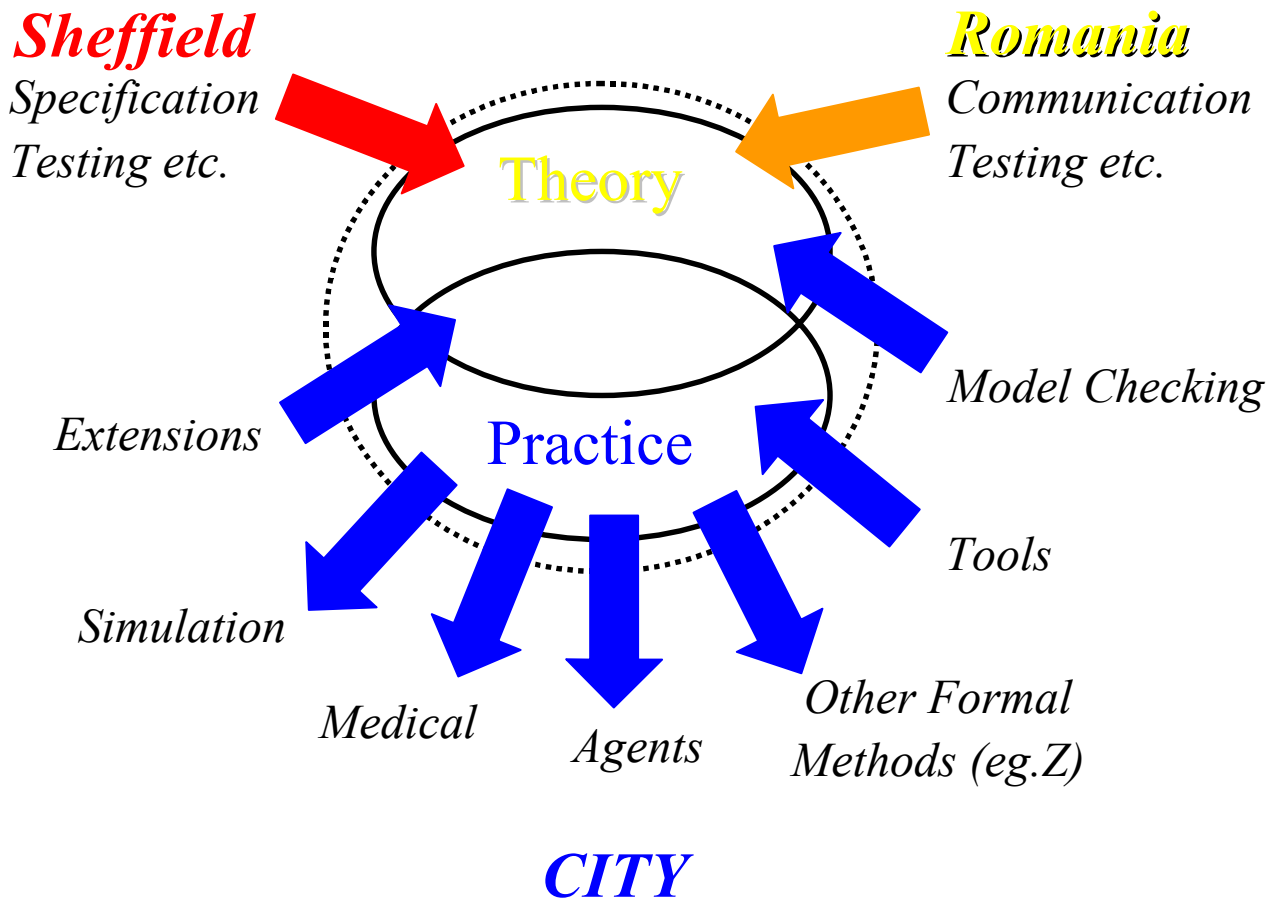


RESEARCH IN X-MACHINES



- **XMDL: A Language for coding X-Machine Specifications**
- **Automatic Code Generation: Translation of X-Machine Specifications to Prolog**
- **Translation of X-Machine Specifications to Z**
- **Using X-Machine to Model and Test Discrete Event Simulation Programs**
- **Model Checking of X-Machine Specifications**
- **Using X-Machine to Model Check Safety Critical Systems**
- **Communicating Object Oriented X-Machines**
- **Using X-machines to Model and Test Reactive Agents**
- **Tools for Test Case generation**
- **Tools for Animation**
- **Tools for Testing Programs resulted from X-Machine Specification**
- **Tools for Validating X-Machine Specifications**
- **Graphical Editor for the Development of X-Machine Specification**

A X-MACHINE EXAMPLE

A STACK WITH 10 ELEMENTS:

$\mathbf{XM} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ where:

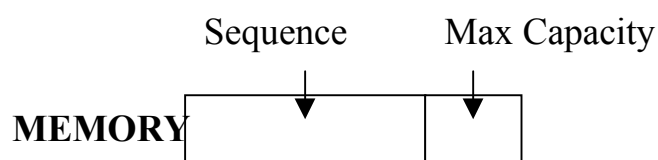
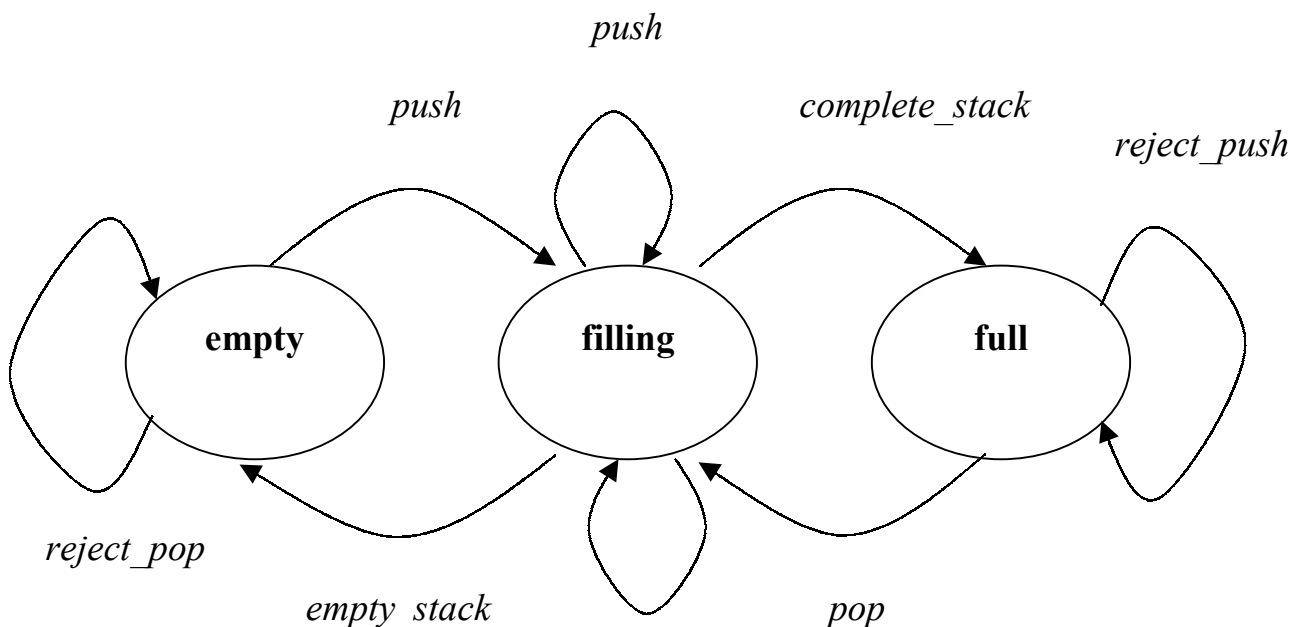
- Σ, Γ is the input and output finite alphabet respectively,
- Q is the finite set of states,
- M is the (possibly) infinite set called memory,
- Φ is the type of the machine M , a finite set of partial functions φ that map a memory state and an input to a new memory state and an output,

$$\varphi: M \times \Sigma \rightarrow \Gamma \times M$$

- F is the next state partial function that given a state and a function from the type Φ , denotes the next state. F is often described as a transition state diagram.

$$F: Q \times \Phi \rightarrow Q$$

- q_0 and m_0 are the initial state and memory respectively.



```

#model stack.

#basic_types = [ELEMENT].

#type capacity = Natural.
#type action = {down_arrow, up_arrow}.
#type messages = {ElementPushed, ElementPoped, StackFull, StackEmpty,
RejectStackFull, RejectStackEmpty}.
#type stack = sequence_of ELEMENT.
#type allelements = ELEMENT union {no_element}.

#states = {empty, filling, full}.

#memory (stack, capacity).

#init_state {empty}.
#init_memory (nil, 10).

#input (action, allelements).
#output (messages, allelements).

```

```

#fun function name INPUT PARAMETERS=
if condition expression then
  OUTPUT PARAMETERS.

```

```

#fun function name (input parameters tuple, memory tuple)=
if condition expression then
  (output parameter tuple, new memory tuple)
where
body expression.

```

```

#fun reject_pop ((up_arrow, no_element), (nil, ?c))=
  ((RejectStackEmpty, no_element), (nil, ?c)).

```

```

#fun push ((down_arrow, ?element), (?stack, ?c))=
if ?number < ?c then
  ((ElementPushed, ?element), (<?element :: ?stack>, ?c))
where
  ?number <- cardinality <?element :: ?stack>.

```

```

#fun reject_push ((down_arrow, ?x), (?stack, ?c))=
if ?number = ?c then
  ((RejectStackFull, ?x), (?stack, ?c))
where
  ?number <- cardinality ?stack.

```

```

#transition (empty, reject_pop)=empty.
#transition (empty, push)=filling.
#transition (filling, push)=filling.
#transition (filling, pop)=filling.
#transition (filling, complete_stack)=full.
#transition (filling, empty_stack)=empty.
#transition (full, reject_push)=full.
#transition (full, pop)=filling.

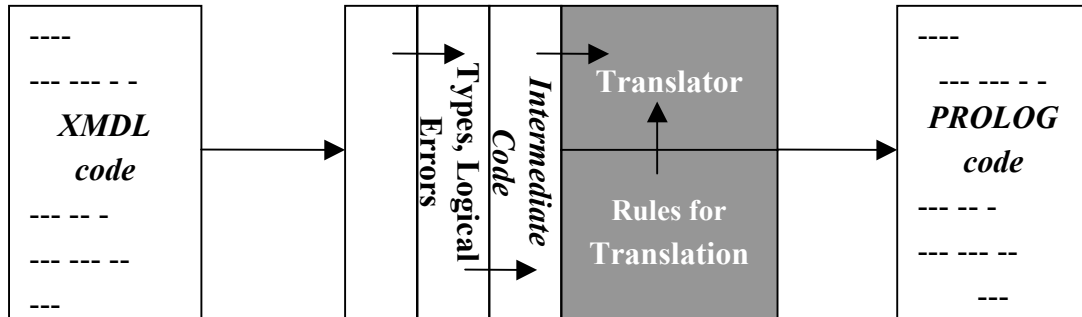
```

```

#end.

```

TRANSLATING XMDL CODE:



Rule of Translation (example):

Whenever you identify this (written in XMDL):

```
#transition ( a state, a function )= a new state.
```

output this (which is the equivalent in Prolog):

```
transition( a state, a function , a new state).
```

PARSING XMDL CODE

DEFINITE CLAUSE GRAMMAR (example):

```
transition -->
    ['#transition'],
    ['('],
    statename,
    [','],
    functionname,
    [')'],
    ['='],
    statename.
```

```
statename --> constant.
```

```
functionname --> constant.
```

```
constant -->
    allowedchar.
```

```
constant -->
    allowedchar,
    constant.
```

```
allowedchar--> letter.
```

```
allowedchar--> digit.
```

```
allowedchar--> special_symbol.
```

```
letter-->      [a].
```

```
. . .
```

```
letter-->      [z].
```

```
digit-->       [0].
```

```
. . .
```

```
digit-->       [9].
```

```
special_symbol--> ['_'].
```

“NORMAL” PROLOG:

```
parse(_,S,[]):- transition(S, []).
```

QUERY:

```
Is the phase #transition (empty, reject_pop)=empty syntactically correct?
```

```
?- parse( [#transition, (,e,m,p,t,y, \, ,r,e,j,e,c,t, _p,o,p), =,e,m,p,t,y],
           []).
```

```
yes
```

DEFINITE CLAUSE GRAMMAR

(Same Example with Parameter and Action)

```
transition -->
    ['#transition'],
    ['('],
    statename(S1),
    [','],
    functionname(F),
    [')'],
    ['='],
    statename(S2)                { my_output_transition(S1,F,S2) }.
```

```
statename(X) --> constant(X).
```

```
functionname(X) --> constant(X).
```

```
allowedchar(L) --> letter(L).
```

```
allowedchar(D) --> digit(D).
```

. . .

“NORMAL” PROLOG:

```
parse(_,S,[]):- transition(S,[]).
```

```
my_output_transition( From, With, Where):-
    write(`transition(`),
    write(From),
    write(`,`),
    write(With),
    write(`,`),
    write(Where),
    write(`).`),
    nl.
```

QUERY:

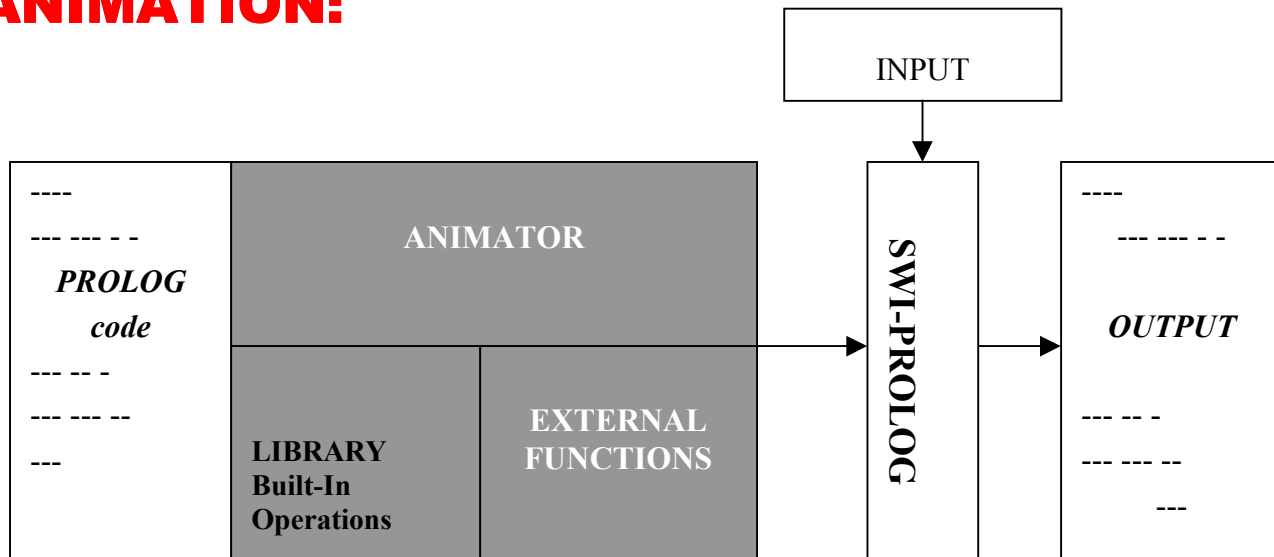
**Is the phase #transition (empty, reject_pop)=empty syntactically correct?
If so, what is the translation in PROLOG?**

```
?- parse( [#transition,(e,m,p,t,y`,` ,r,e,j,e,c,t,_,p,o,p),=,e,m,p,t,y],
          []).
```

```
transition(empty,reject_pop,empty).
```

yes

ANIMATION:



?- animatex.

State : empty Input ?[down_arrow,1].

Applied Function : push

Output: [elementpushed, 1] Memory : [[1], 10] At State :filling

State : filling Input ?[up_arrow,no_element].

Applied Function : empty_stack

Output: [stackempty, no_element] Memory : [[], 10] At State :empty

State : empty Input ?[up_arrow,no_element].

Applied Function : reject_pop

Output: [rejectstackempty, no_element] Memory : [[], 10] At State :empty

State : empty Input ?[down_arrow,1].

Applied Function : push

Output: [elementpushed, 1] Memory : [[1], 10] At State :filling

State : filling Input ?[d_arrow,2].

WARNING: Invalid Input!

WARNING: No Function Applicable!

State : filling Input ?[down_arrow,3].

Applied Function : push

Output: [elementpushed, 3] Memory : [[3, 2, 1], 10] At State :filling

TRANSLATION TO Z:

STACK MEMORY

Stack: seq ELEMENT

Capacity: N

STACK STATE

CurrentState: Q

Initialise

STACK MEMORY

STACK STATE

Stack' = <>

Capacity' = 10

CurrentState' = idle

Push

STACK MEMORY

STACK STATE

In? : INPUT

Out!: OUTPUT

In? = (down_arrow, El) \wedge

<El> \wedge Stack < Capacity \wedge

Stack' = <El> \wedge Stack \wedge

Capacity' = Capacity \wedge

CurrentState' = next(CurrentState, Push) \wedge

Out! = ("ElementPushed", El)
