Programming of Mobile Agents and the JavaScript Agent Machine (JAM)

Dr. Stefan Bosse

Version 2018-08-15

Contents

1. Reactive Agents and the ATG Model

The behaviour Φ of reactive agents is centered around the concept of a perception $\rightarrow$ processing $\rightarrow$ reasoning $\rightarrow$ action $\rightarrow$ decision cycle with actions executed within activities.

Phase	Description
Perception	Get input data from the environment. The environmental data consists of data from other agents and sensors
Processing	The new input data is processed
Reasoning	Interpretation of the input data related to the current state of the agent
Action	Modify the environment (platform & agents) using computed output data.
Decision	Decide what to do next

The agent behaviour can be considered as composition of activities (A) and transitions (T) between activities forming an AT graph (ATG). Activities can be considered as sub-behaviour satisfying a particular goal of the agent. The agent model is composed of the ATG representing the control state and a set of body variables representing the data state of an agent. The execution of an agent is encapsulated by a dynamic process. The control state of a process is primarily given by the *next* activity pointer. An agent is processed by a platform (Agent Processing Platform APP). Agent activities perform computation (modification of agent body variables), environmental interaction with other agents and platforms.

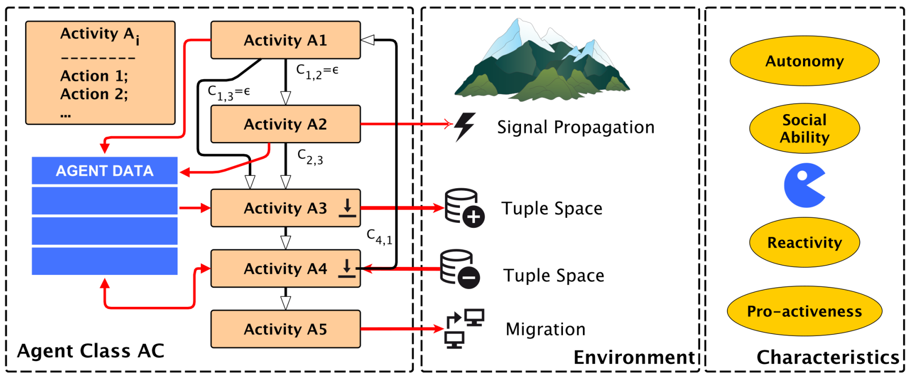


Figure 1. (Left) Agent Behaviour Model: Activity-Transition Graph and body variables (agent data) compose an agent class. (Right) Interaction with the environment (agents and platform) via tuple space access, signals, and mobility.

Environmental interaction consists of the following classes:

- Synchronized data exchange with other agents using tuple space and signal operations
- Creation and destruction of agents (in particular forking of child agents)
- Mobility by migration of an agent process snapshot to another platform node
- Modification of the agent behaviour by changing the ATG (adding, deleting, modifying transitions and activities at run-time)

A tuple space is a database that stores n-dimensional tuples (arity) providing synchronized shared memory. Each element of a tuple (column) stores a value (number, string, object, array). The database is organized with respect to the arity of the tuples. All tuples with a common arity are grouped in a tuple subspace. Tuples are stored by producer agents and are consumed by agents using pattern matching. Tuples are generative, i.e., a tuple can longer exists than the producing agent.

Transitions between activities can be unconditional or conditional based on the evaluation of expressions testing body variables. Some operations, e.g., reading a tuple from the tuple space, can block activity execution and activity transition until an IO event occurs. Therefore, even unconditional transitions are conditional related to the process state of an agent.

2. AgentJS

JavaScript is a widely used language initially designed for dynamic WEB pages. The programming model of JavaScript has two relevant influences: 1. Functional Programming 2. Object-orientated Programming.

AgentJS is a programming language for reactive agents based on the previously introduced ATG behaviour model. AgentJS is syntactically a JavaScript language with some semantic and operational changes. The agent behaviour is specified with an agent constructor function (class template). This function has the following basic structure:

Generic structure of an AgentJS constructor function

```
function ac(options) {
  this.x=0;
  this.y=options.foo;

  this.act = {
    act1: function () { .. },
    act2: function () { .. },
    act3: function () { .. },
    ..
    acti: function () { .. }
  }

  this.trans = {
    act1: function () { return <next> },
    act2: <next>.
    ..
  }

  this.on = {
    '<SIG>': function (arg) { ..},
    ..
    '<ERROR>': function (err) { .. },
  }
  this.next = <start>;
}
```

From an agent class template multiple agent objects can be instantiated. An AgentJS class template defines agent body variables (only accessible within and bound to the agent object by using the `this` object, i.e., `this.XX=<expr>`). An agent can be instantiated with arguments passed to the constructor function parameters. The parameter variables can only be accessed during agent object creation and have to be assigned to agent body variables.

The first required large section `this.act` defines all named activities of the agent. With each activity the body variables can be accessed by the using the `this` object. The AIOS defines a large set of functions that can be used by agents, e.g., the `iter(o,fun)` function that can be used to iterate over arrays and objects. If a function is passed to an AIOS function then agent body variables can be accessed within the function body using the `this` object, too!

The next required section `this.trans` defines all transitions between activities. Each starting activity is referenced by its name followed either by a function, e.g., `act1:function () { return act2}` returning the next activity (conditional or unconditional) or immediately by an activity name, e.g., `act1:act2`.

The event and exception handler section `this.on` is optional and can be used to define user defined signal handler and error handler.

Finally, the definition of the next activity pointer `this.next` and the initial activity is required, e.g., `this.next=act1`.

3. JAM: The JavaScript Agent Machine

Heterogeneous information networks require a unified agent processing platform, which can be deployed on a wide variety of host platforms, ranging from embedded devices, mobile devices, to desktop and server computers. E.g., in a seismic network some measuring stations are attached to buoy or installed on small islands, equipped only with low-power low-resource computers.

To enable seamless integration of mobile MAS in Web and Cloud environments, agents are implemented in JavaScript (JS), executed by the JS Agent Machine (JAM), implemented entirely in JS, too.

JAM can be executed on any JavaScript engine, including browser engines (Mozilla's SpiderMonkey), or from command line using node.js (based on V8) or jxcore (V8 or SpiderMonkey), or a low-resource engine JVM, shown in Fig. 1. The last three extend the JS engine with an event-based (asynchronous using callback functions) IO system, providing access of the local file system and providing Internet access. But these JS engines have high resource requirements (memory), preventing the deployment of JAM on low-power and low-resources embedded devices. For this reason, JVM was invented. This engine is based on jerryscript and iot.js from Samsung, discussed in Gavrin (2015). JVM is a Bytecode engine that compiles JS directly to Bytecode from a parsed AST. This Bytecode can be stored in a file and loaded at run-time. JVM is well suited for embedded and mobile systems, e.g., the Raspberry PI Zero equipped with an ARM processor. JVM has approximately 10 times lower memory requirement and start-up time compared with nodes.js. JAM consists of a set of modules, with the AIOS module as the central agent API and execution level.

JAM is capable of handling thousands of agents per node, supporting virtualization and resource management. Depending on the used JS VM, agent processes can be executed with nearly native code speed. JAM provides Machine Learning as a service that can be used by agents.

JAM is available as an embeddable library (JAMLIB). The entire JAM application requires about 600kB of compacted text code (500kB Bytecode), and the JAMLIB requires about 400kB (300kB Bytecode), which is small compared to other APPs. JVM+JAMLIB requires only 2.7 MB total RAM memory on start-up.

The agent behaviour is modelled according to an Activity-Transition Graph (ATG) model. The behaviour is composed of different activities representing sub-goals of the agent, and activities perform perception, computation, and inter-action with the environment (other agents) by using tuple spaces and signals. Using tuple spaces is a common approach for agent communication, as proposed by Chunlina (2002), much simpler than Bordini (2006) proposed with AgentSpeak. The transition to another activity depends on internal agent data (body variables). The ATG is entirely programmed in JavaScript (AgentJS, see Bosse (2016B) for details).

JAM agents are mobile, i.e., a snapshot of an agent process containing the entire data an control state including the behaviour program, can migrate to another JAM platform. JAM provides a broad variety of connectivity, available on a broad range of host platforms. JAM can be used as a simulation platform integrated in the SeJAM simulator. JAM is capable to execute thousands of agents. The SeJAM simulator is built on top of a JAM node adding simulation control and visualization, and can be included in a real-world closed-loop simulation with real devices.

In real-world application security is an important key feature of a distributed agent platform. The execution of agents and the access of resources must be controlled to limit Denial-of-Service attacks, agent masquerading, spying, or other abuse, agents have different access levels (roles).

There are four levels:

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

The lowest level (0) does not allow 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. The system level can not be negotiated. The capability is node 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 is 20ms)
- 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)

4. 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.

4.1. Agent Scheduling and Check-pointing

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 JS JAM platform. Agents 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. Though 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 a call to a checkpoint function CP() at the beginning of a body of each function contained

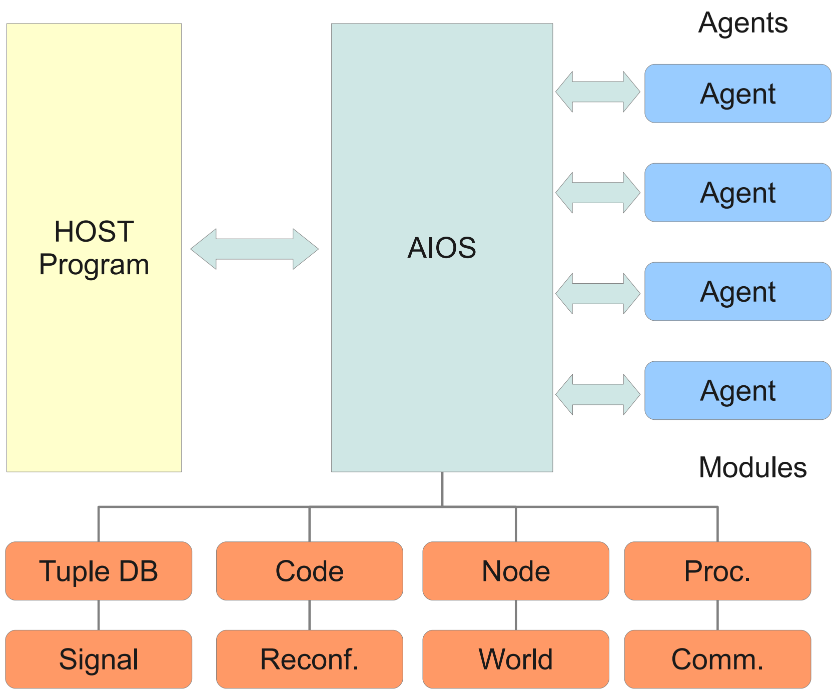


Figure 2. Interface between agents and JAM and between JAM and a host application: Agent Input-Output System (AIOS)

in the agent code, and by injecting the CP call in loop conditional expressions. Though 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 thread 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.

In contrast to the AAPL model that supports multiple blocking statements (e.g., IO/tuple-space access) inside activities, JS is not capable of handling any kind of process blocking (there is no process and blocking concept). For this reason, scheduling blocks can be used in AgentJS activity functions handled by the AIOS scheduler. Blocking AgentJS functions returning a value use common callback functions to handle function results, 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.

Algorithm 1. (*JAM Agent Scheduler*)

```
$\forall node \in world.nodes$ do
 $\forall process \in node.processes$ do
  $\cdot$ $Determine\ what\ to\ do\ with\ prioritized\ conditions:$
   $Order\ of\ operation\ selection:$
   $0.\ Process\ (internal)\ block\ scheduling\ [block]$
   $1.\ Resource\ exception\ handling$
   $2.\ Signal\ handling\ [signals]$
      $-\ Signals\ only\ handled\ if\ process\ priority < HIGH$
      $-\ Signal\ handling\ increase\ process\ priority\ temporarily\ to$
       $allow\ low-latency\ act/trans\ scheduling!$
   $3.\ Transition\ execution$
   $4.\ Agent\ schedule\ block\ execution\ [schedule]$
   $5.\ Next\ activity\ execution$
      $-\ Lowers\ process\ priority$

  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 $do\ nothing$
  else if $not\ process.blocked\ and\ process.block^1[]$
  then $execute\ next\ block\ function$
  else if $agent\ resources\ check\ failed$
  then $raise\ EOL\ exception$
  else if $process.priority < HIGH\ and\ process.signals^1[]$
  then $handle\ next\ signal,\ increase\ process.priority$
  else if $not\ process.suspended\ and\ process.transition$
  then $get\ next\ transition\ or\ execute\ next\ transition\ handler\ function$
  else if $not\ process.suspended\ and\ process.schedule^1[]$
  then $execute\ next\ agent\ schedule\ block\ function$
  else if $not\ process.suspended$
  then $execute\ next\ agent\ activity\ and\ compute\ next\ transition,$
       $decrease\ process.priority$
```

4.2. AgentJS API: Computational Functions

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.

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.

iter

`function(object|array, function (@element,@index?))`
Iteration over object attributes or array elements.

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

copy

`function(array|object) → array|object`
Returns a copy of an array or object. The object may not contain cyclic references.

div

`function(number) → number`
Integer division operation

empty

`function(array|object) → boolean`
Checks if an object or array is empty (`{} []`)

equal

`function(number|string|array|object, number|string|array|object)`
`→ boolean`
Checks equality of numbers, strings, arrays, and objects.

filter

`function(array|object, function (@element,@index?) →boolean)`
`→ array|object`
Filter operation for arrays and objects.

head

`function(array) → *`

Returns head (first) element of array.

int

`function(number) → number`
Returns integer number.

isin

`function(array|object,number|string|(number|string)[]) → boolean`
Checks 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 for arrays and objects.

length

`function(array|object|string) → number`
Returns length of an array, object or string.

map

`function(array|object, function (@element,@index?) → *|none)`
`→ array|object`
Map and filter operation for arrays and objects. If the user function return undefined the element is discarded.

matrix

`function(@cols,@rows,@init) → [] array`
Create a matrix (array of arrays).

max

`function(a:number|array,b?:number) → number`
Returns largest number from two numbers or from 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) → number|array`
Returns negative number or array of numbers.

random

`function(a:number|array|object,b?:number,frac?:number) → number|*`
Returns a random number from the interval [a,b] or an element from an array or object. The optional fraction parameter specified the rounding precision (frac=1 return integer numbers).

sort

`function(array, function (@element1,@element2) → number)`
`→ array`
Sorts an array by 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(array|object,function?) → number`
Returns the sum of elements of an array or attributed 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 argument.

tail

`function(array) → *`
Returns tail (last) element 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};
..
```

4.3. AgentJS API: Environmental Information and Modification

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).

me

`function () → string`
Returns the identity name of this agent.

negotiate

`function (resource:string,value:*,capability?) → boolean`
`with @resource='CPU'|'SCHED'|'MEM'| 'TS'|'AGENT'|'LEVEL'`
Negotiate an agent constraint paramater. Level 0 and 1 agents require a valid access capability.

privilege

`function () → number={0,1,2,3}`
Returns the current privilege level of the agent

4.4. AgentJS API: Tuple Space Operations

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 is used as a key, but that is only a weak constraint. A tuple space has a linear structure. To support complex hierarchical data bases, JAM provides a SQLite data base server and access to this data base for level 3 agents. Tuple spaces can be mapped on tables in this SQL data base.

Examples for tuple access

```
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 ..
  });
```

alt

`function(pattern [],callback:function,all?:boolean,tmo?:number)`
Input operation with multiple search patterns that can have different type interfaces 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

`function (pattern,callback:function (tuple)) → tuple`
Access an evaluator tuple created by a `listen` operation. The evaluator evaluates the given pattern to a tuple and passes the tuple to the callback function.

exists

`function (pattern) → boolean`
Check if a tuple matches the given patterns.

inp

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

`function (pattern,callback:function (pattern) → tuple)`
Install a tuple evaluator (active tuple) that can be accessed by the `evaluate` oepration.

out

`function (tuple)`
Store a tuple in the data base.

mark

`function (tuple,tmo:number)`
Store a tuple with a limited lifetime in the data base.

rd

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

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

ts

`function (pattern,callback:function(tuple)&rarr;tuple)`
Atomic and non-blocking test-and-set operation that can be used to modify a tuple in place found based on the provided pattern.

try_alt, try_inp, try_rd

`function (tmo:number,..)`
Try operation to perform an alternation, input, or read operation with a given timeout.

4.4.1. 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.

Definition 1. (*Active Tuple Template*)

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

4.5. AgentJS API: Signals and Signal Handler

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 localization, 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.

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

```
this.child=none;
this.act = {
  a1: function () {
    this.child=fork({child:none});
  }
  a2: function () {
    Raising of signal
    if (this.child) send(this.child,'PARENT',me());
  }
}
Installation of Signal Handler
this.on : {
  'PARENT' : function (arg) {
    log('Got signal from my parent '+arg);
  }, ..
}
```

send[1,2,3]

`function (to:aid,sig:string|number,arg?:*)`
Send a signal `@sig` (string or number) to an agent with identification string `@to` with an optional argument `@arg`.

broadcast[1,2,3]

`fucntion (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?:*)`
Send 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)`
Suspend agent for a specific time. If `@tmo` is zero, the agent is suspended until it will be woken up by another agent using the `wakeup` operation.

wakeup

`function (aid?:string)`
Wake 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) &rarr; string`
Add and start a new timer that raises the signal `sig` after timeout.

timer.delete

'function (sig:string)
Deletes a timer referenced by the identifier returned from `timer.add`.

4.6. AgentJS API: Agent Control

Agents can be instantiated from an agent class template (previously loaded into the platform) by using the `create` operation with parameter initialization. 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.

create^{1,2,3}

`function (ac:string,[arg`$_1$`,arg`$_2$`,..],level?:number)` $\rightarrow$ `aid`
`function (ac:string,{arg`$_1$`:*,arg`$_2$`:*,..},level?:number)` $\rightarrow$ `aid`
Creates a new agent from agent class `ac` with the given set of arguments. 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 (parameters:{var`$_1$`:*,var`$_2$`:*,..},level?:number)` $\rightarrow$ `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 `var`$_1$`,var`$_2$`,..` passed by the parameters object are overridden on forking with the given values. Note that agent class parameters cannot be accessed after the creation of an aent. The next activity

executed after the fork is either computed by the current transition entry or by a `next` variable override passed with parameter object.

Examples

```
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.

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 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.

Examples

```
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: ..
}
```

4.7. AgentJS API: Mobility

Agent processes can migrate to another node (either physical or logical) 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) -> {tag='DIR.PATH',path:string} ,
  IP   (ip:string) -> {tag='DIR.IP',ip:string} ,
  NODE (node:string) -> {tag='DIR.NODE',node:string} ,
  CAP  (cap:string|capability) -> {tag='DIR.CAP',cap:string|capability}
} : dir
```

moveto

`function (to:dir)`
Migrates current agent to a new node specified by the destination `@to`.

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|[]`
Test 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 `IP('*')`) or Nodes (using pattern `IP('%')`) is returned.

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 ports available on the agent's current node and established links between nodes. IP (UDP/HTTP) links can be established between generic not directional (multicast) IP ports (`DIR.IP("ip:ipport")`) or between directional (unicast) ports, e.g., `DIR.NORTH("ip:ipport")`, commonly connected to a South `DIR.SOUTH("ip:ipport")` port on the remote endpoint. 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.

Example 1. (*Agent forward and backward migration between two nodes*)

```
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
}
```

4.8. AgentJS API: Ad-hoc Connectivity

connectTo[3]

`function connectTo(dir:dir,@options)`
Connect this node to another node using a virtual or physical channel link.

4.9. AgentJS API: Scheduling Blocks

There are many operations that can block (suspend) the agent processing. But the JavaScript programming model does not support code blocking. For this reason, blocking AgentJS/AIOS statement (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 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 initialization, conditional, and next computation function.

4.10. AgentJS API: SQL Operations*

Level 3 (stationary) agents can access or create SQLite data bases. Requires a native sqlite3 plug-in (embedded already in *jx+*, *node.js* requires loading of an external native module).

db.Database

`constructor (filepath:string,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.

createMatrix

`method (matname:string, header:string|number|boolean [], callback?:function) -> boolean`
Creates a new number matrix in the data base. The header argument provides the type interface for all rows.

createTable

`method (tblname:string,header:{},callback?:function) -> boolean`
Creates a new table in the data base

init

Initialize the SQL data base and start server.

insertMatrix

`method (mat:string,row:[],callback?:function) -> boolean`
Insert a new row in an already created matrix

insertTable

`method (tbl:string,row:[]|{},callback?:function) -> boolean`
Insert a new row in an already created table

readMatrix

`method (mat:string,callback?:function) -> [][]|none`
Read entire matrix

readTable

`method (tbl:string,callback?:function) -> {}[]|none`
Read entire table

5. JAM API

The JAM platform is implemented as a *library* that can be embedded in any host application program. The application programs *jamsh* and *jamapp* provide a GUI and shell to the JAM library.

6. JAMLIB

6.1. Synopsis

```
use jam/aios as AI
use jam/mobi as MO
use jam/chan as CH
use jam/node as NO
use jam/ts as TS

class jam = {
  node: object,
  world: object,
  run: boolean,
  options: object,

  syntax: object {
    find: function (@root,@typ,@name),
    location: function (elem,short) -> string,
    name: function (elem) -> string,
    offset: number
  },
  addClass: method (@templates),
  addNode: method (nodeDesc)
    with nodeDesc: {x,y,id},
  addNodes: method (@nodes) -> AI.id [],
  analyze: method (@ac,@options) -> {report:string,interface},
  analyzeSyntax: private method (Esprima.syntax,options:object),
  connectNodes: method (@connections),
  connectTo: method (to:string "<dir->url>|<url>",nodeid?:AI.id),
  connected: method (dir:MO.dir,nodeid?:AI.id) ->
           none|boolean|string|string [],
  compileClass: method (name:string,constructor:function,@options),
  createAgent: method (ac:string,args:[]|{},
                     level:number,className:string) -> AI.id,
  createAgentOn: method (nodeid:AI.id,ac:string,args:[]|{},
                        level:number,className:string) -> AI.id,
  createPort: method (dir:MO.dir,@options,@nodeid) -> CH.channel,
  disconnect: method (to:MO.dir,nodeid?:AI.id),
  emit: method(@event,@arg1,..),
  // Execute an agent snapshot delivered in JSON+ text format
  execute: method (data:string,file?:string),
  executeOn: method (data:string,node:number|string,file?:string),
  extend: method (level:number|number[],
                  name:string,code:function,
                  argn?:number|number []),
  getNode: method (string|number|object) -> NO.node|undefeined,
  getNodeName: method (@nodeNumberorPosition) -> string,
  getWorldName : method () -> string,
  init: method (callback),
  inp: method (TS.pattern) -> TS.tuple|none,
  kill: method (AI.id),
  migrate: method (@data),
  on: method (event:string,handler:function),
  open: method (file:string,@options),
  out: method (TS.tuple),
  rd: method (TS.pattern) -> TS.tuple,
  readClass: method (file:string,@options),
  register: function (node:NO.node),
  removeNode: method (AI.id),
  rm: method (TS.pattern),
  saveSnapshot: method (aid:string,file?:string,kill?:boolean) ->
                string|undefined,
  saveSnapshotOn: method (aid:string,node:number|string,
                          file?:string,kill?:boolean) ->
                  string|undefined,
  schedule: method (),
  setCurrentNode: method (number),
  signal: method (to:AI.id,sig:string,arg:*,@broadcast?),
  start: method (),
  stats: method (@kind) -> object
    with @kind={'process'},
  step: method (steps:number,callback:function),
  stop: method (),
  time: method () -> number,
  ts: method (pat:TS.pattern,callback:function) -> TS.pattern,
  version: method () -> string
}

Jam: constructor (options) -> jam
  with options : {
    connections?,
    print?:function (string) is agent and control message output function,
    print2?:function (string) is agent message only output function,
    provider:function (TS.pattern) -> TS.tuple|none,
    consumer:function (TS.tuple) -> boolean,
    @classes?,
    id?:AI.id     is node identifier,
    verbose?:number,
    TMO?:number   is default cache timeout,
    nolimits?:boolean,
    nowatch?:boolean,
    checkpoint?:boolean,
    log?:{class?:boolean,node?,agent?,parent?,host?,time?,Time?,pid?},
  }
  and connections : {
    $kind : {
      send:function(data:string|buffer,dest:MO.DIR) -> number,
      link?:function (MO.DIR) -> boolean
    } , ..
  }
  and $kind = {north,south,west,east,nw,sw,ne,se,up,down,path,dos,ip}
```

6.2. Description

The JAM library implements JAM and provides an API that can be use dby any host application.

7. Using JAM

7.1. JAM Library

JAM is provided as a library that can be embedded in any host application program written entirely in JavaScript. The library `jamlib` provides a JAM world constructor function `Jam: function (@options)` $\rightarrow$ `jam`. A JAM instance consists of the Agent Input and Output System (AIOS), a world with at least one JAM node, and an agent compiler and analyzer. Multiple virtual nodes can be connected in this world providing an artificial JAM network. Please note that all virtual nodes are executed in one host process and sharing the same AIOS scheduler but having different tuple and agent spaces. Additionally, a JAM node can be connected to other physically separated nodes via IP links. To utilize multi-processor platforms, a physical cluster of nodes can be created.

7.2. Creating a simple JAM Instance

Using the JAM library (`jamlib`) it is easy to create a JAM instance. The following JavaScript code can be started by any command line JS VM, e.g., *node.js* or *jxcore*. Do not forget to initialize and start the JAM instance.

API

```
constructor Jam(@options) → jam;
jam.init: method ();
jam.start: method ();
```

Example

```
var JamLib = require('./jamlib');
var JAM = JamLib.Jam({
  connections:{
    ip:{
      from:'localhost:10001', // Create IP-AMP port
      proto:'udp'
    }
  },
  print:function (msg) {console.log(msg)},
  verbose:JamLib.environment.verbose||1,
});
JAM.init();
JAM.start();
```

After the JAM instance was started it is ready to process agents. Since an IP-AMP (Agent Management Port) communication link was created (listening on IP port 10001), external programs can connect and can access the JAM, e.g., using the *jamp* utility capable to send agent constructor functions (class templates) and to send agent processes ready to start.

An example is shown below. The `helloworld.js` file contains the constructor function `function(options){}` definition for the agent class `helloworld`. The JAM node has an AMP-IP listening on port 10001 (on localhost). A full URL can be specified, too. The constructor function argument(s) can be given in curled parentheses.

```
jamp connect 10001 compile helloworld.js \
     create helloworld {verbose:1} execute
jamp connect 1.1.2.3:10001 compile helloworld.js \
     create helloworld {verbose:1} execute
```

7.3. Adding and Importing Agent Class Templates

An agent class template can be imported (analyzed and compiled) from a file by using the JamLib `readClass(filename,options)` method. The file to be imported can contain one function constructor only (without any module export statements) as defined above or a set of agent class constructor functions exported by `module.exports={ac1:function,ac2:function,..}`. An embedded agent class constructor function can be added by using the `compileClass(classname:string, function, verbose:number)` method.

API

```
constructor Jam(@options) → jam;
jam.compileClass: method (name:string,function,verbose?:number);
jam.readClass: method (file:string,@options);
```

Example

```
var JamLib = require('./jamlib');
var JAM = JamLib.Jam({..});
JAM.init();
JAM.start();
// Import from file
JAM.readClass('agent.js',{verbose:1});
// Embedded constructor function
function ac(options) {
  this.xx=..;
  this.act={..};
  this,trans={..};
  this.next=xx;
}
JAM.compileClass('My Class Name',ac,1);
```

7.4. Creating Agents programmatically

An agent can be instantiated from an agent constructor function using the `createAgent(function|string,arguments:[],level)` method either directly by providing the constructor function, agent object arguments, and the initial agent AIOS level, or by referencing an already compiled agent class.

```
var JamLib = require('./jamlib');
var JAM = JamLib.Jam({..});
JAM.init();
JAM.start();
// Import class ac from file
```

```
JAM.readClass('agent.js',{verbose:1});
// Embedded constructor function
function ac(options) {
  this.xx=options.xx;
  this.act={..};
  this,trans={..};
  this.next=xx;
}
var ag1 = JAM.createAgent('ac',{xx:1},2);
var ag2 = JAM.createAgent(ac,{xx:2},1);
```

7.5. Connecting JAM nodes

Usually JAM nodes are organized in cell- or mesh-like network structures. Two JAM nodes can be connected P2P using directional ports, e.g., `NORTH`, `SOUTH`, `WEST`, or P2N using the IP port `IP`.

Node 1

```
var JamLib = require('./jamlib');
// JAM Node 1
var JAM1 = JamLib.Jam({
  connections:{
    north:{
      from:'hosta:10001', // Create IP-AMP port
      proto:'udp'
    }
  } ..
```

Node 2

```
var JamLib = require('./jamlib');
// JAM Node 2, executed in a different process
var JAM2 = JamLib.Jam({
  connections:{
    south:{
      from:'hostb:10002', // Create IP-AMP port
      proto:'udp'
    }
  } ..

..
JAM2.connectTo('south->hosta:10001');
```

7.6. Extending AIOS of JAM

The AIOS can be extended easily with user supplied functions.

API

```
constructor Jam(@options) → jam;
jam.extend: method (level:number|number[],name:string,
                    function,argn?:number|number []),
```

Example

```
var JamLib = require('./jamlib');
// Create the JAM instance
var JAM = JamLib.Jam({ .. });
// Extend AIOS
function joke() {
  return "He said: Onions are the only food that can make you cry."+
         "So I threw a coconut in his face."
}
// Extend only level 1/2 agents
JAM.extend([1,2],'joke',joke);

function someagent() {
  this.act = {
    lough: function () { log(joke()) }
  }
}
```

7.7. Extended IO of JAM: Adding tuple providers

Agents can read (consume) tuples from the tuple space provided by each JAM node that are stored by other agents (or the consuming agent). To extend the tuple space, the host application can provide tuples on request and can consume tuples stired by agents. This extends the inter-agent communication to the host application IO system. The provider and consumer functions must be passed by the options object on JAM instantiation.

API

```
use jam/ts as TS
constructor Jam({
    provider:function(TS.pattern) → TS.tuple|none,
    consumer:function(TS.tuple) → boolean
  }) → jam;
```

Example

```
function provider(pat) {
  switch (pat.length) {
    case 2:
      switch (pat[0]) {
        case 'SENSOR2':
          return [pat[0],(256*rnd())|0];
      }
      break;
  }
}

function consumer(tuple) {
  switch (tuple.length) {
    case 3:
      switch (tuple[0]) {
        case 'ADC':
          console.log('Host application got '+tuple);
          return true;
      }
      break;
  }
  return false;
}
var myJam = JamLib.Jam({
  consumer:consumer,
  print:console.log,
  provider:provider,
  verbose:0,
});
```

8. JAMSH

8.1. Synopsis

JAMH: JAM Shell

```
Shell Commands
The following shell commands are avaiable:
add({x,y})
  Add a new logical (virtual) node
broker(ip)
```

```
  Start a SIP UDP broker server
connect({x,y},{x,y})
  Connect two logical nodes
connected(to:dir)
  Check connection between two nodes
compile(function)
  Compile an agent class constructor function
create(ac:string,args:*[]|{},level?:number,node?)
  Create an agent from class @ac with given arguments @args and @level
env
  Shell environment including command line arguments a:v
exit
  Exit shell
extend(level:number|number[],name:string,function,argn?:number|number[])
  Extend AIOS

http(ip,dir,index?)
  Create and start a HTTP file server
inp(pattern:[],all:boolean)
  Read and remove (a) tuple(s) from the tuple space
kill(id:string)
  Kill an agent (id="*": kill all)
link(to:dir)
  Connect two phyiscal nodes
lookup(pattern:string,callback:function (string [])
  Ask broker for registered nodes
log(msg)
  Agent logger function
open(file:string)
  Open an agent class file
out(tuple:[])
  Store a tuple in the tuple space
port(dir,options,node)
  Create a new physical communication port
rd(pattern:[],all:boolean)
  Read (a) tuple(s) from the tuple space
rm(pattern:[],all:boolean)
  Remove (a) tuple(s) from the tuple space

script(file:string)
  Load and execute a jam shell script
setlog(<flag>,<on>)
  Enable/disable logging attributes
signal(to:aid,sig:string|number,arg?:*)
  Send a signal to specifid agent
start()
```

```
  start JAM
stats(kind:"process"|"node"|"vm")
  Return statistics
stop()
  stop JAM
ts(pattern:[],callback:function(tuple)->tuple)
  Update a tuple in the space (atomic action) - non-blocking
time()
  print AIOS time
unlink(to:dir)
  Disconnect remote endpoint
verbose(level:number)
  Set verbosity level
```

8.2. Description

The JAM Shell *jamsh* is a command line interpreter that provides direct access to the JAM libraray *jamlib*. Commands can be executed either from command line (of the shell) or by a script.

8.3. Networking

Networking consists of the creation of ports and links between ports (and JAM nodes). Commonly multicast IP ports are used in the Internet domain. A multicast IP port can conenct with an arbitrary number of IP ports of remote JAM nodes. All ports provide an Agent Management Port (AMP) interface used to transfer agent code, signals, tuples, and control messages between JAM nodes. In the Internet or Intranet domain there are three different communciation protocols that can be used to transport AMP messages: UDP, TCP, and HTTP. Different IP ports using different protocols can coexist on a JAM node. All IP ports are handled by an internal router.

Note: An IP port can be defined by using the `DIR.IP("ip:port")` directional type. Other port directions like `DIR.NORTH("ip:port")` can be used, too. But these port directions support unicast ports only.

8.3.1. UDP Unicast ports and links

```
port(DIR.IP(ip:number|string="<ip>:<port>"),
     {proto:'udp',multicast:false,verbose:1});
link(DIR.IP(ip:number|string="<ip>:<port>"));
```

8.3.2. UDP Multicast ports and links

```
port(DIR.IP(ip:number|string="<ip>:<port>"),
     {proto:'udp',verbose:1});
link(DIR.IP(ip:number|string="<ip>:<port>"));
```

8.3.3. UDP Multicast ports using broker service

Client Side

```
port(DIR.IP("*"),{proto:'udp',broker:"<ip>:<port>",
                  name:'/domainX/'+name("node"),
                  multicast:true,verbose:1});
link(DIR.IP("/domainX/B"));
lookup(DIR.PATH('/domainX/*'),function (result) {
  log('lookup: '+result)
});
```

Broker Server

```
broker("<ip>:<port>");
```

8.4. Example

```
// Agent Class Construtor
function fib(args) {
  this.todo = args.val;
  this.output = [];
  this.f = function(n) {
        return n < 2 ? n : this.f(n-2) + this.f(n-1)
  }

  this.act = {
    calculate: function() {
      var n = head(this.todo)
      this.todo = filter(this.todo, function(elem) { return elem != n })
      var result = this.f(n)
      this.output.push(result)
    },
    print: function() {
      var next = head(this.output)
      this.output = filter(this.output, function(elem) { return elem != next })
      log(next)
    }
  }

  this.trans = {
     calculate: function() {
       return empty(this.todo) ? print : calculate
     },
    print: function() {
      if(empty(this.output)) {
        log('Killing agent')
        kill()
      }
      return print
    }
  }

  this.next = calculate
}
// Compile agent class and add it to the world library
compile(fib)
// Start JAM scheduler loop
start()
// Create an agent from already compiled class
create('fib', {val: [10, 5]})
```

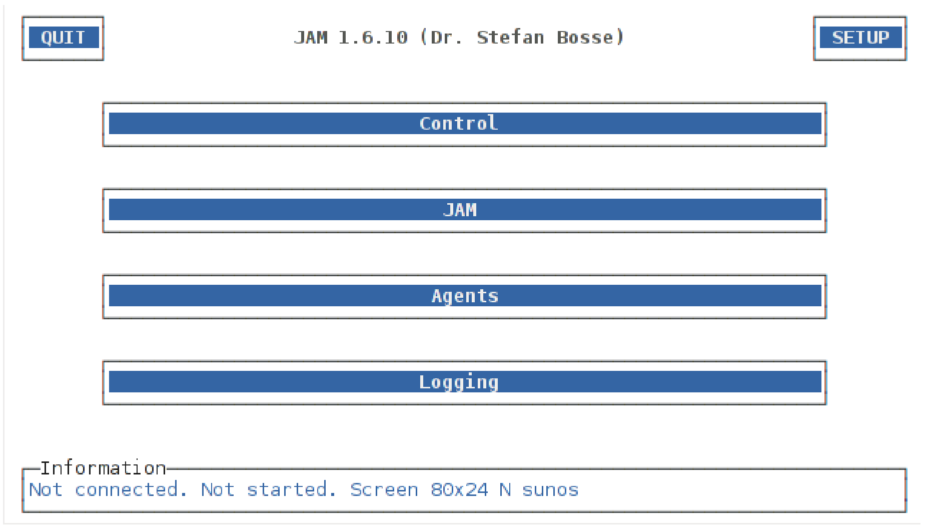
9. JAM APP

The JAM APP is a JAM node with a GUI (terminal based). It can be configured by users and automates the deplyoment of JAM, especially on mobile devices.

The GUI is organized in pages similar to a mobile App layout. The navigation can be done by using the top button row that provides left and right page buttons.

9.1. GUI

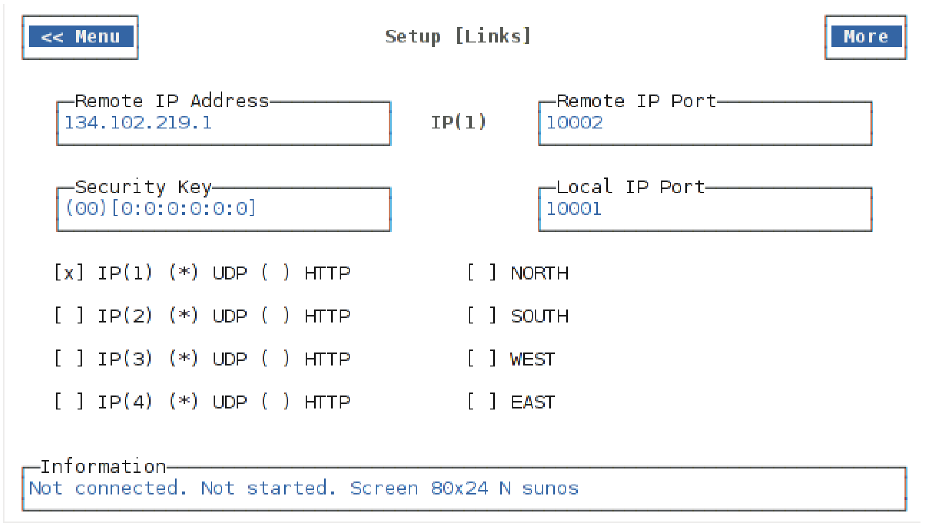
The start screen provides the main menu:



Information
Not connected. Not started. Screen 80x24 N sunos

Setup

Upper right button selects the entry page of the setup menu. The first page configures JAM links. There are up to four IP Point-to-Network (P2N) links that can be established to other JAM nodes and four “directional” P2P links (Direction North, South, West , East). Each links consists of a remote endpoint IP address and IP port, an optional security key, and an optional local endpoint IP port. Usually all four IP links share the same local IP endpoint. A link can be established via the unreliable UDP or by a (less efficient) reliable HTTP protocol.



Remote IP Address: 134.102.219.1
Remote IP Port: 10002
Security Key: (00)[0:0:0:0:0:0]
Local IP Port: 10001

```
[x] IP(1)  (*) UDP ( ) HTTP        [ ] NORTH
[ ] IP(2)  (*) UDP ( ) HTTP        [ ] SOUTH
[ ] IP(3)  (*) UDP ( ) HTTP        [ ] WEST
[ ] IP(4)  (*) UDP ( ) HTTP        [ ] EAST
```

Information
Not connected. Not started. Screen 80x24 N sunos

Control
The control menu provides actions for starting and stopping the JAM. Furthermore, an intial network connection to other JAM nodes can be started. The parameters for network connectivity is defined in the setup menu section. The `Reset` button resets the entire JAM instance that is created on the first start action by using the `Start` button. A reset automatically disconnects the JAM from any other JAM node.

JAM

Agents

Logging

9.2. Options

The JAM App can be started from command line with the following options.

style:*style*
Select a different GUI (color) styles: `black`|`invert`|`simple`

mode:server
Starts the App in server mode (i.e., command line mode) without a GUI.

config:*file*
Loads an alternative configuration file.

9.3. Configuration File

The default configuration file is `jam.app.config`. The configuration file contains JAM and GUI related entries as shown below. Note that if node and wolrd names are specified in the configuration file then they are used permanently. To get a new random node-world name pair the `nodename` and `worldname` entries have to be set to value `null`. The new names are saved in the configuration file automatically.

```
{
  "agents": {
    "level": 1
  },
  "domain": "default",
  "expert": false,
  "keyboard": false,
  "public": {
    "IP(1)": {
      "address": "134.102.219.1",
      "port": 10002,
      "local": "10001",
      "enable": true
    }
  },
  "nodename": "hamuxujo",
  "links": [
    "IP(1)",
    "IP(2)",
    "IP(3)",
    "IP(4)",
    "NORTH",
    "SOUTH",
    "WEST",
    "EAST"
  ],
  "log": {
    "node": false,
    "agent": true,
    "class": false,
    "time": false
  },
  "logJam": {
    "world": false,
    "node": false,
    "pid": false,
    "time": false
  },
  "proto": "udp",
  "script": "load('node.js');\ncreate('node',{repeat:10});\n",
  "security": false,
  "sensors": true,
  "simple": false,
  "verbose": 0,
  "worldname": "HAMUXUJO"
}
```

9.4. Node Management Agent

A JAM APP can connect automatically to specified destination nodes. But connections can be unreliable and being unlinked after multiple communication failures. There is no reconnect feature implemented in JAM. To ensure permanent connectivity (a link) to another node, a node management agent should be started that monitors pre-configured links and executes an initial and succeeding connect requests. Furthermore, a node management agent can perform additional repeating tasks, like collecting of data from tuple spaces and storing data in file databases (SQL) or vice versa.

A node management agent must be started with privilege level 3 to enable system access operation (connect, file database access, ..). This can be done in the script entry of the configuration file: `load('node.js'); create('node',{..},3)`. A node management agent can fail, too. To ensure permanent presence of this management agent, an agent monitor can be used to start and monitor an agent. If the agent terminates (e.g., due to a failure), a new instance is started automatically. In the configuration file use instead: `load('node.js'); monitor('node',{..},3)`

Example

```
function node(options) {
  this.text=options.text;
  this.repeat=options.repeat||1;
  this.time=1000;
  this.links=[];
  this.connects=[];
  this.sensors={};
  this.config={};
  this.pendingjobs=[];
  this.verbose=1;

  this.act={
    init: function () {
      var conn;
      log('Starting. I am from class '+myClass());
      if (privilege() == 3) {
        this.config=config();
        if (this.verbose) log(this.config);
        iter(this.config.public,function (addr,ch) {
          if (addr.enable && addr.address && addr.port) {
            conn={address:addr.address,port:addr.port,time:time(),state:false};
            this.connects.push(conn);
          }
        });
        log('Connects: ',this.connects);
      }
      try_rd(0,['SENSORS',_],function (t) {
        if (t) log('Sensors available: ',t[1])
      });
    },
    percept: function () {
      var curlinks;
      log('Percepting ..');
      // Get all currently linked nodes
      curlinks=link(DIR.IP('*'));
      log(curlinks);
      try_rd(0,['SENSOR','cpu',_],function (t) {
        log(t);
        if (t) this.sensors.cpu=t[2];
      });
    },
    service: function () {

    },
    wait: function () { log('Sleeping ..'); sleep(this.time) },
    end: function () { log('Terminate!'); kill() }
  };
  this.trans={
    init:percept,
    percept:function () { return this.pendingjobs.length?service:wait },
    service:wait,
    wait:function ()  { this.repeat-; return this.repeat?percept:end }
  };
  this.next=init;
}
```

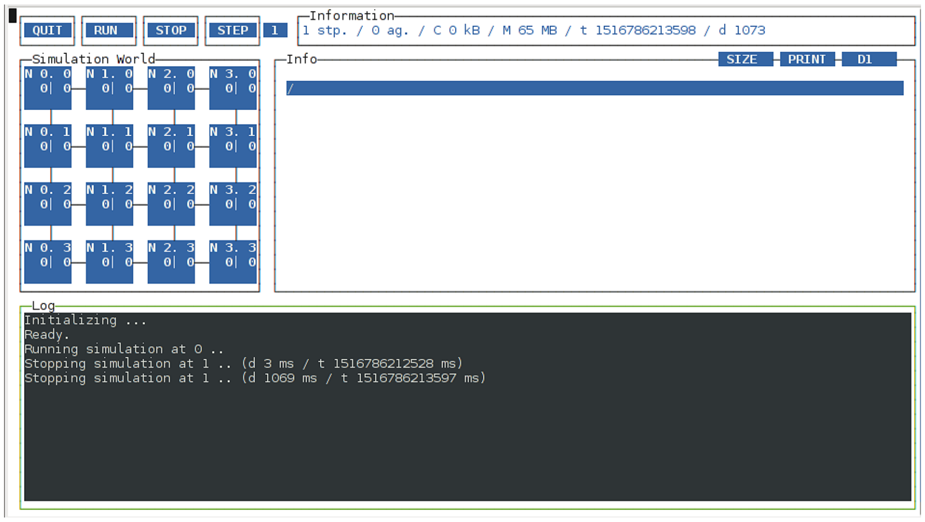
10. Simulation

The JAM can be used for agent-based simulation and simulation of MAS worlds, too. The Simulation Environment for JAM (SEJAM) adds visualization and control layers on the top of the JAM. This enables the simulation of virtual worlds consisting of multiple logical JAM nodes connected, e.g., in mesh-like networks by virtual channels.

Due to the deployment of a real JAM inside the simulator, the simulator can be connected to real world networks enabling hardware-in-the-loop simulations.

There are two simulators available:

SEJAM
SEJAM provides a simple terminal-based GUI and can be executed on any platform.

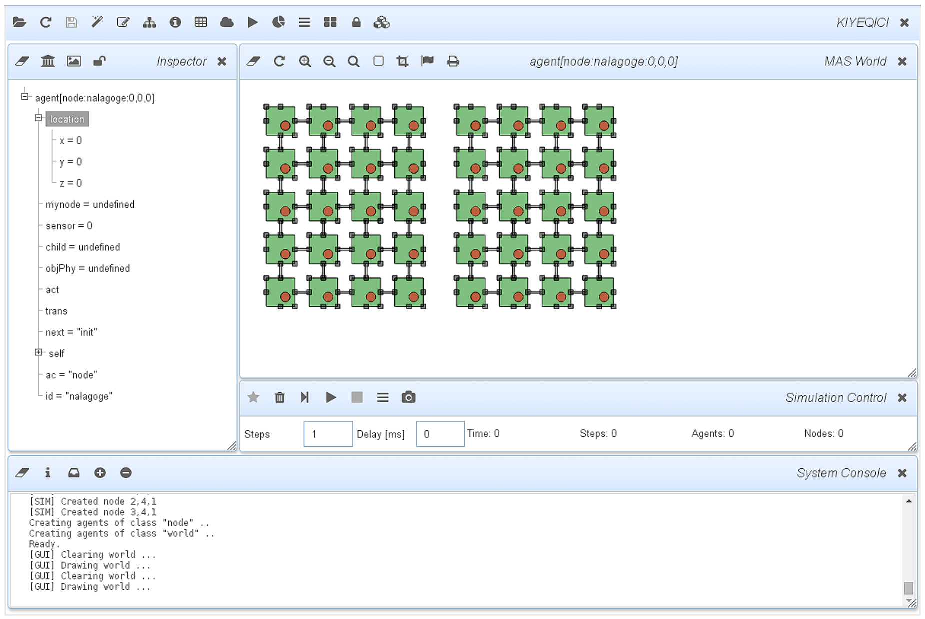


```
1 stp. / 0 ag. / C 0 kB / M 65 MB / t 1516786213598 / d 1073
```

```
Initializing ...
Ready.
Running simulation at 0 ..
Stopping simulation at 1 .. (d 3 ms / t 1516786212528 ms)
Stopping simulation at 1 .. (d 1069 ms / t 1516786213597 ms)
```

SEJAM2

SEJAM2 provides a complex GUI with additional analysis and visualization tools based on node webkit (node + chromium browser).



```
agent[node:nalagoge:0,0,0]
  location
    x = 0
    y = 0
    z = 0
  mynode = undefined
  sensor = 0
  child = undefined
  objPhy = undefined
  act
  trans
  next = "init"
  self
  ac = "node"
  id = "nalagoge"
```

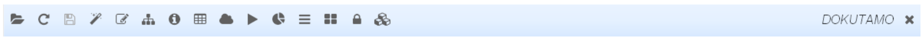
Steps 1 Delay [ms] 0 Time: 0 Steps: 0 Agents: 0 Nodes: 0

```
[SIM] Created node 2,4,1
[SIM] Created node 3,4,1
Creating agents of class "node" ..
Creating agents of class "world" ..
Ready.
[GUI] Clearing world ...
[GUI] Drawing world ...
[GUI] Clearing world ...
[GUI] Drawing world ...
```

The SEJAM2 simulator window consists of the following sub-windows:

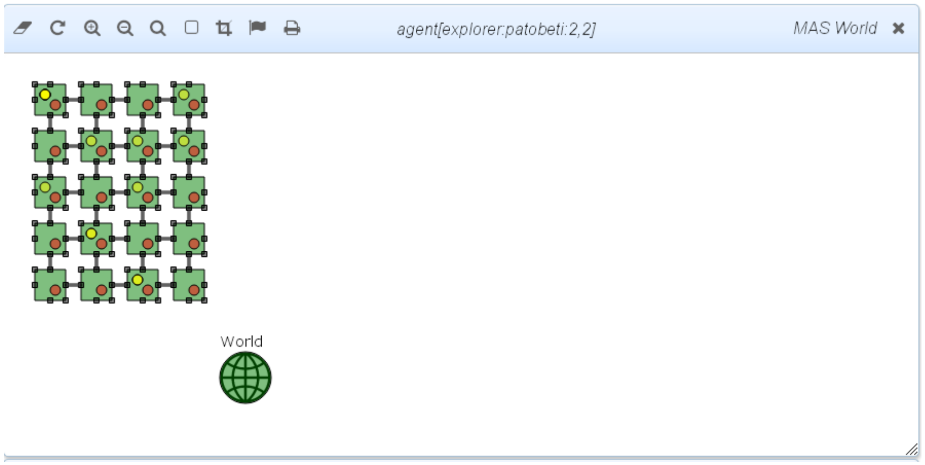
Main Menu and Control Bar

The main control bar provides buttons for the following operations: *Open* *(a simulation model); Reload (a model); Save (a model); Wizard (for a new model, not implemented yet); Code Editor (for simulation model and included files); Show/Hide Inspector; Show/Hide Information Window; Reports (not implemented yet); Show/Hide MAS World; Show/Hide Simulation Control; Show/Hide Plot Window; Show/Hide Configuration Menu; Auto Layout; Lock/Unlock position of sub-windows; Show/Hide Physical Simulation World*.



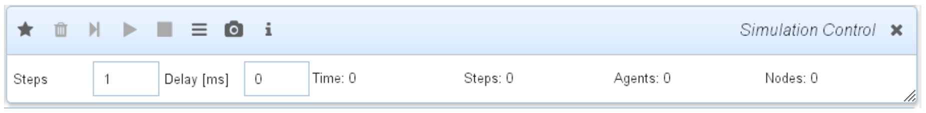
MAS World

The visualization of the MAS and JAM world. Nodes, links, and agents can be selected to display information in the inspector window about the selected object. The window provides the following buttons: *Erase (visualization); Redraw; Zoom in; Zoom out; Zoom selection; Zoom fit in window; Select multiple objects in a region for that information is shown in the inspector; Show/Hide object flags (names); Print world*.



Simulation Control

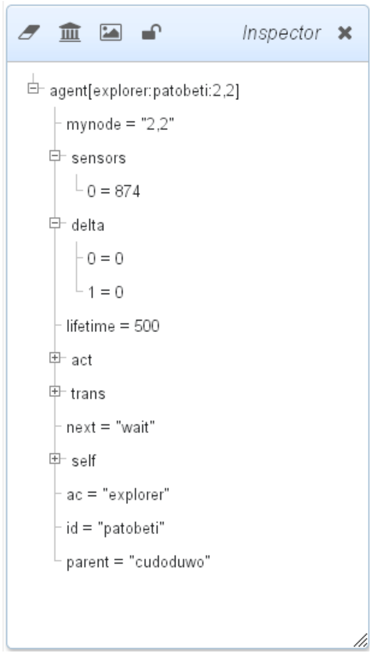
The simulation can be controlled and monitored by using the control buttons of the simulation control window. The second row of the window provides basic simulation settings and monitors: The number of simulation steps executed in step mode, the delay between two simulation steps, the current simulation time (in step or millisecond units), the current simulation step, the total number of agents existing in the simulation world, and the number of nodes. The button bar provides the following operations: *Creation and destruction of the simulation world; Start step mode; Start run mode; Stop simulation; Configuration setup; Enable/disable Recording; Create and show a report*.



Steps 1 Delay [ms] 0 Time: 0 Steps: 0 Agents: 0 Nodes: 0

Inspector

The inspector can be used to explore different objects: Agents; Nodes; Links. Objects selected in the simulation world are displayed automatically. The following buttons are available: Erase inspector window; Display current simulation model; Display all current GUI objects in the simulation world; Lock/Unlock inspector.



```
agent[explorer:patobeti:2,2]
  mynode = "2,2"
  sensors
    0 = 874
  delta
    0 = 0
    1 = 0
  lifetime = 500
  act
  trans
  next = "wait"
  self
  ac = "explorer"
  id = "patobeti"
  parent = "cudoduwo"
```

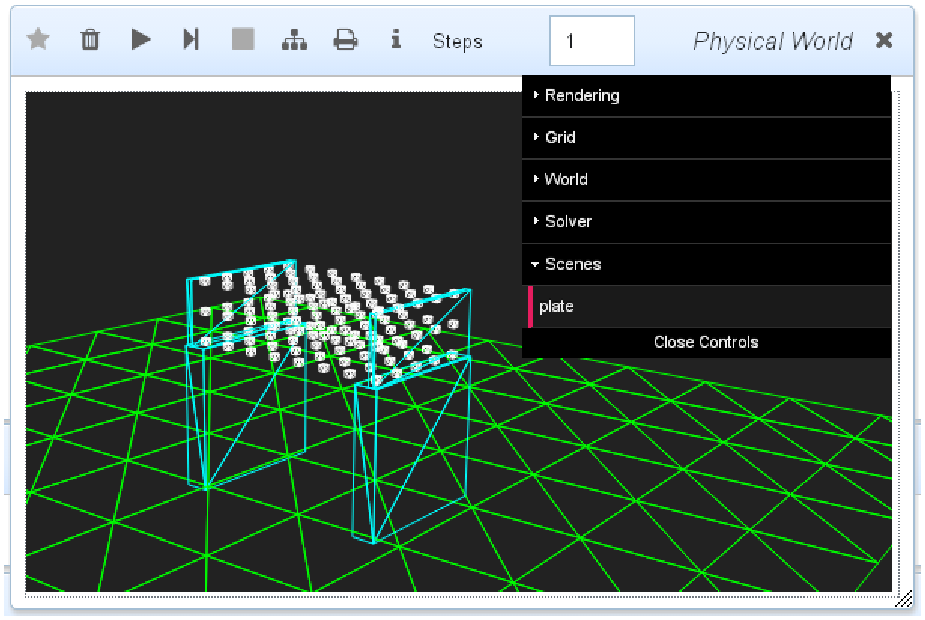
Physical Simualation World

This window provides a visualization of a physical world simulated with the multi body physics simulator CANNON and the physical simulation control bar. Usually phyiscal simulations are controlled by the MAS simulation (e.g., the world agent). A physical simulation world consists of a scene. Within the scene window there is a another sub-menu that can be used to configure the physical simulation and to select scenes.

The button bar provides the following operations: *Create world; Delete world; Start simulation; Step simulation; Stop simulation; Show physical model in inspector; Print current scene; Make a report*.

10.1. Physical Simulation

The physical world is defined in accordance to the CANNON scene modelling. There are masses and connections between masses. The following example defines a plate consisting of multiple rows, columns, and layers, which are laid down on two pillars. The configuration of the phyiscal model is defined in the main simulation model that is passed by the `settings` parameter to this scene constructor function. The `world` paramater provides access of the GUI and CANNON object.



Example Model

```
/** Defines a physical simulation scene
 *  used by the CANNON multi-body physics simulator.
 *  Must return the physical objects that can be accessed
 *  by SEJAM agents.
 *
 */
  /*
  **  X <--+ Z   External coordinates
  **       |
  **       v
  **       Y
  **
  **
  **   x <--+ z  Internal coordinates
  **          |
  **          v
  **          y
  */

function (world,settings) {
  var CANNON=world.CANNON,
      GUI=world.GUI,i,j,
      mass = (settings && settings.mass)?settings.mass:1,
      X=settings.model.world.meshgrid.cols,
      Y=settings.model.world.meshgrid.rows,
      Z=settings.model.world.meshgrid.levels,
      Height=20,
      damping=settings.model.parameter.damping||3,
      stiffness=settings.stiffness||settings.model.parameter.stiffness||50,
      Mass=200,
      MC=5,
      holes=settings.model.parameter.holes ||[];
      // hole=none;

  function contains (vl,v) {
    for(var i in vl) if (equal(vl[i],v)) return true;
    return false;

  }
  function matrix(n,m,k) {
    var x,y,z,mat;
    mat=new Array(n);
    for(x=0;x<n;x++) {
      mat[x]=new Array(m);
      for(y=0;y<m;y++)
        mat[x][y]=new Array(k);
    }
    return mat;
  }

  var constraints = [];
  var bodies = [];
  var springs = [];
  var masses = matrix(X,Y,Z);
  var loadings=[];

  world.gravity.set(0,0,-10);
  world.camera.position.set(150,130,70);
  world.camera.up.set(0,0,1);
  world.camera.fov=5.0;
  var groundMaterial = new CANNON.Material("groundMaterial");

  // Ground
  var groundShape = new CANNON.Plane();
  groundShape.color = 0x00ff00;
  var ground = new CANNON.Body({ mass: 0, material: groundMaterial });
  ground.addShape(groundShape);
  ground.position.z = 0;
  world.addBody(ground);
  GUI.addVisual(ground);

/*
  var fixedBody = new CANNON.Body({mass: 0,
                                   material: groundMaterial  });
  var fixedPlane = new CANNON.Plane();
  fixedPlane.color = 0x00ffff;
  fixedBody.addShape(fixedPlane);
  var rot = new CANNON.Vec3(1,0,0)
  fixedBody.quaternion.setFromAxisAngle(rot, Math.PI/2)
  fixedBody.position.set(0,0,0);
*/
  function makeWalls() {
    var h,h2;
    var fixedBody = new CANNON.Body({mass: 0,
                                     material: groundMaterial  });
    h=Height/2+2.0;
    var fixedShape = new CANNON.Box(new CANNON.Vec3(X*2.5,2,h));
    fixedShape.color = 0x00ffff;
    fixedBody.addShape(fixedShape);
    fixedBody.position.set((X-1)*2.5,0,h+0.5);
    world.addBody(fixedBody);
    GUI.addVisual(fixedBody);
    fixedBody = new CANNON.Body({mass: 0,
                                 material: groundMaterial  });
    fixedShape = new CANNON.Box(new CANNON.Vec3(X*2.5,2,h));
    fixedShape.color = 0x00ffff;
    fixedBody.addShape(fixedShape);
    fixedBody.position.set((X-1)*2.5,(Y-1)*5,h+0.5);
    world.addBody(fixedBody);
    GUI.addVisual(fixedBody);
    h2=(Z-1)*5+1;
    fixedBody = new CANNON.Body({mass: 0,
                                 material: groundMaterial  });
    fixedShape = new CANNON.Box(new CANNON.Vec3(X*2.5,0.5,h2/2));
    fixedShape.color = 0x00ffff;
    fixedBody.addShape(fixedShape);
    fixedBody.position.set((X-1)*2.5,-1,2*h+h2/2+0.5);
    world.addBody(fixedBody);
    GUI.addVisual(fixedBody);
    fixedBody = new CANNON.Body({mass: 0,
                                 material: groundMaterial  });
    fixedShape = new CANNON.Box(new CANNON.Vec3(X*2.5,0.5,h2/2));
    fixedShape.color = 0x00ffff;
    fixedBody.addShape(fixedShape);
    fixedBody.position.set((X-1)*2.5,(Y-1)*5+1,2*h+h2/2+0.5);
    world.addBody(fixedBody);
    GUI.addVisual(fixedBody);
  }

  function makeLoad(x,y,r,m) {
    var h=2;
    if (!r) r=10;
    var bShape = new CANNON.Cylinder(r,r,h,16);
    bShape.color='red';
    var b = new CANNON.Body({ mass: m||Mass });
    b.addShape(bShape);
    b.position.set(x,y,Height+4.0+h/2+(Z-1)*5+1.0+0.5);
    bodies.push(b);
    loadings.push(b);
  }

  function makeBox(x,y,z) {
    var bShape = new CANNON.Box(new CANNON.Vec3(0.5,0.5,0.5));
    var b = new CANNON.Body({ mass: mass });
    // bShape.grid=3;
    b.addShape(bShape);
    b.position.set(x,y,z+Height);
    bodies.push(b);
    return b;
  }

  function connect(bodyA,bodyB,settings) {
    var sAB, localPivotA,localPivotB,constraint,
        dir=new CANNON.Vec3();
    sAB = new CANNON.Spring(bodyA, bodyB, {
              stiffness:stiffness+(MC-2*MC*Math.random()),
              damping:damping,
              computeRestLength:true
          });
      // world.log(sAB.restLength);
    springs.push(sAB /*,sBA*/);
    world.addSpring(sAB);
    if (!bodyA.springs) bodyA.springs={};
    if (!bodyB.springs) bodyB.springs={};
    bodyB.gridPosition.vsub(bodyA.gridPosition,dir);
    bodyA.springs[dir.x+','+dir.y+','+dir.z]=sAB;
    dir=dir.negate();
    // bodyB.springs[dir.x+','+dir.y+','+dir.z]=sAB;
    return sAB;
  }

  function makePlate(l,n,m,d) {
    var dx=5,dy=5,dz=5,b,i,j,k,u,
        x=0,y=0,z=dz*m,offInd=0,bA,bB;
    function get(i,j,k,d) {
      if (d) i+=d[0], j+=d[1], k+=d[2];
      if (masses[i] && masses[i][j] && masses[i][j][k]) return masses[i][j][k];
      else return none;
    }
    for(k=0;k<l;k++) {
      z=dz*m;
      for(j=0;j<m;j++) {
        y=0;
        for(i=0;i<n;i++) {
          if (!contains(holes,[k,i,j])) {
            b=makeBox(x,y,z);
            masses[k][i][j]=b;
            b.gridPosition=new CANNON.Vec3(k,i,j);
          }
          y=y+dy;
        }
        z=z-dz;
      }
      x=x+dx;
    }
    for(k=0;k<m;k++) {
      for(j=0;j<n;j++) {
        for(i=0;i<l;i++) {
          var vec = [

            [0,1,0],
            [1,1,0],
            [1,0,0],
            [1,-1,0],

            [0,0,1],
            [0,-1,1],
            [-1,-1,1],
            [-1,1,1],
            [0,1,1],
            [1,1,1],
            [1,0,1],
            [1,-1,1]

          ];
          for(u in vec) {
            bA=get(i,j,k);
            bB=get(i,j,k,vec[u]);
            if (bA && bB) connect(bA,bB);
          }
        }
      }
    }
  }
  makePlate(X,Y,Z);
  makeWalls();
  // makeLoad(10,15,2,5);

  for(i=0; i<constraints.length; i++)
      world.addConstraint(constraints[i]);

  for(i=0; i<bodies.length; i++){
      world.addBody(bodies[i]);
      GUI.addVisual(bodies[i]);
  }

  world.addEventListener("postStep",function(event){
    for(var i in springs) {
      springs[i].applyForce();
    }
  });

  // A reporter returning a table
  // First row must be the header of the table
  function report() {
    var
      i,j,k,
      max=0,min=100000,
      tbl1=[
        ['Node','Spring','Force','Displacement']
      ],
      tbl2=[
        ['X','Value']
      ];
    for(k=0;k<X;k++) {
      for(j=0;j<Z;j++) {
        for(i=0;i<Y;i++) {
          if (!contains(holes,[k,i,j])) {
            var node = masses[k][i][j];
            node.springs.forEach(function (s,sp) {
              tbl1.push([[k,i,j].join(','),sp,
                         Math.abs(s.force),s.length-s.restLength]);
              max=Math.max(max,Math.abs(s.force));
              min=Math.min(min,Math.abs(s.force));
            });
          }
        }
      }
    }

    tbl2.push(['Force Min,',min]);
    tbl2.push(['Force Max.',max]);

    return {
      Springs:tbl1,
      Global:tbl2,
    };
  }

  return {
    masses:masses,
    loadings:loadings,
    map: function (id) {
      // Map logical node [i,j,k] to respective mass body
      try { return masses[id[0]][id[1]][id[2]] } catch (e) {};
    },
    report: report
  }
}
```

10.2. Table Reports

Same as with the MAS simulation a report function can defined returning tables that can be displayed on clicking on the information button. A table is an array of rows, where the first row is the header (i.e., `string []`). Multiple tables can be reported. Each table is stored in the record returned by the report function.

```
function report() {
  return {
    tbl1: [['Head1','Head2',..],[Col1,Col2,..],[..]],
    tbl2: [['Head1','Head2',..],[Col1,Col2,..],[..]],
    ..
  }
}
```