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

Dr. Stefan Bosse
28.1.2018

Contents

1. Reactive Agents and the ATG Model 1
2. AgentJS 3
3. Agent Input-Output System (AIOS) 4
   3.1. Computational Functions 5
   3.2. Tuple Space Operations 8
   3.3. Signals and Signal Handler 10
   3.4. Agent Control 11
   3.5. Mobility 13
   3.6. Scheduling Blocks 14
4. Using JAM 15
   4.1. JAM Library 15
   4.2. Creating a simple JAM Instance 15
   4.3. Adding and Importing Agent Class Templates 16
   4.4. Creating Agents programmatically 17
   4.5. Connecting JAM nodes 17

1. Reactive Agents and the ATG Model

The behaviour \( \Phi \) 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

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perception</td>
<td>Get input data from the environment. The environmental data consists of data from other agents and sensors</td>
</tr>
<tr>
<td>Processing</td>
<td>The new input data is processed</td>
</tr>
<tr>
<td>Reasoning</td>
<td>Interpretation of the input data related to the current state of the agent</td>
</tr>
<tr>
<td>Action</td>
<td>Modify the environment (platform &amp; agents) using computed output data.</td>
</tr>
<tr>
<td>Decision</td>
<td>Decide what to do next</td>
</tr>
</tbody>
</table>

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.

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 sub-space. 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.

(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.

2. AgentJS

JavaScript is 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. Agent Input-Output System (AIOS)

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

3.1. 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**

```javascript
function(number) \rightarrow number
```

Returns absolute value of number.

**add**

```javascript
function(a:number|array|object, b:number|array|object)
\rightarrow 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,* ) → boolean
  Checks existence of an element 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.

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;
  });
```
```javascript
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};
```

### 3.2. 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.

#### Examples for tuple access

```javascript
out(['SENSORA',100, true]);
inp(['SENSORA', _, _], function (tuple) {
if (tuple) this.s =tuple[1];
});
rm(['SENSORA', _, _], true);
try_rd(0, function (tuple) { .. });
alt([['SENSORA', _, _], ['SENSORB', _], ['EVENT'], ], function (tuple) {
if (tuple && tuple[0]=='EVENT') { .. }
else ..
});
```

alt

```javascript
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 con-
```
sumed and passed to the callback function. If there are multiple tuples matching a specific pattern and the flag is set then all matching tuples are consumed and returned.

**collect**

```plaintext
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**

```plaintext
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).

**exists**

```plaintext
function (pattern) → boolean
```

Check if a tuple matches the given patterns.

**inp**

```plaintext
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 then 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.

**out**

```plaintext
function (tuple)
```

Store a tuple in the data base.

**mark**

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

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

**rd**

```plaintext
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**

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

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

**store**

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

Stores a tuple in a remote TS specified by path `to` (a node destination).

**try_alt, try_inp, try_rd**

```plaintext
function (tmo:number,..)
```

Try operation to perform an alternation, input, or read operation with a given timeout.
3.3. 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.

Types and Templates

```plaintext
type aid = string

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

```plaintext
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());
    }
}
```

10
Installation of Signal Handler

```javascript
this.on: {
  'PARENT': function (arg) {
    log('Got signal from my parent ' + arg);
  }, ..
}

send1,2,3
  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.

broadcast1,2,3
  function (class:string, range, @sig, @arg?)
  Broadcasts a signal to multiple agents of class @class with the specified range.

sendto1,2,3
  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).

3.4. 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 function (ac:string,[argval1,argval2,...],level?:number) → aid
create function (ac:string,{arg1:*,arg2:*,...},level?:number) → 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!
```

```
fork function ({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,... are overridden on forking with the given values.
```

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:..
}

3.5. 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, 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 ,
```javascript
NW, NE, SW, SE,
PATH {tag='DIR.PATH', path:string},
IP {tag='DIR.IP', ip:string},
CAP {cap:string}
}

: dir

moveto
  function (to:dir)
  Migrates calling agent to new node specified by the destination @to.

opposite
  function (dir) \(&rarr;\ dir)
  Returns the opposite (back) direction (if any) of the given direction. E.g.,
  opposite of NORTH is SOUTH.

link
  function (dir) \rightarrow boolean[string][]
  Test a link direction. Should be used prior to migration (migration with not
  available link direction causes an exception).

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);
  }
}

3.6. 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).
function(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.

4. Using JAM

4.1. JAM Library

JAM is provided as a library that can be embedded in any host application pro-
gram 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.

4.2. 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.

Example

```javascript
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 \texttt{jamp} utility capable to send agent constructor functions (class templates) and to send agent processes ready to start.

An example is shown below. The \texttt{helloworld.js} file contains the constructor function \texttt{function(options){}} definition for the agent class \texttt{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.

\begin{verbatim}
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
\end{verbatim}

4.3. Adding and Importing Agent Class Templates

An agent class template can be imported (analyzed and compiled) from a file by using the JamLib \texttt{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 \texttt{module.exports={ac1:function,ac2:function,..}}. An embedded agent class constructor function can be added by using the \texttt{compileClass(classname:string, function, verbose:number)} method.

\begin{verbatim}
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);
\end{verbatim}
4.4. 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.

```javascript
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);
```

4.5. 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

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

Node 2

```javascript
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');