Chapter 11

Simulation

Simulation of the AAPL Agent Behaviour Model, the Agent Processing Platforms, and the Simulation of Sensor Networks

epubli, ISBN 9783746752228 (2018)

This chapter addresses various simulation approaches to study the operational behaviour of MAS and the agent process platform themselves, distinguishing between behavioural and platform simulation.

11.1 The SeSAm Agent Simulator

Agent-based simulation involves different tasks [KLU09]: The Design of a simulation model, the implementation of a computer simulation model, the observing and controlling of the simulation, the observing and immersive testing, calibration and experimentation, and finally the output interpretation.

The *SeSAm* (Shell for Simulated Agent Systems, details can be found in [KLU09]) offers a GUI-based modelling, simulation, and visualization environment that was developed to address most of the previously mentioned tasks. A *SeSAm* simulation model describes the elements of a multi-agent model: The structure and dynamics of agents and their environment, the configuration of situations, instrumentation, and the experimental setup including the visualization.

For a runnable simulation the following parts must be addressed and the steps must be performed [KLU09]:

1. Modelling of the data set with built-in and user defined primitive and data types;
2. Declaration of Agents, Resources, and the world environment providing the structure and the dynamics of the entire system to be simulated;
3. Configuration and Communication (Protocol) Declaration;
4. Integration to the Simulation Run Configuration;
5. Experiment and Interface Declaration.

In *SeSAm*, there are agents capable of moving in a two-dimensional world. A geometric object can be assigned to an agent, visualizing the current position of the agent in the world. The world is treated as a non-mobile agent, too. Finally, there are resources that are passive. The structure diagram of the *SeSAm* simulation model is shown in Figure *11.1*.

An agent consists of body variables, a geometric shape (optional), and a reasoning engine describing the agent behaviour. The reasoning engine is modelled with an activity-transition graph, similar to the *AAPL* model. An activity executes actions that can have different effects (the outcome):

- Modification of the body variables of the agent or the public visible body variables of other agents, resources, and the world;

epubli, ISBN 9783746752228 (2018)

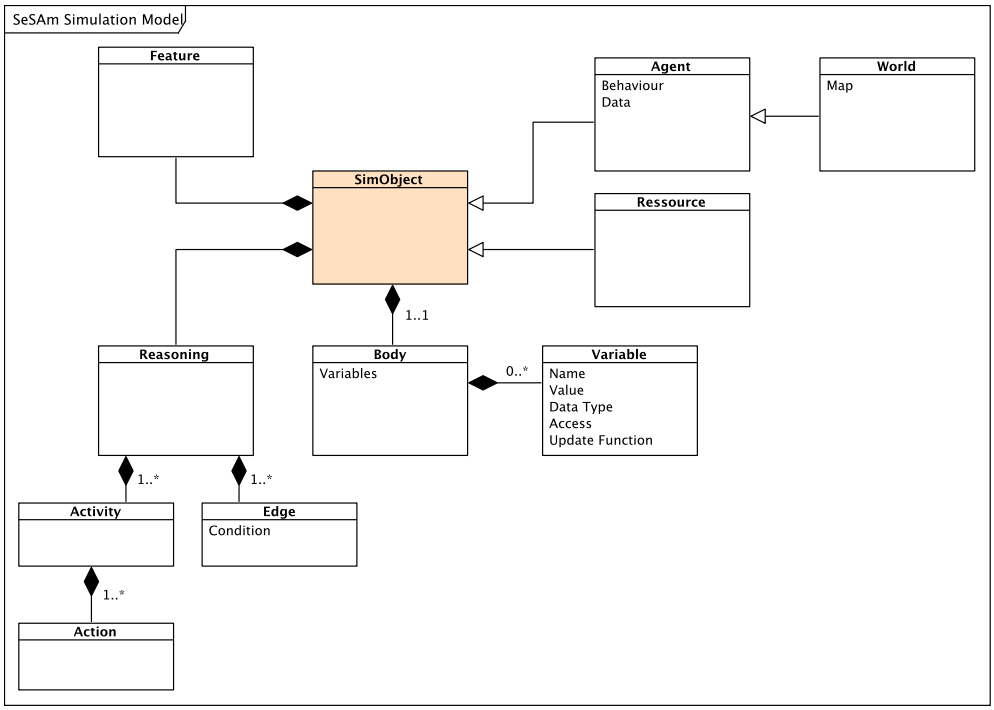

Fig. 11.1 *SeSAm Simulation Model Diagram*

- Change of the spatial location or the geometric shape of an agent;
- Creation of new or the destruction of existing agents or resources;
- Social contact to other agents and perception by reading public visible body variables of other agents.

In contrast to the *AAPL* behaviour model, there is only one control thread, preventing the implementation of signal handlers directly. Furthermore, *SeSAm* agents interaction bases on the shared memory paradigm. And finally, the ATG cannot be modified at simulation time (no activities and no transitions can be added, modified, or removed).

The *SeSAm* run-time simulator schedules all agent updates (executing activities) randomly. An agent update cycles starts with the world update first. This is the only inherent synchronisation constraint in the simulation run and simulation environment seen by the agents.

11.2 Behavioural AAPL MAS Simulation

The behavioural simulation of the agents based on the *AAPL* model using the *SeSAm* agent behaviour model, which represents only a partial subset of the *AAPL* model, and hence requires the application of some transformation rules:

- ***a.*** Each *AAPL* agent class AC_i is implemented with a *SeSAm* agent class S_i;
- ***b.*** Each *AAPL* subclass $AC_{i,j}$ is implemented with a *SeSAm* agent class S_j. At run-time different agents exist derived from each subclass;
- ***c.*** Functions and procedures of an *AAPL* agent class AC_i must be implemented with *SeSAm* feature class F_i;
- ***d.*** *AAPL* signal handlers required for the parent-child agent group communication must be implemented with a separated *SeSAm* agent class $S_{i,sig}$. At simulation time each *AAPL* agent having signal handlers is associated with a shadow agent of the class $S_{i,sig}$;
- ***e.*** Signals ξ are passed by synchronized queues;
- ***f.*** *AAPL* activities a_i that contain blocking statements (tuple space access and waiting for time-outs) require a split into a set of computational and blocking *SeSAm* activities $a_i \Rightarrow \{a'_{i,1}, a'_{i,2}, a'_{i,3}...\}$;
- ***g.*** Migration of agents is only virtual by changing the position of a *SeSAm* agent and connecting the agent to the new node agent infrastructure. Migration of agents requires the migration of the shadow agents (signal handler agents), too.

In addition to real hardware and software implemented agent processing platforms there is the capability of the simulation of the agent behaviour, mobility, and interaction on a functional level. The *SeSAm* simulation framework offers the necessary platform for the modelling, simulation, and visualization of mobile multi-agent systems deployed in a two-dimensional world. The behaviours of agents are modelled with activity graphs (specifying the agent reasoning machine) close to the *AAPL* model. Activity transitions depend on the evaluation of conditional expressions using agent variables. Agent variables can have a private or global (shared) scope. Basically, *SeSAm* agent interaction is performed by modification and access of shared variables and resources (static agents). In addition to the agent reasoning specification there are global visible feature packages that define variables and function operating on these variables. Features can be added to each agent class. Agents can change their position in the two-dimensional world map enabling mobility, and new agents can be created at run-time by other agents. The *SeSAm* framework was chosen due to the activity-based agent behaviour and

epubli, ISBN 9783746752228 (2018)

the data model, which can be immediately synthesized from the common *AAPL* source and can be imported by the simulator from a text based file stored in *XML* format. This model exchange feature allows the tight coupling of the simulator to the synthesis framework.

In principle, *AAPL* activity graphs can be directly mapped on the *SeSAm* agent reasoning model. But there are limitations that inhibit the direct mapping. First of all, *AAPL* activities (IO/event-based) can block (suspend) the agent processing until an event occurs. Blocking agent behaviour is not provided directly by *SeSAm*. Second, the transition network can change during run-time. Finally, the handling of concurrent asynchronous signals used in *AAPL* for inter-agent communication cannot be established with the generic activity processing in *SeSAm* (the provided exception handling is only used for exceptional termination of agents).

For this reason, the agent activity transitions including the dynamic transition network capability are managed by a special transition scheduler, shown in Figure *11.2*. This transition scheduler handles signals and timers, too, which are processed prioritized and passed to the signal scheduler. Each agent activity is activated by the transition scheduler. After a specific activity was processed, the transition scheduler is activated and entered again. An *AAPL* activity can be split in computational and IO/event-based sub-activities in the presence of blocking statements (e.g. *in* and *rd* tuple space interaction).

There is a special node agent implementing the tuple database with lists (partitioned to different spaces for each dimension), and managing agents and signals actually bound to this particular node.

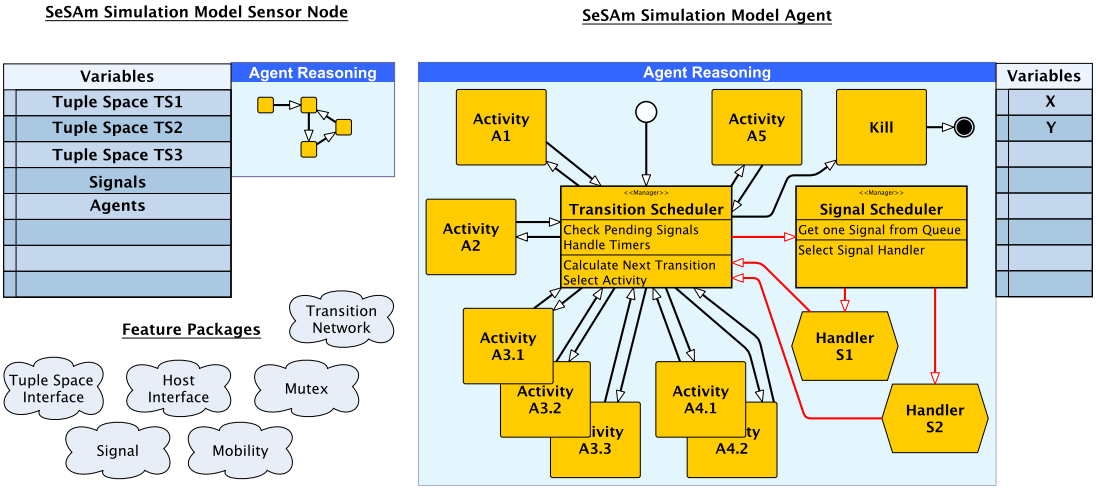

Fig. 11.2 *AAPL behavioural simulation model mapped on the SeSAm MAS model*

Concurrent manipulation of lists is non-atomic operations in *SeSAm*, and hence requires mutual exclusion.

The *AAPL* mobility, interaction, configuration, and replication statements are implemented by feature packages.

11.3 Simulation of Real-world Sensor Networks

The simulation of the operation of entire sensor networks deploying MAS commonly requires real data from the environmental world, which does not exist. To overcome this limitation, the *SeSAm* agent simulator was embedded in a database centric unified simulation environment, connecting the MAS simulator with FEM and numerical computation programs (e.g., *MATLAB*), shown in Figure *11.3*. This approach introduces multi-domain and multi-scale simulation capabilities.

The central part of the simulation framework is an SQL database server enabling the data exchange and synchronization between different programs, mainly the Multi-Agent Simulator *SeSAm* and the *MATLAB* program used, for example, for inverse numerical computations.

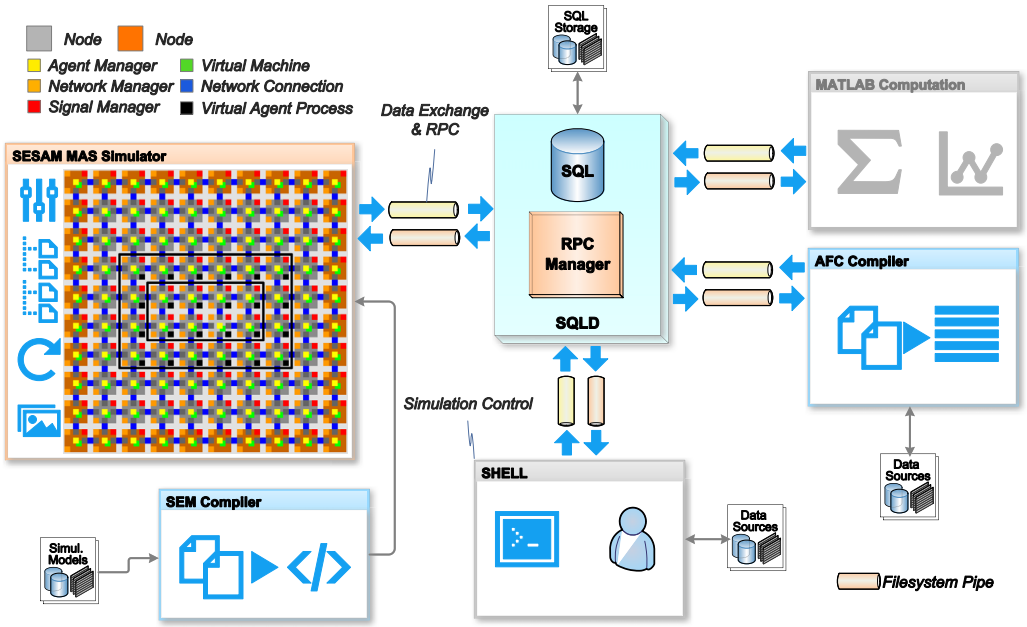

Fig. 11.3 *Simulation Framework with a database approach: Multi-Agent Simulator SeSAm, MATLAB, and other utility programs are exchanging data and synchronizing using an SQL database server, which provides an RPC interface for synchronization, too.*

A Remote Procedure Call (RPC) interface provides synchronization between the programs of the framework. All programs are communicating with the database server by using named file system pipes.

This approach has the advantage to require only a native file system interface for connecting a heterogeneous program environment, supported basically by all programs. No program modification and no special database or inter-process communication modules are required.

11.4 PCSP Platform Simulation

This section will demonstrate that agent-based simulation is suitable to for the simulation of the *PCSP* agent processing platform itself and large scale distributed networks, e.g., sensor networks, using the agent-based *SeSAm* simulator. Simulation and analysis of parallel and distributed systems are a challenge. Performance profiling and the detection of race conditions or deadlocks are essential in the design of such systems, where the agent processing platform is a central part. Furthermore, platform simulation allows the estimation and optimization of static resources like agent tables or queues, completed with the ability to study the temporal behaviour of the entire network including communication treated as a distributed virtual machine, e.g., identifying bottlenecks for specific task situations, hard to monitor in technical systems.

Behavioural simulation [BOS14A][BOS14B] maps agents of the MAS to be tested directly and isomorphic on agent objects of the simulation model. Platform simulation uses agents to implement architectural blocks like the agent manager or activity processes. Hence, agents of the MAS are virtually represented by the data space of the simulator, and not by the agent objects themselves.

The simulation of the processing platform with large scale networks processing large scale MAS aid to modify and refine the *PCSP* architecture, and to tune the static resource parameters like token pool and queue sizes or activity process replication to optimise timing. The platform simulation allows a fine-grained estimation of the required resources.

11.4.1 The Simulation Model

The networks to be simulated (aka. the simulated world) consist of nodes arranged in a two-dimensional mesh grid, with each node connected to his four neighbour nodes, shown in Figure *11.4* for a 10 by 10 sensor network with dedicated computational nodes at the outsides of the network. The entire platform and network system is partitioned into different non-mobile agent and resource classes (a resource is a passive agent with a data state only):

epubli, ISBN 9783746752228 (2018)

World Agent. The world agent creates all node agents and provides some network wide services. The world agent implements a reduced physical environment, e.g., by creating sensor signals or by disabling (destroying) connections between network nodes. Connections are represented by resources (passive agents providing only a geometric shape and body variables).

Node Agent. Each node is represented by a node agent, basically providing a common interface to data structures and tables required by the node managers and the activity processes. The node agent creates all the platform agents at the beginning of the simulation run.

Manager Agent. There is one "agent manager" agent for each agent class that is supported on the network node platform.

Network Manager Agents. There are two network manager agents. One input network manager agent handling incoming messages from neighbour nodes, decoding messages, creating agent or signal tokens, and finally passing the tokens to the agent or signal manager. The second output network manager agent is responsible for encoding and sending of messages carrying agent states or signals.

Activity Process Agents. For each agent class and each activity process of an agent class there is one activity processing agent performing token-based agent processing. The sub-states of an activity process are implemented by a simple sub-state selector and token loop-backing providing a sub-state *FSM*. Each activity process agent has local storage and a global visible token input queue.

Monitor Agent. There is one monitor agent per world collecting temporal resolved statistical data, finally writing the results to a *CSV* data file.

Token and Queues. The agent token queues are implemented with lists in the body variable space of each node agent. The size of the list can be monitored at run-time to detect resource underflow. Mutex-guarded operations (*inq*,*outq*) allow concurrent access to the queues by different agents (manager, activity processes,..). Tokens are record structures with additional descriptive entries like the current queue they are stored in.

Virtual Agent. For visualization and debugging there is a mobile virtual agent resource representing an agent to be processed by a specific agent node platform. The virtual agent references the data and control state of an agent.

epubli, ISBN 9783746752228 (2018)

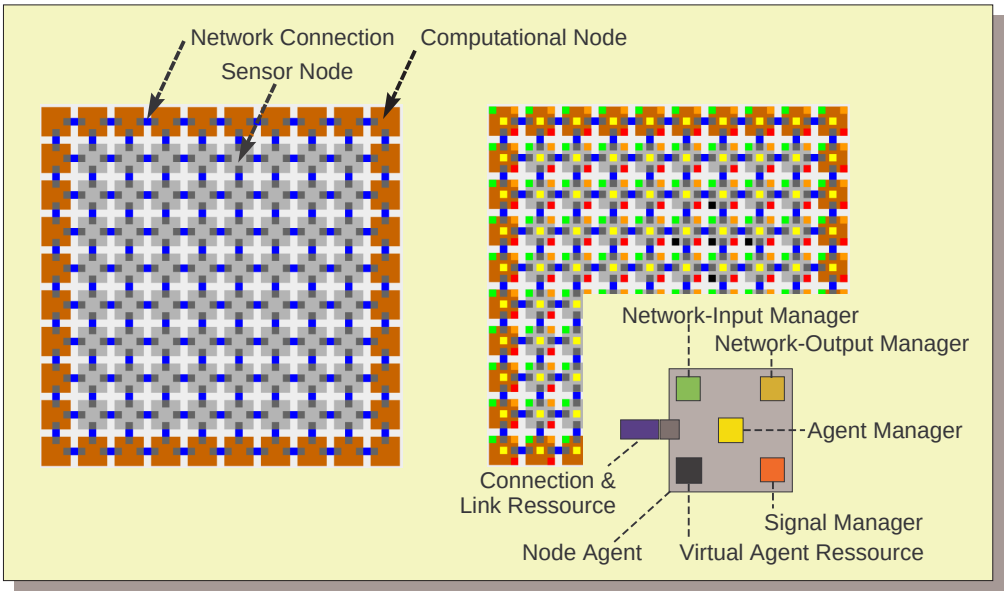

Fig. 11.4 *Simulation world of a sensor network (left) consisting of 10x10 nodes and the network populated with non-mobile platform and virtual mobile agents (right)*

11.4.2 Performance Analysis with an Use-case

The platform simulation was compared with the behavioural simulation from previous work in Table *11.1* for the simulation of a self-organizing MAS used for feature extraction in a sensor network. The behaviour model of the MAS is described in detail in [BOS14B]. It bases on a distributed divide-and-conquer approach. The number of (non-mobile) agents implementing the processing platform depends mainly on the number of activities decomposing the agent behaviour and the number of agent classes to be supported on the platform. For this example, the platform simulation model requires five times more agents and twenty times more computing time than the behavioural model. But the required resources and computing time for the fine-grained platform simulation is still reasonable and can be handled well with low end computers.

The analysis of a simulation run is shown in Figure *11.5*. It shows the temporal resolved analysis of the population of explorer agents of the MAS and the utilization of the *PCSP* network for nodes processing actually agents. There are nodes capable to process up to four agents simultaneous (speed-up 4, in different activity states and processes). The mean speed-up factor is about 1.5 for all nodes actually processing agents. Both the platform and behavioural simulation deliver the same computational results of the distributed MAS.

epubli, ISBN 9783746752228 (2018)

	Behavioural Simulation	*Platform Simulation*
Number of Agents and Resources (dynamic=mobile)	static:300 agents, 700 res.; dynamic: 130 explorer agents	static: 1600 agents, 700 res.; dynamic: 130 virtual agent resources
Simulation time including setup of simulation, with a correlated cluster scenario of 8 nodes, until MAS has finished work.	60 simulation steps in 5 s (on 1.2 GHz Intel U9300, 3GB)	280 simulation steps in 60 s (on 1.2 GHz Intel U9300, 3GB)

Tab. 11.1 *Comparison of behavioural and platform simulation of the same MAS* [BOS14B] *using the SeSAm simulator*

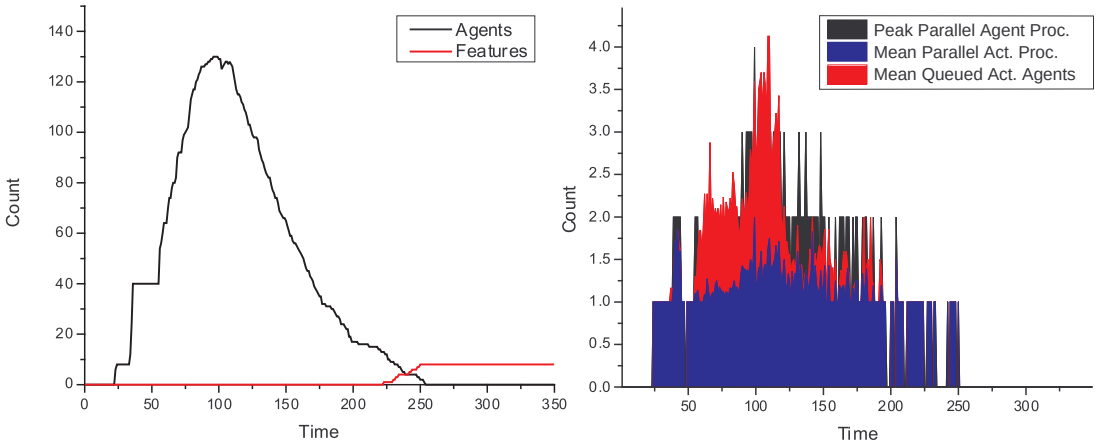

Fig. 11.5 *Analysis of the MAS simulation: (Left) Temporal development of the agent population (explorer agent) and the rise of found features in the sensor network (Right) Utilization of the platform processes (peak parallel agent processing on one node, mean parallel active processes per node, and mean agent tokens queued per active node).*

11.5 The SEM Simulation Programming Language

The activity-based procedural-functional *SEM* programming language provides statements for the description of the behaviour of state-based agents by using activities and transitions creating an activity flow and which enable activities. Activities and functions provide generic functional expressions and procedural statements to perform computation and actions. The *SEM* programming language is a text level design interface for the Multi-Agent System (MAS) simulation environment *SeSAm*, which provides only a GUI model entry level. It is closely related to the *SeSAm* agent and world model, but it is extended with some convenient functions and statements, easing the design of complex MAS.

11.5.1 SEM Classes Model

An agent belongs to a particular agent class defining body variables, activities, and transitions between activities, summarized in Tables *11.2* and *11.3*. The simulation world is treated as a non-mobile agent, too (without any geometric shape associated). Resources are passive agents without a reasoning behaviour, consisting of data and an optional geometric shape only. Feature classes are packages that are composed of user defined types, functions, and variables, which can be imported and used by agent, world, and other feature classes.

SEM Statement	*Description*

`agent ac [[(shapedef)]] =`
`  definitions:`
`  variables`
`  import`
`  activities`
`  transitions`
`end;`
	Definition of an agent class consisting of variable definitions, import of feature classes, activities, and transitions. The shape definition parameter list is optional, defining the visual appearance of an agent (geometric object).

`world wc =`
`  definitions:`
`  variables`
`  import`
`  activities`
`  transitions`
`end;`
	Definition of a world class consisting of variable definitions, import of feature classes, activities, and transitions.

Tab. 11.2 *SEM Agent, Word, Resource, and Feature Class Definitions*

SEM Statement	Description

`resource` *rc* ⟦`(`*shapedef*`)`⟧ `=`
  definitions`:`
  variables
  import
`end;`
	Definition of a resources class consisting of variable definitions and import of feature classes. The shape definition parameter list is optional.

`feature` *fc* `=`
  definitions
`end;`
`use` *fc*`;`
	Definition of a feature class including variable and function definitions.
	Within agent, world or feature classes other feature classes can be imported by using the `use` statement

Tab. 11.2 *SEM Agent, Word, Resource, and Feature Class Definitions*

SEM Statement	Description

`activity` *a* `=`
  `[`*statements*`]` `before;`
  `[`*statements*`];`
  `[`*statements*`]` `after;`
`end;`
	Definition of an agent activity representing the state of the agent. Statements of the activity are executed in sequential order before, immediately, or after an activity was activated by a transition.

`transition (`A_i`,`A_j`,`$cond_{ij}$`);` `transition (`A_i`,`A_j`);`	Definition of agent state transitions (conditionally depending on the evaluation of a boolean expression and unconditional).

Tab. 11.3 *SEM Class Statements*

11.5.2 SEM Definitions, Expressions, Values, and Types

The *SEM* language supports the definition of values (immutable or mutable) and functions in the same language style using the `def` statement. A function is considered as being a dynamic value with parameters, summarized in Table *11.4*.

epubli, ISBN 9783746752228 (2018)

SEM Statement	Description
`type e = {X,Y,Z,..};`	Definition of an enumeration (sum) type with symbolic type elements *X*, *Y*, *Z*...
`type r = ( x=T1, y=T2, .. ); ⇒ fun r: T1*T2*.. -> r;`	Definition of a record structure type consisting of elements *x*, *y*.. with specified data types *DT*. Each record type definition introduces an automatic definition of a type constructor function with the same name, which can be used in expressions to create a record type value.
`def x = expr end;` `def mutable x = expr end;` `def private mutable x = expr end;` `def x:DT = expr end;`	Definition of global read only, global mutable, and private data storage objects of data type *DT* derived from the initial value or specified with an explicit type declaration.
`def f = fun x,y,.. -> expr end;` `def f = fun x:T1, y:T2,.. -> expr end;`	Definition of a (named) function with function parameters *x*, *y*. The data type of the parameters can be specified explicitly by an additional type declaration. Type inference of function parameters is limited.
`fun F : T1*T2*.. -> RT ;`	Declaration of a function type interface (supported only for built-in primitive functions)

Tab. 11.4 *SEM Definition of types, values, and functions*

Expressions have a specific data type and are composed of values of the same data type *DT*. Expressions can appear in assignments, transitions, and conditional branch statements, summarized in Table 11.5.

The set of supported expression operators include arithmetic operators {+,-,*,/,%}, relational operators {<,>,=,<>,>=,<=}, and Boolean operators {and,or,not}.

Type	*Value*	*Description*
integer	-2, -1, 0, 1 ,2 ,3 , 4, ..	Signed integer number (decimal format).
double	-2.41,..,2.41, ..	Floating type number (decimal format).
string	"abc"	String (character text array) .
boolean	true, false	Boolean value.
α list	$\{v_1,v_2,..\}$ {}	List of values and empty list
α array	$[\|v_1,v_2,..\|]$ [||]	Array of values and empty array. There is no array type in *SeSAm*, therefore it is emulated with lists (or hash tables).
record	$T(v_1,v_2,..)$ $T(e_1{:}v_1,..)$	Record value constructor function for record type *T* with optional labels specifying the record element.
objinstance	^class	Instance of an agent class
DT -> double REAL -> INT REAL -> INT REAL -> INT REAL -> INT DT -> INT DT -> CHAR	float(x) round(x) trunc(x) floor(x) ceiling(x) int(x) char(x) (x::DT)	Type conversion (applicable to expressions, values, variables), type casting (applicable to variables only).

Tab. 11.5 *SEM constant values and types*

All operators of an expression must have the same type. Explicit type conversion can be used to convert a native data type to the expression type.

epubli, ISBN 9783746752228 (2018)

Function application is provided by using the function name and an argument list, which can be empty. Arguments containing expressions are evaluated before function application. Function applications can be embedded in expressions.

11.5.3 SEM Paths

Access of objects (variables) from other agents is performed by using paths. A path selector requires a root variable pointing to a valid simulation object. Path selectors can be used in expressions and on the LHS of an assignment. A variable *x* containing a valid simulation object reference is the root element of the path selector, which resolves a variable *vref* of an agent belonging to the specified agent or world class *class*:

```
def x:simobject = null end;
x := GetSimObjectOfClass(class,id);
x->class->vref
```

11.5.4 SEM Lists, Iterators, Arrays

Lists are dynamic data structures, which can be modified at run-time, summarized in Table *11.6*. Iterators are derived from lists and are used by iteration functions provided by *SeSAm*. Some functions expect lists directly, other require conversion to an iterator object.

There is no array support in *SeSAm*. For this reason, arrays are emulated using hash tables (or lists). The *SEM* language provides limited array support.

The access time of lists in *SeSAm* seems to be constant, that concludes the *SeSAm* handles lists internally as arrays. But an initial allocated (empty) list cannot be created, i.e., the creation of lists from an initial empty list has at least linear complexity.

Type	*Expression/Statement*	*Description*

α list	`L.[expr]`
	`L.[head]`
	`L.[tail]`
		Selection of a list element using an index expression (head and tail are keywords - the tail index is evaluated at run-time).).

α list	`L.[expr] := expr;`
	`L.[head] := expr;`
	`L.[tail] := expr;`
		Modifies a selected list element using an index expression.

α list	$L@\{v_1,v_2,..\}$	Concatenation of lists

Tab. 11.6 *SEM List and set type definitions*

Type	Expression/Statement	Description
α list	*e* :: *L*	Add an element to the top of the list *L*.
α list	*L* ::+ *e*; *L* +:: *e*;	Append new element e at the end or before the head of the list *L*.
α list	*x* ::- *L*; *x* -:: *L*;	Remove last or first element form list *L* and assign the removed element to the variable *x*.
α list α iterator	fun AsIterator: α list ->α iterator; fun AsList: α iterator ->α list;	Functions for conversion between iterators and lists
objin-stance	^class	Instance of an agent class

Tab. 11.6 *SEM List and set type definitions*

11.5.5 SEM Sequential Composition, Branches, and Loops

The programming paradigm of *SeSAm* is basically functional. Only actions in activities are executed procedural. To execute a sequential list of statements S_1; S_2; S_3;..., they must be wrapped in a block statement, and each statement must be separated by a semicolon:

```
[
  statement1;
  statement2;
  ..
  statementn
];
```

Kind	Value	Description

Boolean Branch
 	
Procedural
	`if cond then`
	$statement_1$
	`else` $statement_0$`;`
		Depending on the result of the boolean expression *cond* a branch occurs either to statment1 (expr=true) or to the optional alternative statement0 (cond=false).

Boolean Branch Functional	`if cond then` $expr_1$ `else` $expr_0$	Depending on the result of the boolean expression *cond* either $expr_1$ (*cond*=true) or the required $expr_0$ (*cond*=false) is evaluated and its value is returned

Multi-value Branch
 	
Procedural
	`case expr of`
	`|` v_1 `=>` $stmt_1$`;`
	`|` v_2 `=>` $stmt_2$`;`
	`..`
	`end;`
		Different constant values are compared with the result of the expression *expr* and the respective statements are selected on successful matching. There is no default else case (matching all other values)!

Multi-value Branch
 	
Functional
	`case expr of`
	`|` v_1 `=>` $expr_1$
	`|` v_2 `=>` $expr_2$
	`..`
		Different constant values are compared with the result of the expression *expr* and the respective expressions are evaluated on successful matching. There is no default else case (matching all other values)! A functional case branch must be complete and must contain all possible cases (so it is limited to enumeration types)

Tab. 11.7 *SEM branch statements [can be functional or procedural]*

Kind	Value	Description

Counting Loop	`for i = a to ‖ downto b do`
	`  statement`
	`done;`
	`for i in x ..`
	`for i in {v1,..} .`
		The for-loop executes the loop body statements for each element in the iterator list, either a range of values or a set/list (variable *x*) of values. The loop iterator variable *i* holds the current value taken from the list.
		The range includes the limiting values *a* and *b*.

Conditional Loop	`while expr do`
	`  statement`
	`done;`
		The while-loop executes the loop body as long as the boolean expression *expr* is true. The test of the Boolean expression is performed before each loop iteration

Tab. 11.8 *SEM loop statements*

There are different loop statements available. Each loop repeats the execution of the loop body as long as a Boolean condition is satisfied. A counting loop iterates a list of values, either specified explicitly by a set/list or implicitly by a range set constructor.

11.5.6 SEM Shapes

Agent and resources classes are related to spatially located geometric objects, which can be (initially) specified as a parameter list of a class.

Kind	Parameter	Description
Color	`color:color`	Shape colors: `black, white, grey, lightgrey, yellow, green, blue, red, orange, magenta, cyan`
Geometry	`shape:shape`	Shape geometry: `rectangle, square, circle, ellipse`

Tab. 11.9 *SEM branch statements [can be functional or procedural]*

Kind	Parameter	Description
Geometry	`size:{width, height}`	Shape size specifying the width and height of the shape (double values).
Position	`center:{x0, y0}`	Relative shape center point (double values).
Color	`fill:bool`	Shape fill attribute (true/false).

Tab. 11.9 *SEM branch statements [can be functional or procedural]*

11.5.7 SEM Example

The following example shows a fraction of a simulation model description for a simple non-mobile agent (sampling), which collect sensor data from the world and computes the mean value of the sampled data. Algorithm 11.1 shows the SEM program code for this agent, which uses functions from the feature class *env*, shown in Algorithm 11.2. If the sampling agent detects a significant mean value, it will create another *event* agent, which handles sensor events.

Alg. 11.1 *A simple agent description in SEM*

```
1   agent sampling (color:orange,shape:circle,fill:true,size:{2.0,2.0},
2                   center:{3.0,1.0}) =
3     use env;  import feature class, defines GetMatrixI,...
4
5     def mutable private adc = 0 end;
6     def mutable private mean = 0 end;
7     def mutable Pos = Position(0,0) end;
8     def mutable self:simobject = null end;
9     def mutable myworld:simobject = null end;
10    def mutable Parent:simobject = null end;
11    def mutable sampled = 0 end;
12    def mutable Thres = 100 end;
13
14    activity init =
15      [
16        mean := 0;
17        self := Self();
18        myworld := GetWorld()
19      ];
20    end;
21
22    activity sample =
```

```
23      [
24        adc := GetMatrixI(myworld->myworld->Sensors,
25                          Pos.Y-1,Pos.X-1);
26        mean := (mean + adc)/2;
27        sampled := sampled + 1
28      ];
29    end;
30
31    activity sense =
32      [
33        if mean > Thres then
34          CreateAgent(Pos.X,Pos.Y,^event)
35      ];
36    end;
37
38    activity sleep =
39      [
40        AwaitDelay(10.0)
41      ];
42    end;
43
44    transition(entry,init);
45    transition(init,sample);
46    transition(sample,sense);
47    transition(sense,sleep);
48    transition(sleep,sample,blocked=false);
49  end;
50
```

Alg. 11.2 *SEM feature class env*

```
1   feature env =
2     def private mutable _env_tempobj:simobject = null end;
3     def mutable blocked = false end;
4     ...
5
6     CREATE AGENT OF CLASS agentclass AND  RETURN SIMOBJECT
7     def CreateAgentAndReturn =
8       fun x:integer,y:integer,
9           agentclass:objinstance ->
10      CreateObjectAndRemember(agentclass,
11       (fun obj:simobject ->
12        [
13          _env_tempobj := obj;
14          BeamTo(GetSpatialInfo(obj),
15                 CreatePos((x::double)*10.0+5.0,
16                           (y::double)*10.0+5.0))
17        ];)); _env_tempobj
18    end;
19
20    def AwaitDelay = fun tmo:double ->
```

```
21      if _env_in_await then
22        [
23          if ((GetTime()-_env_start_time)>tmo) then
24          [
25            _env_in_await := false;
26            blocked := false
27          ]
28        ]
29      else
30        [
31          _env_in_await := true;
32          _env_start_time := GetTime();
33          blocked := true
34        ];
35    end;
36
37    GET MATRIX ELEMENT mat(i,j) WITH col=0,..,cols-1,row=0,..,rows-1
38    def GetMatrixI = fun mat:integer list list,
39                         row:integer,col:integer ->
40      GetNth(col,GetNth(row,mat))
41    end;
42  end;
```

11.6 SEJAM: Simulation Environment for JAM

Commonly, execution and simulation platforms are completely different environments, and simulators are significantly slower in the agent execution compared to real-world agent processing on optimized processing platforms. *SEJAM* is a MAS *AgentJS* simulator implemented on top of the *JAM* platform layer, executing agents with the same VM as a stand-alone agent platform would do. This capability leads to a high-speed simulator, only slowed down by visualization tasks and user interaction. Furthermore, multiple simulators can be connected via a stream link (sockets, IP links, etc.), improving the simulation performance by supporting parallel agent processing. Finally, the simulator can be directly connected to any other *JAM* node and can be integrated in real-world JAM networks.

The *SEJAM* simulator can be connected to SQL3 data base servers, storing sensor and monitoring data.

There are two *SEJAM* implementations: (1) A curses-based text terminal version; (2) A *node-webkit* version based on the *webix* and *graphics JavaScript* toolkits.

SEJAM1

The GUI of the *SEJAM1* simulator and the simulation world is shown in Figure *11.6*. The GUI consists of the simulation world, in this example composed of 64 logical nodes connected with virtual circuit links. Each node shape pro-

vides information about the node name in the first row, the number of agents and tuples in the database in the second row, and some flag indicators in the last row, e.g., flags signalling the existence of specific agents or sensor values.

On the right side there is a code and data navigator. Each node can be selected including the world object.

The code navigator can be used to explore node and agent information in *JSON* tree presentation. The bottom part of the simulator contains a logging and message window.

Agents can write messages to this window, and a compacted *JSON* can be printed from selected items in the object navigator tree.

Furthermore, agents executed in the simulator world inherit a special simulation object, which can be used to get specific simulation and world information, e.g., the current simulation step, or support for creation of agents on a specific node, e.g., used by the world agent, that is the only agent created and started at the beginning of the simulation. Multiple simulator worlds (*SEJAM* instances) can be connected enabling the composition of complex simulation worlds.

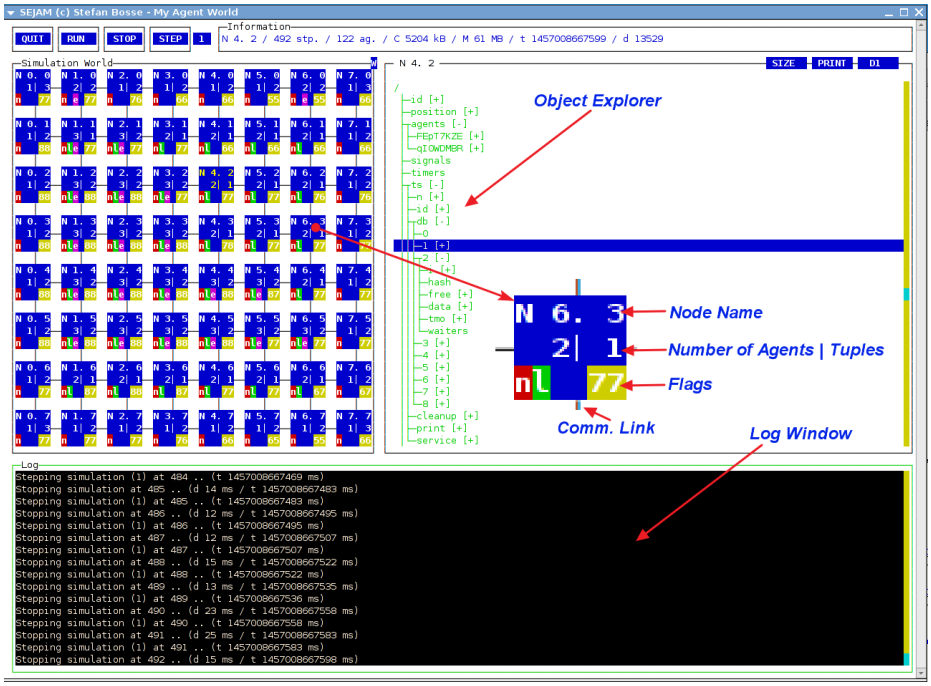

Fig. 11.6 *SEJAM1: Simulation world consisting of logical nodes populated with mobile and non-mobile agents, indicated by markings on the bottom of the node shape (blue rectangle)*

SEJAM2

The *SEJAM2* Simulation Environment provides an extended visualization world (top right window in Figure *11.7*). In the show example the world consists of three floors of a building. Shown are beacons (yellow triangles, each populated with a node agent) and some mobile devices (green rectangles).

The black circles represent communication links of the devices, indicating the communication range. On the left side the object explorer is shown, and on the bottom the logging window. There is an extended simulation control providing a parametrization of the simulation and a statistics monitor module offering graphical plotting capabilities, e.g., of agent populations.

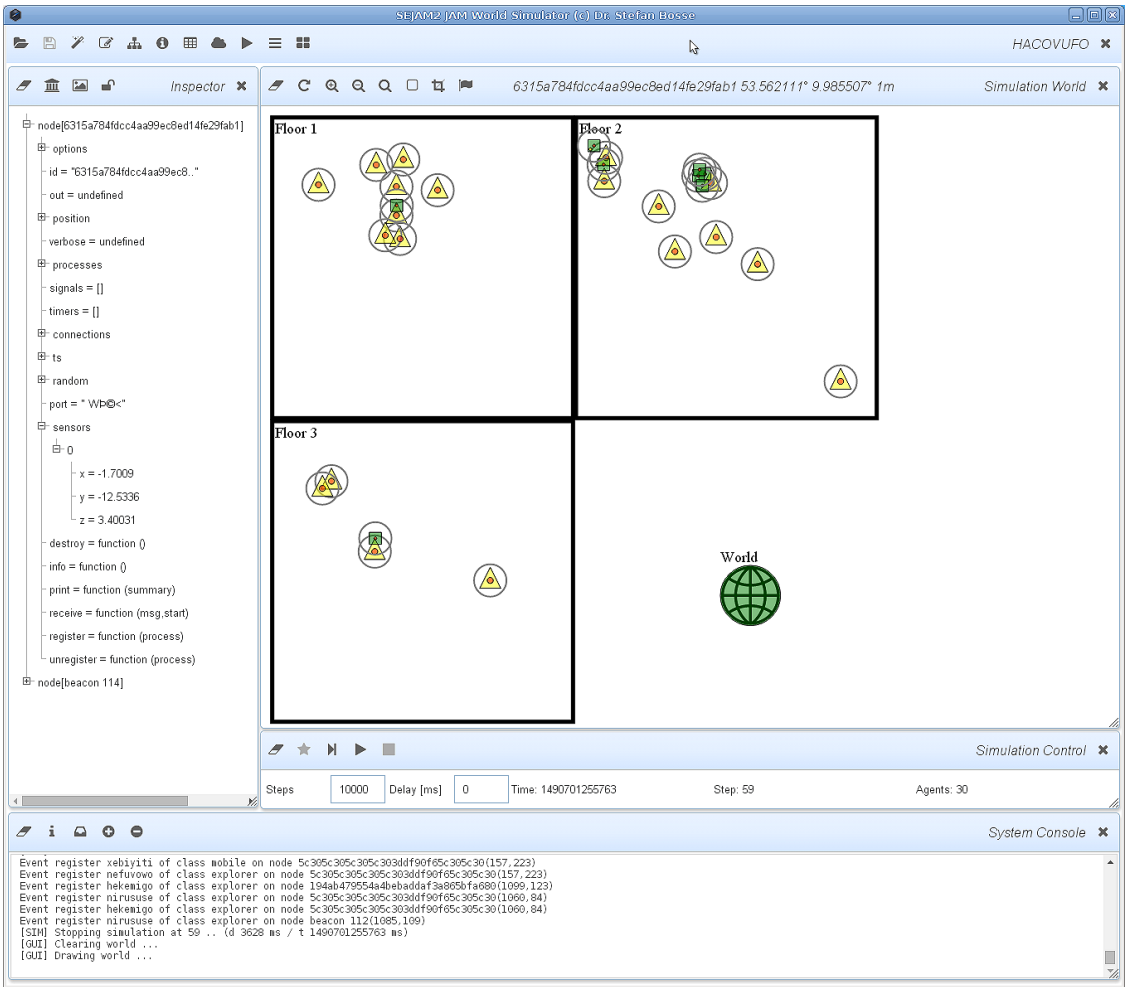

Fig. 11.7 *SEJAM2: JAM with extended visualization and simulation control*

Access to extended simulation objects and functions (e.g., access of data bases or simulation control including shape modification) is granted to agents with an extended privilege level. There is commonly a virtual world agent providing sensor data to logical nodes of the simulation world.

11.7 Multi-Domain Simulation with SEJAM2P

Agents are suitable to interact with real world environments. It is promising to use agents to control physical structures, e.g., adaptive materials. To study agent interaction with physical models, *SEJAM2* was extended with a multi-body physics simulation engine based on the cannon.js kernel and three.js for 3D visualization, shown in Figure *11.8*. The agent and physical model simulations are connected and processed simultaneously enabling agents 1. To access the physical model for getting sensor input, e.g., strain or forces provided by virtual sensors; and 2. To access the physical model to modify it, e.g., by changing the stiffness of damping of springs.

The physical world consists of the mechanical structure given by a physical model (FEM or mass-spring multi-body), sensors, and external loading having impact on the static and dynamic reaction of the structure, including gravity. For the sake of simplicity, the mechanical structure under test is modelled with a mass-spring multi-body system. The multi-body physics simulation is well-known from computer games and animations. It can be easily and efficiently integrated in computational systems like MAS simulators.

Spring and gravity forces have an effect on each mass of the structure. Therefore, the structure will swing until it converges to a static state.

A simulation model consists basically of three parts: (1) The MAS behaviour models; (2) The *JAM* virtual network world; (3) The physical model. All parts are specified in *JavaScript*. In this environment, the physical model can be accessed by all agents.

Typically, the coupling of computational with physical systems by MAS and the effect of MAS actions on the physical system should be investigated. In Figure *11.8*, a MAS is deployed in a three-dimensional mesh-grid network integrated in an artificial mechanical structure. For the sake of simplicity this structure is a plate composed of 5x4x2 mass nodes connected by springs. It is assumed that each network node provides a *JAM* platform to process agents. Each node is connected to up to six neighbours with communication links. Additionally, each node is connected to neighbour nodes by a set of springs, but only a sub-set is controlled by a specific computer and mass node.

In Example *11.1* a typical simulation model is shown. The simulation model is identical to a *SEJAM2* model extended with a physical model section. The model defines two agent classes *node* and *world* and their visualization with shapes in the *classes* section (line 3). Each agent class descriptor consists of

epubli, ISBN 9783746752228 (2018)

the behaviour, visual, and optional event handler parts. All section keywords are highlighted.

The *physics* section (line 29) assigns the physical model to a named scene.

There is an optional parameter section (line 34) defining simulation parameters that can be accessed by the model (agents) and modified at run-time with the GUI.

The final *world* section (line 43) defines the construction of the MAS simulation world. In this simulation, a predefined mesh-grid world is used (line 70), composed of JAM nodes and channel links.

The agent behaviour of the node agent is shown in Example *11.2*. It consists of function constructing the agent object from this template.

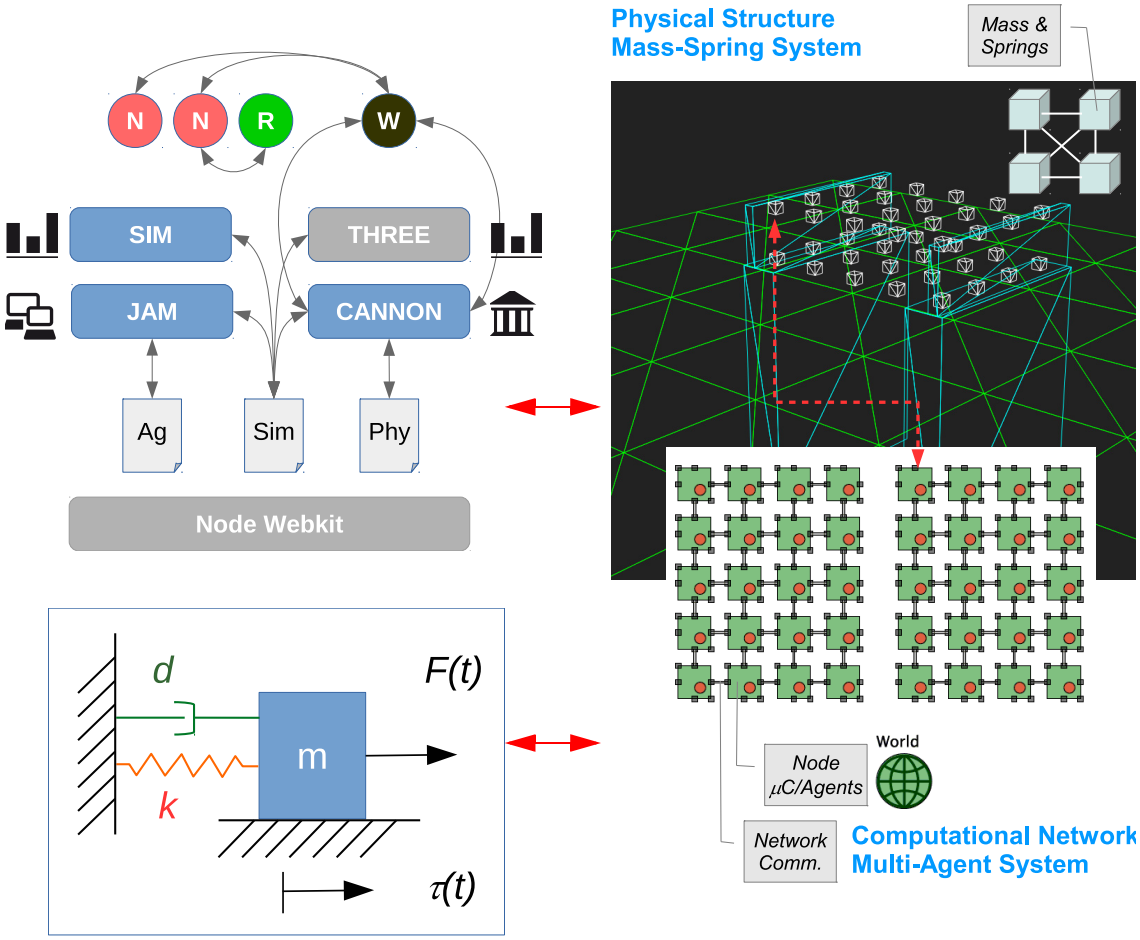

Fig. 11.8 *(Top, Left) Multi-domain simulation coupling a physics and MAS simulator. (Right) Relation of Physical and Logical Models mapping computational on physical mass nodes (Bottom, Left) Physical Mass-Spring Model*

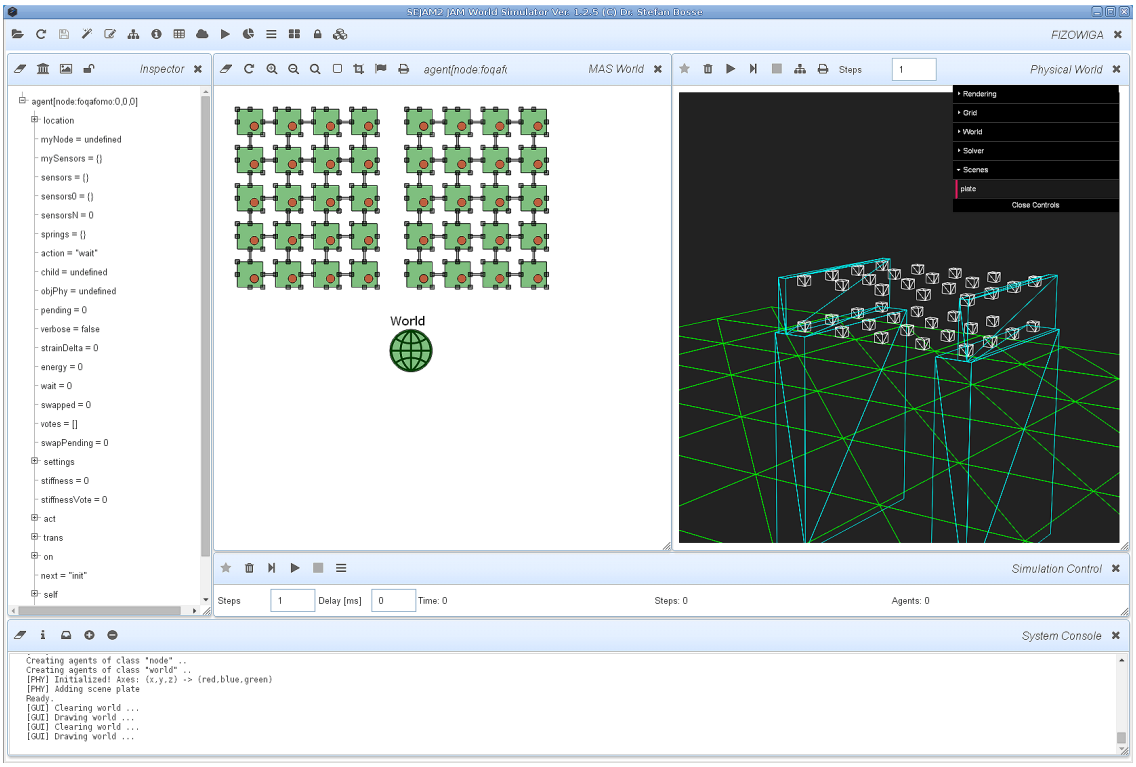

Fig. 11.9 *SEJAM2P: Combined Multi-body Physics and MAS simulator*

The template consists of private body variables (lines 2 to 44), auxiliary functions (lines 47 to 52) used by activity and transition functions, and finally the activity (line 55), transition (line 307, and event handler (line 319) sections.

The last statement defines the initial activity (*next*). In lines 171 to 176 the simulation parameters are accessed updating internal agent parameters.

Finally, the physical model is shown in Example *11.3*, constructing a plate of mass nodes and springs arranged on two fixed walls (see Figure *11.8*, right side). Mostly the *CANNON* API is used to define the physical simulation model, extended with some MAS API functions supporting agent access of this model.

A physical mass node consists of a *CANNON* box defining the geometric shape and a body defining the physical properties.

Ex. 11.1 *Example of a typical SEJAM2P top-level* ***simulation*** *model*

```
1  {
2    name:'My Simulation World',
3    classes : {
4      node: {
5        behaviour:open('node.js'),
```

```
      visual:{
          shape:'circle',
          width:10, height:10, x:5, y:5,
          fill: {color:'red', opacity: 0.5}
      },
      on:{
        'TS.SIG':{
          shape:'circle',
          width:6, height:6, x:5, y:5,
          fill: {color:'black', opacity: 0.5},
          time:10
        },
      }
    },
    world: {
      behaviour:open('world.js'),
      visual:{
          shape:'circle',
          width:50, height:50, center:{x:0,y:0},
          fill: {color:'green', opacity: 0.0}
        }
    }
  },
  physics:{
    scenes:{
      plate:open('plate.js')
    }
  },
  parameter:{
    adapt:'no',
    lowPassK:1.0,
    mark:'strain',
    scale:1/10,
    update:'dynamic',  // 'static' 'dynamic'
    deltaPhy:5,        // termination threshold with update=static
    stepPhy:100
  },
  world : {
    init: {
      agents: {
        node:function (nodeId,position) {
          // Create on each node a node agent, return respective
          // agent parameters! If no agent should be created on the
          // respective node, undefined must be returned!
          if (nodeId!='world')
            return {level:1,args:[
              {x:position.x,y:position.y,z:position.z,
               adapt:'local', // false, // 'local', 'global'
               mark:'strain',
               negotiate:false,
               eps:0.02,
```

```
57                verbose:0,
58                //scale:0.03
59                scale:0.5
60               }]}
61          },
62          world: function(nodeId) {
63            if (nodeId=='world') return {level:3,args:[{model:model}]};
64          }
65        },
66        physics: function (phy) {
67          phy.changeScene('plate',{stiffness:70});
68        }
69      },
70      meshgrid : {
71        // y-axis
72        rows:5,
73        // x-axis
74        cols:4,
75        //z-axis
76        levels:2,
77        // all z-level networks arranged in a matrix
78        matrix:[[0,0],[200,0]],
79
80        node: {
81          // Node ressource visual object
82          visual : {
83            shape:'rect',
84            width:30, height:30,
85            fill: {color:'green', opacity: 0.5}
86          }
87        },
88        // Link port connectors
89        port: {
90          type:'unicast',
91          place: function (node) {return [
92            {x:-15,y:0,id:'WEST'},
93            {x:15,y:0,id:'EAST'},
94            {x:0,y:-15,id:'NORTH'},
95            {x:0,y:15,id:'SOUTH'},
96            {x:-15,y:-15,id:'UP'},
97            {x:15,y:15,id:'DOWN'}
98          ]},
99          visual: {
100           shape:'rect',
101           fill: {color:'black', opacity: 0.5}, width: 5, height: 5
102         }
103       },
104       // Connections between nodes (with virtual port connectors)
105       link : {
106         // Link resource visual object
107         connect: function (node1,node2,port1,port2) {return true},
```

epubli, ISBN 9783746752228 (2018)

```
        type:'unicast',   // unicast multicast
        visual: {
          shape:'rect',
          fill: {color:'#888', opacity: 0.5}, width: 2
        }
      }
    },
    map: function () {
        return [
          model.world.nodes.world(200,280),
        ]
    },
    nodes: {
      world: function (x,y) {
          return {
            id:'world',
            x:x, y:y,
            visual: {
              shape:'icon',icon:'world',
              label:{text:'World', fontSize:14},
              width:50, height:50,
              fill: {color:'black', opacity: 0.5}
            }
          }
      }
    }
  }
}
```

Ex. 11.2 *The node **agent** behaviour model*

```
function node(options) {
  this.location={x:options.x,y:options.y,z:options.z};
  this.myNode=none;
  this.mySensors= {};
  this.sensors={};
  this.sensors0={};
  this.sensorsN=0;
  this.springs={};
  this.action='wait';
  this.child=none;
  this.objPhy = none;
  this.pending=0;
  this.verbose=options.verbose || false;
  this.strain = undefined;
  this.strain0 = undefined;
  this.strainS = undefined;
  this.strainDelta=0;
  this.force = undefined;
  this.force0 = undefined;
```

```
20    this.forceV = undefined;
21    this.energy = 0;
22    this.wait=0;
23    this.swapped=0;
24    this.votes=[];
25    this.swapPending=0;
26
27    this.settings = {
28      adapt : options.adapt||'no',
29      eps : options.eps || 0.2,   // |strain-strain0| notify threshold
30      energyThres : 3,
31      mark: options.mark || 'force',
32      lowPassK:1.0,
33      negotiate : options.negotiate||false,
34      scale : options.scale || 10,
35      strainHigh : 5,
36      strainThres: 2.0,
37      strainLow : 0.5,
38      stiffnessHigh : 80,
39      stiffnessLow : 30,
40      swapHighEnergy:true,
41      vote:10       // |stiffness-stiffnesVote| vote threshold
42    }
43    this.stiffness = 0;
44    this.stiffnessVote = 0;
45
46    // Auxiliary functions
47    this.ofVec = function (o) {..};
48    this.toVec = function (o) {..};
49    this.equalVec = function (v1,v2) {..};
50    this.formatVec = function (v) {..};
51    this.parseVec = function (s) {..};
52    this.strainTostiffness = function (s) {..};
53    this.filter = function (v,v0,k) {..};
54
55    this.act = {
56      init: function () {
57        var n=0;
58        this.myNode=myNode();
59        this.sensors={}; this.stiffness=0;
60        this.objPhy=simu.simuPhy.get(this.myNode);
61        log('Starting at '+this.location.x+','+this.location.y+','+
62            this.location.z+' on node '+this.myNode);
63
64        negotiate('CPU',10000000);
65        negotiate('SCHED',100000);
66
67        if (this.objPhy) iter(this.objPhy.springs,function (s,sp) {
68              this.springs[sp]={stiffness:s.stiffness,alive:-1,flag:false};
69              this.stiffness += s.stiffness, n++;
70          });
```

```
71        this.stiffness /= n;
72      },
73
74      percept: function () {
75        this.action='percept';
76        alt(
77            [['SPRING',_,_],
78             ['SENSOR',_,_],
79             ['VOTE',_,_],
80             ['SWAP',_,_],
81             ['STIFFNESS',_]
82            ],function(ta) {
83          if (!ta) return;
84          iter(ta, function (t) {
85            var p,vote;
86            switch (t[0]) {
87              case 'SENSOR':
88                // Sensor data from neighbour nodes
89                p=t[1];
90                this.action='process';
91
92                if (this.sensors[p]) this.sensors0[p]=this.sensors[p];
93                this.sensors[p]=copy(t[2]);
94                if (!this.sensors[p].stiffness)
95                    this.sensors[p].stiffness=this.springs[p].stiffness;
96                if (this.sensors0[p] == undefined ||
97                    this.sensors0[p].strain != this.sensors[p].strain)
98                    this.action='process';
99                break;
100             case 'SPRING':
101             // Single spring setting
102               p=t[1];
103               this.springs[p]=copy(t[2]); this.action='adapt';
104               break;
105             case 'VOTE':
106               // Vote for node stiffness?
107               if (this.votes.length==0) timer.add(300,'ELECTION');
108               vote={to:t[1],vote:t[2]};
109               vote.vote.from=vote.vote.to;
110               vote.vote.to=undefined;
111               this.votes.push(vote);
112               break;
113             case 'SWAP':
114               // Neighbour node votes (?+-) for stiffness swapping
115               vote={to:t[1],vote:t[2]};
116               vote.vote.from=vote.to;
117               vote.vote.to=undefined;
118               switch (vote.vote.type) {
119                 case 'SWAP?':
120                   this.votes.push(vote); this.action='swap';
121                   break;
```

```
122                 case 'SWAP+':
123                   this.stiffnessVote=vote.vote.stiffness;
124                   this.action='adapt'; this.swapPending--;
125                   break;
126                 case 'SWAP-':
127                   this.swapPending--;
128                   break;
129               }
130               break;
131             case 'STIFFNESS':
132               this.stiffness=t[1];
133               if (this.objPhy) iter(this.objPhy.springs,function (s,sp) {
134                 this.springs[sp]={stiffness:this.stiffness,
135                                   alive:-1,flag:false};
136               });
137               break;
138             }
139           });
140       },true);
141     },
142
143
144     process : function () {
145       var i=0,v,p,s,_strain=0,_forceV=Vec3(0,0,0),id=this.myNode;
146       this.action='percept'; this.strainS=0;
147
148       // Update strain value
149       this.energy=0;
150       this.sensorsN=0;
151       for(p in this.sensors) {
152         v=this.parseVec(p);
153         _forceV = _forceV.vadd(Vec3(v[0],v[1],v[2])
154                         .scale(this.sensors[p].strain*
155                               this.sensors[p].stiffness));
156         s=this.sensors[p].strain;
157         _strain += s;
158         this.strainS += Math.abs(s);
159         this.energy += (s*s*this.sensors[p].stiffness);
160         this.sensorsN++;
161       }
162       if (this.sensorsN>0) this.energy /= this.sensorsN;
163       if (length(this.sensors)>3) {
164         this.strain0=this.strain;
165         this.strain=_strain;
166         this.force0=this.force;
167         this.force=_forceV.length();
168         this.forceV=this.ofVec(_forceV);
169       }
170
171       if (simu.parameter('mark'))
172         this.settings.mark=simu.parameter('mark');
```

```
173       if (simu.parameter('scale'))
174         this.settings.scale=simu.parameter('scale');
175       if (simu.parameter('adapt'))
176         this.settings.adapt=simu.parameter('adapt');
177
178       if (!this.objPhy)
179         simu.changeVisual('node['+id+']',{
180           fill:{
181             color:'white',
182             opacity: 0.5
183           }
184         });
185       else {
186         v=this[this.settings.mark]?this[this.settings.mark]*
187                                    this.settings.scale*2-1.0:0;
188         if (v>0) v=Math.min(v,1); else v=Math.max(v,-1);
189         simu.changeVisual('node['+id+']',{
190           fill:{
191             color:v<0?{color:'blue',value:-v}:{color:'red',value:v},
192             opacity: 0.5
193           }
194         });
195       };
196
197       if (this.energy > this.settings.energyThres && this.swapPending==0)
198         this.action='notify';
199
200       this.wait=100;
201
202       if (this.strain==undefined || this.strain0 == undefined ||
203           Math.abs(this.strain0-this.strain)>this.settings.eps)
204         this.action='notify',
205          this.strainDelta=(this.strain==undefined ||
206                            this.strain0==undefined)?
207                               this.settings.eps*2:
208                               Math.abs(this.strain0-this.strain);
209     },
210
211     notify: function () {
212       var neighbour,neighbours;
213       this.action='percept';
214
215       // 1. Distribute Sensors
216       if (this.strainDelta>this.settings.eps)
217       {
218         iter(this.sensors,function (s,sp) {
219           var _s,vs=this.formatVec(neg(this.parseVec(sp)));
220           if (s.flag) return;
221           _s=copy(s); _s.flag=true;
222           if (this.springs[sp]) _s.stiffness=this.springs[sp].stiffness;
223           store(DIR.DELTA(this.parseVec(sp)),['SENSOR',vs,_s]);
```

```
224         });
225         this.strainDelta=0;
226       }
227
228       // 2. Negotiate stiffness swap votes
229       if (this.settings.adapt != 'no' &&
230           this.energy > this.settings.energyThres && this.swapPending==0) {
231         // Start swap negotiation
232         // Select a neighbour randomly
233         neighbours=map(this.sensors,function (s,sp) {return sp});
234         neighbour=random(neighbours);
235         this.swapPending++;
236         if (this.child)
237           send(this.child,'VOTE',
238               {type:'SWAP?',to:neighbour,stiffness:this.stiffness,
239                energy:this.energy}
240           );
241
242       }
243     },
244
245     swap: function () {
246       var swapit=false;
247       this.action='percept';
248       this.votes=map(this.votes,function (vote) {
249         if (vote.vote.type=='SWAP?') {
250           if (swapit) {
251             if (this.child)
252               send(this.child,'VOTE',
253                   {type:'SWAP-',to:vote.to,stiffness:this.stiffness,
254                    energy:this.energy});
255             return;
256           }
257           swapit= this.stiffness>vote.vote.stiffness &&
258                   this.energy<vote.vote.energy;
259           if (this.settings.swapHighEnergy && !swapit)
260             swapit=this.stiffness<vote.vote.stiffness &&
261                    this.energy>vote.vote.energy;
262           if (this.child) {
263             send(this.child,'VOTE',
264                 {type:swapit?'SWAP+':'SWAP-',to:vote.to,
265                  stiffness:this.stiffness,energy:this.energy});
266             if (swapit) {
267               this.stiffnessVote=vote.vote.stiffness;
268               this.action='adapt';
269               this.swapped++;
270             }
271           }
272           return;
273         }
274         return vote;
```

```
275       });
276     },
277
278     adapt: function () {
279       if (simu.parameter('lowPassK'))
280         this.settings.lowPassK=simu.parameter('lowPassK');
281       switch(this.settings.adapt) {
282         case 'local':
283           this.stiffness=this.stiffnessVote;
284           iter(this.objPhy.springs,function (s,sp) {
285             if (!this.sensors[sp]) return;
286             s.stiffness = this.filter(this.stiffness,s.stiffness,
287                                       this.settings.lowPassK);
288             this.springs[sp]={stiffness:s.stiffness,alive:-1};
289           });
290           break;
291         case 'global':
292           this.stiffness=this.stiffnessVote;
293           iter(this.objPhy.springs,function (s,sp) {
294             s.stiffness = this.filter(this.stiffness,s.stiffness,
295                                       this.settings.lowPassK);
296             this.springs[sp]={stiffness:s.stiffness,alive:-1};
297           });
298           break;
299       }
300     },
301
302     wait: function () {
303       sleep(this.wait);
304     }
305   };
306
307   this.trans = {
308     init: percept,
309     percept: function () { return this.action},
310     swap: function () { return this.action},
311     notify: percept,
312     process: function () { return this.action},
313     adapt: function () { return percept},
314     elect: function () { return percept},
315     wait: function () { if (this.action != 'percept') return this.action;
316                         else return this.wait<=0?percept:wait}
317   };
318
319   this.on = {
320     'DELIVER': function (_sensors) {
321       var p;
322       for(p in _sensors) {
323         this.sensors[p]=_sensors[p];
324       }
325       this.pending--;
```

```
326     },
327     'ALIVE': function (child) {
328     },
329     'DECLARE': function (vote) {
330     },
331     'ELECTION': function () {
332       var major=0;
333       iter(this.votes,function (vote) {
334         major += vote.vote.stiffness;
335       });
336       major /= this.votes.length;
337       major = (major + this.stiffnessVote)/2;
338       this.votes=[];
339       this.stiffnessVote=major;
340       this.action='adapt';
341     }
342   }
343   this.next=init;
344 }
```

Ex. 11.3 *The **physical** simulation model constructing a plate*

```
1   /** Defines a physical simulation scene
2    *  used in teh CANNON multi-body physics simulator.
3    *  Must return the physical objects that can be accessed
4    *  by SEJAM agents.
5    *
6    */
7     /*
8     **  X <----+ Z   External coordinates
9     **         |
10    **         v
11    **         Y
12    **
13    **
14    **   x <---+ z  Internal coordinates
15    **         |
16    **         v
17    **         y
18    */
19
20  function (world,settings) {
21    var CANNON=world.CANNON,
22        GUI=world.GUI,i,j,
23        mass = (settings && settings.mass)?settings.mass:1,
24        X=settings.model.world.meshgrid.cols,
25        Y=settings.model.world.meshgrid.rows,
26        Z=settings.model.world.meshgrid.levels,
27        Height=20,
28        damping=5,
29        stiffness=settings.stiffness||20,
```

```
30        Mass=200,
31        MC=5,
32        // hole=[1,2,0];
33        hole=none;
34
35    function matrix(n,m,k) {
36      var x,y,z,mat;
37      mat=new Array(n);
38      for(x=0;x<n;x++) {
39        mat[x]=new Array(m);
40        for(y=0;y<m;y++)
41          mat[x][y]=new Array(k);
42      }
43      return mat;
44    }
45
46    var constraints = [];
47    var bodies = [];
48    var springs = [];
49    var masses = matrix(X,Y,Z);
50    var loadings=[];
51
52    world.gravity.set(0,0,-10);
53    world.camera.position.set(150,130,70);
54    world.camera.up.set(0,0,1);
55    world.camera.fov=5.0;
56
57    var groundMaterial = new CANNON.Material("groundMaterial");
58
59    // Ground
60    var groundShape = new CANNON.Plane();
61    groundShape.color = 0x00ff00;
62    var ground = new CANNON.Body({ mass: 0, material: groundMaterial });
63    ground.addShape(groundShape);
64    ground.position.z = 0;
65    world.addBody(ground);
66    GUI.addVisual(ground);
67
68  /*
69    var fixedBody = new CANNON.Body({mass: 0,
70                                     material: groundMaterial  });
71    var fixedPlane = new CANNON.Plane();
72    fixedPlane.color = 0x00ffff;
73    fixedBody.addShape(fixedPlane);
74    var rot = new CANNON.Vec3(1,0,0)
75    fixedBody.quaternion.setFromAxisAngle(rot, Math.PI/2)
76    fixedBody.position.set(0,0,0);
77  */
78    function makeWalls() {
79      var h,h2;
80      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 });
    b.addShape(bShape);
```

```
132     b.position.set(x,y,z+Height);
133     bodies.push(b);
134     return b;
135   }
136
137   function connect(bodyA,bodyB,settings) {
138     var sAB, localPivotA,localPivotB,constraint,
139         dir=new CANNON.Vec3();
140     sAB = new CANNON.Spring(bodyA, bodyB, {
141               stiffness:stiffness+(MC-2*MC*Math.random()),
142               damping:damping,
143               computeRestLength:true
144           });
145       // world.log(sAB.restLength);
146     springs.push(sAB /*,sBA*/);
147     world.addSpring(sAB);
148     if (!bodyA.springs) bodyA.springs={};
149     if (!bodyB.springs) bodyB.springs={};
150     bodyB.gridPosition.vsub(bodyA.gridPosition,dir);
151     bodyA.springs[dir.x+','+dir.y+','+dir.z]=sAB;
152     dir=dir.negate();
153     return sAB;
154   }
155
156
157   function makePlate(l,n,m,d) {
158     var dx=5,dy=5,dz=5,b,i,j,k,u,
159         x=0,y=0,z=dz*m,offInd=0,bA,bB;
160     function get(i,j,k,d) {
161       if (d) i+=d[0], j+=d[1], k+=d[2];
162       if (masses[i] && masses[i][j] && masses[i][j][k])
163         return masses[i][j][k];
164       else return none;
165     }
166     for(k=0;k<l;k++) {
167       z=dz*m;
168       for(j=0;j<m;j++) {
169         y=0;
170         for(i=0;i<n;i++) {
171           if (!equal([k,i,j],hole)) {
172             b=makeBox(x,y,z);
173             masses[k][i][j]=b;
174             b.gridPosition=new CANNON.Vec3(k,i,j);
175           }
176           y=y+dy;
177         }
178         z=z-dz;
179       }
180       x=x+dx;
181     }
182     for(k=0;k<m;k++) {
```

```
183       for(j=0;j<n;j++) {
184         for(i=0;i<l;i++) {
185           var vec = [
186             [0,1,0],
187             [1,1,0],
188             [1,0,0],
189             [1,-1,0],
190             [0,0,1],
191             [0,-1,1],
192             [-1,-1,1],
193             [-1,1,1],
194             [0,1,1],
195             [1,1,1],
196             [1,0,1],
197             [1,-1,1]
198           ];
199           for(u in vec) {
200             bA=get(i,j,k);
201             bB=get(i,j,k,vec[u]);
202             if (bA && bB) connect(bA,bB);
203           }
204
205         }
206       }
207     }
208   }
209   makePlate(X,Y,Z);
210   makeWalls();
211   // makeLoad(10,15,2,5);
212
213   for(i=0; i<constraints.length; i++)
214       world.addConstraint(constraints[i]);
215
216   for(i=0; i<bodies.length; i++){
217       world.addBody(bodies[i]);
218       GUI.addVisual(bodies[i]);
219   }
220
221   world.addEventListener("postStep",function(event){
222     for(var i in springs) {
223       springs[i].applyForce();
224     }
225   });
226
227
228   return {
229     masses:masses,
230     loadings:loadings,
231     map: function (id) {
232       // Map logical node [i,j,k] to respective mass body
233       try { return masses[id[0]][id[1]][id[2]] } catch (e) {};
```

```
234       }
235     }
236
237 }
238
```

11.8 Further Reading

1. P.-O. Siebers and U. Aickelin, *Introduction to Multi-Agent Simulation*, Encyclopaedia of Decision Making and Decision Support Technologies, 2008.
2. U. Wilensky and W. Rand, *An Introduction to Agent-Based Modeling - Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. MIT Press, 2015, ISBN 9780262731898
3. A. M. Uhrmacher and D. Weyns, Eds., *Multi-agent systems: simulation and applications*, CRC Press, 2009, ISBN 9781420070231

epubli, ISBN 9783746752228 (2018)