passing tokens to outgoing queues, which can depend on conditional expressions.

Agent Managers

*The agent, signal, and network manager*s provide a node level interface for agents, and are responsible for the creation, control (including signals, events, and transition network configuration), and migration of agents with network connectivity, implementing a main part of an operating system. The agent manager controls the tuple-space database server and signals events required for IO/event-based activity processes. The agent manager uses agent tables and caches to store information about created, migrated, and passed through agents (required, for example, for signal propagation), as shown in Figure 6.7.

Migration

Migration of agents requires the transfer of the agent data and the control state of the agent together with a unique global agent identifier (extending the local *ID* with the agent class and the relative displacement of its root node) encapsulated in messages. The control state consists of the next activity to be processed after migration and the current setting of the transition table. This approach minimizes network load and energy consumption significantly. Migration of simple agents results in a message size between 100-1000 bits. The agent start-up time after the data transfer is low (about some hundred clock cycles).

Transition Network

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

Tuple-Space Database

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

Signals

Signals must be processed asynchronously. Therefore, agent signal handlers are implemented with a separate activity process pipeline, one for each signal handler. For each pending agent signal, the signal manager injects a signal token in the respective handler process pipeline independent of the processing state of the agent. Remote signals are processed by the signal and network managers, which encapsulate signals in messages sent to the appropriate target node and agent.

Resources

A rough *estimation of the resource requirements* R for the hardware implementation of the agent processing architecture supporting a set of N different agent classes $\{AC_i\}$ is shown in Equation 6.1 [BOS14A], with each class having M_i activities, T_i transitions, D_i data cells with a resource weight w_{data}, and $w_{act,i,j}$ for each activity, and a maximal number of managed agents for each class $N_{agents,i}$.

The tuple space database requires $w_{ts,i}*S_i$ resources for each supported n-dimensional space. The C_x values are control parts independent of the above values.

For example, assuming simplified four agent classes with N =16 agents for each class, each class requires D=512 bit memory, M=10 (w_{act}=500), T=16, three tuple spaces (1,2,3) with S=32 (and w_1=32, w_2=64, w_3=128) entries each, and w_{data}=4, w_{queue}=150, w_{sched}=60, w_{cond}=50, w_{act}= 500, C_{sched}=5000, C_{comm}=10000, C_{ts}=1000 (based on experimental experiences, all *w* and *C* values in eq. gates units), which results in 189400 eq. gates for the HW implementation.

$$\begin{aligned} R \simeq \quad & (w_{data} \sum_{i \in AC} N_i^{agents} D_i) + \\ & C_{sched} + w_{sched} (\sum_{i \in AC} M_i) + w_{sched} \max(N_i^{agents}) + \\ & C_{comm} + (w_{queue} + w_{cond})(\sum_{i \in AC} T_i) + (\sum_{i \in AC} \sum_{j \in AT_i} w_{i,j}^{act}) + \\ & C_{ts} + (\sum_{i \in TS} w_i^{ts} S_i) \end{aligned} \tag{6.1}$$

Power/Efficiency

Agents are often heavy-weighted processing entities interpreted by software-based virtual machines. In contrast, in the proposed RTL architecture the agent behaviour is mapped on finite state machines and a data path with data word length scaling, offering minimized power- and resource requirements, both in the control and data path. Most activity statements are executed by the platform in one or two clock cycles! All commonly administrative parts like the agent manager, communication protocols, and the tuple-space database commonly part of an operating system are implemented in hardware, offering advanced computational power enabling low-frequency and low-power designs, well suited for energy-autonomous systems. Transition network changes can be performed within a few clock cycles.

Use-Case Example

A use-case implementing the Explorer SoMAS, which is presented in Section 9.2 and based on the *AAPL* model from Algorithm 9.1., with the *RPCSP* platform should demonstrate the suitability and scalability of the *RPCSP* platform for SoC microchip level designs. The synthesis results of the hardware implementation for one sensor node are shown in Table 6.1, which are in accordance with the previous made resource estimation. The *AAPL* specification was compiled to the *ConPro* programming model and synthesized to an RTL implementation creating *VHDL* models. Two different target technologies were synthesized by using gate-level synthesis: 1. FPGA, Xilinx XC3S1000 device target using Xilinx ISE 9.2 software, 2. ASIC standard cell LIS10K library using the Synopsys Design Compiler software. The agent processing architecture consisted of the activity process chain for the explorer and node agent, the agent manager, the tuple-space database (supporting two- and three-dimensional tuples with integer type values), and the communication unit.

AAPL & CP Synthesis	***FPGA/XC3S1000 Synthesis***	***ASIC LSI10K Synthesis***
AAPL Source: 200 lines	LUTs (4-input): 10826 (70 %)	Equation NAND Gates: 309502
CP Source: 1615 lines	FLIP-FLOPs: 2415 (15 %)	Comb. Gates: 95354
VHDL Source: 37171 lines	BLOCK RAMs: 19 (80 %)	Non-comb. Gates: 214148
CP Processes: 28	Max. Clock: 85 MHz	Chip area (180 nm Tech.): 7 mm^2
CP Queues: 14		

Tab. 6.1 *High-level and gate-level synthesis results for one sensor node* [BOS14A]

6.4 Platform Simulation

This section will summarize 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. Simulation of parallel and distributed systems is a challenge. Performance profiling and the detection of race conditions or dead locks are essential in the design of such systems, where the agent processing platform is a central part. Details of the simulation techniques can be found in Section *11.4*.

Behavioural simulation (see Section *11.2* for details) maps agents of the MAS directly and isomorph 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 agents themselves.

The *SeSAm* simulation framework offers the platform for the modelling, simulation, and visualization of mobile multi-agent systems employed in a two-dimensional world. The behaviours of agents are modelled with activity graphs (specifying the agent reasoning machine) close to the *AAPL* model. Activity transitions depend on the evaluation of conditional expressions using agent variables. Agent variables can have a private or global (shared) scope. Basically *SeSAm* agent interaction is performed by modification and access of shared variables and resources (static agents). In addition to the agent reasoning specification there are global visible feature packages that define variables and function operating on these variables. Features can be added to each agent class.

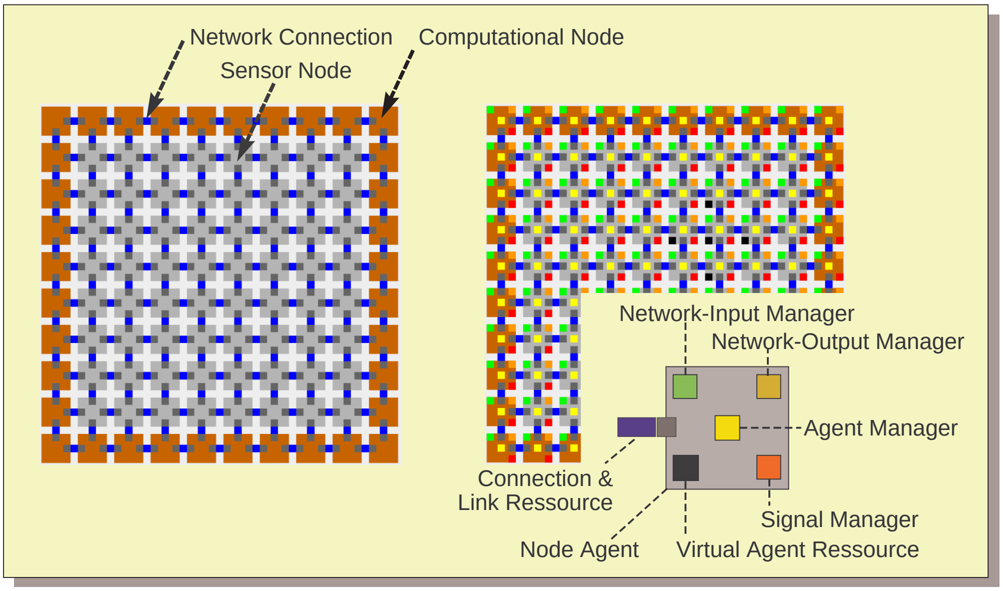

Fig. 6.10 *Simulation world of the sensor network (left) consisting of 10x10 nodes and the network populated with agents (right)*

Agents can change their position in the two-dimensional world map enabling mobility, and new agents can be created at run-time by other agents. The *SeSAm* framework was chosen due to the activity-based agent behaviour and the data model, which can be immediately synthesized from the common *AAPL* source and can be imported by the simulator from a text based file stored in *XML* format that can be compiled from the textual language *SEM* (see Section 11.5 for details). This model exchange feature allows the tight coupling of the simulator to the synthesis framework.

The simulation world is shown in Figure 6.10. The network is a two-dimensional mesh grid. Each node can be connected with four neighbours. Communication connections can be enabled and disabled.

Each node provides multiple stationary agents: a node agent, an agent manager agent, providing the administrative service of the *RPCSP* platform, a signal manager agent, and network manager agents.

In principle, *AAPL* activity graphs can be directly mapped on the *SeSAm* agent reasoning model. But there are limitations that inhibit the direct mapping. First of all, *AAPL* activities (IO/event-based) can block (suspend) the agent processing until an event occurs.

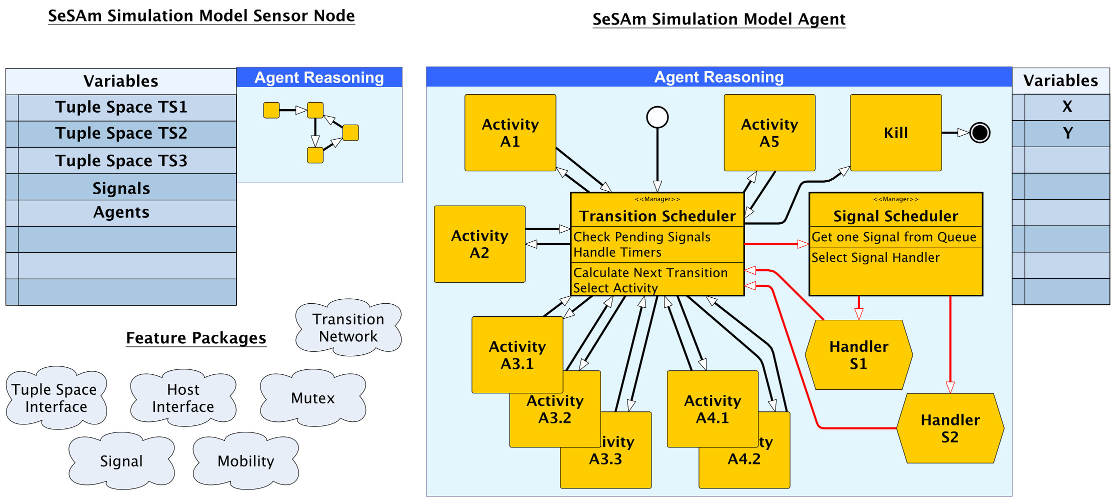

Fig. 6.11 *Simulation Model Architecture used in the SeSAm MAS Simulator for the simulation of AAPL agents with the RPCSP platform*

Blocking agent behaviour is not provided directly by SeSAm. Second, the transition network can change during run-time. Finally, the handling of concurrent asynchronous signals used in *AAPL* for inter-agent communication cannot be established with the generic activity processing in *SeSAm* (the provided exception handling is only used for exceptional termination of agents).

For this reason, the agent activity transitions including the dynamic transition network capability are managed by a special transition scheduler, shown in Figure 6.11. This transition scheduler handles signals and timers, too, which are processed prioritized and passed to the signal scheduler. Each agent activity is activated by the transition scheduler. After a specific activity was processed, the transition scheduler is activated and entered again. An *AAPL* activity can be split in computational and IO/event-based sub-activities in the presence of blocking statements (e.g., *in* and *rd* tuple space interaction).

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

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

6.5 Heterogeneous Networks

It was already pointed out that the hardware, software, and simulation platforms are equal on operational and interface level. Mobility of agents is provided by transferring the state of the agent, consisting of the data state (content values of all body variables) and the control state (next activity and the transition table). The agent behaviour and state machine is immobile and implemented in each node of the network. This feature enables the composition of heterogeneous networks using nodes from all of these platform classes based on inter-node communication with different connections technologies:

1. Sensor Networks and Hardware Platforms: Serial links with byte-stream protocols (aka. RS232)
2. Software Platforms: TCP/UDP/IP protocols
3. Simulation Platforms: TCP/UDP/IP protocols and socket interfaces

There is one common *APPL* behaviour and programming model that is synthesized to different platform classes, as shown in Figure 6.12.

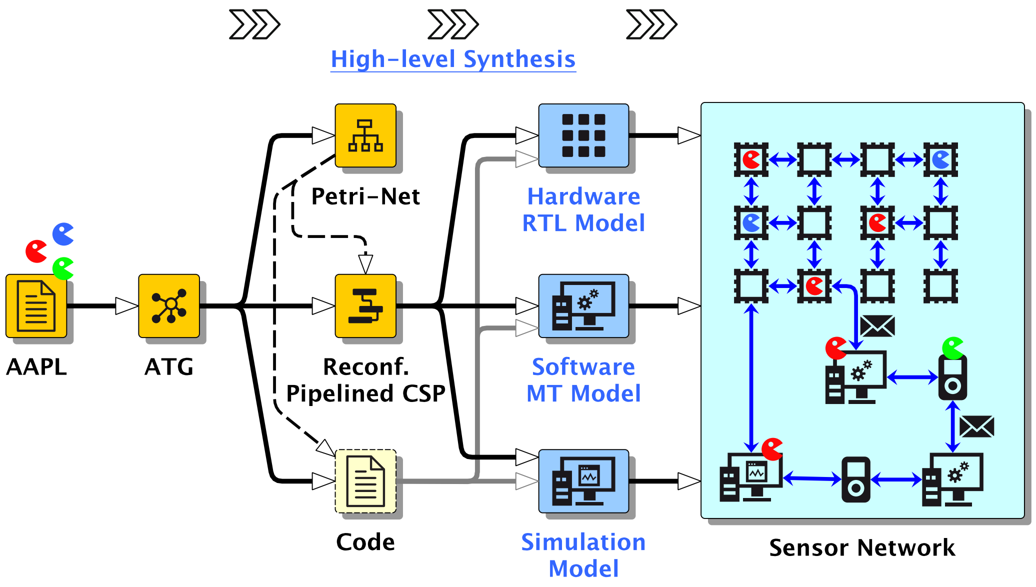

Fig. 6.12 *From one common AAPL programming level to heterogeneous distributed networks [RTL: Register-Transfer Level, MT: Multi-Threading, CSP: Communicating Sequential Processes]*

6.6 Further Reading

1. M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, 2013, ISBN 3642320260
2. M. Fingeroff, *High-Level Synthesis Blue Book*, Xlibris Corp. 2010
3. P. Arató (Autor), T. Visegrády, I. Jankovits, *High Level Synthesis of Pipelined Datapaths*, Wiley, 2001, ISBN 9780471495826
4. G. Ku, David C., DeMicheli, *High Level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic publishers, 1992, ISBN 9781475721171