Agent Input-Ouput System (AIOS)

The Agent Input-Output System (*AIOS*) is the *interface and abstraction layer* between agents programmed in *AgentJS* and the agent processing platform (*JAM*). Furthermore, it provides an interface between host applications and *JAM*. A *JAM* instance consists of multiple modules:

- Node
- World
- Code/Process (control and modification)
- Tuple space
- Signals
- Mobility
- Network and Communication (AMP)
- Scheduler
- Security
- Watchdog
- Artificial Intelligence (optional):
  - Machine Learning (ML)
  - Constraint Solving Programming (CSP)
  - Logic Programming and Satisfiability Solver (SAT)

The modules are accessible (directly or indirectly) by the agents via the AIOS, shown in Fig. [#aios1].

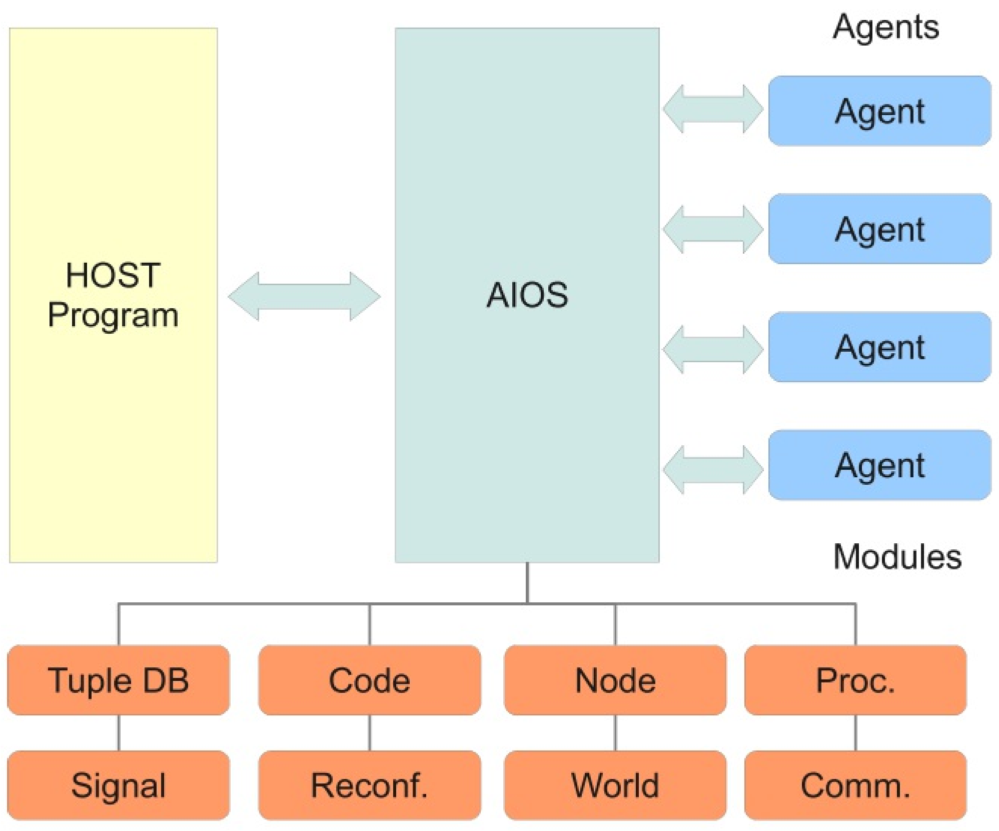

Figure [#aios1]. Interface between agents and JAM and between JAM and a host application by the Agent Input-Output System (AIOS).

Agent Roles

Security is provided by the agent platform by assigning execution roles and levels to agents. The roles are dynamic and can be changed at run-time. The execution of agents and the access of resources is controlled by those roles to limit Denial-of-Service attacks, agent masquerading, spying, or other abuse:

There are four levels:

0. Guest (not trustful, semi-mobile)
1. Normal (maybe trustful, mobile)
2. Privileged (trustful, mobile)
3. System (highly trustful, locally only, non-mobile)

The lowest level (0) does not allow tuple space access, agent replication, migration, or the creation of new agents. The JAM platform decides the security level for new received agents. An agent cannot create agents with a higher security level than its own. The highest level (3) has an extended AIOS with host platform device access capabilities. Agents can negotiate resources (e.g., CPU time) and a level raise secured with a capability-key that defines the allowed upgrades (defined by the services, e.g., agent role service or other resources like tuple space access). The system level can not be negotiated. The capability is node ans service specific. A group of nodes can share a common key (identified by a server port). A capability consists of a server port, a rights field, and an encrypted protection field generated with a random port known by the server (node) only and the rights field.

Among the AIOS level, other constrain parameters can be negotiated using a valid capability with the appropriate rights:

- Scheduling time (longest slice time for one activity execution, default value is between 20-200ms)
- Run time (accumulated agent execution time, default is 2s)
- Living time (overall time an agent can exist on a node before it is removed, default is 200s)
- Tuple space access limits (data size, number of tuples)
- Memory limits (fuzzy, usually the entire size of the agent code including private data, actually not limited)
- Network links and connectivity (supported by the AMP module)

Agent Scheduling

JS has a strictly single-threaded execution model with one main thread, and even by using asynchronous callbacks, these callbacks are executed only if the main thread (or loop) terminates. This is the second hard limitation for the execution of multiple agent processes within one *JAM* platform. Agent processes are scheduled on activity level, and a non-terminating agent process activity would block the entire platform. Current JS execution platform including VMs in WEB browser programs provide no reliable watchdog mechanism to handle non-terminating JS functions or loops. Although some browsers can detect time outs, they are only capable to terminate the entire JS program. To ensure the execution stability of the *JAM* and the *JAM* scheduler, and to enable time-slicing, check-pointing must be injected in the agent code prior to execution. This step is performed in the code parsing phase by injecting checkpoint functions `CP()` at the beginning of a body of each function contained in the agent code, and by injecting the `CP` function calls in loop expressions. Although this code injection can reduce the execution performance of the agent code significantly, it is necessary until JS platforms are capable of fine-grained check-pointing and agent process scheduling with time slicing. On code-to-text transformation (e.g., prior to a migration request), all `CP` calls are removed.

AIOS provides a main scheduling loop. This loop iterates over all logical nodes of the logical world, and executes one activity of all ready agent processes sequentially. If an activity execution reaches the hard time-slice limit, a `SCHEDULE` exception is raised, which can be handled by an optional agent exception handler (but without extending the time-slice). This agent exception handling has only an informational purpose for the agent, but offers the agent to modify its behaviour. All consumed activity and transition execution times are accumulated, and if the agent process reaches a soft run-time limit, an `EOL` exception is raised. This can be handled by an optional agent exception handler, which can try to negotiate a higher CPU limit based on privilege level and available capabilities (only level-2 agents). Any ready scheduling block of an agent and signal handlers are scheduled before activity execution.

After an activity was executed, the next activity is computed by calling the transition function in the transition section (or just applying an unconditional value). If the activity is blocked (agent is suspended, except signal handling), the next transition is computed after the resume of the agent process.

In contrast to the *AAPL* model that supports agent process blocking on statement level, eventually allowing multiple blocking statements (e.g., IO/tuple-space access) inside activities, JS is not capable of handling any kind of process blocking of user instructions (there is no process and blocking concept). For this reason, an activity may only contain one blocking statement, and the blocking is applied to the entire activity after the control flow of an activity function terminates.

Multiple blocking statements require scheduling blocks that can be used in *AgentJS* activity functions (at the end) handled by the *AIOS* scheduler. Blocking *AgentJS* functions with a pending result use common callback functions to pass function results to the agent, e.g., `inp(pat,function(tup){..})`.

A scheduling block consists of an array of functions (micro activities), i.e., `B(block) = B([function(){..}, function(){..},...])`., executed one-by-one by the AIOS scheduler. Each function may contain a blocking statement at the end of the body. The `this` object inside each function references always the agent object. To simplify iteration, there is a scheduling loop constructor `L(init, cond, next, block, finalize)` and an object iterator constructor `I(obj, next, block, finalize)`, used, e.g., for array iteration. Agent execution is encapsulated in a process container handled by the AIOS. An agent process container can be blocked waiting for an internal system-related IO event or suspended waiting for an agent-related AIOS event (caused by the agent, e.g., the availability of a tuple). Both cases stops the agent process execution until an event occurred.

The basic agent scheduling algorithm is shown in the following algorithm and consists of an ordered scheduling processing type selection, i.e., partitioning agent processing in agent activities, transitions, signals, and scheduling blocks. In one scheduler pass, only one kind of processing is selected to guarantee scheduling fairness between different agents. There is only one scheduler used for all virtual (logical) nodes of a world (a *JAM* instance). A process priority is used to alternate activity and signal handling of one agent, preventing long activity and transition processing delays due to chained signal processing if there are a large number of signals pending.

![#jamsched] Algorithm

```
∀ node ∈ world.nodes do
 ∀ process ∈ node.processes do
  Determine what to do with prioritized conditions:
    Order of operation selection:
    0. Process (internal) block scheduling [block]
    1. Resource exception handling
    2. Signal handling [signals]
       - Signals handled if process priority<HIGH
       - Signal handling increase process priority
         temporarily to allow low-latency activity
         and transition function scheduling!
    3. Transition execution
    4. Agent schedule block execution [schedule]
    5. Next activity execution
       - Lowers process priority
  if process.blocked or process.dead or
   process.suspended and process.block=[] and
   process.signals=[] or
   process.agent.next=none and process.signals=[] and
   process.schedule=[]
    then do nothing
  elseif not process.blocked and process.block≠[]
    then execute next block function
  elseif agent resources check failed
    then raise EOL exception
  elseif process.priority < HIGH and process.signals≠[]
    then handle next signal, increase process.priority
  elseif not process.suspended and process.transition
    then get next transition
    or execute next transition handler function
  elseif not process.suspended and process.schedule≠[]
    then execute next agent schedule block function
  elseif not process.suspended
    then execute next agent activity and
      compute next transition,
      decrease process.priority
```

Algorithm [#jamsched]. JAM Agent Scheduler

Agent Class Template

Agents are created from constructor functions providing an ATG behaviour template, shown in Def. [#agentjstempl].

![#agentjstempl]

```
function agentclass(p1,p2..) {
    // Body Variables //
    this.v1 = initial value
    this.v2 = initial value
    // Activtites //
    this.act = {
        a1: function () { .. },
        a2: function () { .. },
        ..
    }
    // Activity Transitions //
    this.trans = {
        a1: anext,
        a2: function () { return anext },
        ..
    }
    // Signal Handler //
    this.on = {
        signal: function (arg) { .. },
        ..
    }
    // Current and Initial Activity //
    this.next = ainit
}
```

Definition [#agentjstempl]. Basic structure of an agent class constructor function

A constructor function defines a set of parameters, body variables, activity and transition function objects, an optional signal handler function object, and the reserved *next* body variable pointing to the current (or after a start the initial) activity.

AgentJS API

The following sections describe the agent programming interface of *AgentJS*. The operations visible to agents depend on their privilege level. Operations restricted to privilege levels are marked.

Computation

There are various powerful and extended computational functions that can be used by agents. Please note that for some reason arrays and objects cannot be iterated in agent processes by using the `for(p in a)` statement. Instead the `iter` function has to be used. Furthermore, the `this` object inside function callbacks references always the agent object, i.e., body variables and functions can be accessed by the `this` object.

Operations

abs

```
function (number) → number
```

Returns absolute value of number.

add

```
function (a:number|array|object,
          b:number|array|object)
  → number|array|object
```

General purpose addition operation for scalar numbers, arrays, and objects of numbers.

angle

```
function (a:[number,number]|{x:number,y:number},
          b?:[number,number]|{x:number,y:number}
  → number
```

assign

```
function (src:array|object|string,dst:array|object|strin
g)
  → array|object|string
```

Assigns (copies) all elements from source to destination object that is returned. The destination object class is preserved.

concat

```
function (array|string|object,
          array|string|object)
  → array|string|object
```

Concatenation operation for arrays, strings, and objects.

contains

```
function (array|object,
           (number|string)|(number|string)[])
         → boolean
```

Tests existence of an element or an array of elements in an array or object (attribute)

copy

```
function (array|object|string)
  → array|object|string
```

Returns a copy of an array, object, or string. The object may not contain cyclic references.

delta

```
function (a:[]|{},
          b:[]|{}
  → []|{}
```

Computes the delta vector of two vectors.

distance

```
function (a:[]|{},
          b:[]|{}
  → []|{}
```

Computes the distance of two vectors.

div

```
function (number,number)
         → number
```

Integer division operation

empty

```
function (array|object|string)
         → boolean
```

Checks if an object, string, or array is empty (`{}` `[]` "")

equal

```
function (number|string|array|object,
           number|string|array|object)
          → boolean
```

Tests equality of numbers, strings, arrays, and objects.

filter

```
function (array|object,
           function (@element,@index?)
                    → boolean)
          → array|object
```

Filter operation for arrays and objects.

flatten

```
function ('a array array,
           level?:number )
          → 'a array
```

Flattens elements of an array up to specified level (default:1).

head

```
function ('a array) → 'a
```

Returns head (first) element of an array.

int

```
function (number) → number
```

Returns integer number.

isin

```
function (array|object,
          number|string|(number|string)[])
         → boolean
```

Tests existence of an element in an array or object (attribute). The element can be an array, too.

iter

```
function (array|object,
          function (@element,@index?))
```

Iterator over arrays and objects.

last

```
function (array|object|string)
         → 'a
```

Returns the last element of an array, object, or string.

length

```
function (array|object|string)
         → number
```

Returns length of an array, number of attributed of an object, or length of string.

map

```
function (array|object,
          function (@element,@index?) → *|none)
         → array|object
```

Map and filter operation for arrays and objects. If the user function returns undefined the element is discarded.

matrix

```
function (@cols,
          @rows,
          init:number|function)
         → [][]
```

Creates a matrix (array of arrays).

max

```
function (a :number|array,
          b?:number)
         → number
```

Returns largest number from two numbers or array of numbers.

min

```
function (a :number|array,
          b?:number)
         → number
```

Returns smallest number from two numbers or from array of numbers.

neg

```
function (number|array|object)
         → number|array|object
```

Returns negative number, negative elements of an array or object of numbers.

random

```
function (a :number|array|object,
          b?:number, frac?:number)
         → number|*
```

Returns a random number within the interval [*a*,*b*] or an element from an array or object. The optional fraction parameter specifies the rounding precision (frac=1 return integer numbers).

reduce

```
function ('a array,
          function ('a,'a) → 'b )
         → 'b
```

Reduces elements of an array to a compound value using a user function.

reverse

```
function ('a array|string)
         → 'a array|string
```

Reverses elements of an array or string.

sort

```
function ('a array,
          function (@element1,@element2) → number)
         → 'a array
```

Sorts an array with a user function returning {-1,0,1} numbers. Descending order is reached if a<b return a positive value, otherwise if a negative value is returned an ascending order is reached.

sum

```
function ('a array|object,
          map?:function (e) → 'b)
         → ('a|'b) number
```

Returns the sum of elements of an array or attribute values of an object. The optional user mapping function can be used to return a computed value for each element.

string

```
function (*) → string
```

Returns string representation of the argument.

tail

```
function ('a array)
         → 'a array
```

Returns tail (last) elements of array.

zero

```
function (number|array|object)
```

```
         → boolean
```

Checks if a number, all elements of an array or all attributes of an object are zero.

Examples

```
this.a=[1,2,3];
this.o={real:2.0,img:3.1};

this.sq = function (objORarray) {
  var res=0;
  iter(objORarray,function (elem,index) {
    res=res+elem*elem;
  });
  return res;
}
..
    var x,y,z;
    x=this.sq(a); // x==14
    y=this.sq(o); // y==13.61
    z=sum(a);     // z==6
    if (zero(this.o)) this.o={real:1.0,img:1.0};
..
```

Usage of computational functions and user defined functions assigned to agent body variables.

Environment

info

```
function (@kind) → {}
typeof @kind = 'node' | 'version' | 'host'
```

Returns environmental information. Supported information requests are:

- *node*
- *version*
- *host*

The node information request returns:

```
{id       : id string,
 position : {x:number, y:number},
 location : undefined | {lat:number, lon:number},
 type     : string}
```

The node type is a string identifier from the set:

```
typeof @type = {'shell', 'webshell', 'relay', 'webapp',
'mobileapp'}
```

The host information request returns information about the host platform:

```
{type:string='node' | 'browser'}
```

myClass

```
function () → string
```

Returns the class of the agent (if known). Same result is returned by accessing the `this.ac` variable.

myNode

```
function () → string
```

Returns the identity name of the current JAM node.

myParent

```
function () → string
```

Returns the identity name of the parent agent of this agent (if any).

myPosition

```
function ()
  → {x:number,y:number} |
    { ip:string,
      gps:{lat:number, lon:number},
      geo:{city:string,
           country:string,
           countryCode:string,
           region:string,
           zip:string} }
```

Returns the position of the current node (Physical GPS/Geographical or logical position).

me

```
function () → string
```

Returns the identity name of this agent.

clock

```
function (ms:boolean)
         → number|string
```

Returns current system clock in milliseconds (*ms* argument is true) or in time format `HH:MM:SS`.

Tuple Space

Tuple spaces are data bases storing vectors of values. Each tuple has a dimension (the number of values) and a type interface. Tuples can be read or consumed by using patterns. Patterns are like tuple but allowing wild-card values (none). If there is no matching tuple found in the data base, the agent is suspended until a matching tuple arrives or a timeout occurs (by using the `try_*` operations). Since JavaScript programs cannot block, a callback function has to be provided and the blocking operation must be placed at the end of an activity or inside a scheduling block. Commonly the first value of a tuple (a string) is used as a key, but this is only a weak constraint that has not to be satisfied. If the first value is a string it is used as a hash key in the tuple data base speeding up tuple pattern matching. A tuple space has a linear structure and is non-persistent. To support complex hierarchical data bases, *JAM* provides a SQLite data base server and access to this data base for level 3 agents (see section [SQL]). Tuple spaces can be mapped on tables in this SQL data base (by tuple space provider and consumer functions passed to JAM).

Types

```
type tuple =
  (number|string|boolean|
   array|object) []
type pattern =
  (number|string|boolean|
   array|object|null) []
```

Examples

![#agentjsts]

```
out(['MARKING1',1]);
out(['SENSORA',100,true]);
inp(['SENSORA',_,_],function (tuple) {
  if (tuple) this.s =tuple[1];
});
```

```
rm(['SENSORA',_,_],true);
try_rd(0,function (tuple) { .. });
ts(['MARKING',_],function (t) { t[1]++ });
alt([
  ['SENSORA',_,_],
  ['SESNORB',_],
  ['EVENT'],
  ],function (tuple) {
    if (tuple && tuple[0]=='EVENT') {..}
    else ..
  });
```

Example [#agentjsts]. Tuple Access

Operations

alt[1,2,3]

```
function (pattern [],
          callback:function,
          all?:boolean,
          tmo?:number)
```

Selective input operation with multiple search patterns that can have different type signatures and arities. The first tuple matching one of the pattern is consumed and passed to the callback function. If there are multiple tuples matching a specific pattern and the flag is set than all matching tuples are consumed and returned.

collect[1,2,3]

```
function (to:path,
          pattern)
         → number
```

The collect operation moves tuples from this source TS that match template pattern into destination TS specified by path to (a node destination).

copyto[1,2,3]

```
function (to:path,
          pattern)
         → number
```

Copies all matching tuples form this source *TS* to a remote destination *TS* specified by path to (a node destination).

evaluate[1,2,3]

```
function (pattern,
          callback:function (tuple|none))
         → tuple
```

Access an evaluator tuple created by a listen operation. The evaluator evaluates the given pattern to a tuple and passes the tuple back to the callback function of the requesting agent.

exists[1,2,3]

```
function (pattern) → boolean
```

Check if a tuple matches the given patterns.

inp[1,2,3]

```
function (pattern,
          callback:function(tuple|[]tuple|none),
          all?:boolean,
          tmo?:number)
          → tuple|tuple[]|none
```

Consumes a tuple matching the given pattern that is passed to the callback function. If there are multiple tuples matching a specific pattern and the `all` flag is set than all matching tuples (array) are consumed and returned. If there is no matching tuple and `tmo` is zero (immediate reply) or positive (timeout) than the callback handler is called with a none value argument.

listen[1,2,3]

```
function (pattern,
          callback:function (pattern) → tuple)
```

Installs a tuple evaluator (active tuple) that can be accessed by the `evaluate` operation.

out[1,2,3]

```
function (tuple)
```

Stores a tuple in the data base.

mark[1,2,3]

```
function (tuple,
          tmo:number)
```

Stores a tuple with a limited lifetime in the data base.

rd[1,2,3]

```
function (pattern,
          callback:function(tuple|tuple[]|none),
          all?:boolean,
          tmo?:number)
         → tuple|[]tuple|none
```

Reads a tuple matching the given pattern that is passed to the callback function. If there are multiple tuples matching a specific pattern and the `all` flag is set than all matching tuples (array) are read. If there is no matching tuple and `tmo` is zero (immediate reply) or positive (timeout) than the callback handler is called with a none value argument.

rm[1,2,3]

```
function (pattern,
          all?:boolean)
```

Removes a tuple or if the `all` flag is set all matching tuples from the data base.

store[1,2,3]

```
function (to:path,
          tuple)
         → number
```

Stores a tuple in a remote TS specified by path `to` (a node destination). Returns number of stored tuples.

ts[1,2,3]

```
function (
  pattern,
  callback : function(tuple)
             → tuple
) → tuple | null
function (
  pattern,
  timeout : number
) → tuple | null
```

Atomic and non-blocking test-and-set operation that can be used to modify a tuple in place found based on the provided pattern. If the second argument is a number, the current timeout value of a tuple is updated.

alt.try, inp.try, rd.try, evaluate.try[1,2,3]

```
function (tmo:number, ..) → *
```

Try operation and execute an alternation, input, or read operation with a given timeout (Milliseconds). **If there was no matching tuple found and the timeout elapsed the callback is fired with a null argument, followed by the continuation of the agent execution with the next activity.**

```
rd.try(timout,tuple,function (t) {
  if (t) this.data=t,log('GOTIT');
  else log('DONT GOTIT');
})
```

Active Tuples

Passive tuples are produced via the `out` operation and consumed via the `rd` and `inp` operations. Among passive tuples, there are active tuples that are evaluated by a consumer and passed back to the original producer (bidirectional tuple exchange) by using the `listen` and `evaluate` operations.

![[#agentjsacttup]

```
listen(pattern, function (tuple) {
  Modification of tuple: Replace formal
  with actual parameters
  return tuple'
})
evaluate(pattern, function (tuple) {
  Process evaluated tuple
})
```

Definition [#agentjsacttup]: Active Tuple Template

Signals

Signals are used as a low-level inter-agent communication. In contrast to tuple, signals can be send directly to specific agents. Although there are remote tuple space operations, signals should be used for remote agent communication. Signals can carry an argument (data). The delivery of signals is only reliable if the source and destination agents are processed on the same platform node. If the destination agent is processed on a remote platform the signals are delivered as messages to the destination node along the travel path of the destination agent.

There is no agent localisation, and only agent traces are used to deliver a signal to a remote agent, i.e., each node remembers the direction/link an agent used to migrate to another node. Therefore, remote signals can only be send to agents that were previously processed on the node of the source agent!

To enable back propagation of signals, each node remembers the direction/link of incoming signals and its source agent, too. The entries of these trace caches have a timeout and are removed automatically. Each time a signal is propagated along the trace path of an agent, the cache entries of all path nodes are refreshed. After a timeout of a trace cache entry, signals cannot be delivered to an agent along a path using this node!

A signal can be received by an agent by installing a signal handler in the `this.on` section of the agent class.

The destination agent is specified by the agent identifier. Usually agent identifiers should not made be public for security reasons (An agent at least with privilege level 1 can control another agent on the same node if it knows its agent identifier). Hence, signals are often used between parent-child agents. Each child knows the agent identifier of its parent, and vice versa.

Signals should carry only simple arguments. Objects may not contain cyclic references. Complex data structures should only be exchanged between agents by using the tuple space.

Template

The following code template shows agent communication via signals between a parent and its created child agents (using forking). The child agent is created in activity *a1*. After the agent process forking, both agents continue with activity *a2*.

```
// Template
this.child=none;
this.act = {
  a1: function () {
    // Create child agent
    this.child=fork();
  }
  a2: function () {
    // Raise signal
    if (this.child)
      send(this.child,'PARENT','Hello World');
  }
}
this.trans = {
    a1:a2
}
// Installation of signal handlers
this.on : {
  'PARENT' : function (arg,from) {
    log('Got message '+arg+' from '+from);
  }, ..
}
```

Types

```
// Type definitions
type aid = string
type range =
    hops:number |
    region:{dx:number,dy:number,..}
```

Operations

send[1,2,3]

```
function (to:aid,
          sig:string|number,
          arg?:*)
```

Sends a signal `@sig` (string or number) to an agent with identification string `@to` with an optional argument `@arg`.

broadcast[1,2,3]

```
function (class:string,
          range,
          @sig,
          @arg?)
```

Broadcasts a signal to multiple agents of class `@class` with the specified range.

sendto[1,2,3]

```
function (to:dir,
          sig:string|number,
          arg?:*)
```

Sends a signal @sig (string or number) to a remote node specified by @to with an optional argument @arg. If there is an agent on the remote node handling the specific signal it will be passed to the listening agent.

sleep

```
function (tmo:number)
```

Suspends an agent for a specific time (milli seconds). If @tmo is zero, the agent is suspended until it will be woken up by another agent using the wakeup operation or by the same agent via a signal handler.

wakeup

```
function (aid?:string)
```

Wakes up a sleeping agent. Can be called from within an signal handler. If @aid is undefined, the agent calling wakeup will be woken up (if suspended).

timer.add

```
function (tmo:number,
          sig:string,
          arg:*,
          repeat:boolean)
         → string
```

Adds and start a new timer that raises the signal sig after timeout. Returns a timer identifier.

timer.delete

```
function (sig:string)
```

Deletes a timer referenced by the identifier returned from `timer.add`.

Agent Control

Agents can be instantiated from an agent class template (previously loaded into the platform) by using the `create` operation with parameter initialisation. Agent class parameters must be passed immediately to agent body variables. They are not accessible during run-time!The agent class `ac` must be loaded previously as an agent class template and is provided by the platform. Alternatively, the agent class can be a sub-class of the current agent.

Furthermore, agents can be forked from the current agent process inheriting the entire data and control state including the current agent behaviour (activities, transitions, ..). Specific body variables of the forked agent can be overridden by the attributes of the settings object passed on the fork call. Forking discards all current scheduling blocks, in contrast to migration!

A newly created agent is identified by a (node) unique identifier string (commonly 8 characters) that is returned by the create and fork operations.

At least privilege level 1 is required to use these operations.

Agent Creation Operations

create[1,2,3]

```
function (ac:string,
          [arg1,arg2,..],
          level?:number)
```

```
         → aid
function (ac:string,
          {arg1:*,arg2:*,..},
          level?:number)
         → aid
```

Creates a new agent from agent class `ac` with the given set of arguments. The agent constructor function *ac* must be available on the platform. Agent class arguments are passed to agent class parameters during the creation or forking process. Arguments can either be passed in an array matching parameters in the order they are defined, or by using an argument object with arbitrary parameter order. Optionally the privilege level of the new agent can be specified, otherwise the new agent inherits the level of the creating agent. The highest level is limited to the level of the creating agent! The initial activity executed by the newly created agent is specified by the constructor function in the `next` attribute.

fork[1,2,3]

```
function (parameter:{var1:*, var2:*,..},
          level?:number)
         → aid
```

Forks a copy of the current agent process inheriting the entire data and control state of the parent agent. The new child agent can reference its parent agent by the `this.parent` attribute or by using the `myParent` function. The child agent body variables `var1`, `var2`, .. passed by the parameters object are overridden on forking with the given values. Note that only existing agent body variables (with a defined value) can be overriden.

Note that the original agent class parameters cannot be accessed after the creation of an agent. The next activity executed after the fork is either computed by the current transition entry or by a `next` variable override by the parameter object.

Example

```
id = create('explorer',{dir:DIR.NORTH,radius:1});
child = fork({x:10,y:20});
kill(child);
```

Among the creation and destruction of agents, the agent behaviour can be modified by agents by adding, deleting, or updating of transitions and activities (modification of the ATG). Only whole activities can only be changed and not code parts. There are two objects accessible by agents providing modification operations: `act` and `trans`. ATG transformations can be temporarily, e.g., used to create child agents with different or reduced behaviour.

Agent Behaviour Operations

act.add

```
function (act:string,
          code:function)
```

Adds a new activity `@act` with the given code to the current agent object.

act.delete

```
function (act:string)
```

Deletes an activity `@act` from the current agent object.

act.update

```
function (act:string,
          code:function)
```

Updates code of activity @act of the current agent object.

trans.add

```
function (trans0:string,
          code:function|string)
```

Adds a new transition starting from activity @trans0 with the given code to the current agent object.

trans.delete

```
function (trans0:string)
```

Deletes a transition from activity @trans0 from the current agent object.

trans.update

```
function (trans0:string,
          code:function|string)
```

Updates code of transition starting from activity @trans0 of the current agent object.

Example

```
this.act = {
  a1: function () {..},
  a2: function () {
    act.delete(a1); trans.delete(a1);
    act.add('b1', function () {
      this.sensor=[]; ..});
    trans.update(a2, function () {
      return this.sensor.length>0?b1:a3 });
  },
  a3: ..
  ..
};
this.trans = {
  a1: a2,
  a2: a3,
  a3: ..
}
```

Process Control

The main control flow of and agent is related to the ATG and (conditional) transitions itself. An agent can call blocking statements within an activity. A blocked activity stops agent execution until an event occurs. But signal handlers can be still executed even the agent is in a blocked state. Among external suspend-wakeup control, the agent itself can suspend and resume its execution explicitly by the following operations. Blocking statements may only occur at the end of an activity (or at least there may be only one blocking statement in one activity).

sleep

```
function (millisec?:number)
```

Suspends agent execution (current activity) for a specific amount of time (milli seconds resolution) or until a wakeup operation (from within a signal handler) is executed.

wakeup

```
function (process?)
```

Wakes up a sleeping (suspended) agent process.

Agent Mobility

Agent processes can migrate to another physical or logical node by transferring its current control and data snapshot via a message over a transport channel. The destination (specified by the transport channel) is selected by a direction `DIR`. If the `moveto` operation is executed at the end of an activity or the current scheduling block is empty after migration, the next activity is computed after migration on the new JAM node.

If a migration to a specific host or in a specific direction is not possible, a `MOVE` exception is thrown.

Types

```
enum DIR = {
  NORTH , SOUTH , WEST , EAST ,
  LEFT , RIGHT , UP , DOWN,
  ORIGIN ,
  NW , NE , SW , SE ,
  PATH (path:string),
  IP   (ip:string),
  NODE (node:string),
  CAP  (cap:string|capability),
  North (string),
  South (string),
  West (string),
  East (string),
} : dir
```

Operations

moveto[1,2]

```
function (to:dir)
```

Migrates current agent to a new node specified by the destination `@to`. If the node is not reachable the agent is killed if it not cathes the `MOVE` exception.

opposite

```
function (dir) → dir
```

Returns the opposite (back) direction (if any) of the given direction. E.g., opposite of `NORTH` is `SOUTH`. In the case of IP links and migration the *opposite* operation can return the IP address or the node name of the last node, i.e., `opposite(DIR.IP())` and `opposite(DIR.NODE())`, respectively.

link

```
function (dir)
  → boolean|string|string[]
```

Tests a link direction. Should be used prior to migration (migration with not available link direction causes an exception). In the case of multi-cast links (e.g., IP), a list of connected/reachable IPs (routes, using pattern `DIR.IP('*')`) or Nodes (using pattern `DIR.IP('%')`) is returned.

Reachable nodes from unicast IP-P2P links can be asked by using the direction constructors `DIR.North('%')` and so on.

Examples

```
// Activity in agent class template
move : function () {
  if (this.verbose>0) log('Move -> '+this.dir);
  if (!this.goback) this.backdir=opposite(this.dir);
  switch (this.dir) {
    case DIR.NORTH: this.delta.y--; break;
    case DIR.SOUTH: this.delta.y++; break;
    case DIR.WEST:  this.delta.x--; break;
    case DIR.EAST:  this.delta.x++; break;
  }
  if (this.dir!=DIR.ORIGIN && link(this.dir)) {
    this.hop++;
    moveto(this.dir);
  }
}
```

The possible migration directions depend on the network toplogy and the ports available on the agent's current node and the established links between nodes.

IP (UDP/TCP/HTTP) links can be established between generic bidirectional (multicast) IP ports with (`DIR.IP("ip:ipport")`) or between unidirectional (uni-cast) ports, e.g., `DIR.NORTH("ip:ipport"))`, commonly connected to a South port on the remote endpoint given by `DIR.SOUTH("ip:ipport"))`, respectively.

Generic IP ports can spawn arbitrary mesh grids. Alternatively, a destination node can be specified, i.e., `DIR.NODE(nodeid)`.

After an agent migration, the agent can retrieve its backpropagation direction, i.e., last node identifier or IP address by using the `opposite(DIR.NODE())` and `opposite(DIR.IP())` operations, respectively.

```
function mi(dest){
  this.src=null;
  this.dest=dest;
  this.act={
    init:function ()    {
     log('Starting on '+myNode())},
    goto: function ()   {
     log('Going to '+DIR.print(this.dest));
     if (link(this.dest)) moveto(this.dest);
     else log('No route')},
    goback: function () {
      this.src=opposite(DIR.NODE());
      log('Going back to '+DIR.print(this.src));
      moveto(this.src)},
     end: function ()    {
      log('End'); kill() }
  }
  this.trans={
    init:goto, goto:goback, goback:end
  }
  this.next=init
}
```

Example. Agent forward and backward migration between two nodes

Security

Changing of agent privilege levels and roles requires secured capabilities. Furthermore, agents can use capability protection to ensure authentication and authorisation of operations.

negotiate

```
function (resource:string,
          value:*,
          capability?)
         → boolean
```

Negotiates an agent constraint parameter. Level 0 and 1 agents require a valid access capability with sufficient rights (0x80). The `LEVEL` resource is the agent privilege level. Supported resources are:

```
typeof
@resource='CPU'|'SCHED'|'MEM'|'TS'|'AGENT'|'LEVEL'
```

privilege

```
function ()
  → number={0,1,2,3}
```

Returns the current privilege level of the agent

Capability

```
type port = string[6]
type privat = {
    prv_obj : number[0..65535],
    prv_rights : number[0..255],
    prv_rand : port
}
type capability = {
    cap_port: port,
    cap_prv: privat
}
```

Operations

The following capability and security functions are available.

Port

```
function (port_vals: numner [])
         → port
```

Creates a port (if *port_vals* is undefined a null port is returned).

Port.toString

```
function (port) → string
```

Returns a string representation of a port

(`XX:XX:XX:XX:XX`)

Port.ofString

```
function (string) → port
```

Returns a port from a string representation (`XX:XX:XX:XX:XX`)

Port.unique

```
function () → port
```

Returns a fresh unique port from a random generator.

Private

```
function (obj:number,
          rights:number,
          rand:port)
         → privat
```

Creates a private object (if *obj* is undefined a null private object is returned).

Private.toString

```
function (privat) → string
```

Returns a string representation of a private object (`obj(rights)[XX:XX:XX:XX:XX]`)

Private.ofString

```
function (string) → privat
```

Returns a private object from a string representation (`obj(rights)[XX:XX:XX:XX:XX]`)

Capability

```
function (port, privat)
  → capability
```

Creates a capability object (if *port* is undefined a null capability object is returned).

Capability.toString

```
function (capability) → string
```

Returns a string representation of a capability object:

```
(`[XX:XX:XX:XX:XX:XX]obj(rights)[XX:XX:XX:XX:XX]`)
```

Capability.ofString

```
function (string) → capability
```

Returns a capability object from a string representation:

```
(`[XX:XX:XX:XX:XX:XX]obj(rights)[XX:XX:XX:XX:XX]`)
```

Connectivity

connectTo[3]

```
function (dir:dir,
          @options) → link
```

Connects this node to another node using a virtual or physical channel link. Common ports are non-directed multi-cast IP ports. E.g., for connecting a node IP port to another IP port of a remote agent platform, the direction argument is `DIR.IP("<ipaddr>:<ipport>")` or by using the remote node name `DIR.NODE(<nodename>)`. Directional ports (supporting uni-cast P2P links only) like `DIR.NORTH` can be connected to another directional port by using the geometric opposite direction (in this example using `DIR.SOUTH` as destination). A different situation occurs if a directional port is established by IP communication (with an IP address and unique IP port). In this case the source port has to be specified (!) with the destination IP as an argument, e.g., `DIR.NORTH("<ipaddr>:<ipport>")`.

Scheduling

There are *AgentJS* operations that can block the agent processing, i.e., suspend the agent process and synchronising with events. But the JavaScript programming model does not support code blocking. For this reason, agent processing can only be suspended in transitions between activity (i.e., the activity is suspended, not the statement). Blocking *AgentJS/AIOS* statements (e.g., `sleep`, `inp`, ..) have to be placed at the end of an activity that is the only scheduling point. And there may be only one blocking statement in an activity. To support scheduling of a sequence of blocking statements, a scheduling block can be defined within an agent activity (but not within a transition that may not block).

B

```
function(block:function [])
```

Defines a scheduling block that is executed after the current activity defining the block has terminated. Each element of the function array is treated as an anonymous (sub-)activity and may contain a blocking statement.

I

```
function (object,
          next:function,
          block:function [],
          finalize:function)
```

Iterates over object or array and applies the function block to each element.

L

```
function (init:function,
          cond:function,
          next:function,
          block:function ]})
```

Loop block iteration with initialisation, conditional, and next computation function.

SQL

Level 3 (stationary) agents can access or create SQLite data bases. Requires either a native sqlite3 plug-in (embedded already in *jx+* and *pl3*, *node.js* requires loading of an external native module), or a pure JavaScript implementation of the sqlite3 data base (default in *JAM*, relies on *emscripten* C2JS cross compilation).

Operations

db.Database[3]

```
function (options?:{mode:"r" | "r+" | "w+"})
```

```
          → sqldb
```

Creates a new data base or opens an existing from a file. A volatile data base can be created in memory by specifying a `:memory:` file path.

sqldb.createMatrix

```
method (matname:string,
          header:string | number | boolean [],
          callback?:function)
         → boolean
```

Creates a new numeric matrix in the data base. The header argument provides the type interface for all rows.

sqldb.createTable

```
method (tblname:string,
        header:{},
        callback?:function)
       → boolean
```

Creates a new data table in the data base. The header object specifies the column names.

sqldb.init

```
method ()
```

Initialize the SQL data base and start server.

sqldb.insertMatrix

```
method (mat:string,
           row:[],
           callback?:function) → boolean
```

Insert a new row in an already created matrix

sqldb.insertTable

```
method (tbl:string,
        row:[]|{},
        callback?:function)
       → boolean
```

Insert a new row in an already created table

sqldb.readMatrix

```
method (mat:string,
        callback?:function)
       → [][]|none
```

Read entire matrix

sqldb.readTable

```
method (tbl:string,
        callback?:function)
       → {}[]|none
```

Read entire table

Meta Data

```
Revision: 15/02/2020
Author: Stefan Bosse
```