Chapter 2

Agent Behaviour and Programming Model

Modelling and Programming Multi-Agent Systems

epubli, ISBN 9783746752228 (2018)

The growing complexity of computer networks and their heterogeneous composition of devices ranging from high-resources server to low-resource mobile devices demands for unified and standardized new data processing paradigms and methodologies, which can be solved by using the distributed Multi-agent computing paradigm. The Internet-of-Things is one major example rising in the past decade, strongly correlated with Cloud Computing and Big Data concepts. One of the early vision (i.e., Mark Weiser 1994) stated the population of the human living environment with distributed and widely connected computing systems performing tasks that support the people but which should be largely unaware of them. The integration of sensor networks providing perception is just an extension of this vision.

There are basically three different layers of data processing in sensing applications, which require processing platforms with different computational power and storage capacity:

Sensing
: Localized Acquisition and Pre-processing of Sensor Data, Sensor Data Fusion

Aggregation
: Distribution and Collection of Sensor Data, globalized Sensor Information Mapping and Sensor Data Fusion

Application
: Presentation of condensed Sensor Information, Storage, Visualization, Interaction.

These different layers can be scattered in different network domains. The sensing layer is usually located in sensor networks, for example, body area networks, the aggregation layer can be found in personal and ambient area networks (PAN/AAN), and finally the application layer can be found in ambient area and wide area networks (WAN).

The characterization of sensor network features and their operational capabilities can be further divided into the following classes and terms, handled by all three layers of the sensing application, shown in Figure *2.1*:

- Processing
- Communication
- Messaging
- Storage
- Ontologies and Data Models
- Manageability
- Security

epubli, ISBN 9783746752228 (2018)

The deployment of Mobile Multi-Agent systems offers a unified service capability covering all functional layers and sub-levels, ranging from sensing to application.

In this chapter, an activity- and state-based agent behaviour and multi-agent interaction model is introduced, finally leading to the Activity-Transition-Graph Agent Programming Model and Language *AAPL*. The *AAPL* model bases on dynamic activity-transition graphs. One of the main advantage of this behaviour model is the low dependency on the infrastructure where the agents are deployed and being processed, enabling the modelling and implementation of complex MAS with a unified and suitable programming language independent of the underlying agent process platform. The *AAPL* model supports traditional imperative computation, instantiation, mobility, behaviour modification, and inter-agent communication.

Furthermore, in this chapter the *AAPL* model is expressed by common calculi of mobile processes and graphical representations of the deployment of agents in arbitrary networked environments by using the Bigraph model. The relation of *AAPL* and the mapping of common agent models like BDI on *AAPL* is explained, showing that *AAPL* is the foundation for higher level agent and interaction models.

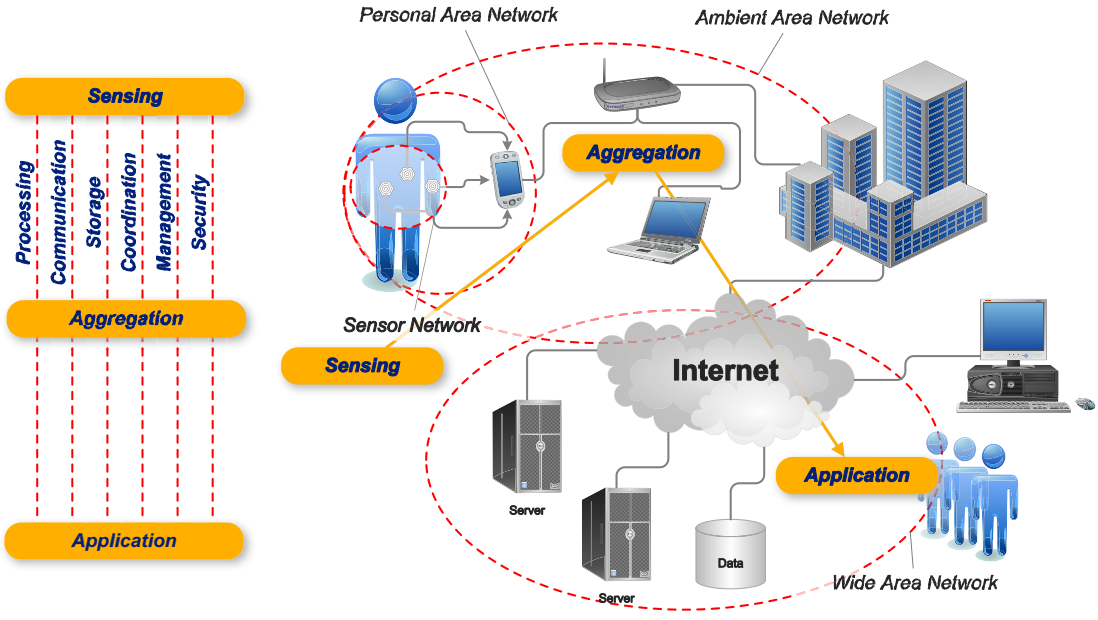


Fig. 2.1 *A heterogeneous network environment with PAN, AAN, and WAN domains (on network level) and sensing, aggregation, and application domains on the use-case level*

2.1 The Agent Computation and Interaction Model

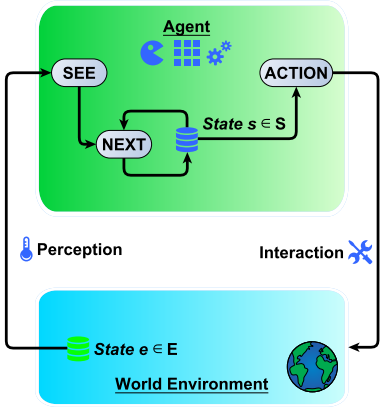
An agent can be considered as a computational unit situated in an environment and world, which performs computation, basically hidden for the environment, and interacts with the environment to exchange basically data. A common computer is specialized to the task of calculation, and interaction with other machines is encapsulated by calculation and performed traditionally by using messages. An agent behaviour can be reactive or proactive, and it has a social ability to communicate, cooperate, and negotiate with other agents. Pro-activeness is closely related to goal-directed behaviour including estimation and intentional capabilities.

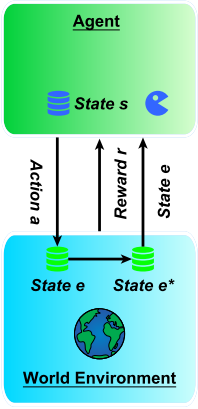
Agents record information about an environment state $e \in E$ and history $h: e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots$ Let $I = S \times D$ be the set of all internal states of the agent consisting of the set of control states S related to activities and internal data D. An agent's decision-making process is based on this information. There is a perception function *see* mapping environment states on perceptions, a function *next* mapping internal states and perceptions $p \in Per$ to internal states (state transition), and the action-selection function *action*, which maps internal states on actions $a \in Act$, shown in Definition 2.1.

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

Def. 2.1 *Agent processing and state change by applying three basic functions in a service loop*

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





The strategies can be used to modify the agent behaviour. Rewarded behaviour learning was addressed in [JUN12], for example, based on Q-learning.

The actions of agents modify the environment, which is seen by the agent, thus the agent is part of the environment. The change of the environment due to the agent actions can be effective immediately or delayed, evaluating the history and past. The actions of pure reactive agents base only on the current perception, and usually an action has an effect on the environment immediately.

An agent behaviour model can be partitioned into the following tasks, which must be reflected by an agent programming language model by providing suitable statements, types, and structures:

Computation

One of the main tasks and the basic action is computation of output data from input data and stored data (history). Principally functional and procedural (or object-orientated) programming models are suitable, but history incorporating computed data and storage is handled only by the procedural programming model consequently.

Communication

Communication as the main action serves two canonical goal tuples: (data exchange and synchronization), (interaction with the environment and with other agents). The latter goal tuple can be reduced to agent interaction only if the environment is handled by an agent, too. Communication between agents can link single agents to a Multi-agent system, by using peer-to-peer or group communication paradigms.

Mobility

Mobility of agents increases the perception and interaction environment significantly. Mobile agents can migrate from one computing environment to another finally continuing there their processing. The state of an agent, consisting of the control and data state, must be preserved on migration.

Reconfiguration

Traditional computing systems get a fixed behaviour and operational set at design time. Adaptation in the sense of behavioural reconfiguration of a system at run-time can significantly increase the reliability and efficiency of the tasks performed.

Replication

These are the methods to create new agents, either created from templates or by forking child agents, which inherit the behaviour and state of the parent agent, finally executing in parallel. Replication is one of the ma-

jor agent behaviours to compose distributed computational and reactive systems.

Agents and Objects

Modern data processing is often modelled based on object-orientated programming paradigms. But there is a significant difference between agents and objects. Objects are computational units encapsulating some state and are able to perform actions (by applying methods) and communicate commonly by message passing. Objects are related to object-orientated programming and are not (or less) autonomous in contrast to agents, and they are commonly immobile. The common object model has nothing common with proactive and social behaviour. But agents can be implemented on top of the object model with methods acting on objects. The modification of the behaviour engine is basically not supported by the object programming model. Agents decide for themselves, in contrast objects require external computational units, like operating systems or users, for the decision-making process. Though object-orientated programming can be extended by parallel and concurrent processing (multi-threading), multi-agent systems are inherently multi-threaded.

To summarize, agents are characterized by their autonomy (without or less intervention of users), reactivity, social ability, and pro-activeness.

2.2 Activity-Transition Graphs

The behaviour of an activity-based agent is characterized by an agent state, which is changed by activities. Activities perform perception, plan actions, and execute actions modifying the control and data state of the agent. Activities and transitions between activities are represented by an activity-transition graph (ATG). The transitions start activities commonly depending on the evaluation of agent data (body variables), representing the data state of the agent, as shown in Figure 2.2.

An activity-transition graph, related to the agent classes, discussed later, consists of a set of activities $\mathcal{A}=\{A_1,A_2,..\}$, and a set of transitions $\mathcal{T}=\{T_1(C_1),T_2(C_2),..\}$, which represent the edges of the directed graph. The execution of an activity, composed itself of a sequence of actions and computations, is related to achieving a sub-goal or a satisfying a prerequisite to achieve a particular goal, e.g., sensor data processing and distributions.

Usually agents are used to decompose complex tasks is simpler ones, based on the composition of MAS. Agents can change their behaviour based on learning and environmental changes, or by executing a particular sub-task with only a sub-set of the original agent behaviour.

epubli, ISBN 9783746752228 (2018)

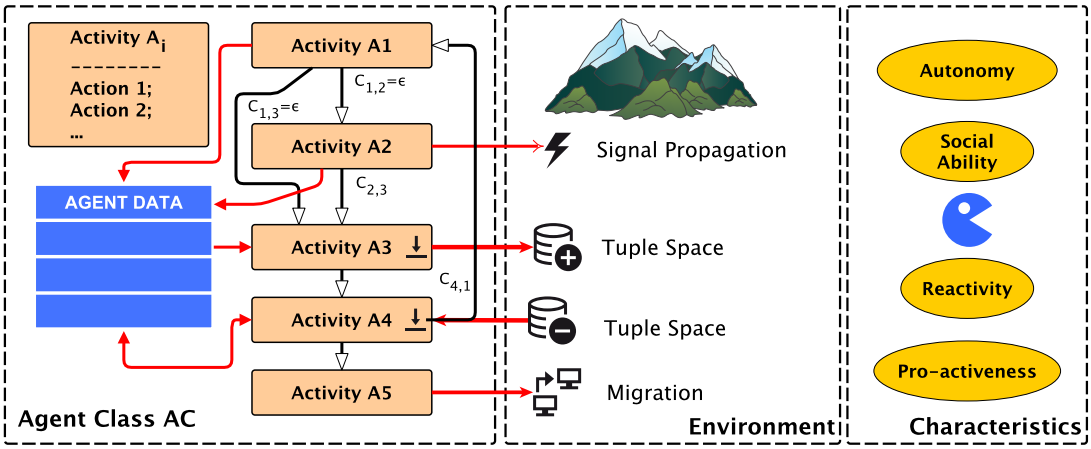


Fig. 2.2 *(Left) Agent behaviour given by an Activity-Transition Graph and the interaction with the environment performed by actions executed within activities (Right) Agent Characteristics*

The ATG behaviour model is closely related to the interaction of agents with the environment, here mainly by exchanging data by using a tuple space database, and migration. Message passing between agents is available by passing signals that execute signal handler on the destination agent asynchronously.

The execution of signal handlers changes the agent data and hence has an impact on activity transitions.

The characteristics of agents can be classified in autonomy, social ability and social interaction, reactivity with respect to changes of the environment and learning based on history and rewards, and finally pro-activeness by making assumptions about the estimated change of the environment resulting from actions performed by the agent.

2.3 Dynamic Activity-Transition Graphs (DATG)

An ATG describes the complete agent behaviour. Any sub-graph and part of the ATG can be assigned to a sub-class behaviour of an agent. Therefore, modifying the set of activities $\mathcal{A}$ and transitions $\mathcal{T}$ of the original ATG introduces several sub-behaviours implementing algorithms to satisfy a diversity of different goals.

The reconfiguration of activities $\mathbb{A}=\{\mathcal{A}_1 \subseteq \mathcal{A}, \mathcal{A}_2 \subseteq \mathcal{A}, ..\}$ derived from the original set $\mathcal{A}$ and the modification or reconfiguration of transitions $\mathbb{T}=\{T_1, T_2, ..\}$ enables dynamic ATG composition and agent sub-classing at run-time, shown in Figure 2.3.

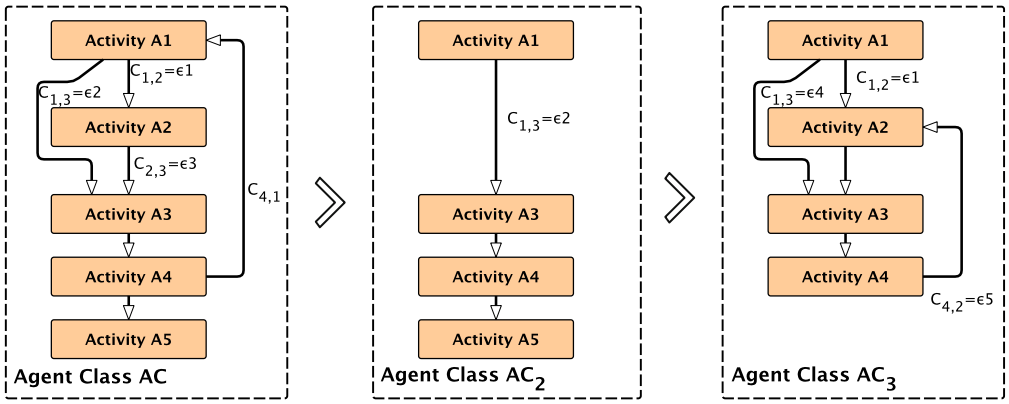


Fig. 2.3 *Dynamic ATG: Transformation and Composition*

AAPL: The Activity-based Agent Programming Language

The Activity-based Agent Programming Language (AAPL) implements the DATG agent model programmatically with a set of operations and a pre-defined structure of the agent class template. Though *AAPL* has a concrete syntax, it is still more a meta language used to implement agents with different existing programming languages. A detailed discussion of *AAPL* can be found in Sec. 2.9.

2.4 Agent Classes

Behaviour. A particular agent class AC_i is related to the previously introduced ATG that defines the run-time behaviour and the computational action performed by agents.

Perception. An agent interacts with its environment by performing data transfer using a unified tuple-space with a coordinated database-like interface. Data from the environment influences the following behaviour and action of an agent. Data passed to the environment (e.g., the database) influences the behaviour of other agents.

Memory. State-based agents perform computation by modifying data. Since agents can be considered as autonomous data processing units, they will primarily modify private data, and a computational outcome using this data will be transferred to the environment. Therefore, each agent and agent class will include a set of body variables $V=\{v_1{:}DT, v_2{:}DT, ..\}$, which are modified by actions in activities and read in activities and transitional expressions (with DT: set of supported data types).

Parameter. Agents can be instantiated at run-time from a specific agent class creating agents with equal initial control- and data states. To distinguish individual agents (creating individuals), an external visible parameter set $P=\{p_1{:}DT, p_2{:}DT, ..\}$ is added, which get argument values on instantiation, enabling the creation of different agents regarding the data state. Inside an agent class, parameters are handled like variables.

To summarize, an agent class is fully defined by the tuple:

$$
\begin{aligned}
&AC_i = \langle A, T, V, P \rangle \\
&A = \{a_1, a_2, .., a_n\} \\
&a_i = \{i_1, i_2, .. \mid i_u \in ST\} \\
&V = \{v_1, v_2, .., v_m\} \\
&P = \{p_1, p_2, .., p_i\} \\
&T = \{t_{ij} = t_{ij}(a_i, a_j, cond) \mid a_i \xrightarrow{cond} a_j; i, j \in \{1, 2, .., n\}\}
\end{aligned}
\tag{2.1}
$$

2.5 Communication and Interaction of Agents

Communication and interaction is discussed in detail in Section 3. In the agent behaviour model used in this work the agents interact with each other by exchanging data using a tuple database as a shared object supporting synchronized and atomic read, test, remove, and write operations. Agents can communicate and synchronize peer-to-peer by using signals, which can be delivered to remote execution nodes, too.

A tuple space is basically a shared memory database used for synchronized data exchange among a collection of individual agents providing an essential MAS interaction paradigm. The scope and visibility of a tuple space database can be unlimited and visible and distributed in the whole network, or limited to a local scope, e.g., network node level. A tuple space provides abstraction from the underlying platform architecture, and offers a high degree of platform independence, vital in a heterogeneous network environment.

A tuple database stores a set of n-ary data tuples, $tp_n=(v_1, v_2, .., v_n)$, a n-dimensional value tuple. The tuple space is organized and partitioned in sets of n-ary tuple sets $\nabla=\{TS_1, TS_2, .., TS_n\}$. A tuple is identified by its dimension and the data type signature. Commonly the first data element of a tuple is treated as a key. Agents can add new tuples (output operation) and read or remove tuples (input operations) based on tuple pattern templates and pattern matching, $pat_n=(v_1, x_2?, .., v_j, .., x_j?, .., v_n)$, a n-dimensional tuple with actual and formal parameters. Formal parameters are wild-card place-holders, which are replaced with values from a matching tuple and assigned to agent variables. The input operations can suspend the agent processing if there is actually no

matching tuple is available. After a matching tuple was stored, blocked agents are resumed and can continue processing. Therefore, tuple databases provide inter-agent synchronization, too. This tuple-space approach can be used to build distributed data structures and the atomicity of tuple operations provides data structure locking. The tuple spaces represents the knowledge of agents.

2.6 Multi-Agent Systems and Networked Processing

Almost every agent-based system consists of multiple agents. Multi-agent systems emphasize on multiple commonly distributed agents and the communication amongst them.

There is a multi-agent system (MAS) consisting of a set of individual agents $AG=\{ag_1, ag_2, ..\}$. Each agent of the MAS belongs to a specific agent behaviour class $AC=\{AC_1, AC_2, ..\}$ from which it was instantiated. Agents initially belonging to a super class AC_i can change their behaviour at run-time by composing different sub-classes $\{AC_{i,1}, AC_{i,2}, ..\}$, sharing activities and transitions of the super class.

Distributed Networks. In a specific situation an agent ag_i is located at and processed on a network node $ND_{m,n}$ (e.g. microchip, computer, virtual simulation node) at a unique spatial location (m,n). There is a set of different nodes $NW=\{ND_1, ND_2, ..\}$ arranged in a mesh-like network NW with peer-to-peer neighbour connectivity (e.g. two-dimensional grid), shown in Figure 2.4.

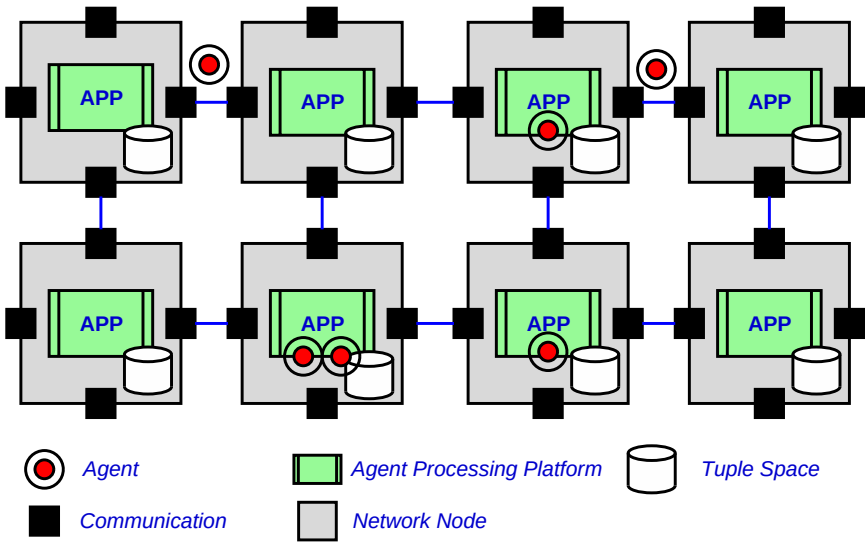


Fig. 2.4 *Network of agent processing nodes*

Each node is capable to process a number of agents $n_i(AC_i)$ belonging to one agent behaviour (super) class AC_i, and supporting at least a subset of $AC' \subseteq AC$. An agent (or at least its state) can migrate to a neighbour node where it continues working. The agent processing platform available on each network node is capable to modify the agent behaviour based on the DATG paradigm.

Communication. In parallel and distributed systems the communication, synchronization, and data exchange of a collection of data processing units (processes or agents) gains significant importance. A common approach for parallel systems is a shared memory based communication paradigm, but which generates a high computational dependency of the processing units among themselves and regarding the platform. Loosely coupled distributed systems like MAS require a different communication strategy. One well-known and common distributed interaction model is the tuple-space. Agents can communicate with each other by accessing a tuple space database service available on each network node and which is provided by the agent processing platform. A tuple space is a logically shared memory and is used for synchronized data exchange between producer and consumer, a common approach for solving communication problems of loosely coupled autonomous or semi-autonomous processing units.

A tuple space is basically a shared memory database used for synchronized data exchange among a collection of individual agents, which was proposed in [CHU02] and [QIN10] as a suitable MAS interaction and coordination paradigm. The scope and visibility of a tuple space database can be unlimited and visible and distributed in the whole network, or limited to a local scope, e.g., network node level. A tuple space provides abstraction from the underlying platform architecture, and offers a high degree of platform independence, vital in a heterogeneous network environment.

For the sake of simplicity the scope of a tuple space can be limited to the node boundary, such that there are multiple tuple spaces distributed in the network. Information can be carried by mobile agents between nodes. A tuple space communication model has the advantage of shielding the underlying node and agent processing platform. Access of tuple spaces require only a small set of simple operations {`OUT`, `IN`, `RD`}, which transfer tuples between producer or consumer and the database. They are discussed in detail in the next section. Since tuples consist of type-tagged values and patterns the tuple space communication is type-safe and strong computational bindings can be avoided.

2.7 The Big Thing: Domains, Networks, and Mobile Agent Processing

The previous section introduced a unified agent behaviour and programming model offering computation, instantiation of agents, mobility, and multi-agent interaction, and which can be implemented on a diversity of processing platforms. One major goal of the deployment of MAS is overcoming heterogeneous platform and network barriers arising in large scale hierarchical and nested network structures, consisting and connecting, e.g., the Internet, sensor networks, body networks, production and manufacturing Cyber-Physical System (CPS) networks, shown in Figure *2.5* on the left.

The large diversity of execution platforms, network topologies, services provided by network nodes, and the programming environments require a unified and abstract behavioural and structural representation model. The Bigraphical model proposed by Robin Milner models the entire "computing" environment with place and link graphs, composing finally bigraphs [MIL09], shown on the right of Figure *2.5*. They include agents, and they are offering an unified model and platform for ubiquitous systems and the foundation for an Ubiquitous Abstract Machine, and supporting reconfigurable spaces (dynamic topologies). Bigraphs virtualize communicating processes (agents) and information objects (tuple-spaces),

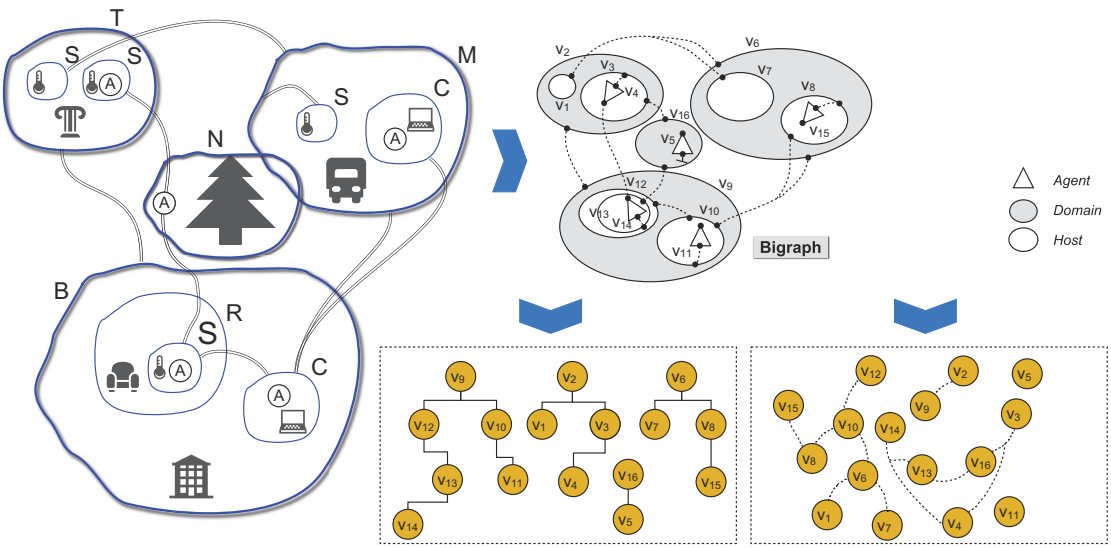


Fig. 2.5 *From physical maps (left) to unified logical maps: link (right, bottom) and structure place (middle, bottom) graphs composing bigraphs (right, top) [S: Sensor, T: Technical Structure, M: Mobile Device, N: Net. Router, B: Building, R: Room, C: Computer, A: Agent]*

They originate in process calculi for concurrent systems, especially the pi-calculus [MIL99] and the calculus of mobile ambient environments [CAR00A] for modelling spatial configurations of networks with a dynamic topology.

The environment consists of places where computation occurs, e.g., computers, agents, rooms, buildings, machines, technical structures, and so on. The links are abstract, providing the possibility of interaction between different places, i.e., transferring of agents and their mobile processes. Agents are treated as active computational units (subjects). Places introduce spatial and logical bindings. Bigraphs allow the nesting of nodes and places, natural for many real-world computing environments, and they can be applied for wide reactive systems. All nodes have a fixed number of ports, providing an end-point for links. Agents have two ports: a processing port link and an interaction (communication) link. Bigraphs, which represents the system state, can be modified by the application of reaction rules, which changes the linking and place relations. Bigraphs can be composed of other bigraphs matching inner and outer interfaces.

A link is a hyper-edge connection that connects nodes, outer, and inner names, where names are open linkings that support additional connectivity, i.e., used for the dynamic composition of bigraphs at "run-time". Connectivity not only provides the platform for agent migration between different places, it provides information exchange, which is provided here by location place-bounded tuple-spaces and signals. Migration of mobile processes is just another form of interaction with and the modification of the environment.

2.7.1 AAPL Agents in the Bigraph Model

To adapt this Bigraphical Reactive System (BRS) model to MAS it is necessary to distinguish subjects (entities that can perform actions, the agents) and objects (here data, tuples, tuple-spaces, signals, and processing platforms themselves).

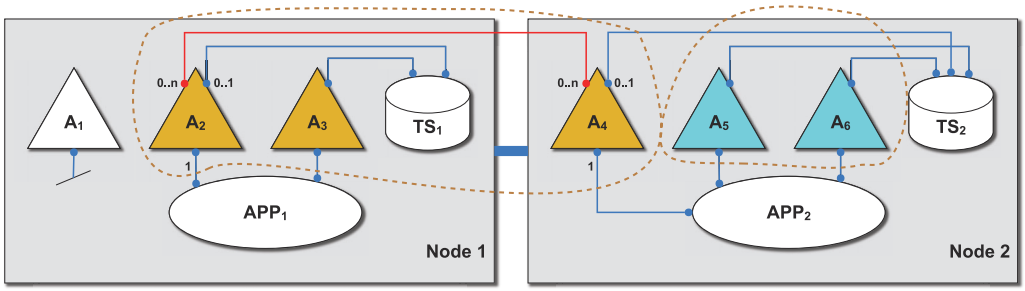


Fig. 2.6 *AAPL agents in the Bigraph Model with a bottom port for the APP link and top port for tuple space and signal link ports. Shown are two connected nodes.[A: Agent, APP: Agent Processing Platform, TS: Tuple Space]*

AAPL agents have different ports in terms of the Bigraph model. One static port is the platform link, required to execute an agent process. Another port is used for the linking of an agent with a tuple-space, with #=1. An *AAPL* agent can have only one tuple-space access and link at any time maximal.

The propagation of signals introduce further ports and dynamic links to other agents, with #=0..n, shown in Figure 2.6.

The communication links create virtual domains, in Figure 2.6 these are the agent group sets $\{Ag_2, Ag_3, A_4\}$ and $\{A_5, A_6\}$. These virtual domains are dynamic, regarding the spatial location and extension, and the agents that are part of the virtual domain. Often agent parent-child trees spawn the virtual domains using signal interaction, but agents of initially different virtual domains can interact by using the tuple-spaces, extending and merging different virtual domains. The spatial extension of virtual MAS domains is constrained by the connectivity graph of the processing nodes. Signal propagation from a source to a destination agent requires the connectivity of nodes if the agents are executed on spatially different nodes. Tuples stored in tuple-spaces are persistent. That means a tuple t, which was produced by an agent Ag_1 and stored in a tuple-space TS_1, and agent Ag_1 is finally migrating to another node location, can be consumed by a different agent Ag_2, now having a historical relation and link to the other agent Ag_1.

2.7.2 Heterogeneous Sensor Networks in the Bigraph Model

Sensor networks consists of multiple sensor nodes connected in mesh-like network topologies. Commonly sensing applications are partitioned in on-line and off-line parts, spatially resulting in different networks, and temporarily resulting in computations performed in real-time and not-real-time. An example is shown in Figure 2.7, where a material-integrated sensor network with sensor nodes capable of measuring strain in a technical structure is connected to an external computation network performing inverse numerical computations for deriving the mechanical load information from the sensor data. Mobile agents are used to distributed data within and between these networks.

All sub-networks and the mobile agents passing network boundaries are treated unified in the Bigraph model.

epubli, ISBN 9783746752228 (2018)

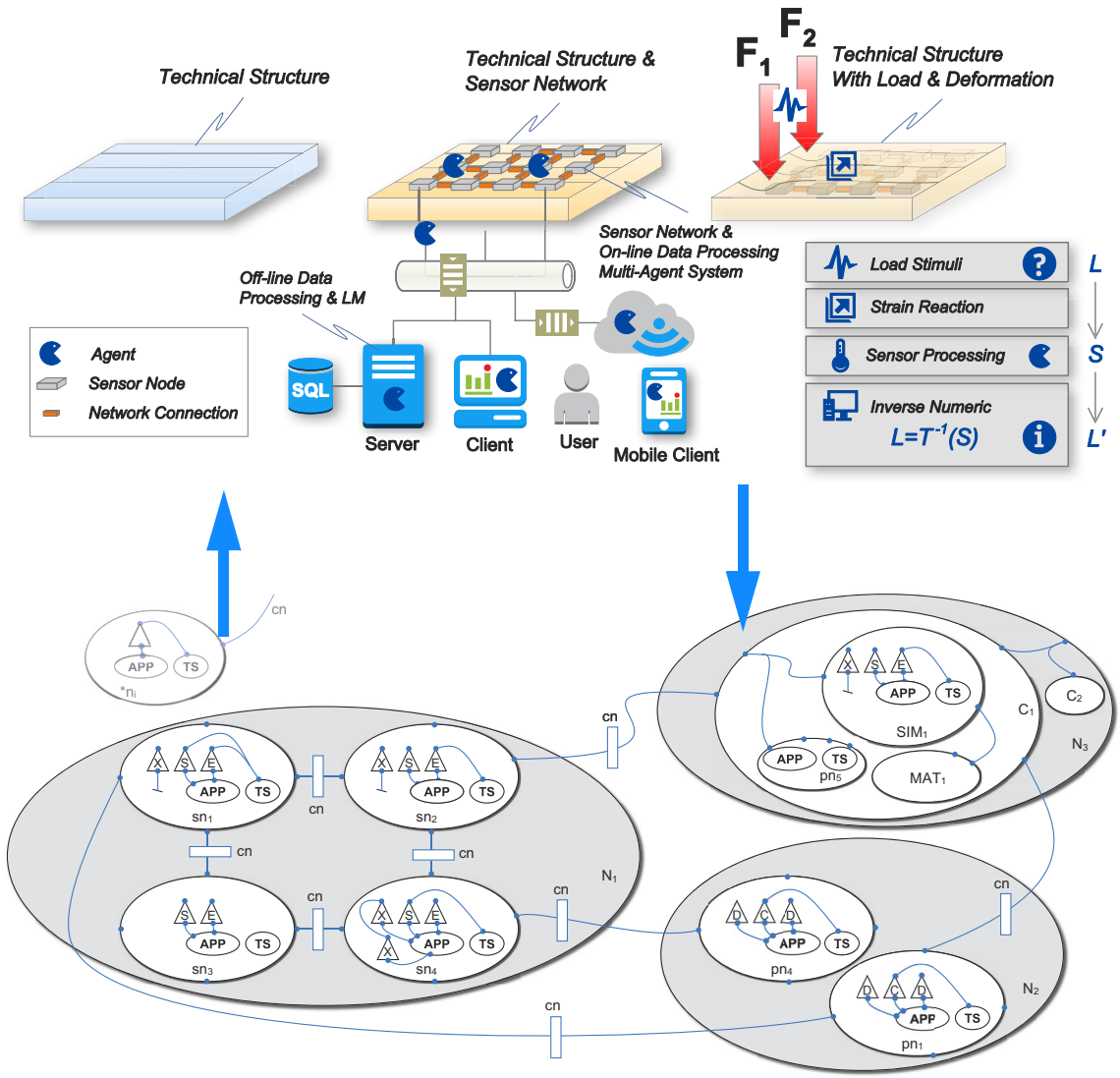


Fig. 2.7 *Top: A Sensorial Material with a material-embedded sensor network connected to a computational network, partitioning sensing and computation in on-line and off-line domains. Agents can migrate between different networks and hosts (sensor nodes, computers, servers, mobile devices). Bottom: Bigraph of the environment [sn/pn: Sensor/Computational Node, cn: Communication Channel, N: Network, C: Computer, SIM: Agent Simulator, MAT: Matlab]*

2.8 Distributed Process Calculus

Up to here, the deployment and execution of mobile agents in a network is considered as being a dynamic distributed system, with static and spatially fixed nodes and their resources, and dynamic agents regarding their control and location state.

The π-Calculus introduced by Milner (1992) and the extended asynchronous distributed π-Calculus introduced by Hennessy [HEN07] (aDΠ) are common formal languages for concurrent and distributed systems, suitable for studying the behaviour and reaction of distributed and concurrent systems including dynamic changes caused by mobility. The π-Calculus bases on the Calculus of Communicating Systems (CCS), partial related to the CSP model by Hoare, which is introduced in Chapter 5. In the CCS model the connection topology is static and cannot change while the system evolves. But most distributed systems, especially in the context of sensorial systems, are highly dynamic. The π-Calculus introduces dynamic aspects, i.e., the dynamic creation of communication channels, which enable dynamic changing structures of the topology of networks. The aDΠ-Calculus extends the π-Calculus with the concept of (structural) domains, resources associated with domains, and migration of processes, close to the MAS paradigm.

Basically all distributed systems can be abstracted by decomposing the system in mobile processes. Though mobile processes are closely related to the agent model, there were several non-agent related distributed systems investigated in the past, e.g., the distributed operating system (DOS) Amoeba [MUL90].

In the following the relationship of the *AAPL*/DATG behaviour and interaction model with the modified distributed Π-Calculus is pointed out (based on the original aDΠ-Calculus), moving the view of point from spatially located agents in a distributed inter-connected system to one unified concurrent system with dynamic virtual communication channels.

Processes are the main execution units, synchronizing by communicating using channels, which are shared objects supporting read and write operations. A channel is related to a name and values, to be communicated i.e., names, which can be channel names, too. The application of the channel operations results in a behavioural transition of processes. In the aDΠ-Calculus channels are global and can be created dynamically at run-time. A created channel can be bound to a particular set of processes.

Figure 2.8 shows the transformation of a distributed network populated with mobile agents to one unified asynchronous and concurrent communicating process system, consisting only of processes and virtual communication channels. Agents are represented by communicating processes.

The migration of agents creates new communication channels, enabling the interaction with other agents.

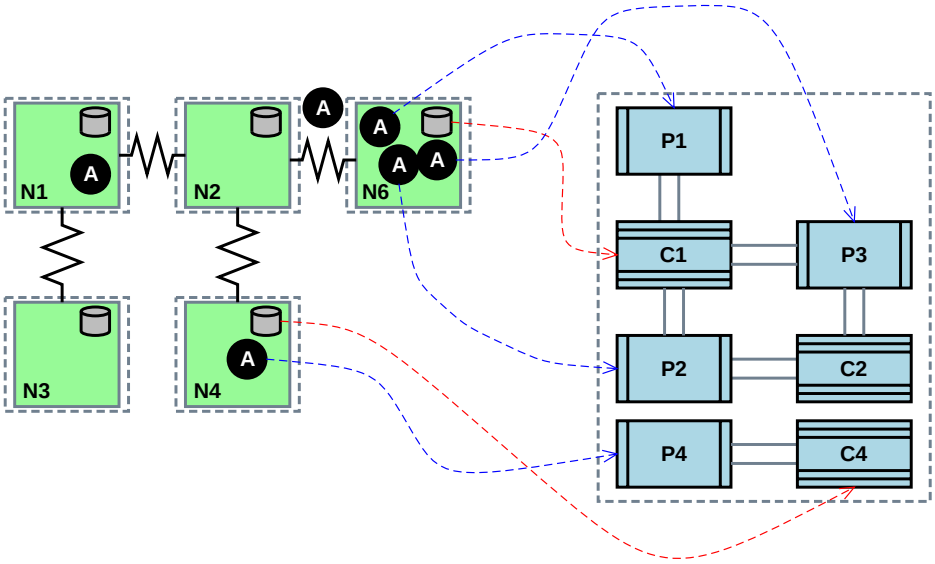


Fig. 2.8 *From a spatially distributed network populated with agents to a unified concurrent CSP model [N: Network node, A: agent, P: Process, C: Communication Channel]*

The scope, i.e., the boundary of the interaction domain spawned by channels, depends on the specific interaction object model. In the *AAPL* model tuple-spaces and signals are used for inter-agent communication and synchronization. These both communication paradigms can be mapped on (global) channels in the sense of the Π-Calculus. The agent-related communication calculus is discussed in Section *3.5*.

Definition *2.2* summarizes the distributed Π syntax, which is inspired and derived from the original mobile process aDΠ and the *KLAIM* calculi [NIC98], basing on the composition of process terms using channel-based communication, scoping procedures for variables and channels, local binding of identifiers, parallel composition, conditional branching, and recursion. A locality is a symbolic name for a site with a logical spatial location. A distributed network consists of a set $L=\{l_1, l_2, ..\}$ of localities, here related to network nodes in the context of the introduced Bigraphs. Programs can be structured over distributed environments by using localities while ignoring their precise allocations and spatial position. There is a distinguished locality *self* that programs can use to refer to their execution site. The local scope determines the reduction of process terms, e.g., due to communication, which is explained in Section *3.5*.

Def. 2.2 *Distributed ΠΠ Process Algebra Language Syntax (adapted from aDΠΠ, KLAIM)*

```
P,R,B ::= Process Terms
        u   Communication channel identifier
        x   Variable identifier
        x@l u@l P@l  Identifier bound to location l
        x@d u@d P@d  Identifier bound to domain d
        a . P  Action prefixing (a: communication,..)
        u!<v>  Output of a value using channel u
        u?(x) . R  Input from a channel u
        (new n) R  Channel name creation bound to R
        (new x,y,..) R  Variable name creation bound to R
        if e then R1 else R2   Conditional Branching
        match e0 with e1:R1 e2:R2 ..  Matching and Alternation
        R1 || R2  Parallel Composition
        R1 | R2  Choice (Alternation)
        rec x • B  Recursive process definition with
            recursion variable x and process body B
        *P  Iteration of P (≡ rec x • P(x) )
        P1;P2;P3;..  Sequence
        stop ⊥  Termination / Blocked process state
        P → P'  Reduction rule (Evolving of processes)
        goto(l) . P  Migration of process to location l
        P{v/x}  Replaces all occurrences of variable x
            in process P with value v
        Sig(ε)  Type signature of expression
        vi ← ε   Data Assignment to variable vi
```

The following Definition 2.3 summarizes the simplified representation of the *AAPL*/ATG agent behaviour model with the Π-Calculus. Each agent ATG is related to one meta-process $\mathbb{P}$, consisting of a set of transitional process representations $\{P_1, P_2, ..\}$ related to the activity processing, which changes by activity transitions. Activities consist of computational parts and I/O and event-based statements (communication and mobility). The I/O statements can block the agent processing, which splits an activity process P_i in sub-process states represented by sub-processes $\{P_{i,1}, P_{i,2}, ..\}$ extensively discussed in the platform Chapters 6 and 7.

Agents are bound to locations $L=\{l_1, l_2, ..\}$, in the context of the *AAPL* model these are the spatially different agent processing nodes of the network. Nodes of a network belong to domains, for example, binding nodes with the same agent process platform architecture. An agent can access resources that are bound to a location, too. In the *AAPL* context these are the tuple-spaces, which are the data resources, and the agent platform itself, which is an execution resource.

The signal handlers introduced later are mapped on processes, too. The interaction between the sending and the receiving agent, as well between the receiving agent and the signal handler processes is performed by channels, too.

Def. 2.3 *Simplified representation of the agent behaviour model (left) by the Π-Calculus (right) [Ag: Agent, P: Process]*

Agents		**Π-Calculus**
Agent Ag_n := <$\mathcal{A},\mathcal{T}$>	⇔	$\mathbb{P}_n \equiv \{P_{n,1},P_{n,2},..\}$
Transition $A_i \rightarrow A_j \mid c_{ij}$		$\tau.P_{n,i} \rightarrow^{cij} P_{n,j}$
Mobility		
Ag_n: moveto(*DIR*)	⇔	goto l_2.P: $P_{n,i}@l_1 \rightarrow P_{n,j}@l_2$
Tuple-Space Access		
Ag_a: out(v_1,v_2,..)	⇔	(new $\underline{c_d},\underline{c_a}$) $\underline{req}$!<OUT,$\underline{c_d},\underline{c_a}$>.$\underline{c_d}$!<**v**>.$\underline{c_a}$?().$P_{a,j}$
Ag_b: in(v_1,x_1?,..)	⇔	(new $\underline{c_d},\underline{c_a}$) $\underline{req}$!<IN,$\underline{c_d},\underline{c_a}$>.$\underline{c_d}$!<**v**>.$\underline{c_a}$?(**x**).$P_{b,j}$
Signal Propagation		
Ag_i: send(Ag_j,*S*,*arg*)	⇔	$\underline{S}$!<arg>
Instantiation of Agents		
Ag_a: fork(v_1,..)	⇔	(new p_1,..) (P_a || $P_b\{v_1/p_1,..\}$)
Ag_a: new(v_1,..)	⇔	(new p_1,..) (P_a || $P_b\{v_1/p_1,..\}$)

The propagation of signals from one agent $Ag_a@l_1$ to another agent $Ag_b@l_2$, which can be located on different node, create virtual domains, primarily spawned by agent parent-child tree relationships. Usually signals are used only within this tree because the agent destination identifier must be provided, commonly known only by parent-child groups.

The tuple-space access, which is limited to agents executed on the same node only, creates polyadic channels, which are local resources (communication resources), too. Indeed, there is one channel for each common tuple pattern exchanged by producers and consumers, explained in Section 3.5.

The instantiation of agents creates new processes, either forked from a parent process or created from a template. The parameters of the newly created process are substituted by the argument values. Migration of agents results in a process and location transition (i.e., after the migration the state of an agent has changed), related to the `goto` statement. The body variables of an agent are mobile resources.

In the *AAPL* context activity processes can be blocked related to I/O activities, i.e., waiting on available tuples or time-outs, which can be expressed by channel actions. The transition from one outgoing to multiple incoming activi-

ties $A_i \rightarrow \{A_j{:}c_{ij}, A_k{:}c_{ik}, ..\}$ can be blocked if there is actually no condition c for the respective transition that can be satisfied. Both reasons for blocking of the agent processing and the evolving of the process is shown in Definition 2.4. If the satisfaction of a transition condition depends on the processing of a signal handler, serving a signal send by another agent and modifying agent data, than an activity process transition can be considered as being I/O related by using a channel invocation, too, discussed in Section 3.5.1.

Def. 2.4 *Evolving of agent processes and blocking of agent processing [Ag: Agent, P: Process, Process index n,i,u: agent n, activity i, sub-state u]*

Computation
Agent Ag_n : $P_{n,i,u}$. $v_i \leftarrow \varepsilon$. $P_{n,i,v}$

I/O Event
Agent Ag_n : $P_{n,i,u}$. ch?(..) . $P_{n,i,v}$

Activity Transition
Agent Ag_n : $P_{n,i} \rightarrow$ (c_{ij}=true.$P_{n,j}$ | c_{ik}=true.$P_{n,k}$ | ..)
Agent Ag_n : $P_{n,i} \rightarrow$ (c_{ij}=true.$P_{n,j}$ | ch?(..).c_{ik}=true.$P_{n,k}$ | ..)

2.9 AAPL Programming Model and Language

2.9.1 Overview and Summary

The *AAPL* programming model should optimally match the requirements of MAS deployed in unreliable sensor and generic distributed networks, keeping low-resource nodes with low computational power in mind. On one hand, *AAPL* should reflect the core concepts of agents, on the other hand *AAPL* should provide core concepts of traditional programming language to ease the programming of widely used algorithms.

The agent behaviour is partitioned and modelled with an activity graph, with activities representing the control state of the agent reasoning engine, and conditional transitions connecting and enabling activities, shown in Figure 2.9 Activities provide a procedural agent processing by sequential execution of imperative data processing and control statements.

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

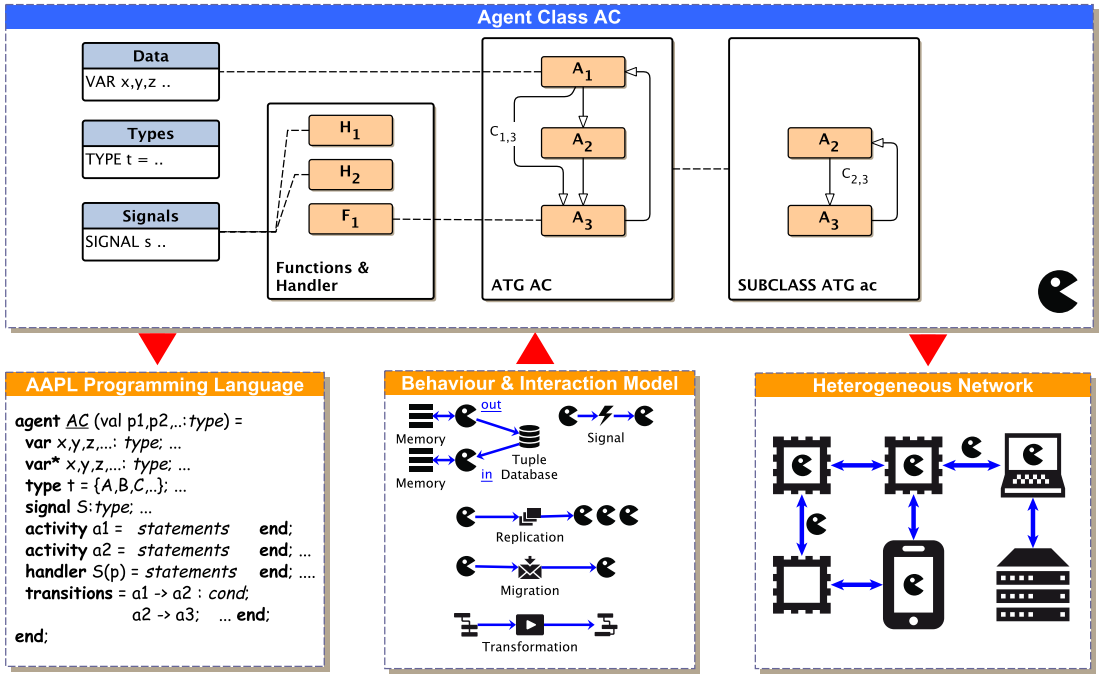


Fig. 2.9 *Agent behaviour programming level with activities and transitions (AAPL, left); agent class model and activity-transition graphs (top); agent instantiation, processing, and agent interaction on the network node level (right)* [BOS14B].

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

***Multi-Agent Systems**: There is a multi-agent system (MAS) consisting of a set of individual agents $\{a_1,a_2,..\}$. There is a set of different agent behaviours, called classes $\mathcal{C}=\{AC_1, AC_2,..\}$. An agent belongs to one class. In a specific situation an agent a_i is bound to and processed on a network node $N_{m,n}$ (e.g. microchip, computer, virtual simulation node) at a unique spatial location (m,n). There is a set of different nodes $\mathcal{N}=\{N_1, N_2,..\}$ arranged in a mesh-like network with peer-to-peer neighbour connectivity (e.g. two-dimensional grid). The node connectivity may be dynamic and changing over time. Each node is capable to process a number of agents $n_i(AC_i)$ belonging to one agent behaviour class AC_i, and supporting at least a subset of $\mathcal{C}' \subseteq \mathcal{C}$. An agent (or at least its state) can migrate to a neighbour node where it continues working. Each agent class is specified by the tuple $AC = \langle A,T,F,S,H,V \rangle$. A is the set of activities (graph nodes), T is the set of transitions connecting activities (relations, graph edges), F is the set of computational functions, S is the set of signals, H is the set of signal handlers, and V is the set of body variables used by the agent class.*

Ex. 2.1 *Example of a simple AAPL Agent class definition*

AAPL ⇔ **Short Notation**

```
type TSKEY={ADC,SENSOR,SENSOREV};
agent mean_filter(thr:integer)
  var m:integer; var *x:integer;
  activity A1 = in(ADC,x?); end;
  activity A2 = m:=(m+x)/2;
    out(SENSOR,m); end;
  activity A3 =
    out(SENSOREV,m);
    kill($self); end;
  transitions =
    A1 -> A2 : m < thr;
    A1 -> A3 : m >= thr;
    A2 -> A1;
end;
```

```
κ: {ADC, SENSOR, SENSOREV}
ψ mean_filter: thr → {
   Σ: {x,m}
   α A1 : {  ∇⁻(ADC,x?) }
   α A2 : {  m←(m + x)/2;
             ∇⁺(SENSOR,m) }
   α A3 : {  ∇⁺(SENSOREV,m);
             ⊗($self) }

   Π : {
      A1→A2 | m<thr
      A1→A3 | m≥thr
      A2→A1 }
}
```

Plans are related to *AAPL* activities and transitions close to conditional triggering of plans. Tables *2.1* and *2.2* at the end of this section summarize the available language statements. Their effects on a multi-agent system is shown in Figure *2.10*. Beside the *AAPL* programming language there is a short notation, which has the same operational semantic, but offers a more compact *AAPL* representation, which ease the understanding of more complex agent behaviours.

The following Example*2.1* poses an agent class definition with a simple agent that has the goal to collect sensor values (`ADC`), finally computes the mean value (`SENSOR`) of all samples. If a threshold `thr` is reached, an event tuple (`SENSOREV`) is generated.

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

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

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

Agent interaction and synchronization is provided by the previously introduced tuple-space database server available on each node (related to [CAB95]). An agent can store an n-dimensional data tuple *(v1,v2,..)* in the database by using the `out(v1,v2,..)` statement (commonly the first value is treated as a key). A data tuple can be removed or read from the database by using the `in(v1,p2?,v3,..)` or `rd(v1,p2?,v3,..)` statements with a pattern template based on a set of formal (variable,`?`) and actual (constant) parameters. These operations block the agent processing until a matching tuple was found/stored in the database. These simple operations solve the mutual exclusion problem in concurrent systems easily. Only agents processed on the same network node can exchange data in this way. Simplified the expression of beliefs of agents is strongly based on *AAPL* tuple database model.

epubli, ISBN 9783746752228 (2018)

Tuple values have their origin in environmental perception and processing bound to a specific node location.

The existence of a tuple can be checked by using the `exist?` function or with atomic test-and-read behaviour using the `in?/rd?` functions. A tuple with a limited lifetime (a marking) can be stored in the database by using the `mark` statement. Tuples with exhausted lifetime are removed automatically (by a garbage collector). Tuples matching a specific pattern can be removed with the `rm` statement. A Remote Procedure Call operation is supported by the `eval(v1,p2?,v3,..)` primitive that stores a partially evaluated tuple in the database that is consumed by a service agent, processing it and returning the fully evaluated tuple to the database again, which is finally passed to the original client evaluation call.

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

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

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

Reconfiguration: Agents are capable to *change their transitional network* (initially specified in the transition section) by changing, deleting, or adding (conditional) transitions using the `transition♦(`A_i`,`A_j`,predcond)` statements. *This behaviour allows the modification of the activity graph, i. e., based on learning or environmental changes, which can be inherited by child agents.* The modification can be restricted to a sub-class transition set, which is useful for child agent generation. Additionally, the ATG can be transformed by adding or

removing activities using the `activity♦(Ai,Aj,...)` statements, which is only applicable for dynamic code-based agents not considered here.

Kind	*AAPL Statement*	*Description*

Agent Class Definition	`agent AC (parameters) =`
	`  definitions`
	`  activities`
	`  transitions`
	`  subclasses`
	`end;`
		Defines a new agent class *AC* with optional parameters. The class body consists of variable, activity, and transition definitions.

Agent Subclass Definition	`subclass SC =`
	`  definitions`
	`end;`
		Defines a new agent subclass *SC* part of a root class *AC*.

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

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

Activity	`activity A =`
	`  statements`
	`end;`
		Defines a new agent activity *A*, which can be bound to a subclass *SC*.

Composition	`activity+ (id,a1,a2,...);` `activity- (id,a1,a2,...);`	Adds or remove activities at run-time to/from a specific agent *id*.

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

Tab. 2.1 *Summary of the AAPL statements used to define the agent behaviour and control*

Kind	AAPL Statement	Description

Reconfiguration and Composition	`transition+ [SC] (a1,a2,c);`
	`transition* [SC] (a1,a2,c);`
	`transition- [SC] (a1,a2);`
	`(id,..)`
		Changes transitions at runtime (add, replace all, remove all). Can be applied to a subclass *SC* or to a specific agent *id* only.

Mobility	`moveto(`*DIR*`|`*PATH*`);`
	`moveto(`*dx*`,`*dy*`,..);`
	`.. link?(`*DIR*`|`*PATH*`) ..`
	`.. link?(`*dx*`,..) ..`
	DIR`={NORTH, SOUTH,`
	`WEST, EAST, ORIGIN, ..}`
	PATH `=`*IP*`|`*CAP*`|`*URL*`|..`
		Migrates agent to a neighbour node. The connectivity can be tested by using the `link?` operation.

Tab. 2.1 *Summary of the AAPL statements used to define the agent behaviour and control*

Kind	AAPL Statement	Description

Signal	`signal S:datatype;`
	`handler S(x) =`
	`statements`
	`end;`
	`send(id,S,v);`
	`reply(S,v);`
	`broadcast(AC,DX,DY,S,v);`
	
	`sendto(to,S,v);`
		Definition of a signal *S* that can be processed by a signal handler similar to an activity. Signals are either send to a specific agent *id* or send to all agents of a specific class within a region (Agent-to-Agent). A signal receiver can send a signal reply back to the original sender. The *sendto* operation sends a signal to a specific node that is further passed to all listening agents on this node (Agent-to-Node).

Tab. 2.2 *Summary of the AAPL statements used for interaction and communication*

Kind	AAPL Statement	Description

Tuple Space Database	`out(v1,v2,..);`
	`.. exist?(v1,?,..) ..`
	`in(v1,x1?,v2,x2?,..);`
	`rd(v1,x1?,v2,x2?,..);`
	`in?(timeout,v1,..);`
	`rd?(timeout,v1,..);`
	`mark(timeout,v1,v2,..);`
	`rm(v1,?,..);`
	`alt((pat1)|(pat2)|..]);`
	`alt?(timeout(pat1)|..]);`
		Coordinated data exchange by agents using the tuple space operations with tuples and patterns. A marking is a tuple with a limited lifetime. Commonly, the first tuple value is treated as a key, e.g. classifying the tuple. The *alt* operation listens on a set of tuple patterns. If there is a tuple matching one of the patterns, the tuple is consumed.

Tuple Space Database	`eval(id,v1,x1?,v2,x2?,..);`	Injection of an active tuple that is only partially evaluated (containing empty elements) that is consumed by a server agent processing the tuple and storing the evaluated tuple. The evaluated tuple is passed back to the initiator of the evaluation operation.

Distributed Tuple Space	`copyto(to,v1,x1?,v2,x2?,..);`
	`collect(to,v1,x1?,v2,x2?,..);`
	`store(to,v1,v2,..);`
		Remote tuple space access. The `copyto` operation copies all tuples matching the pattern to the specified node tuple space. The `collect` operation moves all matching tuples, and the `store` operation is the remote `out` operation.

Timer and Blocking	`timer+(timeout,S);`
	`timer-(S);`
	`sleep; wakeup;`
		A timer can be used to raise a signal *S*. Agents can be suspended and be woken up.

Neighbourhood	`who?;`
	`who?(DIR|PATH|RANGE)`
	`who?(AC,DIR|PATH|RANGE)`
		Returns a list of agents found on this node or in the neighbourhood, optionally limiting to a specific class.

Tab. 2.2 *Summary of the AAPL statements used for interaction and communication*

2.9.2 Signal Classes

Agent-to-Agent (A2A) Signals

Signals are lightweight messages that are delivered to specific agents (Agent-to-Agent A2A), in contrast to the anonymous tuple exchange. One major issue in distributed MAS is remote agent communication between agents executed on different network nodes. Though an agent can be addressed by an unique identifier, the path between a source and destination agent is initially unknown. For the sake of simplicity and efficiency, a signal from a source node *A* can only be delivered to a destination agent currently on node *B* iff the destination agent was executed (or created) on node *A* some time ago. I.e., two agents must have been executed on the same node in the past. Agent migration and signal propagation is recorded by the agent platform using look-up table caches with time limited entries and garbage collection.

Agent-to-Node (A2N) Signals

The signal delivery along migration paths based on the destination agent identifier (private, uni-cast) or the agent class (public, broadcast) is not appropriate in all use cases. Therefore, signal delivery of signals to specific remote platforms (remote signalling) based on paths specified by the signal sender agent is available. The destination platform node broadcasts the signal to all listening agents executed on this particular node. To simulate private A2A unicast (or multi-cast) communication, agents can use a randomly generated signal name only known by the sender and the receiver. This new approach enables interaction between agents never executed on the same node. Furthermore, these remote signals are used to implement distributed tuple-spaces, discussed later.

2.9.3 Distributed Tuple-Spaces

The tuple exchange between agents is limited to the node level. To establish distributed tuple spaces, tuple migration using the `collect`, `copyto`, and `store` operations are available, which can be performed by agents. This feature enables the composition of distributed tuple-spaces controlled by agents. The *collect* and `copyto` operations transfer tuples from the local tuple-space to a remote using pattern matching, similar to the `inp` and `rd` operations. The *store* operation send a tuple to a remote tuple-space, similar to the *out* operation.

epubli, ISBN 9783746752228 (2018)

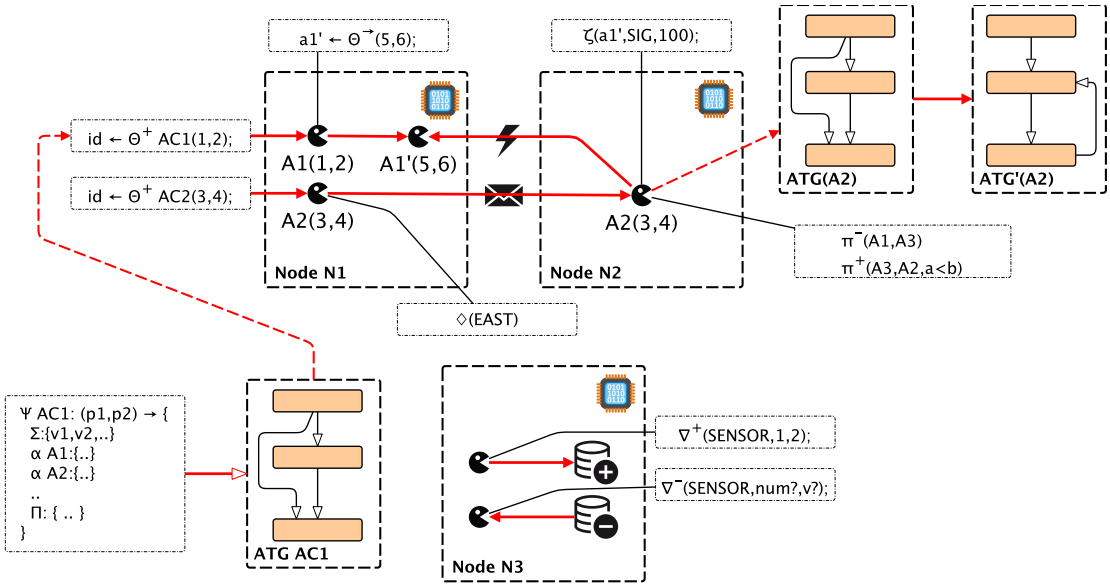


Fig. 2.10 *Effects of AAPL control statements at run-time*

2.9.4 AAPL Agent Classes and Agent Instantiation

The *AAPL* agent behaviour model is closely related to the DATG model. An agent behaviour description is encapsulated in an agent class and consists initially of a set of activities $A=\{a_1, a_2, ..\}$ performing actions (computation and interaction with the environment) and a set of transitions $T=\{t_1, t_2, ..\}$ defining the activation of activities based on the previous activity already executed and the internal data state, as already defined in Equation 2.1.

There are multiple different methods to create new agents at run-time:

1. Agents can be instantiated from an original root agent class. If there are more than one transition sets in the agent class definition, a specific set can be chosen and activated for the new created agent.
2. Agents can be instantiated from a subclass of an original root agent class.
3. Agents can be forked from a parent agent. The child agents inherit the agent behaviour including the current transitions, and the data (content of body variables). A child agent must use the same transition set of the parent agent. A transition set switch is impossible.
4. New agents are composed of an original or already modified agent behaviour (ATG and data, e.g., sub-classing). Though only sub-classing is possible by creating a subset of activities and transitions, free com-

position is possible at run-time, but can be limited by the underlying processing platform.

The agent class consists of a unique name, an optional parameter list, and the agent class body, shown in Definition 2.5.

Def. 2.5 *Summary of agent class definition and instantiation (left AAPL syntax, right short notation) [AC: Agent class, sc: agent subclass]*

Definition of an Agent Class

```
agent AC [[(p1:DT,p2:DT,..)]] =
begin
  definitions
    body variables    var  v1:T; ..
    body variables*   var* v1:T; ..
    [[ signals ]]       signal s:T; ..
    [[ exceptions ]]    exception e;
    [[ types ]]         type T =  ..;
  activities       activity a = .. end;
  transitions      transitions = .. end;
  [[ functions ]]    function f(..) = .. end;
  [[ handler ]]      handler s(..) = .. end;
  [[ subclasses ]]   subclass sc = .. end;
 end;
```

Short Notation

```
Ψ AC:(p1,p2,..) →
{
   Σ : { .. }
       σ : { .. }
   ξ : { .. }
   ε : { .. }
   κ: { .. }
     α a: { .. }
   Π : { .. }
     f : (p1,..) → { .. }
   ξ s:(p) → { .. }
     φ ac: { .. }
}
```

Creation of Agents

```
id := new AC(x1,x2,..);
id := fork (x1,x2,..);
id := fork sc(x1,x2,..);
id := new AC;
.. ATG composition ..
run(id,x1,x2,..);
```

```
id ← Θ+ AC(x1,x2,..)
id ← Θ→(x1,x2,..)
id ← Θ→ sc(x1,x2,..)
```

The body consists of the definition of body variables, optional signals, types, and exceptions. Body variables can be defined as persistent (`var`) or temporary (`var *`). The data content of persistent variables is preserved on migration and is inherited by child agents. Variables defined with the `var*` statement are temporary variables. They are not part of the mobile data state of an agent and are not transferred during agent migration and is not inherited by child agents.

The following example demonstrates the *AAPL* agent class definition for a simple MAS exploring a region of interest (ROI) and collecting sensor data within the ROI by sending out child agents. The agent class Sample consists of five activities and transitions. Agent parent-child interaction is done by using signals carrying an argument (the computational result of the child agents). In

the case of more complex data transfer between agents the tuple-space must be invoked, and signals are used only to synchronize an event notification. The main activity is *percept*, which creates the child agents by forking.

Ex. 2.2 *AAPL definition of a simple agent class and instantiation of agents at run-time using the fork and new statements. The goal of the MAS is to collect sensor values in a ROI within the given radius and finally computes the mean values of all samples.*

```
1   type Direction = {NORTH,SOUTH,EAST,WEST,ORIGIN};
2   type Delta = (dx:integer,dy:integer);
3   type keys = {SENSOR,MEAN};
4   signal DELIVER:integer;
5
6   agent Sample (dir: Direction, radius: integer[1..16]) =
7     var mean: integer; s:integer; deltaV: Delta; dirs:Direction set;
8
9     activity start =
10      mean := 0; deltaV := Delta(0,0);
11    end;
12
13    activity percept =
14      enoughinput := 0;
15      if not rd?(0,SENSOR,s?) then s := 0; end;
16      mean := mean+s;
17      transition*(percept,move);
18      if not inbound(dir) then dirs := {}
19     elsif deltaV.y = 0 and deltaV.x > 0 then dirs := {NORTH,SOUTH,EAST}
20     elsif deltaV.y = 0 and deltaV.x < 0 then dirs := {NORTH,SOUTH,WEST}
21      elsif deltaV=Delta(0,0) then dirs := {NORTH,SOUTH,WEST EAST}
22      elsif deltaV.y > 0 then dirs := {NORTH}
23      else dirs := {SOUTH} end;
24      for nextdir in dirs do
25        incr(enoughinput);
26        eval(fork(nextdir,radius));
27      end;
28      transition*(percept,goback,enoughinput=0);
29      wait for child agents delivering sensor values
30    end;
31
32    activity move =
33      mean := 0;
34      deltaV := deltaV+delta(dir);
35      moveto(delta(dir));
36    end;
```

```
37
38  activity goback =
39    if deltaV <> Delta(0,0) then
40      deltaV := deltaV+delta(backdir(dir));
41      moveto(delta(backdir(dir)));
42    end;
43  end;
44
45  activity deliver =
46    if deltaV = Delta(0,0) then
47      out(MEAN,mean/(radius*2+1)**2)
48    else
49      signal($parent,DELIVER,mean);
50    end;
51  end;
52
53  handler DELIVER(v:integer) =
54    mean := mean + v;
55    decr(enoughinput);
56  end;
57
58  function delta(dir:Direction):Delta =
59    case dir of | NORTH -> Delta(0,1) | WEST -> Delta(-1,0) | .. end;
60  end;
61  function inbound(nextdir:Direction):bool =
62    case nextdir of | NORTH -> dy<radius | WEST -> dx > -radius | ..
63    end;
64  end;
65  function backdir(dir:Direction):Direction =
66    case dir of | NORTH -> SOUTH | WEST -> EAST | .. end;
67  end;
68
69  transitions =
70    start -> percept;
71    percept -> move;
72    goback -> deliver;
73    deliver -> exit
74  end;
75
76  .. some other agent ..
77  id := new Sample(ORIGIN,2);
```

2.9.5 Agent Identification in AAPL

The *AAPL* offers symbolic variables for the identification of agents, `$self` for self-referencing and `$parent` for the identification of the parent agent if the

current agent was forked. There are basically two approaches assigning agents a unique identifier, a natural number:

1. A local unique identifier number handled by the agent manager extended with a delta-distance vector Δ for migrated agents;
2. A random number in the range [a..b].

The first approach requires the locking of an agent identifier number on the node where the agent was created until the agent process is terminated, which can happen on a different node. The first approach ensures the uniqueness of agent identifiers and the required bit-width of the identifier depends on the maximal extent of the network and the maximal number of agents that can be processed on a node. The second approach requires a large value range to minimize the collision probability, which usually requires at least 16 bit for an identifier.

2.9.6 AAPL Activities, Transitions, Composition, and Subclasses

The *AAPL* definition of the agent ATG behaviour is partitioned in the definition of activities and a transitions block. An activity is like a procedure that can use local storage defined at the beginning of the activity body. There is a sequentially ordered execution model for the statements of the activity body. Activities perform computation, modification of agent data and global data, migration, communication, and agent management. An activity can use temporary local storage variables, only visible in the activity body and valid only during the activity is processed, defined in the beginning of the activity body.

The transitions section of an agent class definition specifies an initial transition set with conditional and unconditional transitions between activities of the current agent class, shown in Definition *2.6*. There may be several transitions out going from the same activity and different incoming transitions from other activities.

Transitions can be assigned to a subclass. A subclass transition usually set only handles a part of the ATG. The root class always handles the entire ATG. Subclasses can be switched and modified at run-time providing a sub-classing of agents and their respective ATG behaviour. The main usage and the highest benefit of transition set switching is achieved during forking or creating of new agents. The deployment of multiple transition sets (sub-classes) provides an efficient way to create new (child) agent behaviours by still preserving the current (parent) behaviour.

epubli, ISBN 9783746752228 (2018)

Def. 2.6 *AAPL agent activity and transition definitions, which can be encapsulated in sub-classes.*

Activity Definition

```
activity ac_i =
  definitions   var x:DT; var* l:DT;
  statement;
  statement;
  ..
end;
```

Short Notation

```
α ac_i { ..
  Σ: {x_1,..} σ:{l_1,..}
  statement;
  statement;
  ..
}
```

Transitions Definition

```
transitions =
  a_i → a_j ⟦ : cond ⟧ ;
  ..
end;
```

```
Π {
  a_i → a_j ⟦ : cond ⟧
  ..
}
```

Subclass Definition

```
subclass sc =
  definitions   var x:DT; var* l:DT;
  .. imports    use x;
  activities    activity a_j = .. end;
  .. imports    use a_j;
  transitions   transitions = .. end;
end;
transitions sc = .. end;
```

```
φ sc: { .. }
↓ x
↓ a_j
π { .. }
```

Creation of Agents and Subclass Agents

```
id := new AC(x_1,x_2,..);
id := new AC.sc(x_1,x_2,..);
id := fork (x_1,x_2,..);
id := fork sc(x_1,x_2,..);
id := new ⟦ empty ⟧ AC;
.. ATG composition ..
run(id,x_1,x_2,..);
```

```
id ← Θ+ AC(x_1,..)
id ← Θ+ AC.sc(x_1,..)
id ← Θ→ (x_1,..)
id ← Θ→ sc(x_1,x_2,..)
id ← Θ AC

id' ← Θ→ (id,x_1,..)
```

Activities and transitions can be bound to subclasses, creating ATG sub-graphs. All subclasses initially belonging to the root class. Creating of agents from this root class therefore includes all sub-classes. But child agents can be forked by using the subclasses, shown below. The effect of sub-classing at run-time depends strongly on the agent platform architecture, which may only partially support sub-classing at run-time. Subclasses can import activities (and variables) from the parent class.

Subclass definitions allow the partition and decomposition of the entire ATG in sub- ATGs. A subclass restricts the root class (which includes all the subclasses) to a restricted or scope-limited set of variables, transitions, and activities.

If there are no variable definitions inside a subclass, the full root data set is inherited. If there is at least one subclass variable definition, no root variables are inherited!

All activities defined in the subclass are only visible and usable inside the subclass. All root class activities for which subclass activities have transitions to must be imported. The transition set of a subclass is independent of the root class transition set and does not inherit any root class or other subclass transitions.

Some special statements or statements with a special semantic appearing inside the subclass section are listed below:

var x:DT; var* l:DT;

Defines persistent and temporary data only used by subclass agents.

use A,T,v;

Imports an activity, transition, or variable from the root or another subclass.

SC.A

Reference of a subclass activity

A subclass definition can be used to define a subset of transitions only, which can be simplified by marking a transition definition section with the subclass name.

An agent class can define additional computational functions and procedures, which can be used in activities or conditional transition expressions, shown in Definition 2.7. Functions can be used in transitional expressions if they have no side effects (i.e., not modifying body variables and not using I/O statements).

Def. 2.7 *AAPL function and procedure definitions*

Function Definition	Short Notation
function f(par_1,par_2,..):T =	f:(x,y,..) → {..}
definitions	
statements	
return *expr*;	
end;	
procedure p(par_1,par_2,..) =	p:(x,y,..) → {..}
definitions	
statements	
return;	
end;	

2.9.7 AAPL Data Types

Core Types

The *AAPL* core type set consists of integer, natural, float, boolean, char, and string (text) types, summarized in Definition 2.8. The integer type can be sub-typed (value range) to support efficient program and hardware synthesis from *AAPL* specifications. Additionally, there is a low-level word type for generic use (bit-vector type). Though real type value arithmetic is expensive regarding hardware resources, the generic real (floating-point) and the fixed point real type are included in the set of core data types. The fixed point real type can be limited by a sub-range specification, providing important information for the hardware synthesis and enabling the transformation of real to integer type arithmetic, vital for low-resource processing platforms.

Def. 2.8 *AAPL core data types and variable definitions*

```
Core Data Types
DT = {integer, natural, word, real, fixed, boolean, char, text}

Agent Body Variable Definitions
var x: DT; var* x: DT;
var x: integer [[A..B]];              [[sub-range]]
var w: word[bit-size];                true-bit scaled
var r: real [[A..B]]                  scaled fixed point real type
var f: fixed[N1..N2,F1..F2]           scaled fixed point real type
```

Arrays

One- and multidimensional arrays of scalar and composed types are supported, summarized in Definition 2.9.

Def. 2.9 *AAPL array definition (type and object instantiation in one statement)*

```
Array Definition
var A[size]: T;
var A[a..b]: T;
var M[size1,size2,..]: T;

Array Element Access
  A[index] := expr;
  x := A[index];
  y := M[index1,index2,..];
```

Composed Types

AAPL provides common record structure, enumeration, and static set types, summarized in Definition *2.10*. There are no record type references to avoid a restriction to memory-based data processing architectures. List types are provided, but list objects are unavailable on all platforms. Lists are ordered, in contrast to sets, which are always unordered (the order is insignificant). Sets objects are available on all platforms (*PCSP*/*PAVM*).

Def. 2.10 *AAPL definition of user defined types (product and simple sum types) and object instantiation*

```
Record Types
type Tr = (e1:T, e2:T, ..);          var x:Tr;
          .. x := Tr(v1,v2..); .. x.v1 := x.v2; ..

Symbol (Enumeration) Types
type Ts = {S1 ,S2, .. };             var x:Ts;
          .. x := S1; ..
```

Def. 2.11 *AAPL definition of sets and list types*

```
Set Type
type Ts = {S1 , S2, .. };            var e:Ts;
                                     var s:Ts set;
..
  s := {};                           Empty set
  s := s + {S2,S3};                  Add elements to the set
  s := s - {S3};                     Remove elements from set
  e := Random(s);                    Return one element of s
  for e in s do ..                   Iterate over sets
  for e in {S2,S3} do ..
..
List Type
                                     var l:T list;
..
  l := [];                           Empty list
  l := l + [v1;v2];                  Add elements to the tail
  e := Random(s);                    Return one element
  for e in l do ..                   Iterate over lists
  for e in [e2;e3] do ..
..
```

Set Types

Sets can be constructed from symbolic enumeration types, shown in Definition *2.11*. A set contains each enumeration element only zero or one times. Initially a set variable is empty (empty set {}). New set elements can be added by using set addition and subtraction operations (+,–). Internally set variables are represented with bit-fields, with each bit field assigned to one symbol element. Set operations (+,–) will enable or disable the respective bits of the bit-field variable.

2.9.8 AAPL Computational Statements

The set of computational statements consists of data (assignment) and control flow related statements. Computational statements can be used in the body of activities, handlers, functions, and procedures, shown in Definition *2.12*. Expressions can be used additionally in transition sections. Expressions are used in assignments, branches, function applications, and loops. There are arithmetic, relational, boolean, and bit-wise logical operations.

Def. 2.12 *AAPL expressions and data statements [x: variable, v: value, ε: expression]*

```
Assignment and Expressions

x := ε;
          ε ::= v ‖ x ‖ (tp) ‖ ε op ε ‖ ( ε )
          op ::=
            + - * / mod |x|
            and or not land lor lnot lsl lsr
            < <= > >= = <>
          tp ::= ε ‖ ε , tp
incr(x[,Δ]); decr(x[,Δ]);    ↔     x:=x +/- Δ;  (w/o Δ: Δ=1)
```

2.9.9 AAPL Control Statements

There are different branch statements available passing the program flow to an alternative statement or a block of statements depending on boolean conditions or value matching expressions, summarized in Definition *2.13*. Branches with static conditions (which can be resolved at compile time) can appear on module top-level, too., supporting conditional compiling (there is no syntax preprocessing in *AAPL* like in *C*).

There are different loop statements available for repetitive execution of statements. Each loop repeats the execution of the loop body as long as a boolean condition is satisfied. A counting loop iterates a list of values, either specified explicitly by a set/list or implicitly by a range set constructor. The for-loop can be used to iterate over sets and lists, too. Furthermore, multi-iterator loops are supported.

epubli, ISBN 9783746752228 (2018)

Def. 2.13 *AAPL control statements [x: variable, v: value, ε: expression]*

```
Conditional Control Statements

if ε then S1; ⟦ else S0; ⟧ end;
if ε1 then .. elsif ε2 then .. ⟦ else .. ⟧ end;
case ε of | V1 => S1; | V2 => S2; .. ⟦ else S0; ⟧ end;

Iterative Control Statements

for i := a to ⟦ downto b ⟦ by c ⟧ do S; end;
for i in x do S; end;
for i in x, j in y,.. do S; end;
while ε do S; end;
repeat S; until ε;
```

2.9.10 AAPL Communication

Tuple-spaces and signals are the central inter-agent communication services provided by the agent processing platform. There is one set of n-dimensional spaces accessible in each node domain. Hence, tuple-space communication can only be used by agents currently processed on the same network node. In contrast, signals can be propagated in the network domain. Signals can be used to send events carrying simple data (one argument value) to other agents, commonly instantiated from the same root agent class. The agent identity number must be known by the sender. To receive signals, an agent must implement a signal handler, primarily modifying agent data. Tuple-spaces can be used by different agents (not necessarily belonging to the same class) to exchange simple or complex data with a producer-consumer synchronization model. Tuple-spaces can be accessed without any special environment like a handler, hence being more flexible.

Signals

A signal can be defined within an agent class or outside on top-level, optionally specifying the data type of the signal argument. The signal API is given in Definition 2.14.

Signal handlers are defined within the agent class. There may be at most one handler for each signal. Signal handlers are usually executed pre-emptive and concurrently to an activity processing of an agent. Signals are queued by the agent processing platform, but not necessarily preserving the order of the arrival of different signals. Signals are addressed by giving the agent identity number, which is unique in each node domain, and in the global domain extended by the relative displacement vector if an agent has migrated to another node. Hence, signals are usually only used within groups of agent, mostly parent-child groups.

epubli, ISBN 9783746752228 (2018)

Def. 2.14 *AAPL Signal Communication*

Signal Definition

```
signal S1,S2,.. [[ : DT ]];
```

Signal Raising

```
send(ID,Si,[[arg]]);                    ID={$parent,$self,agent-id}
reply(Si,[[arg]]);
broadcast(AC,DX,DY,S,[[arg]]);
sendto(TO,Si,[[arg]]);                  TO={DIR,PATH,node-id}
```

Signal Handler Definition

```
handler Si [[([[val]] p)]] = .. end;
```

Timer Installation and Signal Handler

```
signal Ti [[ : DT ]];
timer+(timeout,T);
timer+(timeout,T,V);
timer-(T);
handler Ti [[([[val]] p)]] = .. end;
```

A signal can be sent from an agent to itself, commonly used with timers raising a signal after a time-out has occurred, shown in Definition 2.14.

Peer-to-peer signals are commonly used to synchronize parent-child agents. The broadcast of a signal to all agents of a specific agent class and within a limited spatial range (in node units) goes beyond the parent-child relationship of agents, and is more generic.

Group specific signal propagation is actually not supported because there is no agent group management at all. Groups must be managed by the agents themselves, for example, shown in the distributed feature recognition Algorithm 9.1 using tuple-space interaction and temporary tuple markings. An agent that received a signal can reply (within the currently executed signal handler) with any signal to the original sender using the `reply` operation.

Tuple Spaces

Multi-Agent interaction and synchronization is provided by a tuple-space database server available on each node (service provider), summarized in Definition 2.15.

epubli, ISBN 9783746752228 (2018)

Def. 2.15 *AAPL tuple space communication*

Generation of Tuples and Markings	Short Notation
out(v_1,v_2,..);	$\nabla^{+}(v_1,v_2,..)$
mark(*timeout*,v_1,v_2,..);	$\nabla^{T}(v_1,v_2,..)$
Search and Removal of Tuples	
in(v_1,x_1?,..);	$\nabla^{-}(v_1,x_1?,..)$
in(v_1,x_1?,?,?,..);	$\nabla^{-}(v_1,x_1?,?,?,..)$
stat := try_in(*timeout*,..,);	$\nabla^{?-}(tmo,..)$
Search and Reading of Tuples	
rd(v_1,x_1?,..,v_2,..);	$\nabla^{\%}(v_1,x_1?,..)$
rd(v_1,x_1?,?,?,..,v_2,..);	$\nabla^{\%}(v_1,x_1?,?,?,..,v_2,..)$
stat := try_rd(*timeout*,..);	$\nabla^{?\%}(tmo,..)$
Testing of Tuple Existence	
exist?(v_1,?,?,..);	$\nabla^{?}(v_1,?,?,..)$
Removal of Tuples	
rm(v_1,?,?,v_2,..);	$\nabla^{\times}(v_1,?,?,..)$
Generation of Active Tuples	
eval(id,v_1,?,?,v_2,..);	$\nabla^{+-}(\mathrm{id},v_1,?,?,..)$
Distributed Tuple Space/Remote Tuple Access	
store(*to*,v_1,v_2,..);	$\nabla^{+}(v_1,v_2,..) \rightarrow to$
collect(*to*,v_1,x_1?,..);	$\nabla^{-}(v_1,x_1?,..) \rightarrow to$
copyto(*to*,v_1,x_1?,..,v_2,..);	$\nabla^{\%}(v_1,x_1?,..) \rightarrow to$
Multi-Pattern Selector	
res:=alt((v_1,x_1?,..)|(..));	$\nabla^{*-}((v_1,x_1?,..)\|(..))$
res:=try_alt(*tmo*,(v_1,x_1?,..)|(..));	$\nabla^{?*-}(tmo,(v_1,x_1?,..)\|(..))$

An agent can store an n-dimensional data tuple (v_1, v_2, ..) in the database by using the out(v_1,v_2,..) statement (commonly the first value is treated as a key). A data tuple can be removed or read from the database by using the in(v_1,p_2?,v_3,..) or rd(v_1,p_2?,v_3,..) statements with a pattern template consisting of a set of formal (variable,?) and actual (constant) parameters, which can be expression terms. These operations block the agent processing until a matching tuple is found and was stored in the database by another agent. There are non-blocking versions of these operations, respectively in?(*tmo*,..) and rd?(*tmo*,..), limiting the blocking with a time-out (which can be zero), and combine a tuple existence test with the input operation.

These simple operations solve the mutual exclusion problem in concurrent systems easily. Only agents processed on the same network node can

exchange data in this way. Simplified the expression of beliefs of agents is strongly based on the *AAPL* tuple database model. Tuple values have their origin in environmental perception and computation bound to a specific node location.

The existence of a tuple can be checked by using the `exist?` function or with atomic test-and-read behaviour using the `in?/rd?` functions. A tuple with a limited lifetime (a marking) can be stored in the database by using the `mark` statement. Tuples with exhausted lifetime are removed automatically (by a garbage collector). Tuples matching a specific pattern can be removed with the `rm(`v_1`,?,`v_3`,..)` statement.

The `alt` operation enables listening on tuples matching different patterns. If a tuple matches one of the patterns it is returned. The `alt` operation always return the tuple or none.

2.9.11 AAPL Migration

Migration of agents (preserving the local data and processing state) to a neighbour node is performed by using the `moveto(`*DIR*`)` statement, assuming the arrangement of network nodes in a mesh- or cube-like network. To test if a neighbour node is reachable (testing connection liveliness), the `link?(`*DIR*`)` statement returning a Boolean result can be used, summarized in Definition *2.16*.

Def. 2.16 *AAPL mobility and connectivity [DIR: symbolic direction, d: relative numeric direction value for a specific dimension)*

Migration	**Short Notation**
moveto(*DIR*|*PATH*)	⇔(*DIR*)
moveto(d_1,d_2,..)	⇔(d_1,d_2,..)
moveto(*id*)	⇔(*id*)
Link Test	
link?(*DIR*|*PATH*)..	?Λ(*DIR*) ?Λ(d_1,d_2,..) ?Λ(*id*)
Directions	
type DIR = {NORTH, SOUTH, WEST, EAST}ω = {..}	

The set of symbolic directions is defined by a relation function mapping symbolic directions names, e.g., {NORTH, SOUTH, WEST, EAST}, on a delta distance vector specifying a neighbour node location. Alternatively, the delta distance vector can be used immediately in the `moveto` and `link?` operations, .i.e., `moveto(`d_1,d_2,..`)` and `link?(`d_1,d_2,..`)` with d_i={-1,0,1}, shown in Definition *2.16*. To extend the deployment of agents to generic network environments found on the Internet domain a specific node identifier *id* can

be passed instead a geometrical distance vector, for example, a URL or IP address.

2.9.12 AAPL Reconfiguration

Behavioural Modification. An activity itself is immutable, but the composition of ATGs at compile and at run-time is dynamic. An arbitrary set of different ATGs $G=\{g_1, g_2, ..\}$ with $g_i=\langle A' \subseteq A, T' \subseteq T, V' \subseteq V \rangle$ can be composed of a static set of activities and transitions known at compile time. The capabilities of the ATG composition and reconfiguration at run-time depends on the particular agent processing platform and the agent behaviour implementation (programmable contrary to application specific). But at least the reconfiguration by modifying the set of active transitions is available on all platforms.

The activity composition of an agent at run-time can be limited by the agent platform due to side effects and storage dependencies existing in the activities (though storage references exists in transitions, too), the implementation depends on the used platform architecture, and is primarily only supported by sub-classing of the ATG itself, which must be known at design time.

Def. 2.17 *AAPL behavioural reconfiguration and composition modifying the ATG*

Transitions	Short Notation
`transition+ [[C]] ([[id,]] a1,a2,c);`	π^+ `[[C]] ([[id,]] a1,a2,c)`
`transition* [[C]] ([[id,]] a1,a2,c);`	π^* `[[C]] ([[id,]] a1,a2,c)`
`transition- [[C]] ([[id,]] a1,a2);`	π^- `[[C]] ([[id,]] a1,a2,c)`
Activities	
`activity+ [[C]]([[id,]] a1,a2,...);`	α^+ `[[C]]([[id,]] a1,a2,...)`
`activity- [[C]]([[id,]] a1,a2,...);`	α^- `[[C]]([[id,]] a1,a2,...)`

The transition reconfiguration can be used to create child agents with a modified behaviour forked from a parent agent, shown in Example 2.3. Before a child agent is created (either forked or created from the template using the `fork` or `new` operations, respectively) the transition network can be modified and immediately restored after the child agent was instantiated. Basically the same behaviour can be achieved by using behavioural subclasses.

In this example, a parent agent performs a transition from the *replicate* to the *wait* activity, whereas the child agent transitions after forking to the *move* activity. In the first case (a) the *move* activity is part of the root class (top-level), and in the second case (b) it is part of the subclass *childsc*.

Ex. 2.3 *Practical use of the AAPL transition reconfiguration for behavioural modification with (a) Transition reconfiguration, and (b) Subclass specification for the instantiation of child agents*

```
(a)
  activity replicate =
    transition*(replicate,move);
    eval(fork(..));
    transition*(replicate,wait);
  end;

  activity wait = .. end;
  activity move = .. end;
  transitions =
    ..
    replicate -> wait;
    ..
  end;
```

⇔

```
(b)
  activity replicate =
    eval(fork childsc(..));
  end;
  transitions =
    ..
    replicate -> wait;
    ..
  end;
  subclass childsc =
    use replicate;
    activity move = .. end;
    transitions =
      replicate -> move;
    end;
  end;
```

Ex. 2.4 *Free ATG composition at run-time using the configuration operations*

```
id := new empty AC;
activity+(id,A1,A4,A6,..);
transition+(id,A1,A4,cond);
transition+(id,A4,A6);
run id(arg1,arg2,);
```

epubli, ISBN 9783746752228 (2018)

The Example *2.4* shows a free ATG composition from an original root agent class by adding activities and transitions. A new empty class template is required.

2.9.13 AAPL Exception Handling

Exceptions are used to leave a control environment, for example a function, loop, or branch. Exceptions are propagated beyond control environments until an exception handler or handler environment catches the exception. Otherwise, an uncaught exception fault appears. In contrast to signals (which can be delivered to other agents), exception cannot carry arguments. The *AAPL* exception management and statements are shown in Definition *2.18*.

An exception raised within a nested control environment (nested branches/loops or function calls) is passed to the next higher environment level until a handler environment is reached. Exception handler environments can be nested, too. Exception not caught by a particular handler (without `else`-branch) are re-raised. If the exception reaches the activity control boundary, either a specific exception handler is called if defined, or the agent is terminated. Exception raised in functions called from activities are propagated to the calling activity.

Def. 2.18 *AAPL exception handling*

```
Exception Definition
exception ex1,ex2,..;

Exception Handler Environment (inside program blocks)
try
  statements;
except
 | ex1 => stmt1;
 | ex2 => stmt2;
 else stmt0;
end;

Exception Raising
raise exi;

Global Exception Handler Definition
handler exi = .. end;
```

2.10 AAPL Agents, Platforms, Bigraphs, and Mobile Processes

2.10.1 Networks of Agent Platforms

Mobile agents based on the *AAPL* model are associated with mobile processes that can migrate between network nodes, introduced in Sections 2.7 and 2.8. In Chapters 6 and 7 two different Agent processing platforms (APP) will be introduced (non-programmable application-specific and programmable APP). The *AAPL* behaviour and programming model is a common source for both APP implementations of agents. The state of a mobile agent process is a tuple $Ast=\langle Cst,Dst,Fst\rangle$, consisting of the control (*Cst*), data (*Dst*), and configuration (*Fst*) sub-state, independent of the implementation of the process and the APP used to execute a process. The migration of a process preserves the agent process state *Ast*. In the case of the programmable APP the agent processes are related to program code specifying the agent behaviour with embedded data and control sections, which carry the entire agent process state. Migration of agents means the transmission of program code including the agent behaviour. In the case of the non-programmable APP only the process state is transmitted, without the agent behaviour model, which is implemented in the APP, as shown in Figure 2.11.

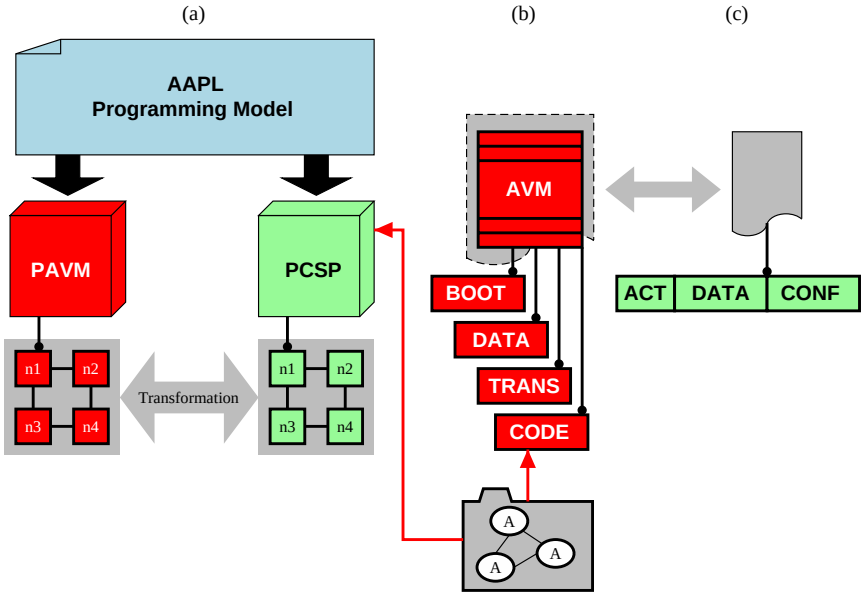


Fig. 2.11 *Two different Agent Processing Platform architectures and the association with mobile processes representing the state of the process. (a) Deployment of both APPs in heterogeneous networks with two different sub-networks (b) Mobile processes with program code embedding the process state (c) Mobile processes with a state vector only*

The composition of heterogeneous networks consisting of sub-networks with nodes of different APP classes requires a transformation module at least on one of the connected nodes of different sub-networks. These different APP class networks are related to different domains (sites) in the Bigraph model. The transformation module can use code templates and code morphing to transform a pure state-vector based mobile process in a code-based. The other direction is much more complicated, requiring the extraction of the agent state vector from the program code containing the embedded agent state. The transformation can be eased by adding transformation activities to the agent class.

2.10.2 AAPL MAS and the Π–Calculus

In Section *2.8* the relation of the *AAPL* behaviour and interaction model with the Π-Calculus was introduced. The following Example *2.5* shows a simple MAS behaviour model. A parent agent sends out child agents to each reachable neighbour node with the goal to deliver sensor values from the nodes, finally computing the mean value.

Ex. 2.5 *Simple explorer MAS behaviour model*

```
agent explorer(dx,dy) =
  var s,s1,dxb,dyb,n: integer; signal WAKEUP;
  activity init =
    dxb:=-dx; dyb:=-dy;
    if (dx,dy) <> (0,0) then moveto(dx,dy); end;
  end;
  activity percept =
    in(SENSOR,s?);
    if (dx,dy) = (0,0) then
      n:=0; out(SENSORMEAN,s);
      transition*(percept,init);
      if link?(-1,0) then eval(fork(-1,0)); incr(n); end;
      if link?(1,0) then eval(fork(1,0)); incr(n); end;
      if link?(0,-1) then eval(fork(0,-1)); incr(n); end;
      if link?(0,1) then eval(fork(0,1)); incr(n); end;
      transition*(percept,finalize,n = 0);
    end;
  end;
  activity goback = moveto(dxb,dyb); end;
  activity deliver =
    in(SENSORMEAN,s1?); s1:=s1+s; out(SENSORMEAN,s1);
    send($parent,WAKEUP);
    kill($self)
  end;
  activity finalize = kill($self); end;
  handler WAKEUP = decr(n); end;
  transitions =
```

```
    init -> percept;
    goback -> deliver;
end;
```

A possible evolving of such multiprocess system in terms of the Π-Calculus in shown in Example 2.6. It is assumed that another agent instantiates one explorer agent with the *AAPL* `new explorer(0,0)` statement. It is further assumes that the processing nodes are arranged in two-dimensional mesh-network with full connectivity and that tuple-space communication is pure computational and non-blocking. The explorer parent agent will then send out four child agents in all four directions, which will finally return and deliver the collected sensor values. The initial location is l_1, and the neighbour nodes in North, South, West, and East direction are at locations l_5, l_4, l_2, l_3, respectively.

Ex. 2.6 *Possible evolving of a multiprocess-system based on the AAPL behaviour in Example2.5 (Tuple space communication is assumed to be non-blocking and is neglected here) [s: event channel for WAKEUP signal handling for parent-child communication]*

$$(\text{new dx,dy})\ P_{1,\text{init}}@l_1\{0/\text{dx},0/\text{dy}\}$$

$$\downarrow$$

$$(\text{new } \underline{s})\ (P_{1,\text{percept}}@l_1 \ ||\ P_{2,\text{init}}@l_1\{-1/\text{dx},0/\text{dy}\} \ ||\ P_{3,\text{init}}@l_1\{1/\text{dx},0/\text{dy}\} \ ||\ P_{4,\text{init}}@l_1\{0/\text{dx},-1/\text{dy}\} \ ||\ P_{5,\text{init}}@l_1\{0/\text{dx},1/\text{dy}\})$$

$$\downarrow$$

$$\underline{s}?().P_{1,\text{percept}}@l_1 \ ||\ \text{goto}(l_2).P_{2,\text{percept}}@l_2 \ ||\ \text{goto}(l_3).P_{3,\text{percept}}@l_3 \ ||\ \text{goto}(l_4).P_{4,\text{percept}}@l_4 \ ||\ \text{goto}(l_5).P_{5,\text{percept}}@l_5$$

$$\downarrow$$

$$\underline{s}?().P_{1,\text{percept}}@l_1 \ ||\ P_{2,\text{goback}}@l_2 \ ||\ P_{3,\text{goback}}@l_3 \ ||\ P_{4,\text{goback}}@l_4 \ ||\ P_{5,\text{goback}}@l_5$$

$$\downarrow$$

$$\underline{s}?().P_{1,\text{percept}}@l_1 \ ||\ \text{goto}(l_1).\underline{s}!<>.P_{2,\text{deliver}}@l_1 \ ||\ \text{goto}(l_1).P_{3,\text{deliver}}@l_1 \ ||\ \text{goto}(l_1).P_{4,\text{deliver}}@l_1 \ ||\ \text{goto}(l_1).P_{5,\text{deliver}}@l_1$$

$$\downarrow$$

$$\underline{s}?().P_{1,percept}@l_1 \;||\; \underline{s}!<>.P_{3,deliver}@l_1 \;||\; P_{4,deliver}@l_1 \;||\; P_{5,deliver}@l_1$$

$$\downarrow$$

$$\underline{s}?().P_{1,percept}@l_1 \;||\; \underline{s}!<>.P_{4,deliver}@l_1 \;||\; P_{5,deliver}@l_1$$

$$\downarrow$$

$$\underline{s}?().P_{1,percept}@l_1 \;||\; \underline{s}!<>.P_{5,deliver}@l_1$$

$$\downarrow$$

$$P_{1,finalize}@l_1$$

2.11 AAPL Agents and Societies

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

1. The Micro-social level characterized by a tight bounding of agents, supported by *AAPL* parent-child groups with forking of the control and data state.
2. The group level characterized by a composition of larger structures and organisations, supported by *AAPL* mobility, replication, reconfiguration, and tuple-space coordination.
3. The global society level characterized by the dynamics of numerous agents at their evolution, supported by the *AAPL* mobility crossing platform barriers and the DATG reconfiguration and sub-classing capability. Organisations profit from behavioural differentiation and the constitution of specialists, well matching the DATG sub-classing feature.

The micro-macro relationship in Multi-Agent Systems and the role of the agent society created by organisation and interaction of agents is illustrated in Figure 2.12.

The structure of organisation is not a static structure, it is mainly a result of effects and interactions. The Bigraph model introduced in Section 2.7 can reflect such dynamic structures.

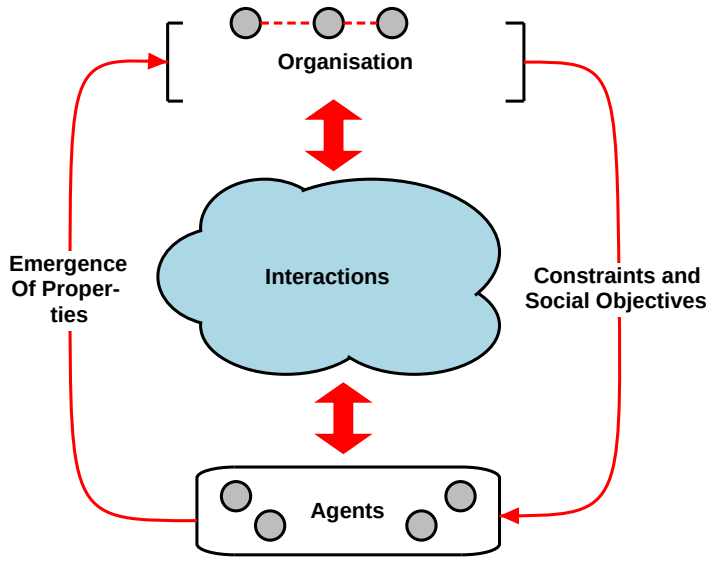


Fig. 2.12 *Agent societies by organisation structures and interaction*

2.12 AAPL Agents and the BDI Architecture

2.12.1 The BDI Architecture

The Belief-Desire-Intention (BDI) architecture is well-known agent behaviour and interaction model capable of rational behaviour and practical reasoning [WOO99], in contrast to, for example, procedural reasoning architectures (like PRS). The BDI architecture was originally proposed in [RAO95].

The top of Figure *2.13* outlines the BDI agent architecture and its components. The bottom part shows the relationship of the *AAPL* model with the BDI architecture, discussed below.

There are seven main components consisting of functions and databases that implements the rational behaviour [WOO99]:

1. A set of beliefs $\mathbb{B}$, representing information the agent has collected and computed from the environment.
2. A set of current Desires $\mathbb{D}$, the options, representing possible action executions and expected outcomes.
3. A set of current Intentions $\mathbb{I}$, representing the current focus of the agent.
4. A Belief Reasoning Function (*BRF*), which gets the input from current set of perception data $\mathbb{S}_n$ and the previous beliefs $\mathbb{B}_{n-1}$, and stores the output in the Belief database $\mathbb{B}$.

5. An Option Generation Function (*OGF*), which determines the available options, the desires of the agents, computed from the current beliefs and intentions.
6. A Filter Function (*FF*) as part of the agent's deliberation process, and which computes the next intentions of the agent.
7. An Action Selection Function (*ASF*), which selects actions to be performed basing on the current intentions.

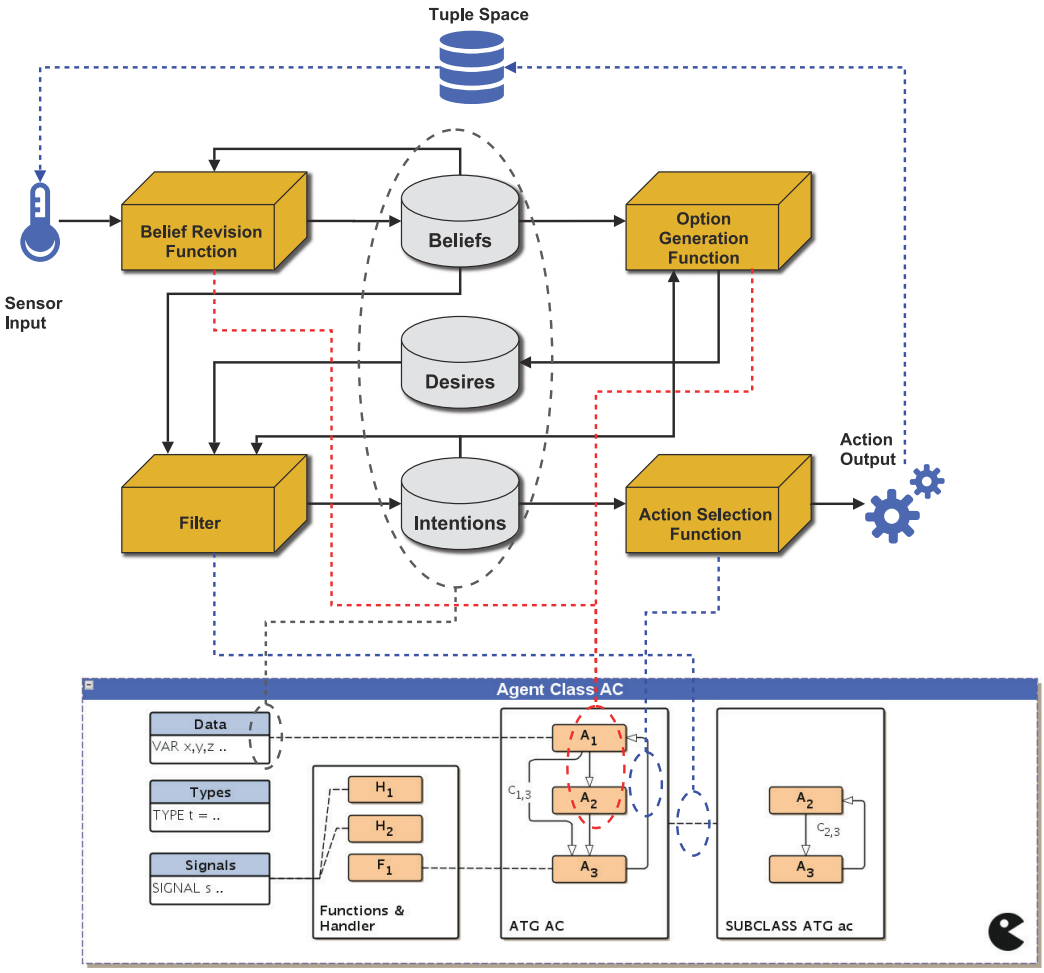


Fig. 2.13 *Relation of the BDI agent architecture to the AAPL behaviour model (agent class)*

The following equation summarizes the functions involved in BDI reasoning process. The state of an agent at any moment is given by the tuple ⟨𝔹, 𝔻, 𝕀⟩ with $\mathbb{B} \subseteq \boldsymbol{\mathcal{B}}$, $\mathbb{D} \subseteq \boldsymbol{\mathcal{D}}$, and $\mathbb{I} \subseteq \boldsymbol{\mathcal{I}}$, where $\boldsymbol{\mathcal{B}}$, $\boldsymbol{\mathcal{D}}$, and $\boldsymbol{\mathcal{I}}$ are the sets of all possible beliefs, desires, and intentions The outcome of the filter function *FF* giving the current intentions of the agent are either older intentions or newly adopted options. This is given by the following set constraint.

$$\begin{aligned}
&BRF : \wp(B) \times P \rightarrow \wp(B) \\
&OGF : \wp(B) \times \wp(I) \rightarrow \wp(D) \\
&FF : \wp(B) \times \wp(D) \times \wp(I) \rightarrow \wp(I) \\
&reasoning : P \rightarrow A \\
&execute : \wp(I) \rightarrow A \\
&\forall b \in \wp(B), \forall d \in \wp(D), \forall i \in \wp(I) : FF(b,d,i) \subseteq (I \cup D)
\end{aligned} \tag{2.2}$$

The φ(X) function represents the current sets of beliefs, desires, and intentions, *P* is the current perception, and *A* the currently executed action.

The reasoning function (originally proposed as the action function) repeatedly computes new Beliefs, Desires, and Intentions from the current perception and outputs the current action, shown in the Definition *2.19*.

Def. 2.19 *The BDI reasoning function*

```
1  DEF reasoning = p:P →
2    B := BRF(B,p)
3    D := OGF(D,I)
4    I := FF(B,D,I)
5    execute(I)
```

2.12.2 The AAPL-BDI Relationship

The Belief, Desire, and Intention databases are basically represented by and composed of the body variables of an agent. But agents also store data in the tuple-space database, which can contain BDI data, or at least pre-computed data that has an effect on the current ⟨𝔹, 𝔻, 𝕀⟩ database of an agent. Furthermore, the current transition set $\mathbb{T} \in \boldsymbol{\mathcal{T}}$ of an agent's ATG is embedded in the Intentions and Desire sets.

Sensor input comes from and action output affects tuple-space database data, representing both the environmental world, the agent itself, and other agents state, or at least the public visible parts.

The Belief Reasoning and Option Generation functions are related to the *AAPL* activities, and the Filter and Action Selection functions are related to the transitions set of an ATG.

2.13 Further Reading

1. M. Wooldridge, *An introduction to multiagent systems*, John Wiley & Sons, Ltd, 2009, ISBN 9780470519462.
2. Gerhard Weiss (Ed.), *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence*, The MIT Press, 2000, ISBN 0262232030
3. Jose M. Vidal, *Fundamentals of Multiagent Systems*. 2010.
4. R. Milner, *The space and motion of communicating agents*. Cambridge University Press, 2009, ISBN 9780521738330
5. M. Hennessy, *A Distributed PI-Calculus*. Cambridge University Press, 2007, ISBN 9780521873307
6. Jörg P. Müller, *The Design of Intelligent Agents - A Layered Approach*, LNAI 1177. Springer Berlin, 1996.
7. P. Leitão and S. Karnouskos, *Industrial Agents Emerging Applications of Software Agents in Industry.* Elsevier, 2015, ISBN 9780128003411

epubli, ISBN 9783746752228 (2018)