SEM Programming Language

SEM SeSAm Simulator Programming Language

SYNOPSYS

Introduction to and overview of the Programming Language SEM offering the specification of a simulation and behaviour model for state-based agents used for the SeSAm simulator environment.
Author: Stefan Bosse

DESCRIPTION

The activity-based functional SEM programming language provides statements for describing the behaviour of state-based agents by using activities and transitions connecting and enabling activities. Activities and functions provide generic functional 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 (Klügel et al.), which provides only a GUI. It is closely related to the SeSAm agent and world model, but is extended with some convenient functions and statements, easing the design of complex MAS.

AGENTS

An agent belongs to a particular agent class defining body variables, activities, and transitions between activities, summarized in Table **1**.

Tab. 1. Agent Class Definition

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.

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

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.

def x = *expr* **end**; **def** mutable x = *expr* **end**; **def** private mutable x = *expr* **end**; **def** x:*T* = *expr* **end**;	Definition of global read only, global mutable, and private data storage objects of data type DT derived from the initial value or specifified with an explcit type declaration.

activity a =
  [*statements*] before;
  [*statements*];
  [*statements*] after;
  transitions
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 (conditional depending on evaluation of a boolean expression and unconditional).

SEM Statement	Description
feature fc = *definitions* **end**; open 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 open statement.

DEFINTIONS

Value (variable), function, and type definitions are summarized in Tab. **2**.

Tab. 2. Definition of types, values, and functions

Statement	Description
type e = {X,Y,Z,..};	Definition of an enumeration (sum) type with symbol 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 recory type definition introduces an automatic definition of a type constructor function with same name, which can be used in expressions to create a record type value.

def v = *expr* **end**; **def mutable** v = *expr* **end**; **def** v:*DT* = *expr* **end**;	Definition of an immutable (constant) and mutable value (variable). The type is either derived from the expression (not always possible) or specified explicitly by an additional 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 datatype of the parameters can be specified explicitly by an additional type declaration. Type inference of function parameters is limited.

Statement	Description
`fun F : T1*T2*.. → RT ;`	Declaration of a function type interface (supported only for builtin primitive functions).

SEM DATA TYPES

Predefined ordinal data types `DT` are summarized in Table **7**.

Tab. 3. Ordinal data types DT

APL Statement	Type DT	Description
`integer`	`INT`	Signed integer
`double`	`REAL`	Floating point type
`string`	`STRING`	Text string consisting of characters
`boolean`	`BOOL`	Boolean type (with value set {`true`, `false`}.
`void`	`-`	Empty type
`'a list` `DT list`	`LIST`	Polymorphic and concrete list type
`'a iterator` `DT iterator`	`ITERATOR`	Polymorphic and concrete iterator type
`activity`	`ADT`	Activity type
`color`	`ADT`	Simulation object color type
`objclass`	`ADT`	Agent class type
`objinstance`	`ADT`	Agent class instance type
`position`	`ADT`	Simulation object position type
`simobject`	`ADT`	Simulation object type
`spatialinfo`	`ADT`	Spatial info type

VALUES

Values of differente data types `DT` can appear in expressions and assignments, summarized in Table **9**.

Tab. 4. Values and data types DT

Value	Type	Description
`-2, -1, 0, 1 ,2 ,3 ,4, ...`	`INT`	Signed integer number (decimal format)
`-2.41,..,2.41, ...`	`REAL`	Floating type number (decimal format)
`"abc"`	`STRING`	String (character array)
`true, false`	`BOOL`	Boolean value
`{v1,v2,..}` `{}`	`LIST`	List of values and empty list
`[|v1,v2,..|]`	`ARRAY`	Array of values
`T(v1,v2,..)` `T(e1:v1,...)`	`RECORD`	Record value constructor function for record type T with optional labels specifying the record element.
^class	`CLASS`	Instance of an agent class

EXPRESSIONS

Expressions are used in assignments, branches, function applications, and loops. There are arithmetic, relational, and boolean operations.

Tab. 5. Arithmetic, relational, and boolean/logical (bitwise) operators with applicable data types

Operator	Type	Description
`+,-,*,/`	`INT, REAL`	Addition ,Subtraction (Negation), Multiplication, Division
`%`	`INT`	Modulo division

Operator	Type	Description
< <= > >= = <>	INT, REAL	Lower than, lower equal than, greater than, greate equal than, equal, not equal.
and, or , not	BOOL	Boolean operators
(x::DT)	DT' -> DT	Type casting of variables only
float(x) round(x) trunc(x) floor(x) ceiling(x) int(x) char(x)	DT -> REAL REAL -> INT REAL -> INT REAL -> INT REAL -> INT DT -> INT DT -> CHAR	Type conversion (applicable to expressions, values, variables)

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.

Function application is provided by using the function name and an argument list, which can be empty (Def. **1**). Arguments containing expressions are evaluated before function application. Function applications can be embedded in expressions.

Def. 1. Function application (f) and procedure execution (p) with arguments, w/o arguments

```
dst := f(arg1,arg2,...);
dst := f();
expr(f(...))
p(arg1,arg2,...);
p();
```

PATHS

Access of objects (variables) from other agents is performed by using paths, summarized in Tab. **6**. 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.

Tab. 6. Path selectors

Statement	Description
`def x:simobject = null end;` `x->`*`class`*`->`*`vref`*	A variable x containing a valid simulation object reference is the root element of the path selector, which resolve a variable *vref* of an agent belonging to the specified *class*.

LISTS, ITERATORS, ARRAYS

Lists are dynamic data structures, which can be modified at run-time, summarized in Table **11**. Iterators are are derived from lists and are used by iterative functions provided by SeSAm. Some function except 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. The SEM language provides limited array support.

Tab. 7. Summary of list definitions and their usage

Statement	Data Type DT	Description
`def L:`*`DT`* `list = {} end;`	`* LIST`	Definition of a list variable initialized with an empty list. Initialiazation with empty lists requires the type declaration of the variable, otherwiese type inference is used to determine the list type.
`L := {v1,v2,..};`	`* LIST`	Assignment of a new list to a list variable.

Tab. 8. List and set type definitions (TYPE: structure type)

Statement	Data Type DT	Description
{v1,v2,v3,..}	* LIST	List of values
L.[*expr*] L.[head] L.[tail]	* LIST	Selection of a list element using an index expression (head and tail are keywords - the tail index is evaluated at run-time).
L.[*expr*] := *e*; L.[head] := *e*; L.[tail] := *e*;	* LIST	Modifies a selected list element using an index expression.
L@{v1,v2,..}	* LIST	Concatenation of lists
e :: L	* LIST	Add an element to the top of the list L.
L ::+ *e*; L +:: *e*;	* LIST	Append new element *e* at the end or before the head of the list L.
x ::- L; x -:: L;	* LIST	Remove last or first element form list L and assign the removed element to the variable x.
fun AsIterator: 'a list -> 'a iterator; fun AsList: 'a iterator -> 'a list;	* LIST * ITERATOR	Functions for conversion between iterators and lists.

SEQUENTIAL COMPOSITION

Statements in functions and activities are executed strict sequentially. Statements `S1; S2; S3;...` separated by a semicolon must be wrapped with a block statement, summarized in Tab. **8**

Tab. 9. Sequential composition using blocks

Statement	Description

`[`
`  statement1;`
`  statement2;`
`  ..`
`  statementn`
`];`
	Generic block which can be used in functions or nested activity statements. Statements grouped in a block are executed in a strict sequential order.

BRANCHES

There are different branch statements available. They pass the program flow to an alternative statement or a block of statements depending on values. Branches can appear within block statements in functions or activities. There are functional counterparts to the procedural statements which may appear in expressions (conditional expressions).

Tab. 10. Procedural and functional branch statements

Statement	Kind	Description
if *cond* **then** *statement1* **else** *statement0*;	Boolean Branch Procedural	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).
if *cond* **then** *expr1* **else** *expr2*	Boolean Branch Functional	Depending on the result of the boolean expression *cond* either *expr1* (*cond*=true) or the required *expr0* (*cond*=false) is evaluated and its value is returned.

Statement	Kind	Description
case *expr* **of** | *v1* => *stmt1*; | *v2* => *stmt2*; **end**;	Multi-value Branch Procedural	Different constant values are compared with the result of the expression *expr* and the respective statements are selected on successfull matching. There is no default `else` case (matching all other values)!
case *expr* **of** | *v1* => *expr1* | *v2* => *expr2*	Multi-value Branch Functional	Different constant values are compared with the result of the expression *expr* and the respective expressions are evaluated on successfull 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).

LOOPS

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

Tab. 11. Loop statements

Statement	Kind	Description

for i **=**
  a to|downto b
do
  statement
done;
for i **in** x ..
for i **in**
  {v1,..} ..
	Counting Loop	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*.

while *expr* **do** *statement* **done**;	Conditional Loop	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.

SHAPES

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

Tab. 12. Shape parameter

Statement	Kind	Description
color:*color*	COLOR	Shape colors: grey, lightgrey, yellow, green, blue, red, orange.
shape:*shape*	GEOMETRY	Shape geometry: rectangle, square, circle, ellipse.

Statement	Kind	Description
size:{*width*, *height*}	GEOMETRY	Shape size specifying the width and height of the shape (double value).
center:{*x0*, *y0*}	POSITION	Relative shape center point (double value).
fill:bool	COLOR	Shape fill attribute (true/false).

EXAMPLES

Ex. 1. Resource class definitions

```
resource link(color:grey,
              shape:rectangle,
              size:{2.0,2.0},center:{1.0,1.0}) =
  def mutable Node = Position(0,0) end;
  def mutable Link = NORTH end;
end;
resource connection(color:blue,
                    shape:rectangle,
                     size:{2.0,2.0},
                    center:{1.0,1.0}) =
  def mutable Node1:simobject = null end;
  def mutable Node2:simobject = null end;
end;
```

Ex. 2. Type definitions

```
type Stats = (
  stat_index = integer,
  stat_event = integer,
  stat_dist = integer,
  stat_explorer = integer,
  stat_explorer_child = integer,
  stat_features = integer
);
type Direction = {
  NORTH,
  WEST,
```

```
    EAST,
    SOUTH,
    ORIGIN
  };
  type Event = {
    NOEVENT,
    TIMEOUT,
    WAKEUP
  };
  type Timer = (
    timer_val = integer,
    timer_event = Event
  );
```


Ex. 3. Feature class definitions

```
  feature env =
    def private mutable _env_tempobj:simobject =
     null end;
    def mutable blocked = false end;
    def mutable errno = EOK end;
    def private mutable _env_vd = 0.0 end;
    def private mutable _env_vi = 0 end;
    def private mutable _env_i = 0 end;
    def private mutable _env_j = 0 end;
    def private mutable _env_mat:integer list list =
     {} end;
    def private mutable _env_start_time = 0.0 end;
    def private mutable _env_in_await = false end;
    def private mutable _env_vec:integer list =
     {} end;

    def AbsI = fun v:integer ->
      [
        _env_vd := Abs((v::double));
        _env_vi := (_env_vd::integer)
      ]; _env_vi
    end;

    def AwaitDelay = fun tmo:double ->
      if _env_in_await then
        [
```

```
      if ((GetTime()-_env_start_time)>tmo) then
      [
        _env_in_await := false;
        blocked := false
      ]
    ]
  else
    [
      _env_in_await := true;
      _env_start_time := GetTime();
      blocked := true
    ];
end;

--
-- CREAT AGENT OF CLASS agentclass
--
def CreateAgent = fun x:integer,y:integer,
                      agentclass:objinstance ->
  CreateObjectAndRemember(agentclass,
   (fun obj:simobject ->
    [
      BeamTo(GetSpatialInfo(obj),
             CreatePos((x::double)*10.0+5.0,
                       (y::double)*10.0+5.0))
    ];))
end;
--
-- CREATE AGENT OF CLASS agentclass AND
-- RETURN SIMOBJECT
--
def CreateAgentAndReturn =
  fun x:integer,y:integer,
      agentclass:objinstance ->
  CreateObjectAndRemember(agentclass,
   (fun obj:simobject ->
    [
      _env_tempobj := obj;
      BeamTo(GetSpatialInfo(obj),
             CreatePos((x::double)*10.0+5.0,
                       (y::double)*10.0+5.0))
    ];)); _env_tempobj
```

```
end;
--
-- GET (ONE) AGENT OF SPECIFIED CLASS AT
-- CURRENT POSITION
--
def GetAgent = fun thisclass:objclass ->
  GetFirst(Select(
      (fun obj:simobject ->
        Distance(GetSpatialInfo(obj),
        GetSpatialInfo(GetCurrentSimObject()))
                        = -1),
      GetAllObjectsOfType(thisclass,true,true)
    ))
end;
--
-- GET MATRIX ELEMENT mat(i,j) WITH
-- col=0,..,cols-1,row=0,..,rows-1
--
def GetMatrixI = fun mat:integer list list,
                     row:integer,col:integer ->
  GetNth(col,GetNth(row,mat))
end;
--
-- COMPUTE MAT1-MAT2 (matrix element subtraction)
--
def MatrixSub = fun mat1:integer list list,
                    mat2:integer list list ->
  [
    _env_mat := {};
    _env_j := 0;
    ForElements((fun row:integer list ->
      [
        _env_mat ::+ VectorSub(row,
                        GetNth(_env_j,mat2));
        _env_j := _env_j + 1
      ];
      ),mat1)
  ]; _env_mat
end;

def VectorSub = fun vec1:integer list,
                    vec2:integer list ->
```

```
      [
        _env_vec := {};
        _env_i := 0;
        ForElements((fun col:integer ->
          [
            _env_vec ::+ (col-vec2.[_env_i]);
            _env_i := _env_i + 1
          ];
        ),vec1)
      ]; _env_vec
    end;

    --
    -- SET MATRIX ELEMENT mat(i,j) WITH
    -- i=0,..,cols-1,j=0,..,rows-1
    --
    def SetMatrixI =
      fun mat:integer list list,
          row:integer,col:integer,v:integer ->
      [
        _env_vec := GetNth(row,mat);
        SetNth(col,v,_env_vec);
        SetNth(row,_env_vec,mat)
      ];
    end;

    --
    -- I AM THIS SIM OBJECT
    --
    def Self = fun () -> GetCurrentSimObject()
    end;

    --
    -- CHANGE COLOR
    --
    def SetColor = fun thisobj:simobject,r,g,b ->
      ChangeShapeColor(GetSpatialInfo(thisobj),
                       CreateColor(r,g,b))
    end;
  end;
```

Ex. 4. Agent class definitions

```
agent sampling (color:orange,
  shape:circle,fill:true,size:{2.0,2.0},
  center:{3.0,1.0}) =
  use os;
  use env;
  use ts_client;

  def mutable adc = 0 end;
  def mutable Pos = Position(0,0) end;
  def mutable self:simobject = null end;
  def mutable myworld:simobject = null end;
  def mutable Parent:simobject = null end;
  def mutable sampled = 0 end;

  activity init =
    [
      self := Self();
      ts_init(Parent);
      myworld := GetWorld()
    ];
  end;
  activity sample =
    [
      adc :=
       GetMatrixI(myworld->myworld->Sensors,
                  Pos.Y-1,Pos.X-1);
      out2(ADC,adc);
      sampled := sampled + 1
    ];
  end;
  activity sleep =
    [
      AwaitDelay(10.0)
    ];
  end;

  transition(entry,init);
  transition(init,sample);
  transition(sample,sleep);
  transition(sleep,sample,blocked=false);
```

```
end;
```

VERSION

```
1.2
Last modified: April 8, 2014
```