Chapter 7

PAVM: The Programmable Agent Platform

Agent Processing Platform based on a parallel and token-based Agent FORTH Virtual Machine Architecture

This Chapter introduces a programmable (generic) agent processing platform that is capable of processing *AAPL* based agents. The platform bases on a stack processor architecture with token-based agent processing. Various implementations are presented and discussed supporting the execution and migration of agents in heterogeneous environments including the Internet.

7.1 Stack Machines versa Register Machines

There are basically two different data flow processing architectures:

1. Register-based machines performing the data processing by accessing single registers and organized memory with random access memory behaviour;
2. Stack-based machines performing data processing by operating on the top elements of commonly multiple stacks. The top elements of the stacks are the operands of instructions, and they are directly connected to the computational and control unit of the processor.

Register-based machine commonly using the stack (LIFO) memory model, too, mainly for simplified temporary memory management, but the stack is embedded in the main data and program memory.

The stack storage model is ordered with respect to the Last-In First-Out data item ordering. The access of the stack memory is performed by using a push and a pop operation only. The push operation stores (writes) a new data item on the top of the stack, and the pop operation removes (reads) the top data stack item, shown in Figure 7.1.

This data order requires an ordering of operations of nested expressions that can only be relaxed by overlaying the LIFO access model with a Random Access Model (RAM).

Beneath the memory model differ the instruction set of the machines in the number of operands. Register-based machines usually carry one up to four operands (instruction arguments) resulting in a more complex instruction format, whereby stack-based machines uses mostly zero-operand instructions. These zero-operand instructions get their operands directly from the stacks, and computational results are immediately stored on the stack (with some exceptions like branches). This simple instruction format leads to a significantly simplified and less complex machine architecture.

On one hand there are several stack-based virtual machines. Examples are the *OCaML* byte code machine (with the predecessor ZINC [LER90]), the well-known *JAVA* VM, or the *FORTH* interpreter, discussed below.

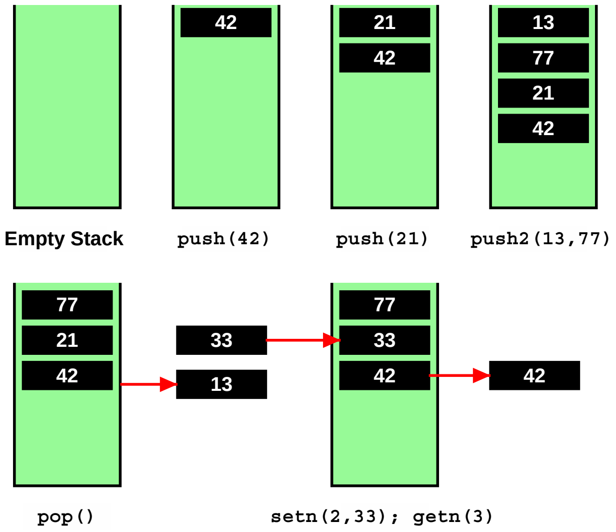


Fig. 7.1 *Stack storage model and simple access with the push(val) and pop() operations, extended and overlaid with a RAM access model using getn(index) and setn(index,val) operations.*

On the other hand, there are numerous register-based virtual machines, for example, the *DALVIK* VM invented by Google for their Android OS (evaluated and compared with the *JAVA* VM in [OH12]). The pro and contra of stack versa register machines are discussed extensively in [CAS08]. Stack-based machines are very popular in the context of functional programming languages, though *JAVA* is an imperative (with respect to the execution model) and memory-based language. Functional programming bases on expression evaluation, which can be efficiently mapped to the stack model, which imply scope bounds of data references constrained by the top elements of the stack, which can be inferred from functional expressions. In contrast, imperative programming (including the object-orientated programming model) with side effects require random access of storage, which can be efficiently supported by the stack model by overlaying the LIFO with a RAM access, extending the operational set with a get and set n-th operation. All modern stack-based VMs extend the stack model with these RAM operations, like the *OCaML* or *JAVA* VM.

The importance of the deployment of virtual machines in heterogeneous and multi-purpose sensor networks was already pointed out in [MUE07], using a software implemented tiny *JAVA* VM capable of interpreting a subset of *JAVA* byte code. Stack-based microprocessor raises in the seventies and eighties of the last century, but mainly disappeared as a general purpose processor.

7.2 Architecture: The PAVM Agent Processing Platform

The abbreviation *PAVM* can be interpreted as the Programmable Agent Forth Virtual Machine, but more precisely the *P* character is the abbreviation for a specific architecture feature, Pipe-lining, related to token-based agent processing, which is now discussed in more detail.

7.2.1 PAVM Overview

The requirements for the agent processing platform can be summarized to:

1. The suitability for microchip level (SoC) implementations;
2. The support of a stand-alone platform without any operating system;
3. The efficient parallel processing of a large number of different agents;
4. The scalability regarding the number of agents processed concurrently;
5. The capability for the creation, modification, and migration of agents at run-time.

Migration of agents requires the transfer of the data and control state of the agent between different virtual machines (at different node locations). To simplify this operation, the agent behaviour based on the activity-transition graph model is implemented with program code, which embeds the (private) agent data as well as the activities, the transition network, and the current control state encapsulated in code frames. It can be handled as a self-contained execution unit. A code frame can be modified by the agent itself or by the agent manager, responsible for the agent process control and the migration of the program code.

The execution of the program by a stack virtual machine (SVM) is handled by a task. The program instruction set consists of zero-operand instructions, mainly operating on the stacks of the VM. The VM platform and the machine instruction set implements traditional operating system services, too, offering a full operational and autonomous platform, with a hybrid RISC and CISC architecture approach. No boot code is required at start-up time. The hardware implementation of the platform is capable of operating after a few clock cycles, which can be vital in autonomous sensor nodes with local energy supply from energy harvesting. An ASIC technology platform requires about 500-1000 k gates (16 bit word size), and can be realized with a single SoC design.

A task of an agent program is assigned to a token holding the task identifier of the agent program to be executed. The token is stored in a queue and consumed by the virtual machine from the queue. After a (top-level) word (one activity in terms of the agent processing) was executed and leaves an empty data and return stack, the token is either passed back to the processing queue or to another queue.

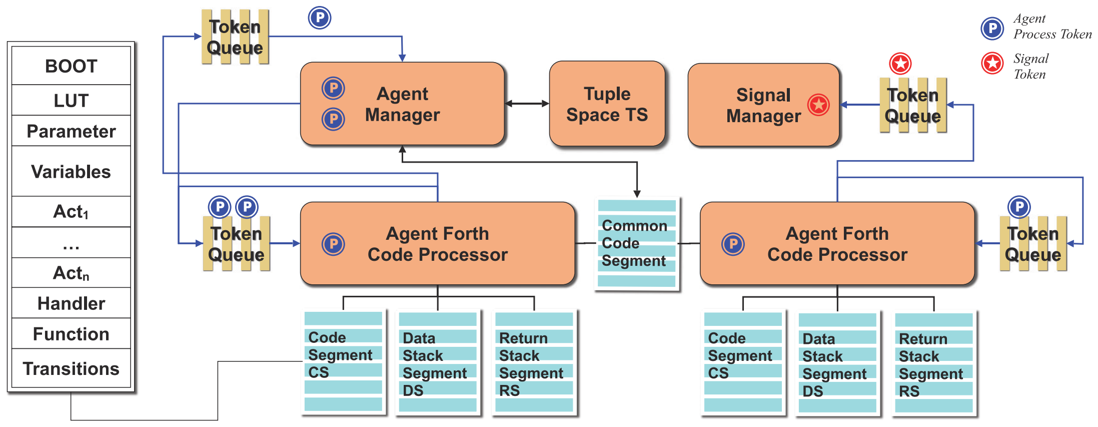


Fig. 7.2 *(Left) Code Frame Layout (Right) The Agent Forth Virtual Machine Architecture with a token-based agent processing.*

The token is passed, e.g., to the agent manager queue, enabling self-scheduling of different agent processes, shown in Figure 7.2.

7.2.2 Platform Architecture

The virtual machine executing tasks is bases on a traditional *FORTH* processor architecture and an extended zero-operand word instruction set (αFORTH), discussed in Section 7.3 Most instructions directly operate on the data (*DS*) and the control (*RS*, return) stack. A code segment (*CS*) stores code frame containing program code with embedded data, shown in Figure 7.3. There is no separate data segment. Temporary data is stored only on the stacks. The program is mainly organized by a composition of words (functions). A word is executed by transferring the program control to the entry point in the *CS*; arguments and computation results are passed only by the stack(s). There are multiple virtual machines with each attached to (private) stack and code segments. There is one global code segment (*CCS*) storing global available functions and code templates that can be accessed by all programs. A dictionary is used to resolve *CCS* code addresses of global functions and templates. This multi-segment architecture ensures high-speed program execution and the local *CS* can be implemented with (asynchronous) dual-port RAM (the other side is accessed by the agent manager, discussed below), the stacks with simple single-port RAM. The global *CCS* requires a Mutex scheduler to resolve competition by different VMs.

The register set of each VM consists of: $\Re$={*CF*, *CFS*, *IP*, *IR*, *TP*, *LP*, *A*, .. , *F*}. The code segment is partitioned in physical code frames. The current code frame that is processed is stored in the *CF* register.

The instruction pointer *IP* is the offset relative to the start of the current code frame. The instruction word register *IR* holds the current instruction. The *LP* register stores an absolute code address pointing to the actual relocation LUT in the code frame, and the *TP* register stores an absolute address pointing to the currently used transition table (discussed later). The registers *A* to *F* are general purpose registers.

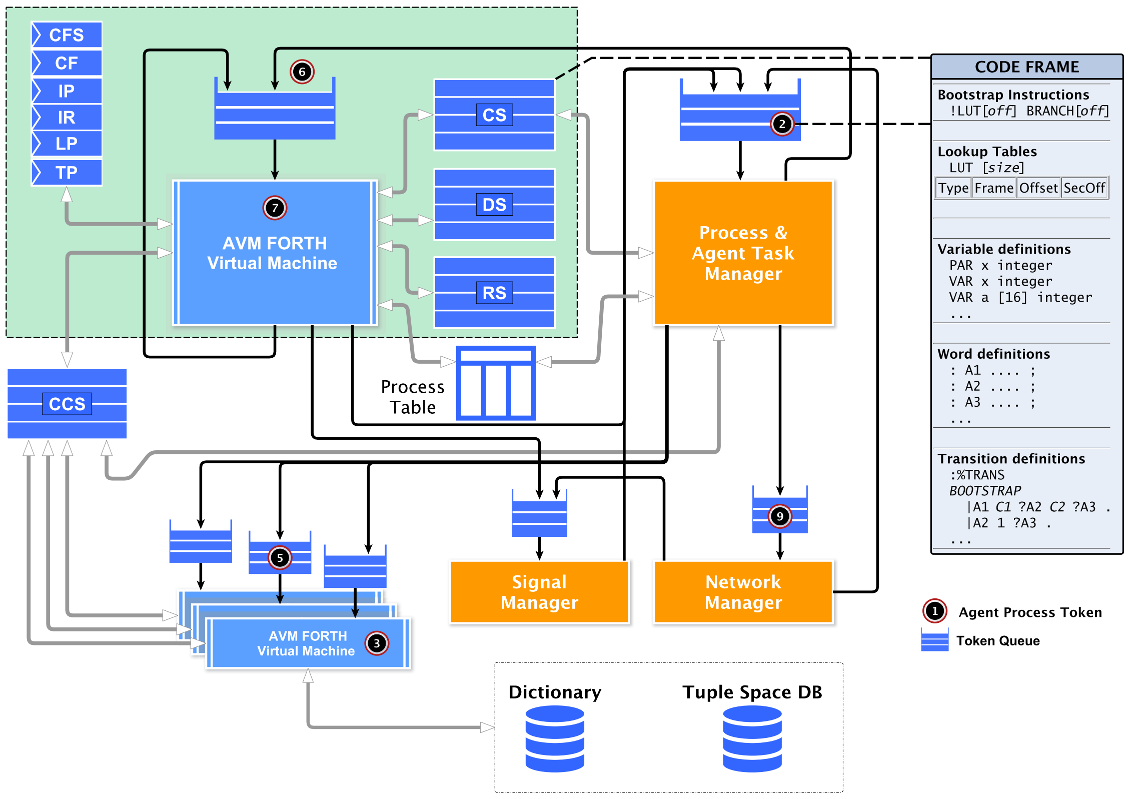
The program code frame (shown on the right of Figure 7.3) of an agent consists basically of four parts:

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

The transition table section can be modified by the agent by using special instructions, explained in Section 7.3.4 Furthermore, new agents can be created by composing activities and transition tables from existing agent programs, creating subclasses of agent super classes with a reduced but optimized functionality. The program frame (referenced by the frame pointer *CF*) is stored in the local code segment of the VM executing the program task (using the instruction pointer *IP*). The code frame loading and modifications of the code are performed by the virtual machine and the agent task manager only.

A migration of the program code between different VMs requires a copy operation applied to the code frame. Code morphing can be applied to the currently executed code frame or to any other code frame of the VM, referenced by the shadow code frame register *CFS*.

Each time a program task is executed the stacks are initially empty. After returning from the current activity execution the stacks are left empty, too. This approach enables the sharing of only one data and return stack by all program tasks executed on the VM they are bound to! This design significantly reduces the required hardware resources. In the case of a program task interruption (process blocking) occurring within an activity word the stack content is morphed to code instructions, which are stored in the boot section of the code frame, discussed later. After the process resumption, the stacks can be restored.



```
Bootstrap Instructions
  !LUT[off]  BRANCH[off]
Lookup Tables
  LUT  [size]
Type Frame Offset SecOff
Variable definitions
  PAR x integer
  VAR x integer
  VAR a [16] integer
  ...
Word definitions
 : A1 .... ;
 : A2 .... ;
 : A3 .... ;
 ...
Transition definitions
 :%TRANS
 BOOTSTRAP
    |A1 C1 ?A2 C2 ?A3 .
    |A2 1 ?A3 .
 ...
```

Fig. 7.3 *The Agent processing architecture based on a pipelined stack machine processor approach. Tasks are execution units of agent code, which are assigned to a token passed to the VM by using processing queues. The control state is stored in and restored from the process table. After the execution, the task token is either passed back to the input processing queue or to another queue of either the agent manager or a different VM. Right: The content and format of a code frame.*

STATE	*VM#*	*CFROOT*	*CFCUR*	*IP*	*ID*	*PAR*	*AWAIT*	*AWARG*	*POS*
Process state	Virtual Machine Number	Root code frame # of process	Current code frame # of process	Last/ Next IP offset	Process identifier number	Id. of parent process	The reason for waiting	Await argument (Key)	Delta position of migrated Process

Tab. 7.1 *Process table (PT) row format and description*

Each VM processor is connected with the agent process manager (PM). The VM and the agent manager share the same VM code segment and the process table (PT). The process table contains only basic information about processes required for the process execution. The column entries of a process table row are explained in Table 7.1.

7.2.3 Token-based Agent Processing

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

Tokens are coloured by extending tokens with a type tag. There are generic processing tokens, signal processing tokens, and data tokens, for example, appearing in compounds with signal processing tokens, discussed later.

Each VM interacts with the process and agent task manager. The process manager passes process tokens of ready processes to the token queue of the appropriate VM. Processes which are suspended (i.e., waiting for an event), are passed back to the process manager by transferring the process token from the current VM to the manager token queue.

7.2.4 Instruction Format and Coding

The width of a code word is commonly equal to the data width of the machine. There are four different instruction code classes:

1. Values
2. Short commands
3. Long command Class A
4. Long command Class B.

A value word is coded by setting the most significant bit of the code word (MSB) and filling the remaining bits (N-1, N machine word size) with the value argument. To enable the full range of values (full data size N bit), a sign extension word can follow a value word setting the n-th bit. A short command has a fixed length of 8 bits, independent of the machine word and data width. Short commands can be packed in one full-size word, for example, two commands in a 16 bit code word. This feature increases the code processing speed and decreases the length of a code frame significantly. The long commands provide N-4 (class A) and N-7 (class B) bits for argument values.

7.2.5 Process Scheduling and VM Assignment

The token-based approach enables fine-grained automatic scheduling of multiple agent processes already executed sequentially on one VM with a FIFO scheduling policy. A new process (not forked or created by a parent) must be assigned to a selected VM for execution. There are different VM selection algorithms available: Round-robin, load-normalized, memory-normalized, random. The VM selection policy has a large impact on the probability of a failure of a process creation and a process forking by a running process, requiring child agents to be created on the same VM!

7.3 Agent FORTH: The Intermediate and the Machine Language

The *FORTH* programming language corresponds to an intermediate programming language level, with constructs from high-level languages like loops or branches and low-level constructs used in machine languages like stack manipulation. The *αFORTH* (*AFL*) instruction set I_{AFL} consists of a generic *FORTH* subset I_{DF} with common data processing words operating on the data and return stack used for computation, a control flow instruction set I_{CF}, i.e., loops and branches, a special instruction set I_{AP} for agent processing and creation, mobility, and agent behaviour modification at run-time based on code morphing, and finally an agent interaction sub-set I_{AI} based on the tuple space database access and signals. The *AFL* language is still a high-level programming language close to *AAPL*, which can be used directly to program multi-agent systems. The *PAVM* agent processing platform will only support a machine language sub-set (*AML*) with a small set of special low-level instructions IAM for process control, so that $I_{\mathrm{AML}} \subset (I_{\mathrm{AFL}*} \cup I_{\mathrm{AM}})$, and with some notational differences. Several complex and high-level statements of I_{AFL} are implemented with code sequences of simpler instructions from the I_{AML} set, with some of them are introduced in Section 7.4 The (current) *AML* instruction set consists of 92 instructions, most of them are common *FORTH* data processing instructions operating immediately on stack values, and 31 complex special instructions required for agent processing, communication, and migration. The *AML* instruction set is unfixed and can be extended, which leads to an increased resource requirement and control complexity of the VM.

7.3.1 Program Code Frame

An αFORTH code frame (see Figure 7.4) starts with a fixed size boot section immediately followed by a program lookup relocation table (*LUT*). The instructions in the boot section are used to

- Set up the *LUT* offset register *LP* (always the first instruction);
- Enable program parameter loading (passed by the data stack);
- Restore stack content after migration or program suspending;
- Branch the program flow to the transition table section.

The program counter *IP* points to the next instruction to be executed. The *LUT* is a reserved area in the program frame initially empty, and is used by the VM to relocate variable, word, and transition table row references. A *LUT* row consists of the entries: {*Type*, *Code Offset*, *Code Frame*, *Secondary Offset*}. Possible row types are: *Type* = {`FREE`, `PAR`, `VAR`, `ACT`, `FUN`, `FUNG`, `SIGH`(*S*), `TRANS`}. The signal handler type `SIGH` is indeed a negative value, specifying the signal number *S* that is related to the signal handler.

Within the program code, all address references from the frame objects, i.e., variables, user defined words, and transitions, are relocated by the *LUT* at run-time. This indirect addressing approach eases the reconfiguration of the program code at run-time and the code migration significantly. If a program frame is executed the first time or after a migration, the code frame is executed on top-level, where the *LUT* is updated and filled with entries by processing all object definitions of the frame from the beginning to the end. Code inside user defined words are bypassed in this initialization phase. Variable, parameter, and word definitions (`var V`, `par P`, `:W`) update at initialization-time entries in the *LUT* (code offset, code frame), and transition branches `?A` updates entries at run-time (secondary offset specifying the relative offset of an activity call in the transition table section). On programming *AFL* level, generic function, activity, and signal handler word definitions are distinguished by different syntax (`:F`, `:*A`, `:$S`), whereas not on machine instruction *AML* level (`DEF`).

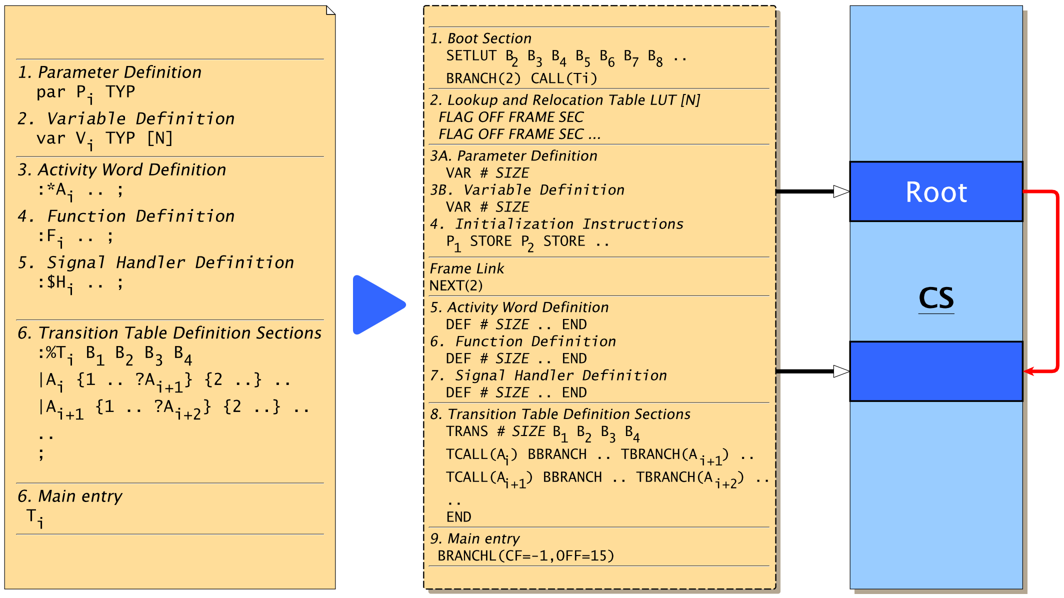
After the *LUT* section, parameter and variable object definitions (private agent data) follow and some top-level instructions used to initialize agent parameters with values passed by the data stack.

The main part of the code frame consists of activity, function, and signal handler word definitions (`:F .. ;`, `:*A .. ;`, `:$S .. ;`, with names F/A/S, respectively).

Finally, the transition table section is defined (`:%T .. ;`). A transition table consists of transition rows, which group all transitions outgoing from one specific activity, discussed later. Because of more than one transition table can be defined, but only one may be used by a process at one time, a top-level transition table call is required at the end of the frame. A transition table contains a small boot section (four words) at the beginning, too. This boot section is used for the control of process resumption after a suspending, whereas the code frame boot section is used primarily to store data.

Beside pure procedural activity words (without any data passing leaving the data stack unchanged) there are functional words passing arguments and results by using the data stack. Words not accessing private agent data can be exported (`::F`) to a global dictionary (transferring the code to a *CCS* frame) and reused by other agents, which can import these functions (`import W`) referenced by their name, which creates a *LUT* entry pointing to the *CCS* code frame and offset relative to this frame. Global functions may not access any private agent data due to the *LUT* based memory relocation.

The code segment of a VM is divided in fixed size partitions to avoid memory management with dynamic linked lists for free and used memory regions and memory fragmentation issues. A code frame always occupies a region of this fixed size code partition, the physical code frame, in the code segment of the respective virtual machine. Therefore, a single code frame is commonly limited to minimal 512 and maximal 2048 words of one code partition, depending on the VM implementation and *CS* overall size. In the case an agent program does not fit in one code partition, physical code frames can be linked forming one logical code frame, shown on the right side of Figure 7.4.



```
1. Parameter Definition
  par P_i TYP
2. Variable Definition
  var V_i TYP [N]
3. Activity Word Definition
  :*A_i .. ;
4. Function Definition
  :F_i .. ;
5. Signal Handler Definition
  :$H_i .. ;

6. Transition Table Definition Sections
  :%T_i B_1 B_2 B_3 B_4
  |A_i {1 .. ?A_i+1} {2 ..} ..
  |A_i+1 {1 .. ?A_i+2} {2 ..} ..
  ..
  ;

6. Main entry
 T_i
```

```
1. Boot Section
  SETLUT B_2 B_3 B_4 B_5 B_6 B_7 B_8 ..
  BRANCH(2) CALL(Ti)
2. Lookup and Relocation Table LUT [N]
 FLAG OFF FRAME SEC
 FLAG OFF FRAME SEC ...
3A. Parameter Definition
  VAR # SIZE
3B. Variable Definition
  VAR # SIZE
4. Initialization Instructions
  P_1 STORE P_2 STORE ..
Frame Link
NEXT(2)
5. Activity Word Definition
  DEF # SIZE .. END
6. Function Definition
  DEF # SIZE .. END
7. Signal Handler Definition
  DEF # SIZE .. END
8. Transition Table Definition Sections
  TRANS # SIZE B_1 B_2 B_3 B_4
  TCALL(A_i) BBRANCH .. TBRANCH(A_i+1) ..
  TCALL(A_i+1) BBRANCH .. TBRANCH(A_i+2) ..
  ..
  END
9. Main entry
 BRANCHL(CF=-1,OFF=15)
```

Fig. 7.4 *Logical code frame structure (left, AFL source code) and (optionally split) physical code frame (middle, AML machine code) with mapping to code partitions in the code segment (right).*

A physical code frame is specified by its address offset in the code segment, or by a partition index number (absolute index), or by a relative physical code frame number, relative to the first root frame of a process. The root frame always has the relative number 1. This relative physical code frame numbering is required to support code frame migration between different VMs, where absolute code frame addresses and index numbers changes, discussed later.

The last two words of the boot section are reserved and are used to control the code frame initialization and the current transition set table. Initially they contain the `{BRANCH(2),CALL(Ti)}` word sequence. At the end of the code frame there is a long branch to the last boot section word finally executing `CALL(Ti)`. If the initialization of the code frame should be omitted, the `BRANCH(2)` word is replaced with a `NOP` operation by using code morphing, discussed in the following sections.

The following subsections introduce the special *AFL/AML* instruction set required for code morphing, agent interaction, agent creation, and mobility. Most instructions get their arguments from the stacks and return results to the stacks. To illustrate the modification of stacks by instructions, a common stack notation is used: $(a_1 \; a_2 \; a_3 \; \text{--} \; r_1 \; r_2 \; r_3)$, showing the relevant top content of the stack before (left part) and after the instruction execution (right part), delimited by `--`. The top element of the stack is the right element (a_3/r_3 in this example). The return stack is prefixed with a `R` character.

7.3.2 Agent Processing

Agent processing involves the execution of activities and the computation and processing of transitions between activities based on private agent data (body variables). The transition computation is stored in the transition table words (`:%TRANS .. ;`). A transition table consists of transition rows, grouping all (conditional and unconditional) transitions for one outgoing activity. Each row starts with an activity word call `|Ai`. After the return from the activity and a process schedule occurred, a new activity transition is computed by evaluating Boolean expressions. The result of the computation is processed by a transition branch operation `?Aj`, which branches to a different transition row (starting with `|Aj`) if the condition is true, otherwise, the next transition is evaluated. If currently no condition is satisfied and the end of a transition row is reached, the process is suspended, and the process token is passed back to the process manager. In this case, the process will be only resumed by the process manager if a signal was delivered to and processed by the process (e.g., an event occurred).

AFL	*AML*	*Stack*	*Description*

`|Ai`	`TCALL(#)`	`( -- )`
		 `R( -- ip cf# )`
			Calls next activity word A_i. The word address offset and the code frame are taken from the LUT. The current code location (a call frame) is stored on the return stack. Only relative frame numbers may be used in call frames to enable process migration.

`?Ai`	`TBRANCH(#)`	`( flag -- )`	Branches to next transition row for start activity A_i if the flag is true. The relative branch displacement for the appropriate `TCALL(#)` target is first searched by using the LUT entry for the respective activity (Sec. off. col.). If this fails, the entire transition section is searched (and the result is cached in the LUT).

`{*n .. }`
 `{n .. }`
	`BBRANCH(Δ)`	`( -- )`	Dynamic block environment: a conditional branch that can be enabled (Δ>0, block disabled) or disabled (Δ<0, block enabled) using the `BLMOD` operation. If the branch is enabled, the block spawned by Δ is skipped.

`.`	`END`	`( -- )`	End marker, which marks the end of a transition table row. The process is suspended if reached in the transition section.

Tab. 7.2 *AFL/AML agent processing control instructions*

AFL	*AML*	*Stack*	*Description*
`?block`	`QBLOCK`	`( flag -- )`	Suspend code processing and saves the stacks if the flag is not zero. If a schedule occurs, the current data and return stack content must be transferred and morphed to the boot code section with a branch to the current *IP*–1, repeating the previous code word execution after process resumption.
`suspend`	`SUSPEND`	`( .. flag -- )` `R( .. -- )`	Suspend the execution. The current *CF* and *IP*+1 are saved in the current transition table boot section with a long branch. If *flag* = 1 then the code frame is fully re-initialized after resumption and the stacks must be already dumped to the boot section. If *flag* = –1 then the boot section is initialized and the stacks are dumped, after resumption the next instruction is directly executed w/o full code frame setup by jumping directly to the transition table boot section.

Tab. 7.2 *AFL/AML agent processing control instructions*

More than one transition table can exist and can be selected by using the `!t` statement. The reconfiguration of the transition table using the `t+`, `t-`, and `t*` statements (add, delete, replace) requires code morphing capabilities, different from those instructions introduced in Section 7.3.4 The reconfiguration of the transition table - basically reduced to enabling and disabling of transitions on machine level - bases on dynamic code blocks `{n .. }`, which can be enabled or disabled using the `blmod` (`BLMOD`) instruction, explained in Table 7.4. The `{*n .. }` block is enabled by default.

The generic program flow can be controlled using *AFL* and common *FORTH* branch and loop statements. On machine level (*AML*), there are only three branch operations: 1. a conditional relative branch `BRANCHZ(ΔIP)`, which redirects the program flow if the top of the data stack is zero, 2. an unconditional relative branch `BRANCH(ΔIP)`, and 3. a long inter-frame branch `BRANCHL(CF,IP)`.

7.3.3 Agent Creation and Destruction

New agents are created (or forked) by using a composition of the `NEW`, `LOAD`, and `RUN` operations, discussed in Section 7.4 The `suspend` (`SUSP`) operation is usually inferred by the compiler in conjunction with other blocking instructions. An agent can be destroyed by using the `kill` operation. Table 7.3 summarizes these operations.

AFL	*AML*	*Stack*	*Description*
`fork`	-	`( arg1 .. argn #args -- pid )`	Forks a child process. The child process leaves immediately the current activity word after forking, the parent process continues after the fork operation.
`create`	-	`( arg1 .. argn #args #ac -- pid )`	Creates a new agent process loaded from the agent class code template *#ac*.
-	`RUN`	`( arg1 .. #args cf# flag -- id )`	Starts a new process with code frame (from this VM), returns the identifier of the newly created process. The arguments for the new process are stored in the boot section in the code frame of the new process. If *flag* = 1 then a forked process is started. The boot section of the new code frame and the boot section of the transition table will be modified.
`kill`	`pid=self: CLEAR EXIT`	`( pid -- )`	Terminates and destroys an agent. For self-destruction the *aid* must be equal –1. Executing an exit operation with an empty stack (clear) terminates an agent.

Tab. 7.3 *AFL/AML agent creation and destruction instructions*

7.3.4 Agent Modification and Code Morphing

Code morphing is the capability of a program to modify its own code or the code of another program. Code morphing is used for:

- Modification of the boot section part of a code frame and the boot section of a transition table;
- Temporary state saving on process forking, migration, and suspend to dump stack data to the boot section;
- Copying variable, word, and transition tables to a new code frame;
- Modification of dynamic blocks, mainly used in the transition table, which enables or disables specific transitions.

Since data is embedded in the code frame of a process, code morphing is used here to modify data, too. Only twelve instructions supporting code morphing are required and are summarized in Table 7.4. There are explicit code morphing operations, which can be used on the programming level, and there are implicit code morphing operations embedded in other control instructions, for example, the `QBLOCK` and `SUSP` operations (see Table 7.2) used for modifying the code frame and transition table boot sections.

New code frames can be allocated using the `new` (`NEW`) instruction. The code frame can be allocated only from the VM of the current process. If the *init* argument is equal 1, then a default (empty) boot and LUT section is created, with sizes based on the current process. The code frame number *cf#* and the offset value *off* pointing to the next free code address in the morphing code frame is returned.

The `load` (`LOAD`) instruction has two purposes. First, it can be used to load a code template from the global code segment CCS, which is resolved by the agent class number using the global dictionary. Second, it can be used to copy the current process code to the new code frame, which was already allocated by using the `new` command. If the template or the current process code spawns more than one frame, additional frames are allocated and linked.

New agents are created (or forked) by using a composition of the `NEW`, `LOAD`, and `RUN` operations, discussed in Section 7.4

Code can be modified by using the `c>` (`TOC`), `v>c` (`VTOC`), `s>c` (`STOC`), and `r>c` (`RTOC`) instructions. All these operations are complex instructions with high operational power supported directly by the VM. Commonly the code morphing instructions are used by the compiler for agent creation, forking, and migration. Some process control instructions like `QBLOCK` or `SUSPEND` use code morphing implicitly to save the data and control state of the process in the boot sections.

All code morphing operations are applied to the code frame, which is referenced with the current value of the *CFS* register, which can point to the current process frame or to any other code frame (limited to the code segment of the current process). The code morphing frame can be loaded by using the `!cf` (`SETCF`) operation. Basically code morphing takes place by transferring code words from the data stack to a specified code offset position in morphing frame by using the `>c` (`TOC`) operation.

Code words (code snippets) can be pushed by using the `c>` (`FROMC`) operation, which copies code words following this word from the current process frame to the data stack. Value literal words can be created and transferred to the morphing frame with values stored on the data stack by using the `v>c` (`VTOC`) operation.

The current entire stack contents can be dumped to the morphing frame by using the `s>c` (`STOC`) operation, excluding the arguments of this operation.

AFL	AML	Stack	Description
`new`	`NEWCF`	`( init -- off`$^{\texttt{init=1}}$ `cf# )`	Allocates a new code frame (from this VM) and returns the code frame number. If *init* = 1 then a default boot and *LUT* section is generated, and the code offset is returned additionally.
`load`	`LOAD`	`( cf# ac# -- )`	Loads the code template of agent class *ac* in the specified code frame number or make a copy of the current code frame (*ac*=–1).
`c>`	`FROMC`	`( n -- c )`	Pushes the *n* following code words on the data stack
`v>c`	`VTOC`	`( v n off -- off' )`	Converts *n* values from the data stack in a literal code word and extension if required. The new code offset after the last inserted word is returned.
`>c`	`TOC`	`( c1 c2 .. n off -- )`	Pops *n* code words from the data stack and store them in the morphing code frame starting at offset *off*.

Tab. 7.4 *AFL/AML code morphing words (off: code offset relative to code segment start, cf#: code frame number, ref#: LUT object reference number)*

AFL	***AML***	***Stack***	***Description***
`s>c`	`STOC`	`( .. off -- off' )` `R( .. -- )`	Converts all data and return stack values to code values and store them in the morphing code frame starting at offset *off*. Return the new offset after the code sequence.
`r>c`	`RTOC`	`( off ref# -- off' )`	Transfers the referenced object (word, transition, variable) from the current process to the morphing code frame starting at offset *off*. Returns the new offset after the code sequence.
`!cf`	`SETCF`	`( cf# -- )`	Switches code morphing engine to new code frame (number). The root frame of the current process can be selected with *#cf*=–1.
`@cf`	`GETCF`	`( -- cf# )`	Gets current code frame number (in CS from this VM).
t+(A_i,b#) t-(A_i,b#) t*(A_i,b#) !t(T_i)	`BLMOD` `TRSET`	`( ref# b# v sel -- )` `( ref# -- )`	Modifies the transition table that can be selected by the `!t` statement. Each transition bound to an outgoing activity is grouped in a dynamic block environment. The transition modifiers reference the block number in the respective transition row. The t-operations are reduced to the AML operation `BLMOD` (modify a dynamic block). `BLMOD` can be used for global dynamic blocks, too (*sel*=1: transition, *sel*=0: activity, *sel*=-1:top-level).

Tab. 7.4 *AFL/AML code morphing words (off: code offset relative to code segment start, cf#: code frame number, ref#: LUT object reference number)*

Entire words from the current process can be copied to the morphing frame by using the `r>c` (`RTOC`) operation.

The transition table can be modified by using dynamic blocks and configurable block branches (`BBRANCH`), which can be enabled or disabled by using the `BLMOD` operation, inferred by high-level transition configuration functions `t+`, `t-`, and `t*` enabling and disabling transitions.

7.3.5 Tuple Database Space

The access of the database tuple space transfers n-ary data tuples to, and reads or removes n-ary tuples from the database, based on pattern matching, which is part of the agent processing platform and directly supported on machine level! Reading and removing of tuples bases on search pattern matching, consisting of actual parameter values and formal parameters replaced with values from matching tuples. The implementation of generic tuple space access - vital to the agent interaction model and heavily used - on machine level is a challenge. Table 7.5 summarizes the *AFL* programming interface and *AML* subset, which reuses some instructions for efficiency. Tuple space operations (`in` and `rd`) can suspend the agent processing until a matching tuple was stored by another agent. This requires a special operational behaviour of the machine instructions for further process management, which must save the control and data state (stack content) of this process in the frame and transition table boot sections.

The implementation complexity of the tuple space together with the code morphing operations is very high. Therefore, the high-level *AFL* input operations (`in`, `tryin`, `rd`, `tryrd`, `?exist`, `rm`) are mapped on a reduced set of machine instructions {`IN`, `RD`} offering an enhanced platform resource sharing, selected by the t-parameter, explained in Tab 7.5. The processing of the `IN` and `RD` operations by the platform VM profits from additional resource sharing in the VM.

AFL	***AML***	***Stack***	***Description***
`out`	`0 OUT`	`( a1 a2 .. d -- )`	Stores a d-ary tuple (*a1*,*a2*,..) in the database.
`mark`	`OUT`	`( a1 a2 .. d t -- )`	Stores a d-ary temporary marking tuple in the database (after time-out *t* the tuple is deleted automatically).

Tab. 7.5 *AFL/AML tuple data space access operations*

AFL	*AML*	*Stack*	*Description*

`in`
`rd`
	`0 IN`
	`0 RD`
	`QBLOCK`
		`( a1 a3 .. p d --`
		`pi .. p2 )`
			Reads and remove or read only a tuple from the database. Only parameters are returned.
			To distinguish actual and formal parameters, a pattern mask *p* is used (n-th bit=1:n-th tuple element is a value, n-th bit=0: it is a parameter and not pushed on the stack).

`tryin`
`tryrd`
	`IN`
	`RD`
		`( a1 a3 .. p d t --`
		`pi .. p2 0 )`
		`( a1 a3 .. p d t --`
		`a1 a3 .. p d t 1 )`
			Tries to read and remove or read only a tuple. The parameter *t* specifies a time-out. If *t* = –1 then the operation is non-blocking. If *t* = 0 then the behaviour is equal to the `rd` operation. If there is no matching tuple, the original pattern is returned with a status 1 on the top of the data stack, which can be used by a following `?block` statement. Otherwise a status 0 is returned and the consumed tuple. Only parameters are returned.

`rm`	`-2 IN`	`( a1 a2 .. p d -- )`	Removes tuples matching the pattern. Is processed with a `IN` operation and *t*=-2.

`?exist`	`-2 RD`	`( a1 a3 .. p d --`
		`0|1 )`
			Checks for the availability of a tuple. Returns 1 if the tuple does exist, otherwise 0. It is processed with a `RD` operation and *t*=–2.

Tab. 7.5 *AFL/AML tuple data space access operations*

7.3.6 Signal Processing

Signals ς:S(*A*) carry simple information *A* that is treated as the (optional) argument of a signal. Signals are delivered to an appropriate signal handler of a specific agent, offering peer-to-peer agent communication. Signals are managed by the node signal manager. A signal *S* is delivered to an agent signal handler `$S` by inserting a signal processing token in the processing queue of the VM responsible for the parent process, followed by a signal and argument data token consumed by the VM immediately if the signal process is executed. The signal argument is pushed on the data stack, and the signal handler word is called. After the return from the signal handler word, the signal processing token is converted in a wake-up event token and passed to the agent manager, which resumes the process in the case the process waits for a signal event. On one hand using signal processing tokens and queues ensures that the parent agent process will not be pre-empted if executed with pending signals. On the other hand, signals can be processed delayed, which is normally not critically, since signal handler should modify only agent data used primarily for the transition decision process.

AFL	***AML***	***Stack***	***Description***
`signal S`	-	`( -- )`	Definition of a signal *S*.
`:$S .. ;`	`DEF`	`( arg -- )`	Definition of a handler for signal *S*. The signal argument is pushed on the top of the data stack.
`raise`	`RAISE`	`( arg sig# pid -- )`	Sends a signal *S* with an argument to the process *pid*.
`timer`	`TIMER`	`( sig# tmo -- )`	Installs a timer (*tmo*>0) raising signal *sig* if time-out has passed. If *tmo*=0 then the timer is removed.

Tab. 7.6 *AFL/AML signal processing instructions*

The raising of a signal from a source process passes an extended signal token to the signal manager, which either generates the above described signal processing token sequence that is passed to the VM processing queue, or encapsulates the signal in a message, which is sent to a neighbour node by the network manager (sketched in Figure 7.3). A process migrating to a neighbour node leaves an entry in a process cache table providing routing path information for message delivery to the migrated process.

7.3.7 Agent Mobility

An agent program can migrate to a different VM on a neighbour node by executing the `move` operation specifying the relative displacement to the current network node, shown in Table 7.7. A code migration is a complex instruction that requires the dump of the control and data state in the boot section of the code frame. The connection status for the link in a specified direction can be tested with the `?link` operation.

AFL	AML	Stack	Description
`move`	`MOVE`	`( dx dy -- )`	Migrates agent code to neighbour node in the given direction. The current data and return stack content are transferred and morphed to the boot code section. The transition boot section is loaded with a branch to the current *IP*+1.
`?link`	`LINK`	`( dx dy -- flag )`	Checks the link connection status for the given direction. If *flag*=0 then there is no connection, if *flag*=1 then the connection is alive.

Tab. 7.7 *AFL/AML agent mobility instructions*

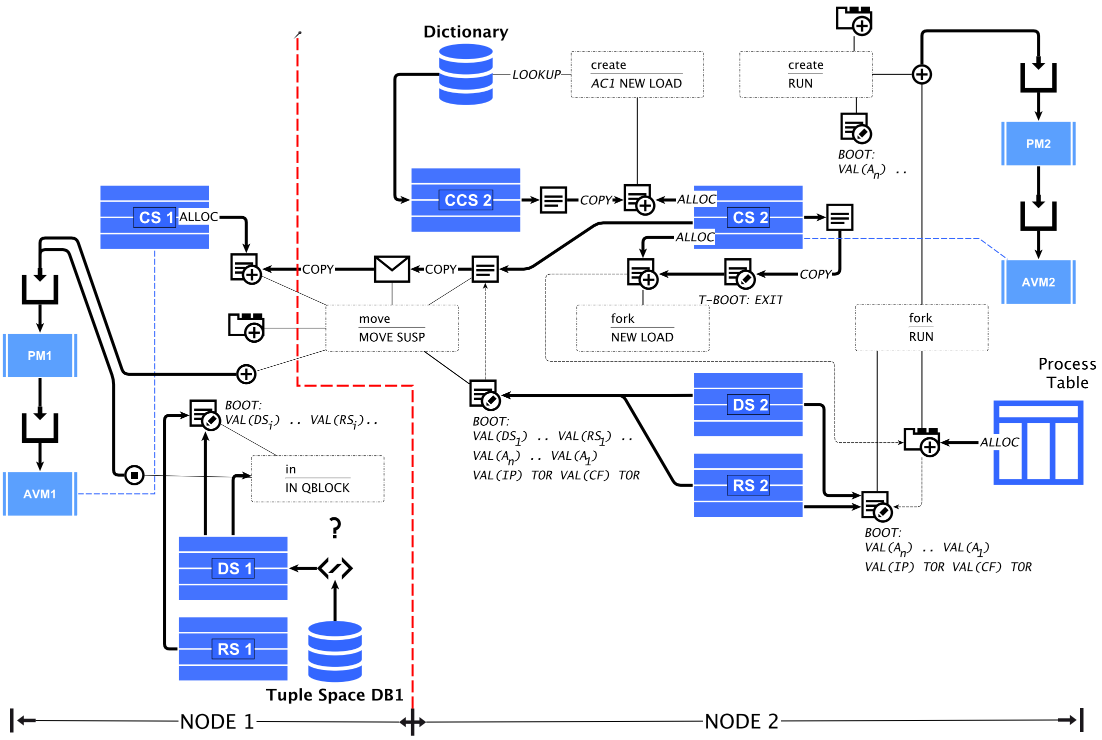


Fig. 7.5 *Effects of AML operations on code and stack memory, process management, and token flow, shown partially for agents processed on two different connected platform nodes (Node 1, Node 2).*

The migration is handled by the agent process and network managers and requires the encapsulation of the code frame(s) in a message container. The header of the container contains some persistent information about a process like the process identifier number, the parent process identifier (if any), and the delta position vector. All remaining information is contained in the program code and is initialized by restarting the program on the new node and VM.

The effects of various important machine instructions are summarized and illustrated in Figure 7.5.

7.3.8 Examples

Take a look at the following very simple *αFORTH* code Example 7.1 implementing an agent performing a mean value computation of sensor values exceeding a threshold (agent parameter `thr`) with two body variables `x` and `m`, one agent class parameter `thr`, three activities {A_1, A_2, A_3}, and a transition network with some conditional transitions. The *AAPL* behaviour model is shown on the right side. The sensor value will be read from the tuple space by using the ∇^- / in instruction in activity A_1 (tuple key ADC). The mean value is computed and stored in the database in activity A_2. It is finally passed to the tuple database in activity A_3 by using the ∇^+ / out instruction if the mean value exceeds a threshold. The agent is terminated after this action.

Ex. 7.1 *Left code shows an αFORTH program derived from an AAPL-based agent behaviour specification on the right side (in short notation).*

```
αFORTH (AFL)                                ⇐        AAPL
1  enum TSKEY ADC SENSOR SENSOREV ;                  κ: {ADC, SENSOR, SENSOREV}
2  par thr integer                                   ψ mean_filter: thr →{
3  var x integer var m integer                         Σ: {x,m}
4  :A1 ADC 0b10 2 in x ! ;                             α A1 : { ∇-(ADC,x?) }
5  :A2 m @ x @ + 2 / m !                               α A2 : { m←(m + x)/2;
6      SENSOR m @ 2 out                                         ∇+(SENSOR,m) }
7  :A3 SENSOREV m @ 2 out $self kill;                  α A3 : { ∇+(SENSOREVENT,m);
                                                                ⊗($self) }
8  :%trans                                             Π : {
9     |A1 m @ thr @ > ?A2 m @ thr @ <= ?A3 .                A1→A2 | m>thr
10    |A2 1 ?A3 .                                            A1→A3 | m<=thr
11    |A3 . ;                                                A2→A3 }
12 trans                                             }
```

This code example requires 74 operational *AML* code words, and the total size of the code frame including the boot section and the LUT created by the *AFC* compiler is only 137 words. The *AFL* parameter definition appearing in line 2 is treated like a variable definition, but with an additional parameter initialization added by the compiler following this definition immediately. After an agent process was instantiated from this program code, the entire program is executed on top-level, and therefore initializing the parameter with values pushed to the data stack in the boot section, also added by the compiler. The last statement in the *AFL* program executes the transition network, starting the execution of the program (in this case calling activity A1).

Ex. 7.2 *Code morphing and agent creation related to the agent behaviour modification.*

```
αFORTH (AFL)          ⇐      AAPL
1  t*(A1,2)                 π*(A1 → A2 | x < y) replace all trans A1->A2
2  t*(A1,2)                 π+(A1 → A2 | x = 0)    add transition A1->A2
3  100 1 fork a !             a ← Θ→(100);             fork child agent
4  100 1 mean_filter create   a ← Θ+ mean_filer(100); create new agent
```

```
        create a !
   1 new !cf ref(A1) r>c ..   a ← Θ+ ();     create new customized agent
   ref(T1) r>c ..             α+ a(A1,A2,..) π+a(A1 → A2 | x < y) ..
   100 1 run                  ⊕a(100)
```

The reconfiguration of the ATG modifying the agent behaviour using code morphing (see Example 7.2) enables agent sub-classing at run-time. This situation occurs in the employment of parent-child systems creating child agents getting an operationally reduced subset from the parent agent. This approach has the advantage of high efficiency and performance due to the reduced code size. New agents can be created by simply forking an existing agent (`fork`), which creates a copy of the parent agent including the data space. New agent programs (with different behaviour) can be created by composing existing activities and by adding different transition tables. The capability to change an existing agent is limited to the modification of the transitions (enabling and disabling of dynamic blocks inside transition rows) and by removing activities. The transition table modification (and activity deletion) is the main tool for run-time adaptation of agents based on learning. The modified agent behaviour can be inherited by forked child agents. In *AFL/AML* customized agents can be assigned only a complete transition table already part of the current agent program.

7.4 Synthesis and Transformation Rules

This section explains the mapping of fundamental concepts of the ATG agent behaviour and *AAPL* programming model and the transformation of the *AFL* program to *AML* machine code, primarily performed by the *AFC* compiler. The composition with only a small set of special *AFL/AVM* instructions is capable of providing agent creation, forking, migration, and modification by using code morphing, directly supported by the VM.

7.4.1 Agent Creation using Code Morphing

New agent processes can be created by using code templates and the `create` statement, by forking the code and the state of a currently running process using the `fork` statement, or by composing a new agent class from the current process.

Creating new and forking child processes are implemented with the previously introduced `NEW`, `LOAD`; and `RUN` machine instruction sequences, defined in Equations 7.1 and 7.2, respectively.

$$\left)\frac{a_1\ a_2\ ..\ a_n\ n_{args}\ ac\ \text{create}}{\text{VAL}(0^{\text{noinit}})\ \text{NEW DUP TOR SWAP LOAD FROMR VAL}(0^{new})\ \text{RUN}}\right. \tag{7.1}$$

$$\left)\frac{a_1\ a_2\ ..\ a_n\ n_{args}\ \text{fork}}{\text{VAL}(0^{\text{noinit}})\ \text{NEW DUP TOR VAL}(\text{-}1^{\text{fork}})\ \text{LOAD FROMR VAL}(1^{\text{fork}})\ \text{RUN}}\right. \tag{7.2}$$

7.4.2 Agent Migration using Code Morphing

Process migration requires the saving of the data and control state of the process in the frame and transition table boot sections. After the migration, the code frame is fully re-initialized including the loading of the process parameters. This is requiring the storage of the process parameter values on the data stack.

The migration is a two-stage process: the first stage is executed by the `MOVE` operation, the second by the `SUSPEND` operation, shown in Equation 7.3.

$$\begin{array}{l} \left)\dfrac{dx\ dy\ \text{move}}{\text{MOVE VAL}(\text{-}1^{\text{root}})\ \text{SETCF VAL}(1^{\text{codeoff}})\ \text{STOC TOR}}\right. \\ \text{REF}(\text{p}_1)\ \text{FETCH .. REF}(\text{p}_\text{n})\ \text{FETCH} \\ \text{VAL(n) FROMR VTOC VAL}(1^{\text{fullinit}})\ \text{SUSPEND} \end{array} \tag{7.3}$$

In Example 7.3 a short *AAPL* program and the transformed corresponding *AFL* program is shown. The *AAPL* program implements a mobile agent travelling along the x-axis in east direction in a mesh like network using the ⇔ / `move` instruction (activity A_2), sampling sensor values using the $\nabla^{\%}$ / `rd` instruction reading the sensor value from the current node tuple space (activity A_1), and computing the mean value of the sensor values. The agent class parameter `dn` determines the extension of the path in node hopping units. If the last node is reached, the computed mean value is stored in the local tuple database using the ∇^{+} / `out` instruction for further processing by other agents, performed in activity A_3, which is started if the hop-counter `dx` is equal to `dn` (increased at each migration). The compiled *AML* code frame requires 181 words including the embedded data space, boot section, and LUT.

Ex. 7.3 *Left code shows an αFORTH program derived from an AAPL-based agent behaviour specification on the right side (in short notation), posing migration of agents.*

```
αFORTH (AFL)                                   ⇐    AAPL
enum TSKEY ADC SENSOR ;                             κ: {ADC, SENSOR}
par dn integer                                      ψ mean_filter: dn → {
var x integer var dx integer                          Σ: {x,dx,m}
var m integer
:A0 0 m ! 0 dx ! 0 dx ! ;                             α A_0 : { m←0; dx←0; }
:A1 ADC 0b10 2 rd x ! ;                               α A_1 : { ∇%(ADC,x?) }
:A2 m @ x @ + 2 / m ! dx @ 1 + dx !                   α A_2 : { m←(m + x)/2;
    1 0 move;                                                  dx←dx+1; ⇔(EAST) }
:A3 SENSOR m @ 2 out self kill ;                      α A_3 : { ∇+(SENSOR,m);
                                                                ⊗($self) }
:%trans                                               Π : {
  |A0 1 ?A1 .                                             A_0→ A_1
  |A1 dx @ dn @ < ?A2 dx @ dn @ = ?A3 .                   A_1→ A_2 | dx<dn
  |A2 1 ?A3 .                                             A_2→ A_3 | dx=dn
  |A3 . ;                                                 A_2→ A_1 }
trans                                                 }
```

7.4.3 Code Frame Synthesis

The compiled AML machine program that was synthesized from the previous Example 7.3 is shown with assembler mnemonics in the following Example 7.4. The program frame is partitioned according Figure 7.4, beginning with a boot section (address range 0-15), followed by the look-up table LUT (start address 16) and the parameter and variable definitions (start address 58) adding embedded data space after each object definition, which is part of the code frame. The LUT reserves 4 words for each object (variable, parameter, activity, function) used in this program frame. The object type (first column) is already filled.

The agent parameter is initialized by the store instruction at address 74 getting the data from stack, which is pushed on the data stack in the boot section (modified at agent process instantiation). For example, if the agent program is created with the parameter `dn` = 5, a typical boot section contains the instruction sequence `SETLUT(18) VAL(5) .. BRANCH(2) CALL(9)`. The first instruction sets the LUT pointer relative to the code frame start, the second pushes the argument on the stack. The branch instruction ensures a complete initialization of the code frame (jumping over the transition section call). The last branch instruction (address range 179-181) jumps back to the last instruction in the boot section calling the transition network.

The boot section is also used for the migration request, performed by the code in the address range 119-133. The move operation itself only prepares the migration (reset of the boot section), which is finalized by the last suspend instruction. The code between modifies the boot section by morphing the actual stack content to instructions in the boot section. After the migration a full code frame initialization is required, therefore requiring the `BRANCH(2) CALL(9)` sequence at the end of the boot section. The boot section of the transition network (address range 150-153) is modified for saving the current control state (that is the instruction pointer) by creating a long branch to next instruction to be executed after migration (within an activity word) and the full code frame initialization.

Each time a code frame is initialized by executing the top-level instructions, the LUT is updated by the `VAR/DEF/TRANS` instructions (updating current code address). This self-initialization approach enables the modification of the code frame, e.g., reconfiguration and re-composition of agent programs.

The execution of the `TCALL` and `TBRANCH` instructions in the transition section rely on the LUT, too. The `TBRANCH` lookups and updates the secondary column (initially zero) of a LUT row for the relative address computation reaching the respective `TCALL`. Again, this approach ensures the highest degree of flexibility and independence from any other computational unit or VM data.

Ex. 7.4 *Compiled AML assembler code from Example7.3. First part: Boot, LUT, variable, and activity/function relocation section (KIND NAME [off0+off1] # LUT), Second part: machine instructions shown in ADDR : AML format.*

```
BOOT                [000000+0]          LUT           LUT [000016+2]
PAR              dn [000058+3] #1
VAR               x [000062+3] #2
VAR              dx [000066+3] #3       VAR             m [000070+3] #4
WORD             A0 [000076+3] #4
WORD             A1 [000089+3] #6       WORD           A2 [000101+3] #7
WORD             A3 [000134+3] #8       TRANS       trans [000147+3] #9

BOOT
0000 : SETLUT 18   0001 : NOP           ..
0014 : BRANCH 2    0015 : CALL 9

LUT
0016 : LUT      0017 : VAL 40
0018 : VAL 2    0019 : DATA     0020 : DATA    0021 : DATA     First LUT row
0022 : VAL 1    0023 : DATA     0024 : DATA    0025 : DATA     Second LUT row
```

```
..
0058 : VAR          0063 : VAL 1        0064 : VAL 1        0065 : DATA
0062 : VAR          0067 : VAL 2        0068 : VAL 1        0069 : DATA
..

Parameter Initialization
0074 : REF 1        0075 : STORE

Activity A0
0076 : DEF          0077 : VAL 5
0078 : VAL 10       0079 : VAL 0        0080 : REF 4        0081 : STORE
0082 : VAL 0        0083 : REF 3        0084 : STORE        0085 : VAL 0
0086 : REF 3        0087 : STORE        0088 : EXIT

Activity A1
0089 : DEF          0090 : VAL 6        0091 : VAL 9        0092 : VAL 1
0093 : VAL 2        0094 : VAL 2        0095 : VAL 0        0096 : IN
0097 : QBLOCK       0098 : REF 2        0099 : STORE        0100 : EXIT

Activity A2
0101 : DEF          0102 : VAL 7        0103 : VAL 33       0104 : REF 4
0105 : FETCH        0106 : REF 2        0107 : FETCH        0108 : ADD
0109 : VAL 2        0110 : DIV          0111 : REF 4        0112 : STORE
0113 : REF 3        0114 : FETCH        0115 : VAL 1        0116 : SUB
0117 : REF 3        0118 : STORE        0119 : VAL 1        0120 : VAL 0
0121 : MOVE         0122 : VAL -1       0123 : SETCF        0124 : VAL 1
0125 : STOC         0126 : TOR          0127 : REF 1        0128 : FETCH
0129 : VAL 1        0130 : FROMR        0131 : VTOC         0132 : VAL 1
0133 : SUSP         0134 : EXIT

Activity A3
0135 : DEF          0136 : VAL 8        0137 : VAL 9        0138 : VAL 1
0139 : REF 4        0140 : FETCH        0141 : VAL 2        0142 : VAL 0
0143 : OUT          0144 : VAL -1       0145 : CLEAR        0146 : EXIT

Transition Network Section
0147 : TRANS        0148 : VAL 9        0149 : VAL 29
0150 : NOP          0151 : NOP          0152 : NOP          0153 : NOP
0154 : TCALL 5      0155 : VAL 1        0156 : TBRANCH 6    0157 : END
0158 : TCALL 6      0159 : REF 3        0160 : FETCH        0161 : REF 1
0162 : FETCH        0163 : LT           0164 : TBRANCH 7    0165 : REF 3
0166 : FETCH        0167 : REF 1        0168 : FETCH        0169 : GE
0170 : TBRANCH 8    0171 : END
0172 : TCALL 7      0173 : VAL 1        0174 : TBRANCH 8    0175 : END
0176 : TCALL 8      0177 : END
0178 : EXIT

Transition Section Call, referenced from Boot section
0179 : VAL 15
0180 : VAL -1
0181 : BRANCHL
```

7.5 The Boot Sections and Agent Processing

The code frame contains different boot sections. This is the main boot section at the beginning of the code frame containing instructions to set up the code frame, and a short boot section embedded at the beginning of each transition network section, shown with an example in Table 7.8.

Boot Section Content	***Description***

`0: SETLUT(Δ) NOP ..`
`   BRANCH(2) CALL(`*Tr*`)`
`0: SETLUT(Δ) VAL(a) VAL(b) ..`
`   BRANCH(2) CALL(`*Tr*`)`
`T: NOP ..`
	Default main boot section setting up the LUT register offset (relative to start of code frame) and finally skips the transition call forcing a full frame setup. The Δ offset specifies the size of the boot section, too. The second boot section contains values that are pushed to the data stack for further processing, e.g., the agent parameters.

`0: SETLUT(Δ) VAL(a) VAL(b) ..`
`   BRANCH(0) CALL(`*Tr*`)`
`T: VAL (IP) VAL(CF) BRANCHL`
	The boot section pushes values on the data stack that are later consumed, e.g., for reprocessing of blocked operations. At the end of the boot section the current transition scheduler is called. The transition scheduler boot section contains a long branch to the next code position after the last that blocked.

`0: SETLUT(Δ) VAL(a) VAL(b) ..`
`   BRANCH(2) CALL(`*Tr*`)`
`T: VAL (IP) VAL(CF) BRANCHL`
	With additional full code frame initialization after migration or forking.

Tab. 7.8 *Example of boot section layouts (AML statements) and their effect on the frame processing [0: Main boot section, T: Transition scheduler boot section]*

7.6 Agent Platform Simulation

The proposed agent processing platform is a massive parallel data processing system. The composition of networks with these processing nodes creates a massive distributed system. The agent behaviour model used in this work reflects the parallel and distributed system. But it is a challenge to test and validate the operational and functional behaviour of a MAS consisting of hundreds and thousands of agents processed on hundreds of agent platform nodes. The monitoring of such a large parallel and distributed system is nearly impossible in a technical real-world system. For this purpose a multi-agent based simulation environment is used to simulate the distributed agent platform network on architectural level. That means that all components, i.e. the VM and the managers, shown in Figure 7.3 are simulated with non-mobile state-based agents and the *SeSAm* simulator [KLU09], simulating the processing of code frames representing agents on the proposed platform architecture. This simulation model uses agents to simulate the processing of the agents. In *SeSAm*, agent behaviour model bases on a similar but simpler ATG model compared with the *AAPL* model introduced in this work. *SeSAm* agents communicate with each other by accessing agent body variables of other agents (that is a shared memory model). This approach is only suitable in a simulation environment, and not in a real world distributed deployment of agents.

Though the simulation model has no fixed timing model regarding the real processing platform (e.g., a microchip), a time step in the simulation is equivalent to the processing of one machine instruction, which correspond roughly to 5-10 clock cycles required in an RTL implementation of the *PAVM* platform for the same code processing. The relationship for a software implementation of the *PAVM* platform is about 100-1000 machine instructions on a generic microprocessor for each simulation step.

The simulation environment addresses two different simulation goals:

1. Test, profiling, and validation of the agent processing platform;
2. Test, profiling, and validation of algorithms and multi-agent system use cases, for example, event-based distributed sensor data processing in sensor networks.

Technical failures like connection losses or complete node failures can be simulated using Monte-Carlo simulation methods.

The entire simulation environment uses a database for storing output and reading input data, e.g., the program code, shown in Figure 7.6. The *SQLD* database server not only provides a standard *SQL* based database interface, it additionally provides a RPC interface, which allows programs to communicate and synchronize with each other.

This feature enables multi-domain simulations, for example, incorporating external mathematical computations with *MATLAB* or FEM simulators for testing and evaluating Structural Monitoring systems (but this is unconcerned in this work). On other hand, the code output of the *AFL* compiler *AFC* can be immediately stored in the database and read by the simulator.

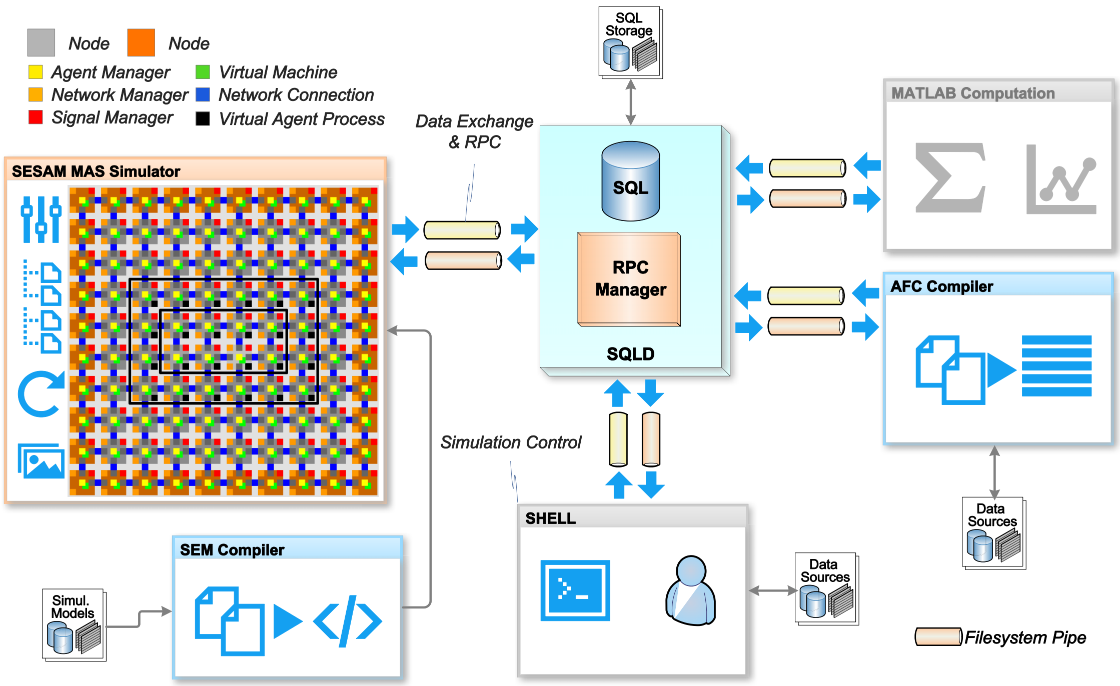


Fig. 7.6 *Simulation environment with the simulation world (left) of a sensor network with sensor nodes containing the PAVM processing platform (with multiple VM and manager components, each simulated using an agent). The simulator operates on a database for storing output and reading input data (e.g., the program code).*

The *SeSAm* simulator has originally only a GUI based programming interface for the composition of the simulation model, which is unsuitable for large models. To overcome this limitation, a textual representation of the SeSAm simulation model with the *SEM* language was developed, which can be compiled with the *SEMC* compiler to a *XML* simulation model, which can be imported directly by the simulator (native model file format).

The simulation world consists of a 10 by 10 mesh network of sensor nodes and some dedicated computational nodes at the outside of the network, which is irrelevant for the following case study. Each network node consists of a process, signal, and two network managers, four virtual processing machines, each with its own code and stack memory segments. The physical code frame size is set to 1024 words. Overall, 400 VMs with a total of 7 million of memory cells are simulated simultaneously. The simulator uses 1621 immobile (*SeSAm*) agents to simulate the platform components and the network.

Furthermore, agents are used to simulate the network connections between nodes (resources in the terms of *SeSAm*). Each sensor node, represented by a node agent, provides a set of sensor values by storing data tuples in the node tuple database, which can be processed by other agents. There is a world agent that updates the sensor values for all nodes. The set of sensor data is read from the *SQL* database, that dimension and the values depend on the use case to be simulated.

7.7 Case Study: A Self-organizing System

In this section, a self-organizing MAS is implemented with *AAPL* and transformed to α*FORTH* to show the suitability and resource requirement of the proposed agent processing platform. The *AFM* machine code is tested and evaluated by using the agent-based platform simulation environment introduced in the previous section.

7.1. The Algorithms

Faulty or noisy sensors can disturb data processing algorithms significantly. It is necessary to isolate noisy from well operating sensors. Usually sensor values are correlated within a spatially close region, for example, in a spatial distributed load measuring network using strain-gauge sensors. The goal of the following MAS is to find extended correlated regions of increased sensor intensity (compared to the neighbourhood) due to mechanical distortion resulting from externally applied load forces. A distributed directed diffusion behaviour and self-organization are used, derived from the image feature extraction approach (proposed originally by [LIU01]). A single sporadic sensor activity not correlated with the surrounding neighbourhood should be distinguished from an extended correlated region, which is the feature to be detected.

The feature detection is performed by the mobile *exploration agent*, which supports two main different behaviour: diffusion and reproduction. The diffusion behaviour is used to move into a region, mainly limited by the lifetime of the agent, and to detect the feature, here the region with increased mechanical distortion (more precisely the edge of such an area). The detection of the feature enables the reproduction behaviour, which induces the agent to stay at the current node, setting a feature marking and sending out more exploration agents in the neighbourhood. The exploration behaviour, the algorithms, and the *AAPL* specification is given in Chapter 9.

The calculation is performed by a distributed calculation of partial sum terms by sending out child explorer agents to the neighbourhood, which itself can send out more agents until the boundary of the region *R* is reached. Each child agent returns to its origin node and hands over the partial sum term to his parent agent. Because of a node in the region *R* can be visited by more than one child agent, the first agent reaching a node sets a marking MARK. If another agent finds this marking, it will immediately return to the parent. This multi-path visiting has the advantage of an increased probability of reaching nodes having missing (non operating) communication links. An *event agent*, created by a sensing agent, finally delivers sensor values to computational nodes, which is not considered here.

The *AFL* program of the *AAPL* Algorithm 9.1 is shown in Algorithm 7.1 incorporating a subclass definition for the neighbourhood perception agents (helpers). Two signal handlers are installed, processing *WAKEUP* and *TIMEOUT* signals.

Alg. 7.1 *AFL program of the explorer agent including the explorer child subclass*

```
enum KEYS
     ANY ADC H MARK FEATURE COMPUTER DISTRIBUTER SENSOR SENSORVALUE;

enum DIR NORTH SOUTH WEST EAST ORIGIN;

const MAXLIVE 1
const E1 3
const E2 6
const ATMO 500
const MTMO 50
const DELTA 30

signal timeout
signal wakeup

par dir integer
par radius integer

var dx integer
var dy integer
var live integer
var h integer
var s0 integer
var backdir integer
var group integer

var enoughinput integer
var die integer
var back integer
var s integer
var v integer

( dir ! radius ! )
```

```

:*init
  0 dx !
  0 dy !
  0 h !
  false die !
  0 10000 random group !
  dir @ ORIGIN <>
  if
    dir @ dup opposite backdir !
    delta move
  else
    MAXLIVE live !
    ORIGIN backdir !
  then
  H self @ 0 3 out
  SENSORVALUE 0b10 2 rd s0 !
;

:*percept
  0 enoughinput !
  ref(percept) 3 t*
  ORIGIN 0 do
    i backdir @ <> i delta ?link and if
      enoughinput @ 1 + enoughinput !
      i radius @ 2 fork drop
    then
  loop
  ref(percept) 1 t*
  ref(percept) 2 t+
  timeout ATMO timer
;

:*reproduce
  live @ 1 - live !
  H self @ 0b110 3 rm
  FEATURE 0b10 2 ?exist
  if
    FEATURE 0b10 2 in v !
  else
   0 v !
  then
  FEATURE v @ 1 + 2 out
  live @ 0 >
  if
    ref(reproduce) 2 t-
    ORIGIN 0 do
```

```
      i backdir @ <> i delta ?link and if
        i radius @ 2 fork drop
      then
    loop
    ref(reproduce) 1 t+
  then
;

:*diffuse
  live @ 1 - live !
  H self @ 0b110 3 rm
  live @ 0 >
  if
    0
    begin
      drop
      NORTH EAST random dup dup
      backdir @ <> swap delta ?link and
    until
    dir !
  else
    true die !
  then
;

:*end
  -1 kill
;

Explorer.child
:*percept_neighbour
  MARK group @ 0b11 2 ?exist not
  if
    MARK group @ 2 MTMO mark
    0 enoughinput !
    SENSORVALUE 0b10 2 rd s !
    H self @
      s @ s0 @ - abs DELTA <=
      3 out

    ref(percept_neighbour) 1 t*
    ORIGIN 0 do
      i backdir @ <> i delta ?link and
      i inbound and
      if
        enoughinput @ 1 + enoughinput !
        i radius @ 2 fork drop
      then
    loop
    timeout ATMO timer

    ref(percept_neighbour) 2 t*
  else
    ref(percept_neighbour) 3 t*
  then
;

:*migrate
  dir @ opposite backdir !
  dir @ delta
  dy @ + dy !
  dx @ + dx !
  dir @ delta move
;

:*goback
  H self @ 0b110 3 -1 tryin
  not if
    h !
  else
    0 h !
  then
  backdir @ delta move
;

:*deliver
  H parent @ 0b110 3 in v !
  H parent @ v @ h @ + 3 out
  0 wakeup parent @ raise
;

:opposite ( dir -- dir' )
  dup NORTH = if drop SOUTH exit then
  dup SOUTH = if drop NORTH exit then
  dup WEST = if drop EAST exit then
  dup EAST = if drop WEST exit then
;

:delta    ( dir -- dx dy )
  dup NORTH = if drop 0 -1 exit then
  dup SOUTH = if drop 0  1 exit then
  dup WEST = if drop -1  0 exit then
      EAST = if        1  0 exit then
  0 0
```

```
;

:inbound  ( dir -- flag )
  dup NORTH = if drop dy @ radius @ negate > exit then
  dup SOUTH = if drop dy @ radius @ < exit then
  dup WEST = if drop  dx @ radius @ negate > exit then
      EAST = if       dx @ radius @ < exit then
  0
;
:$wakeup
  drop
  enoughinput @ 1 - enoughinput !
  H self @ 0b110 3 -1 tryrd
  not if
    h !
  then

  enoughinput @ 1 <
  if
    timeout 0 timer
  then
;

:$timeout
  drop
  0 enoughinput !
;

:%trans
  |init 1 ?percept .
  |percept
    {*1 h @ E1 >= h @ E2 <= and enoughinput @ 1 < and ?reproduce }
    {*2 h @ E1 < h @ E2 > or enoughinput @ 1 < and ?diffuse }
    {3 1 ?migrate }
    .
  |reproduce
    {*1 0 ?end }
    {2  1 ?init } .
  |diffuse
    die @ false = ?init
    die @ true = ?end .
  |percept_neighbour
    {*1 1 ?migrate }
    {2 enoughinput @ 1 < ?goback }
    {3 1 ?goback }
    .
  |migrate 1 ?percept_neighbour .
```

```
    |deliver 1 ?end .
    |goback 1 ?deliver .
    |end .
;

trans
```

Summary: The corresponding AFL program consists of 227 source code lines only, and is compiled to 721 code and 197 data words with a total code frame size of 918 words. The explorer child subclass requires 648 code and data words (resulting in a 30% reduction of the code frame size).

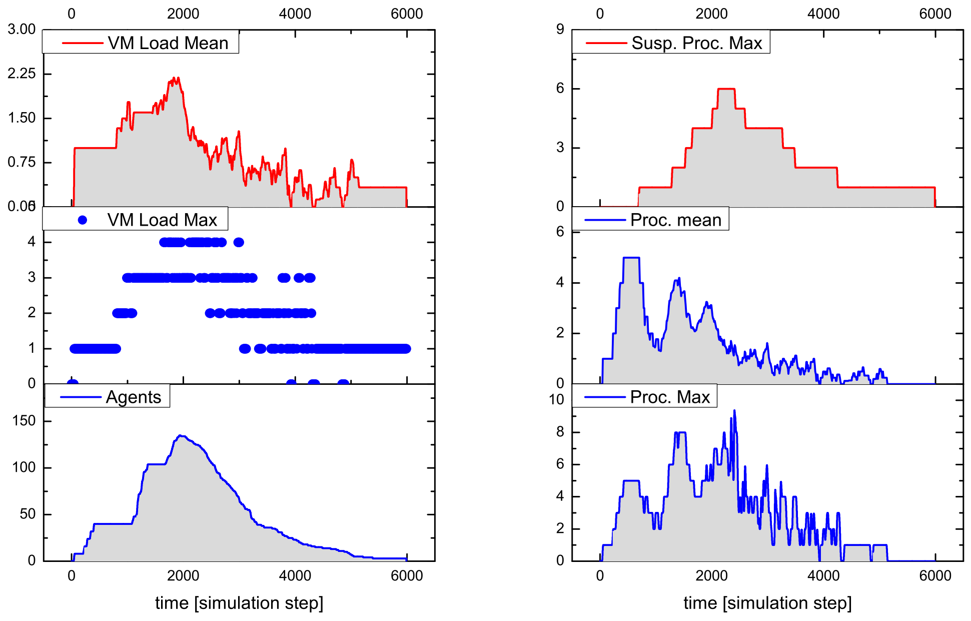


Fig. 7.7 *Analysis results for a typical run of the SoS MAS with a correlated cluster of 4x2 nodes having significant different sensor values compared with the neighbourhood (with 4 VMs/node, max. and mean computation related to nodes in the region of active nodes processing at least one program/agent) [Load: fraction of processing to idle time of a VM set]*

Figures *7.7* and *7.8* summarize the analysis results for a typical simulation run of the above described SoS MAS, with a stimulated sensor network region of four by two nodes having sensor values differing significantly from the neighbourhood (shown in the inner black rectangle in Figure *7.6*). The analysis shows the VM load factor, the agent processing statistics, and the agent population (related to the SoS MAS) in the entire network for the test run. The VM load is the fraction of processing to idle time of a VM, for each VM in the range [0.0,1.0], and is cumulated for all VMs of a node (i.e., the node VM load factor). To clarify it, if all VMs are busy 100% of the time, the node load factor is *x* if the number of VMs per node is *x*. The mean value is an averaged node VM load factor of all nodes that are processing SoS agents, the maximum value is the peak value of one VM of this group.

The MAS population has a peak number of 140 agents, with originally 8 root agents created by the sensor nodes. The analysis evaluates the temporally resolved processing load of the VMs in the extended region populated with explorer and explorer child agents only (in the outer black rectangle in Figure *7.6*).

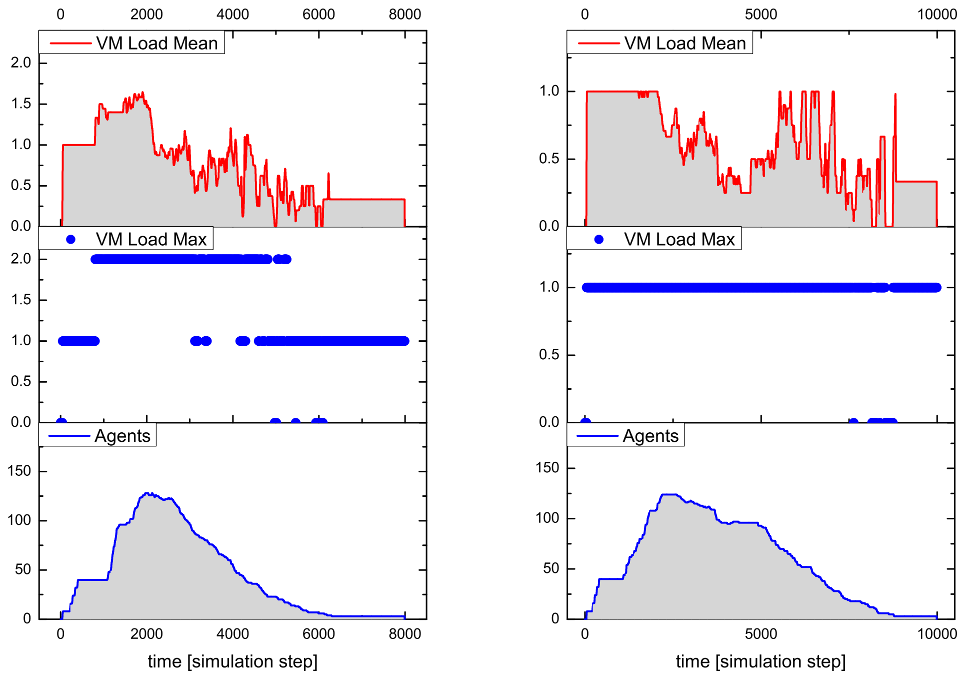


Fig. 7.8 *Analysis results with two VMs (left) and only one VM (right) per node (feature recognized after 6200 and 8800 simulation steps, respectively) [Load: fraction of processing to idle time of a VM set]*

Different platform configurations were investigated with four, two, and one VM(s) per node. Depending on the number of VMs per node, the feature is recognized by the MAS after 5050, 6200 and 8800 simulation steps, respectively, which shows a significant performance decrease by reducing the available number of VMs. The average speed-up compared with one VM for this specific MAS and network situation is 2.2 for four, and 1.5 for two parallel processing VMs. The fine-grained platform simulation of program code processing requires only 100 times more simulation steps than a comparable pure behaviour-based agent simulation (directly implementing *AAPL* agents with *SeSAm* agents). Each explorer agent requires about 1000 machine instruction to achieve its goal and termination. Neglecting communication and migration time, the total computational execution time of an explorer agent on a hardware platform with 10 MHz clock frequency requires less than 1 ms! A node in the populated region processes up to 10 different agent programs, shown in the right diagram of Figure 7.7.

7.8 The JavaScript WEB Platform JAVM

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

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

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

Two main issues arising in Internet applications using mobile agents must be addressed:

1. The definition and the knowledge representation of virtual/artificial neighbourhood connectivity in loosely coupled and hierarchical graph-based networks based on semantic rather on physical connectivity.
2. The visibility and deployment of pure client-side applications like Web browsers and computers hidden in private or restricted networks as agent processing platforms capable of receiving, processing, and sending of agents.

Usually the mobility of *AAPL* agents relies on geometrical neighbourhood structures that are available immediately in wired networks, for example, two-dimensional mesh-like networks embedded in surfaces like aircraft or air-power wings or textiles (wearable computing applications). Wireless network connectivity introduces spatially bounded domains connecting multiple devices (sensor nodes). But nodes connected to the Internet are not initially bounded in spatial domains, requiring the creation and management of logical domains that bind nodes in a meaningful sense.

Usually WEB applications communicate by using the *HTTP* application protocol level. But browser communication is limited to the *HTTP* client capabilities (`GET`, `PUT`, `POST` operations) preventing an agent processing node to be visible in the network that requires *HTTP* server capabilities (`LISTEN`, `RECEIVE`,.. operations).

There are actually inventions to overcome these limitations (i.e., WEBSockets), but there is still no common solution solving this limitation.

To enable the distributed agent processing in browser and applications running on generic computers connected by the Internet, the previously introduced Agent Forth Virtual Machine platform was implemented in JavaScript that can be executed either by a *node.js* interpreter or by any browser capable to execute JavaScript code. The *AFVM* was integrated in a distributed operating system layer, also implemented entirely in JavaScript, discussed in the following subsections. The transition form peer-to-peer networks to routed and hierarchical networks like the Internet requires some methodological and architectural changes, introducing the aforementioned broker service, discussed below.

A Distributed Co-ordination layer (DCL) is introduced to connect application programs on the Internet and NAT networks. The DCL bases on Object-orientated Remote Procedure Calls. A broker server based approach is instead used to connect *HTTP* client-only devices, discussed in the next section.

7.8.1 Capability based RPC

Object-orientated Remote Procedure Calls (RPC) are initiated by a client process with a transaction operation, and serviced by a server process by a pair of get-request and put-reply operations, based on the Amoeba DOS [MUL90]. Transactions are encapsulated in messages and can be transferred between a network nodes. The server is specified by a unique port, and the object to be accessed by a private structure containing the object number (managed by the server), a right permission field specifying authorized operations on the object, and a second port protecting the rights field against manipulation (see [MUL90] for details). All parts are merged in a capability structure, shown in Definition *7.1*.

Def. 7.1 *RPC Capability Structure and textual representation*

```
[srvport] obj(rights)[protport]

srvport,protport: byte[48 bits], XX:XX:XX:XX:XX:XX  X:0-9,A-F
obj: unsigned integer[24 bits]
rights: byte
```

The RPC communication interface is used in this work for the inter-platform communication, for example, for transferring agent program code to another platform or to access distributed file and naming services. The RPC ontology consists of servers and clients communicating by using a set of operations. A server performs a `GETREQ` operation to publish a listening on a public server port, and a client performs a transaction `TRANS` operation to access a server identified by the public server port. Each server handles a set of objects, identified by capabilities that are tuples ⟨*port*, *obj*, *rights*, *rand*⟩, consisting of the server port, an object number, a rights field, and a private protection field authorizing the rights field. A transaction operation transfers object capabilities to the server that handles the request and finally replies by using the

PUTREP operation. Therefore, a client transaction is synchronous and blocks the client process until the reply arrives or an error occurred (time-out). The localization of the server and the routing of the messages is hidden by the RPC layer, or more precisely by the underlying protocol layer, shown in Definition 7.2. The localization is basically performed by broad- or multi-casting LOCATE messages to nodes in the current domain and finally to a limited number of boundary domains. Each node monitors the locally registered servers, and replies with a IAMHERE message. Nodes are identified with ports, too, commonly identical to the server port of a host server, providing core services.

The RPC communication is encapsulated in *HTTP* messages with *XML* content and transferred using the generic *HTTP* protocol. The RPC header and data is stored inside *XML* tags with compacted hexadecimal coded text, on one hand complaining with the *XML* standard, on the other hand reducing and optimizing the payload. The binary byte data is coded with two hexadecimal digits for each data byte. Each RPC server (process) can act as a client, too, and vice versa.

Def. 7.2 *RPC-based client-server communication types, operations, and protocol schema*

```
type port       = byte array;
type capability = {cap_port:port, cap_priv: private};
type private    = {prv_obj: integer, prv_rights: integer,
                   prv_rand: port};
type header     = {h_port: port, h_priv: private,
                   h_command: integer, h_status: integer};

+-----------------------------------------------------+
| reply:header = TRANS(request:header,data:buffer); |   Client
+-----------------------------------------------------+
      ↑ ❻                       ↓ ❶                     Network (Messages ✉)
+-----------------------------------------------------+
|  Locate               |  Foreward                   |   Router/Broker
+-----------------------------------------------------+
  ❷ LOC  ↓  ↑  ❸ IAMHERE       ↓ ❹            ↑ ❺       Network (Messages ✉)
+-----------------------------------------------------+
| request:header*buffer = GETREQ(public:port); ↓ ❹ |
| .. service ..                                     |   Server
| PUTREP(reply:header,data:buffer)              ↑ ❺ |
+-----------------------------------------------------+
```

7.8.2 AFS: Atomic File System Service

The Atomic File System Server (*AFS*) provides a unified file system storage suitable for the deployment in unreliable environments and is independent of lower level storage capabilities. Files are associated with a capability. The capability port is given by the server, and the capability object number identifies the file uniquely. The protected rights field of the capability determines the authorized operations to be applied to a file or the file system.

A file is stored always in a contiguous block cluster of the file system, avoiding a linked free and used block management that offers a low-resource and low-overhead file system with basic real-time feature capabilities. A committed file is immutable (locked, read-only mode) and occupies only one internal node (i-node). Modification of locked files require an uncommitted (unlocked) copy of the original immutable file. Though this approach seems to be inefficient for post modifications of files, it avoids the requirement of file system logging required for a fast crash recovery. Here, after a crash only uncommitted files (occupying only one i-node) must be cleared. The AFS is used in this work mainly for storing agent program code and persistent tuple space data. The simplicity of the AFS enables the implementation in JavaScript and the embedding in browser applications, discussed in the next section.

The set of operations embraces the reading, modification, commitment, creation, and destruction of files. Each file object has a limited lifetime that is decremented periodically by a garbage collector that removes unused entries. Therefore, there is a touch and age operation modifying the lifetime of a file. Only the explicit commitment of a file makes the file persistent. Reading data from and writing data to storage devices is performed through a cache module, speeding up reading and modifying of file data and i-nodes.

7.8.3 DNS: Directory and Naming Service

The Directory and Name Server (*DNS*) provides a mapping of names (strings) on capability sets, organized in directories. A directory is a capability-related object, too, and hence can be organized in graph structures. A capability set binds multiple capabilities associated with the same semantic object, for example, a file that is replicated on multiple file servers. A capability set marks one object capability as the current and reachable object, though this may change any time.

A directory is associated with an internal node and the content (the rows). The directory content is stored in an *AFS* file. Redundancy is offered by the capability sets themselves (replication of objects) and by the *DNS* using two file servers for storing directories in two-copy mode (replication of directories). The immutability of *AFS* files immediately qualifies the immutability of directories, enabling a robust directory system with a fast and stable recovery after a server crash.

The simplicity of the *DNS* enables the implementation in JavaScript and the embedding in browser applications, too, discussed in the next section.

The set of operations embraces the reading, modification, creation, and destruction of directories. Each directory has a limited lifetime that is decremented periodically by a garbage collector that removes unused entries that are not linked anymore. Therefore, there is a touch and age operation modifying the lifetime of a directory.

7.8.4 Broker Service

The integration and network connectivity of client-side application programs like Web browsers as an active agent processing platform requires client-to-client communication capabilities, which is offered in this work by a broker server that is visible in the Internet or Intranet domain, shown in Figure 7.9. To provide compatibility with and among all existing browser, *node.js* server-side, and client-side applications, a RPC based inter-process communication encapsulated in *HTTP* messages exchanged with the broker server operating as a router was invented. Client applications communicate with the broker by using the generic *HTTP* client protocol and the `GET` and `PUT` operations. RPC messages are encapsulated in *HTTP* requests. If there is a RPC server request passed to the broker, the broker will cache the request until another client-side host performs a matching transaction to this server port. The transaction is passed to the original RPC server host in the reply of a *HTTP* `GET` operation.

But the deployment of one central broker server introduces a single-point-of-failure and is limiting the communication bandwidth and the scaling capability significantly. To overcome these limitations, a hierarchical broker server network is used. Each broker in this broker graph can be the root of a subgraph and can be a service end-point (i.e., providing directory and name services), a router between clients and other broker servers, and an interface bridge to a non-IP based network, for example, a sensor network. A broker is just an application program capable of running on any computer visible globally on the Internet or more locally in some Intranet domains. Each node in the network act always as a service end-point and as a logical router, regardless of the server- or client-side visibility.

Communication between broker servers can be established either by using the aforementioned *HTTP* based message passing without a size limitation, or by using the *UDP* with additional data fragmentation and de-fragmentation layers that splits large messages in packets. Alternatively, client applications (using the *node.js* VM) can communicate with a broker server by using *UDP*.

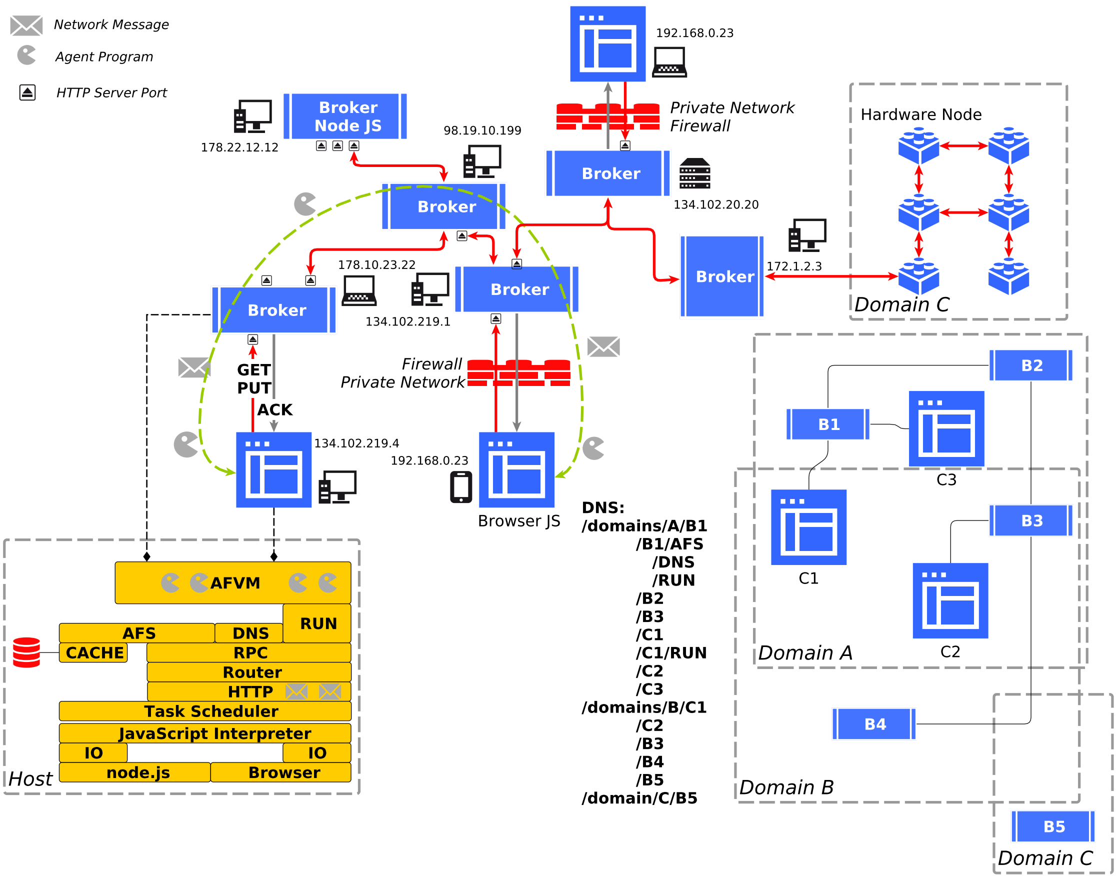


Fig. 7.9 *Different agent processing platforms and nodes are connected in Inter-, Intranet, and dedicated sensor network domains including hardware nodes (embedded and microchip level platform implementations).*

7.8.5 Domains as Organizational Structures and the Directory Name Service

Domains are groups of agent processing nodes that are coupled in a network. Agents can migrate between nodes of a group. A node can be assigned to more than one domain, enabling the migration of agents between domains. Node domain composition bases on

1. Geometrical localization and proximity, basically expressing and simulating neighbourhood connectivity
2. Information and data context

3. Tasks to be performed, cooperative goals to be satisfied
4. Logical network domains

Domains can be expressed by paths similar to directory trees that are handled usually by a file system. In this work a distributed and unified Directory Name Service (DNS) is used that provides a database to publish (capability-name) pairs organized in trees. Each object in the distributed system is related to a capability, which is serviced by a specific server. For example, a file containing the agent program code is serviced by a file server. A directory containing domains is an object, too, handled by the DNS server. An agent platform that processes agents programs is another kind of object, handled by a run server that exists on each node. Agents are objects in this sense, but they don't belong to a specific server, therefore they are handled as mobile and autonomous severs. In Figure *7.9*, an example for a composition of domains consisting of network nodes that are not directly connected is shown.

7.8.6 The Modular Platform Architecture

The agent processing platform is highly modular, shown in Figure *7.10*, consisting of various modules. Basically it consists of the agent code processes (Agent Forth Virtual Machine, *AFVM*, discussed later), an agent manager (*AMAN*) responsible for agent processing control, migration, and interaction, some kind of communication layer (the bare-bone NET module or an Operating System layer), and optional a distributed operating and coordination layer consisting of the file- and naming services including the previously introduced RPC, implemented in JavaScript (*JS*) and executed either using the *node.js* VM or a *JS* capable WWW browser application. The JS platform requires a task scheduler to implement synchronization of parallel tasks. At least one broker server is required in Intra- and Internet domains for connecting pure client-side applications (WWW browser). The agent processing platform is implemented in hardware (SoC design, pure digital logic), in software (stand-alone *C/OCAML*), and in JavaScript. All platform implementations are compatible on communication, operational, and execution level. Platforms that should be visible in Intra- and Internet domains and that are connected indirectly require the RPC, a message Router (used for inter-node server-client communication, too), and *HTTP* connection modules to establish at least agent migration.

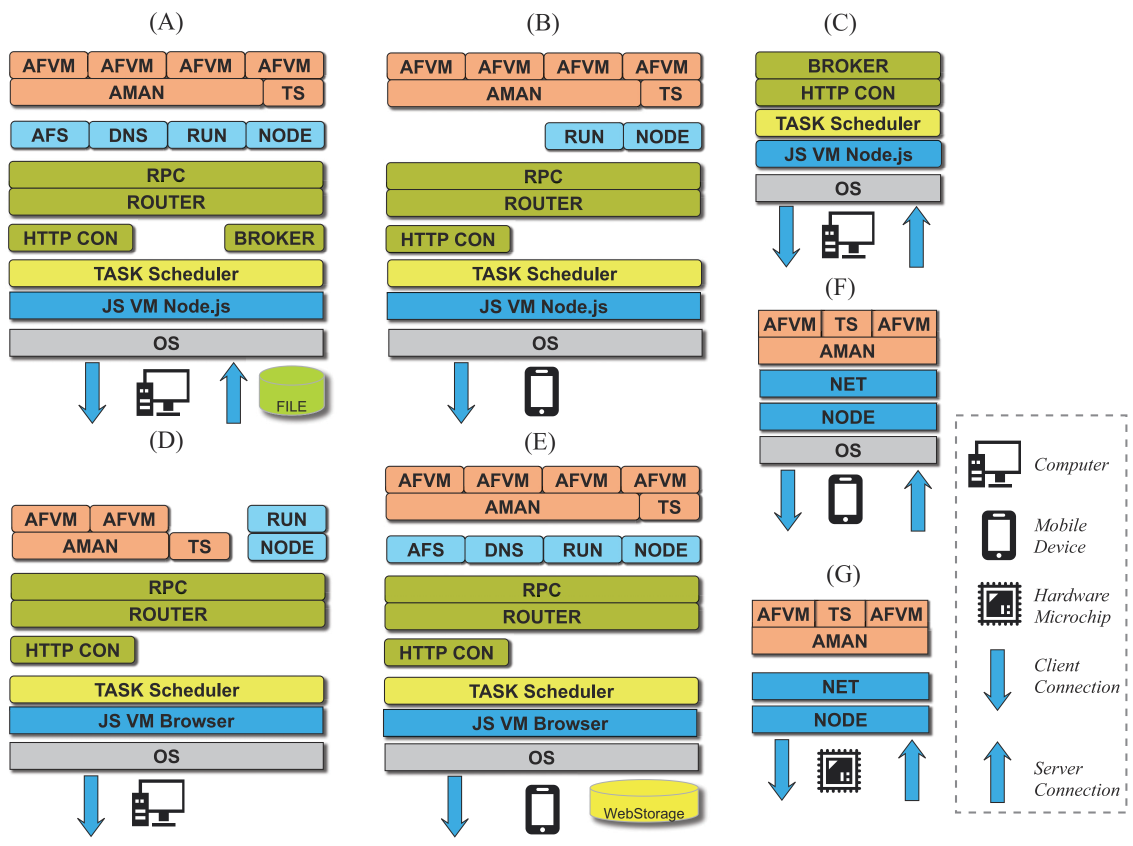


Fig. 7.10 *The modular host platform architecture:*
(A) Full Server-side JavaScript Implementation with File and Naming Service
(B) Client-side JS Implementation
(C) Broker-only Node
(D) Client-side Browser JS Implementation
(E) Client-side Browse JS Implementation with File- and Naming Services using, e.g., WebStorage
(F) Native Software Implementation of the AFVM
(G) Native Hardware Implementation of the AFVM

7.9 Further Reading

1. S. Pelc, *Programming Forth*, May. 2011, ISBN 9780952531050.
2. L. Brodie, *Thinking FORTH*. 2004, ISBN 0976458705
3. R. Wilhelm and H. Seidl, *Compiler Design - Virtual Machines*, Springer Berlin, 2010, ISBN 9783642149085
4. lain D. Craig, *Virtual Machines*, Springer London, 2006, ISBN 9781852339692