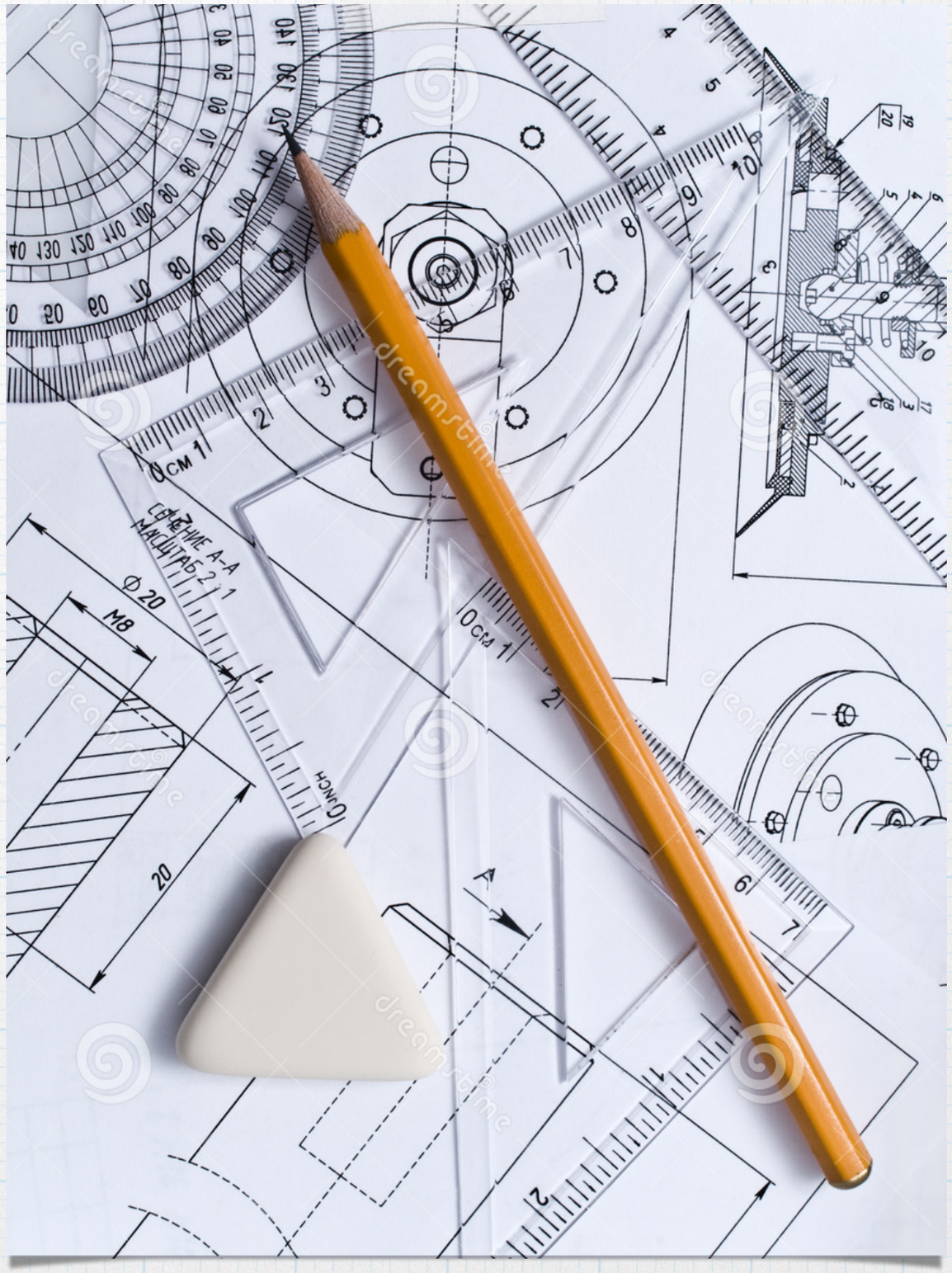
Intuitive Hierarchical State Machine

version 1.0

Victor Li
vli02@hotmail.com



Prerequisite

* UML State Machine @wikipedia
* [https://en.wikipedia.org/wiki/UML_state_machine]
* UML Hierarchical State Machine

Intuitive

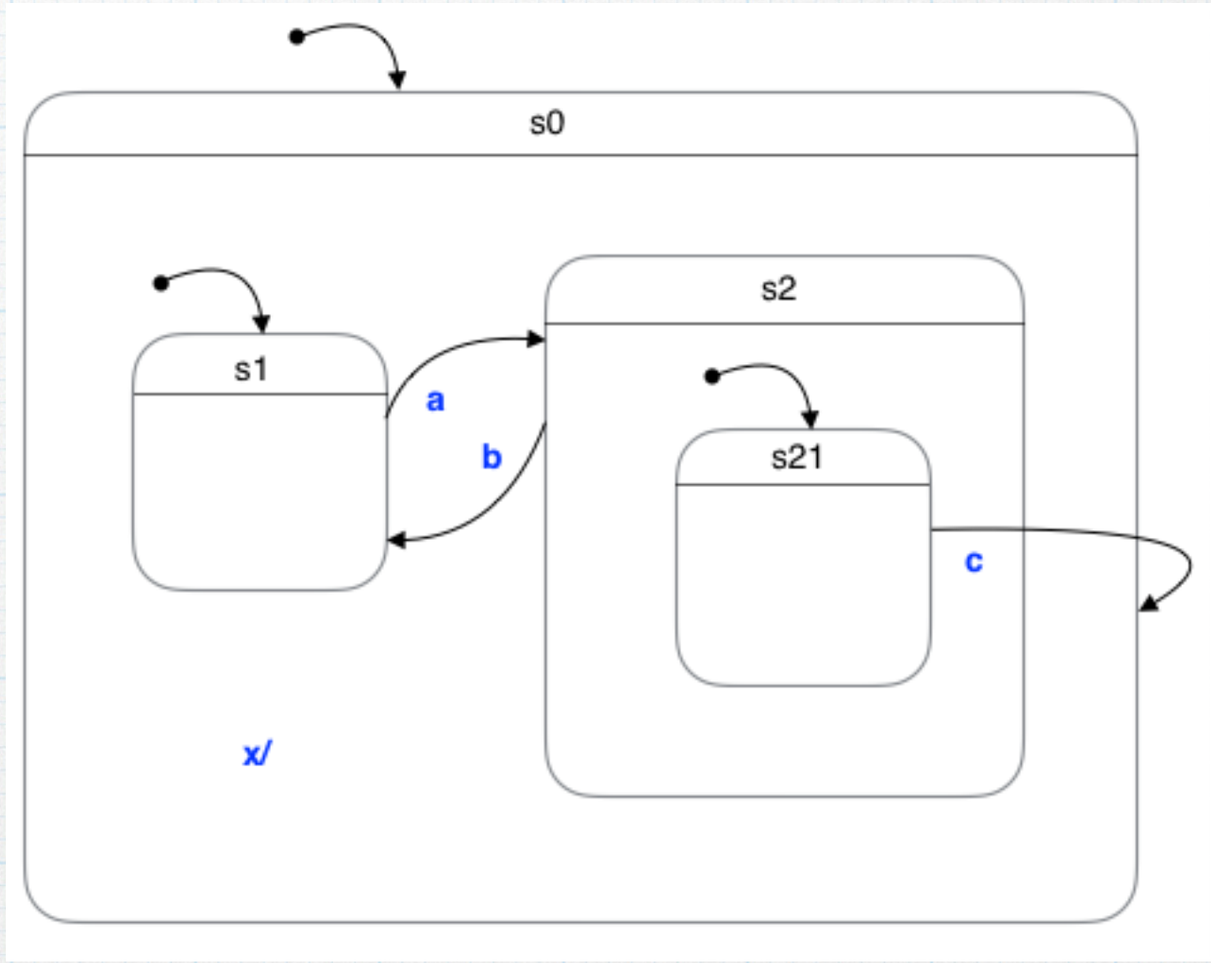
* Use a notation expression language, to
* Describe a hierarchical state machine in a Grammar File, as
* Implementation of the transition logic, in
* A simple, flexible and intuitive format

Essential

* Declare a list of event
* Declare a list of state
* Declare state hierarchical relationship
* Declare state transition rules

A quick sample

```
%event a b c x

%state s0 s1 s2 s21

%hiera s0 (s1, s2 (s21))

%% /* section delimiter */

s0: x -- /* internal trans */
;

s1: a -> s2
;

s2: b -> s1
;

s21: c -> s0
;
```



Benefit

* Simplify state machine design from a large piece of and complicated code, to
  * A small piece of intuitive descriptive text Grammar File
  * Generate high efficient source code, and
  * State machine diagram

Event

```
%event <a list of event separated by space>
```

* The list of event can be one or more events
* The list of event can be in one line or multiple lines
* More than one event declarations can be used in one Grammar File

Event samples

```
%event a
%event b
%event c
%event x

or

%event a b c x

or

%event a b c
       x
```

State

```
%state <a list of state separated by space>
```

* The list of state can be one or more states

* The list of state can be in one line or multiple lines

* More than one state declarations can be used in one Grammar File

Entry/exit action

* `'-> { action }'` following a state is the entry action of the state

* `'<- { action }'` following a state is the exit action of the state

* Inside the curly braces is user code of the action

* Entry/exit actions are optional.

State samples

```
%state s0

%state s1 s2
          s21

%state s3
        -> { printf("s3-entry"); }  /* entry action */
  <- { printf("s3-exit");  }  /* exit action */
s4
  -> { printf("s4-entry"); }
s5
  <- { printf("s5-exit");  }
s6
```

State Hierarchy

```
%hiera <a super state>
        ‘(‘ <a list of sub state separated by space> ‘)’
```

* Multiple hierarchies of super-sub state can be **nestled** in one %hiera declaration, or
* Separated in multiple %hiera declarations

Hierarchical samples

```
%hiera s0 (s1, s2 (s21))

or

%hiera s0 (s1, s2)
%hiera s2 (s21)
```

```
===============

%hiera s0 (s1 (s11), s2 (s21 (s211)))

or

%hiera s0 (s1, s2)
%hiera s1 (s11)
%hiera s2 (s21)
%hiera s21 (s211)
```

Transition rule

```
<a source state> ':' <events> <transitions>
                         [<events> <transitions>]…
';'
```

* Starts with a source state followed by a column
* Ends with a semicolon
* Transitions are attempted from top to bottom, left to right, and
* stop once one is qualified

Transition rule - events

```
<a source state> ':' <events> <transitions>
                         [<events> <transitions>]…
';'
```

* Events can be one or more events separated by comma ','

* An optional pair of parenthesis '(…)' can be used for the events

* Use a dot '.' as event for initial transition from a super state to it's sub state

Transition rule - trans...

```
 [guard] <transition type> [state] [action]
[[guard] <transition type> [state] [action]]…
```

* Transition(s) are one or more possible reaction(s) of the source state to the specified events being received

* Each transition starts with an optional guard condition, followed by

* a notation to indicate transition type, and

* an optional destination state with transition action

Guard Condition

```
' ? ( expr )'
```

* `'expr'` is a logical expression, which
  * should be evaluated as true to enable the transition, or
  * false to disable the transition
  * Initial transition has no a guard condition

Destination Type

`[guard] <transition type> [state] [action] …`

* `'__'` means internal transition
* `'->'` means external transition
* `'|>'` means local transition to sub state
* `'>|'` means local transition to super state

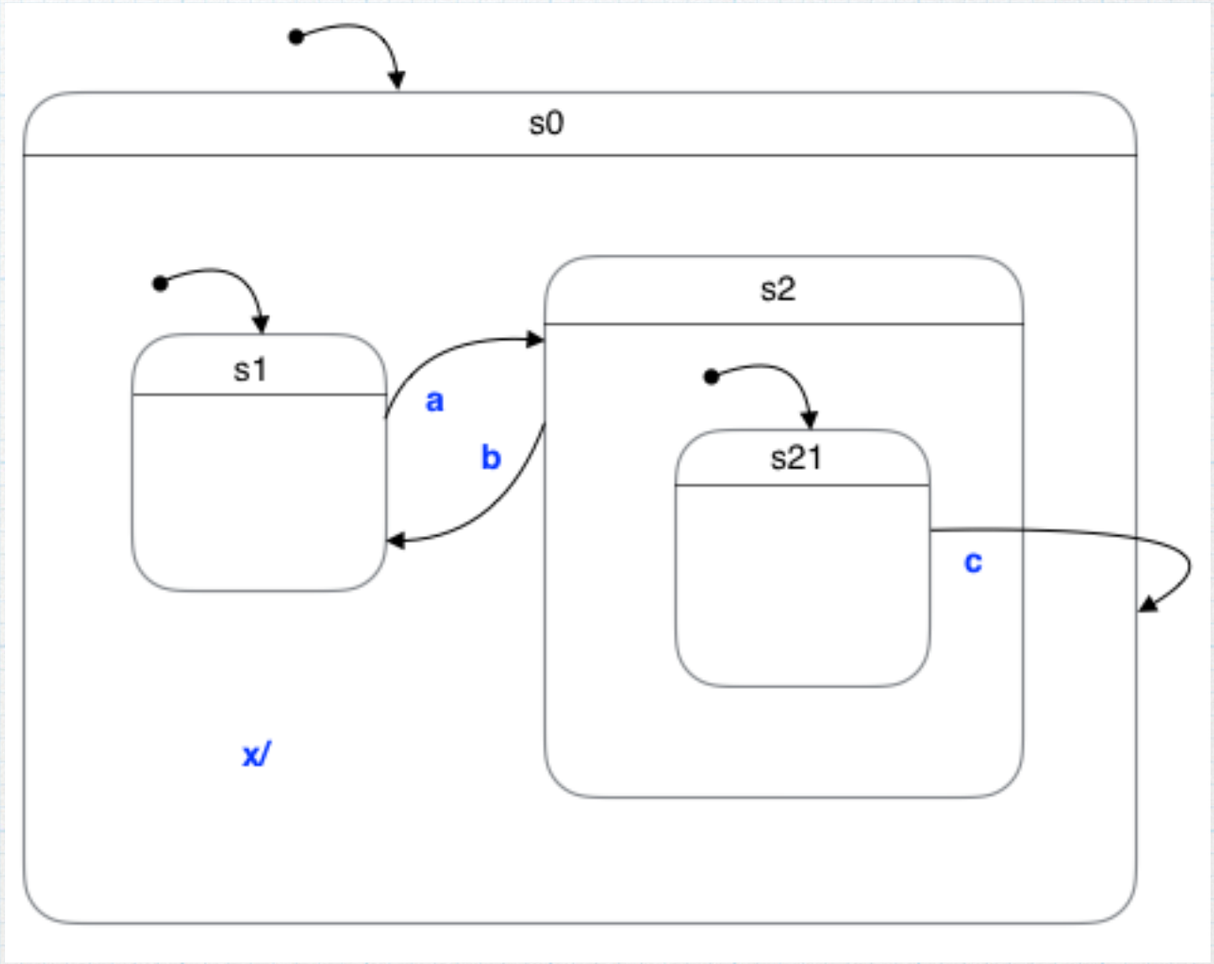
Transition Action

```
'{ action }'
```

* `action` is a piece of user code
  * All transition related data are available to access in the action code, e.g.
  * event ID and data, source state, effective state, destination state, etc...

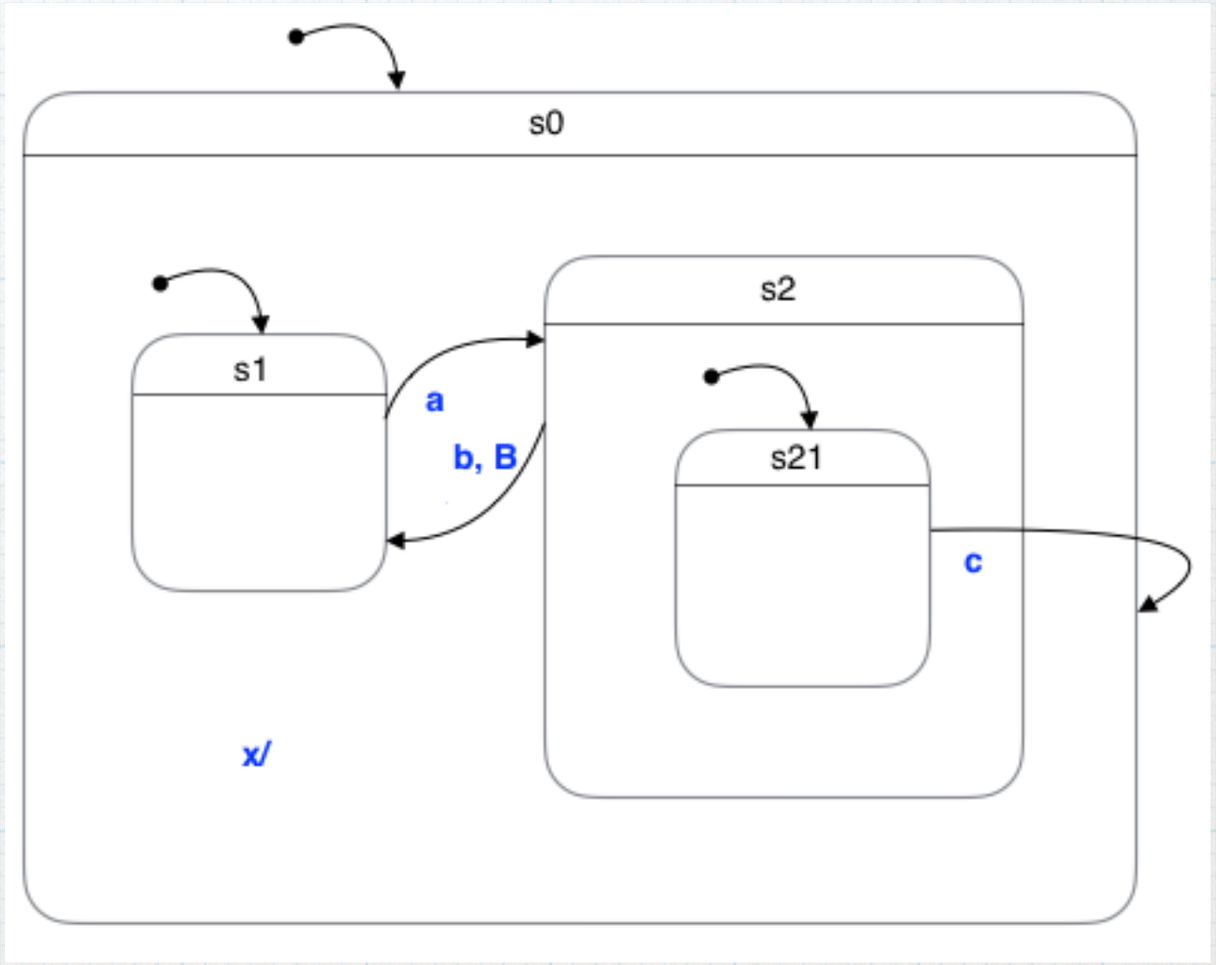
Transition samples - 1

```
s0: . -> s1
;

s0: x --
;

s1: a -> s2
;

s2: . -> s21
    b -> s1
;

s21: c -> s0
;
```



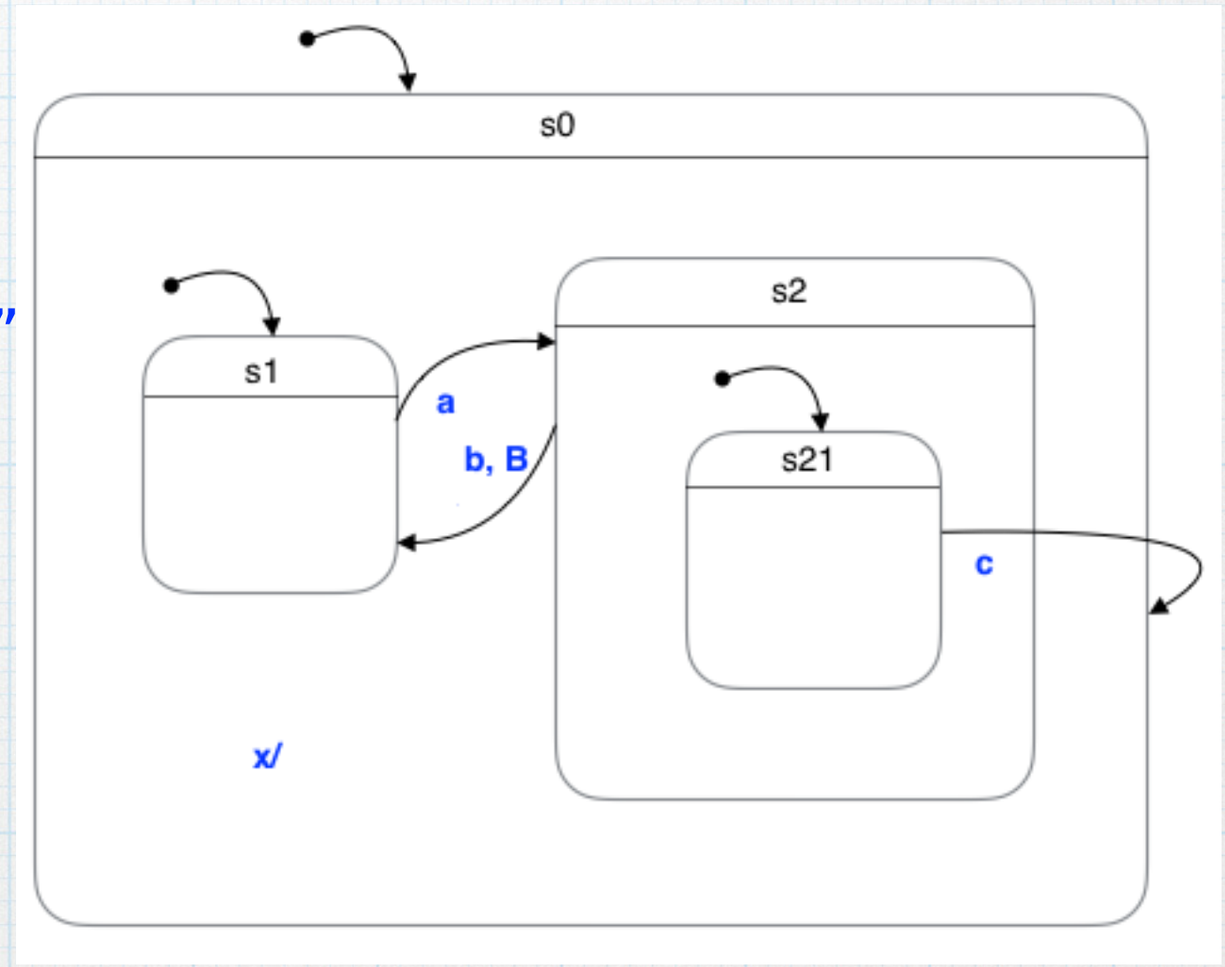
Transition samples - 2

```
s0: . -> s1
;

s0: x --
;

s1: a -> s2
;

s2: . -> s21
    (b, B) -> s1
;

s21: c -> s0
;
```



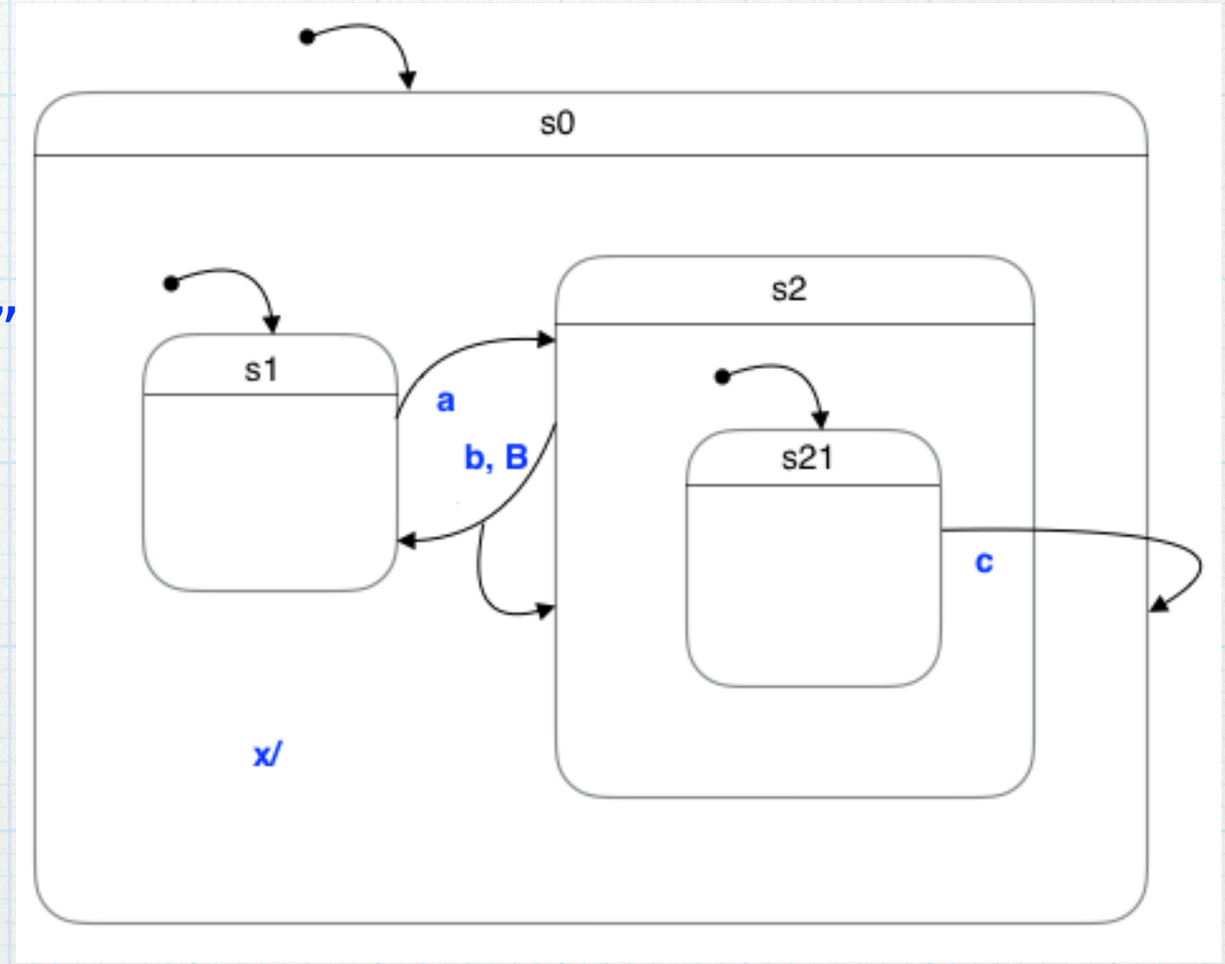
Transition samples - 3

```
s0: . -> s1  { printf("s0-s1 init"); }
;

s0: x --     { exit(0); }
;

s1: a -> s2  { printf("s1-a-s2"
;

s2: . -> s21
    (b, B) -> s1
;

s21: c -> s0
;
```



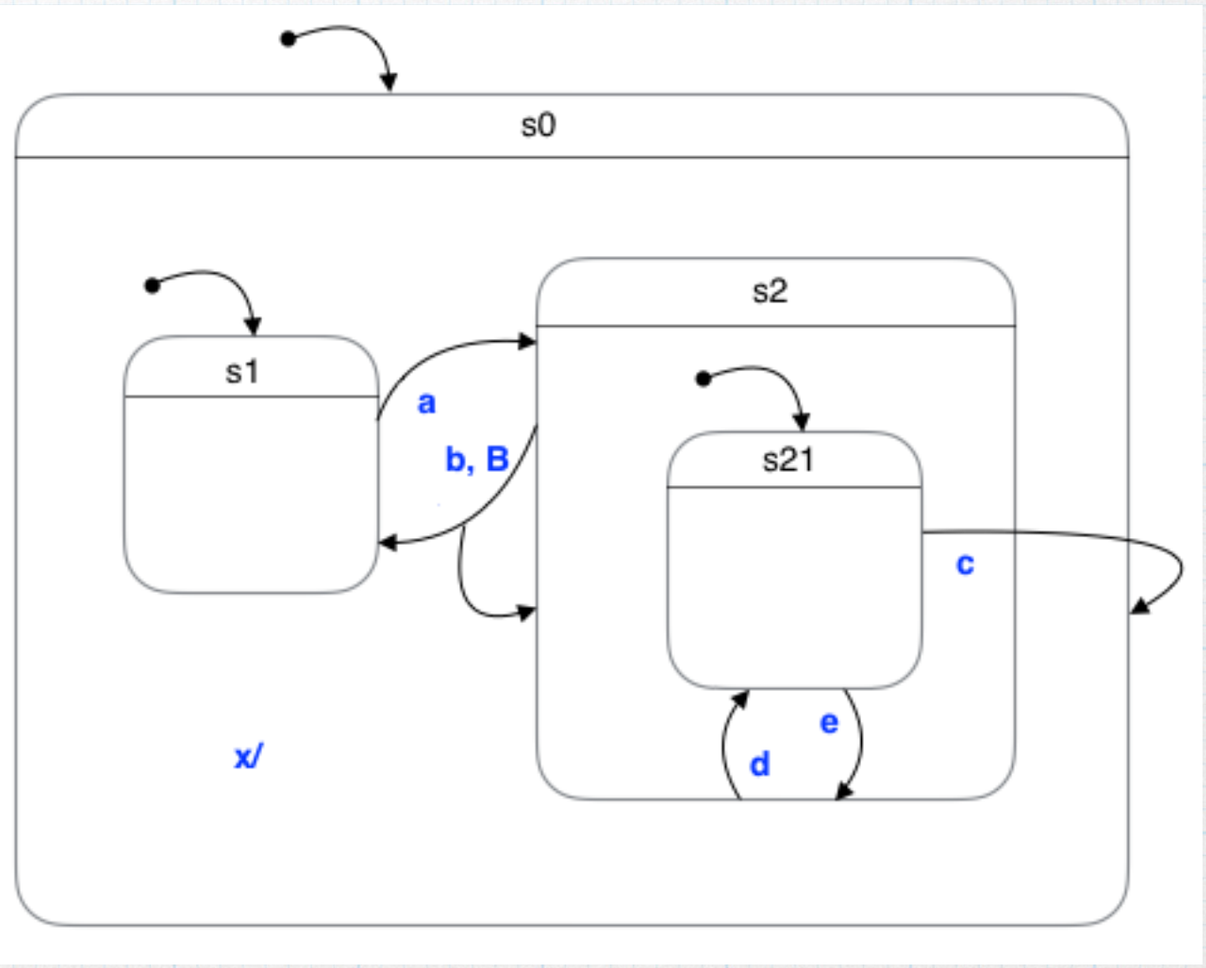
Transition samples - 4

```
s0: . -> s1  { printf("s0-s1 init"); }
;

s0: x --     { exit(0); }
;

s1: a -> s2  { printf("s1-a-s2"
;

s2: . -> s21
    (b, B) ?(test) -> s1 { }
                   -> s2 { }
;

s21: c -> s0
;
```



Transition samples - 5

```
s0: . -> s1  { printf("s0-s1 init"); }
;

s0: x --     { exit(0); }
;

s1: a -> s2  { printf("s1-a-s2"
;

s2: . -> s21
    (b, B) ?(test) -> s1 { }
                   -> s2 { }
    d |> s21  /* local trans */
;

s21: c -> s0
     e >| s2  /* local trans */
;
```



Advanced Features

* Event Deferral
* Wild-character event
* Orthogonal Regions
  * Zone
  * Region
* Run To History

Event Queue

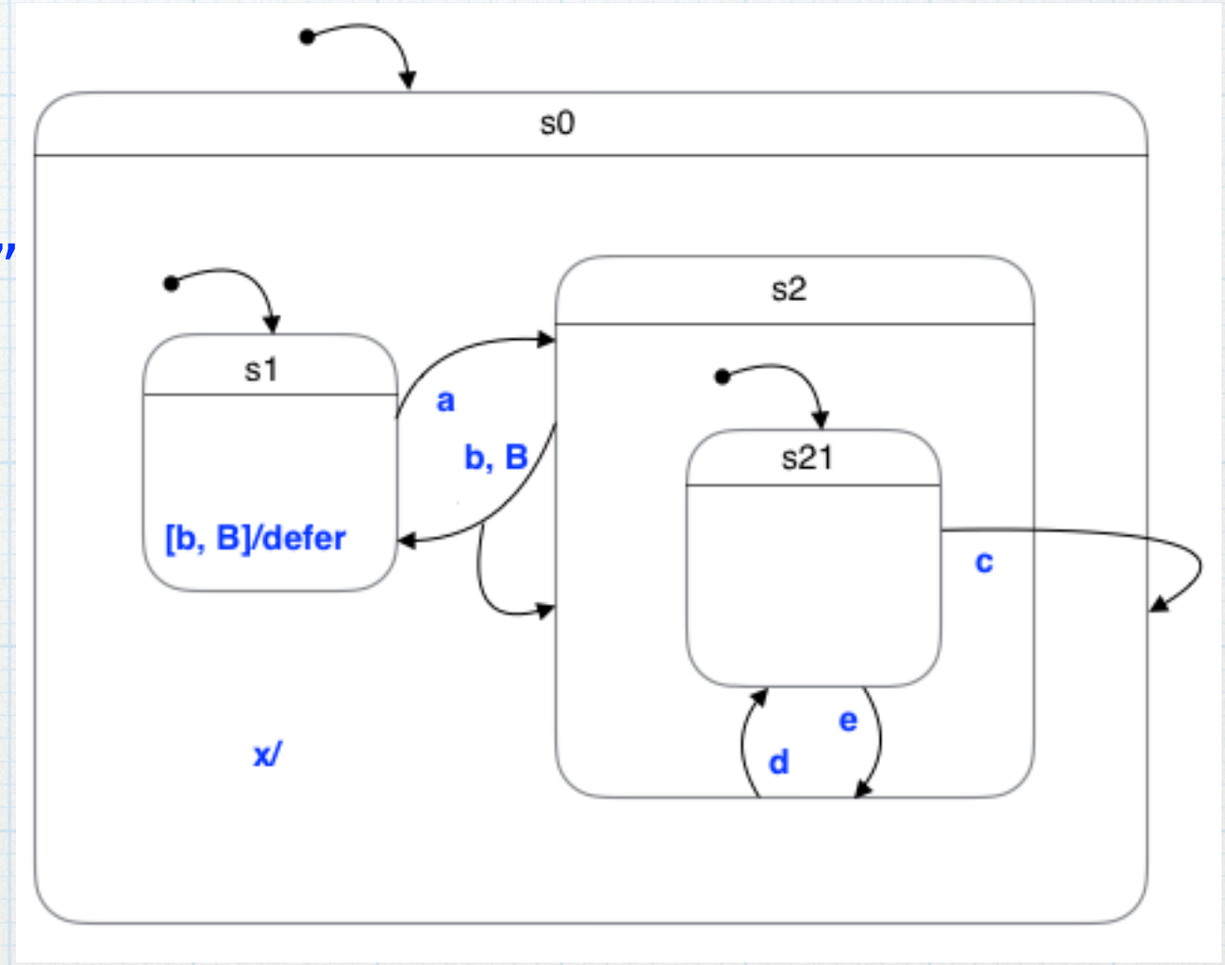
* It is user's responsibility to implement event queuing or dispatching system
* A callback of event receiving function needs to be provided

Event Deferral - 1

* Make internal transition for event which supposes to be deferred, and
* in the transition action, enqueue the event back to the event queuing or dispatching system, so
* this deferred event can be received in the right order after a state change

Event Deferral sample 1

```
s1: a -> s2  { printf("s1-a-s2")
     (b, B) --
                { /* enqueue */ }
;
```



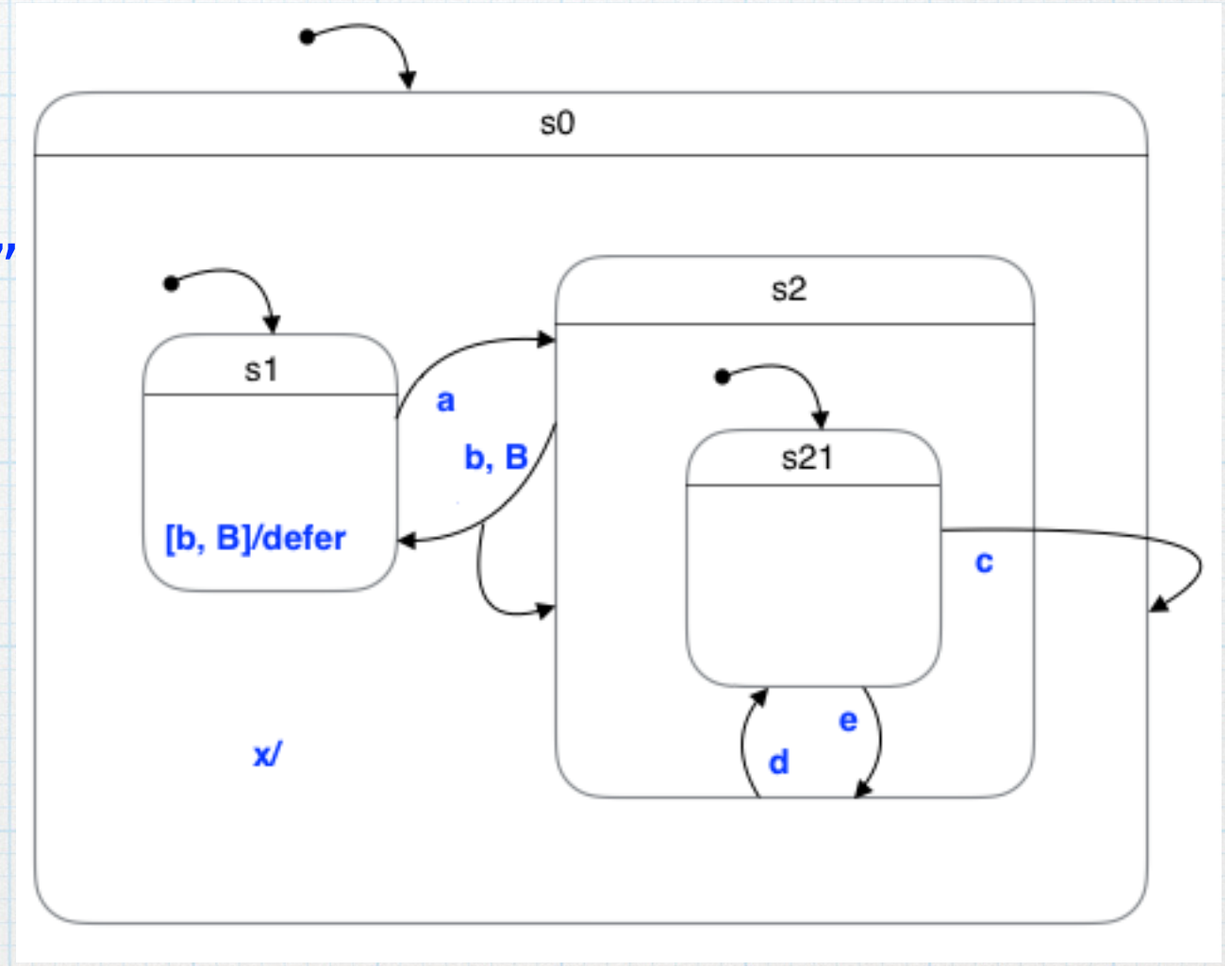
Event Deferral - 2

```
<a source state> ':' <events> '<<' ';'
```

* Use a state transition rule with deferral transition type `'<<'`
  * This transition has no guard and action
  * A callback of deferral event enqueuing function needs to be provided

Event Deferral sample 2

```
s1: a -> s2  { printf("s1-a-s2")
     (b, B) <<
;
```

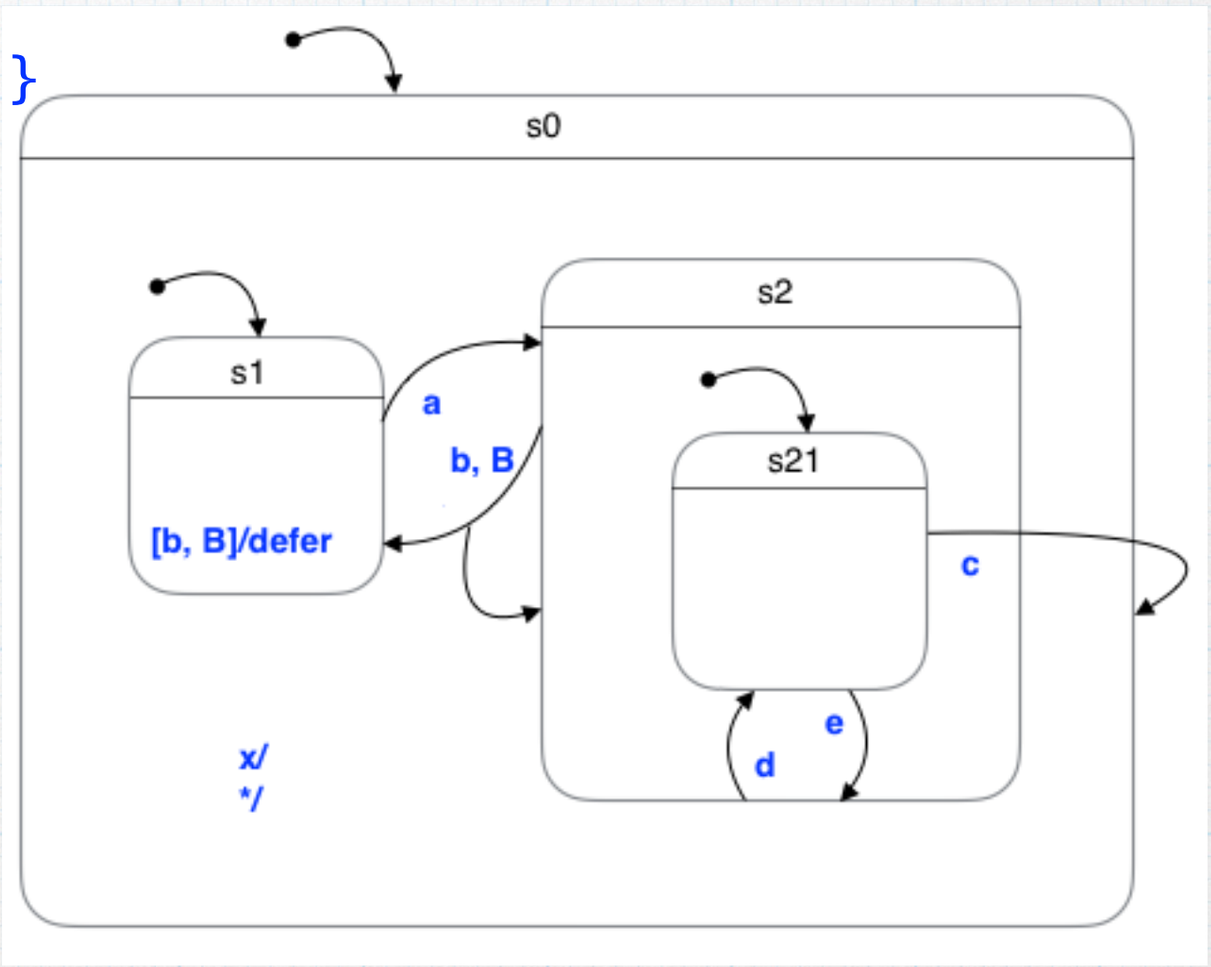


Wild-char Event (*)

- '*' can be used as a wild-char event in a transition rule, which
  - is to match any event

Wild-char Event sample

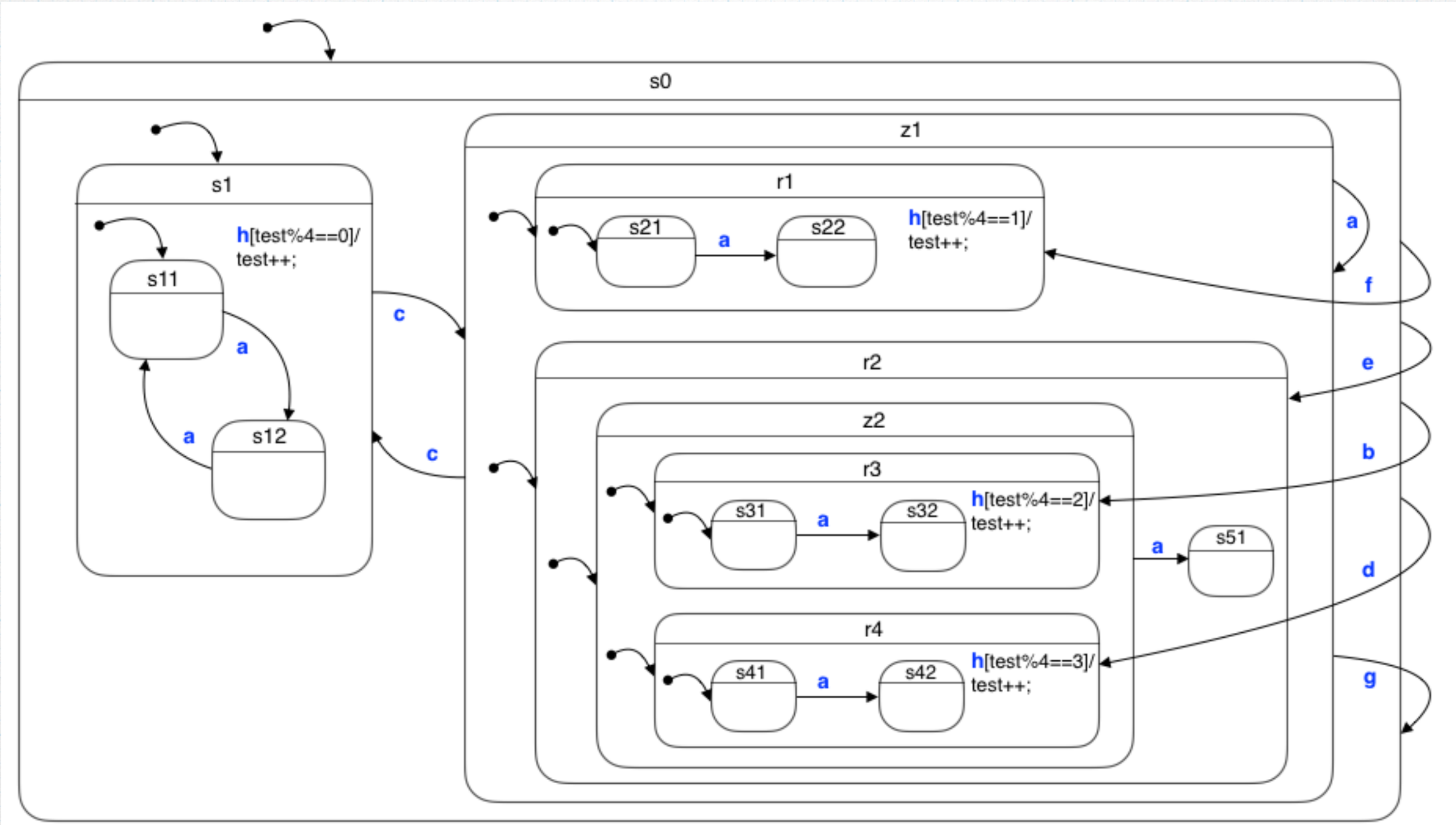
```
s0: . -> s1  { printf("s0-s1 init"); }
    x --     { exit(0); }
    * --     { /* match any */ }
;
```



Zone State

* A super state which has more than one initial sub states, and
* More than one initial transitions to these initial sub states;
* Transition to this super state triggers these initial transitions;
* Which forms orthogonal regions.

Zone State sample



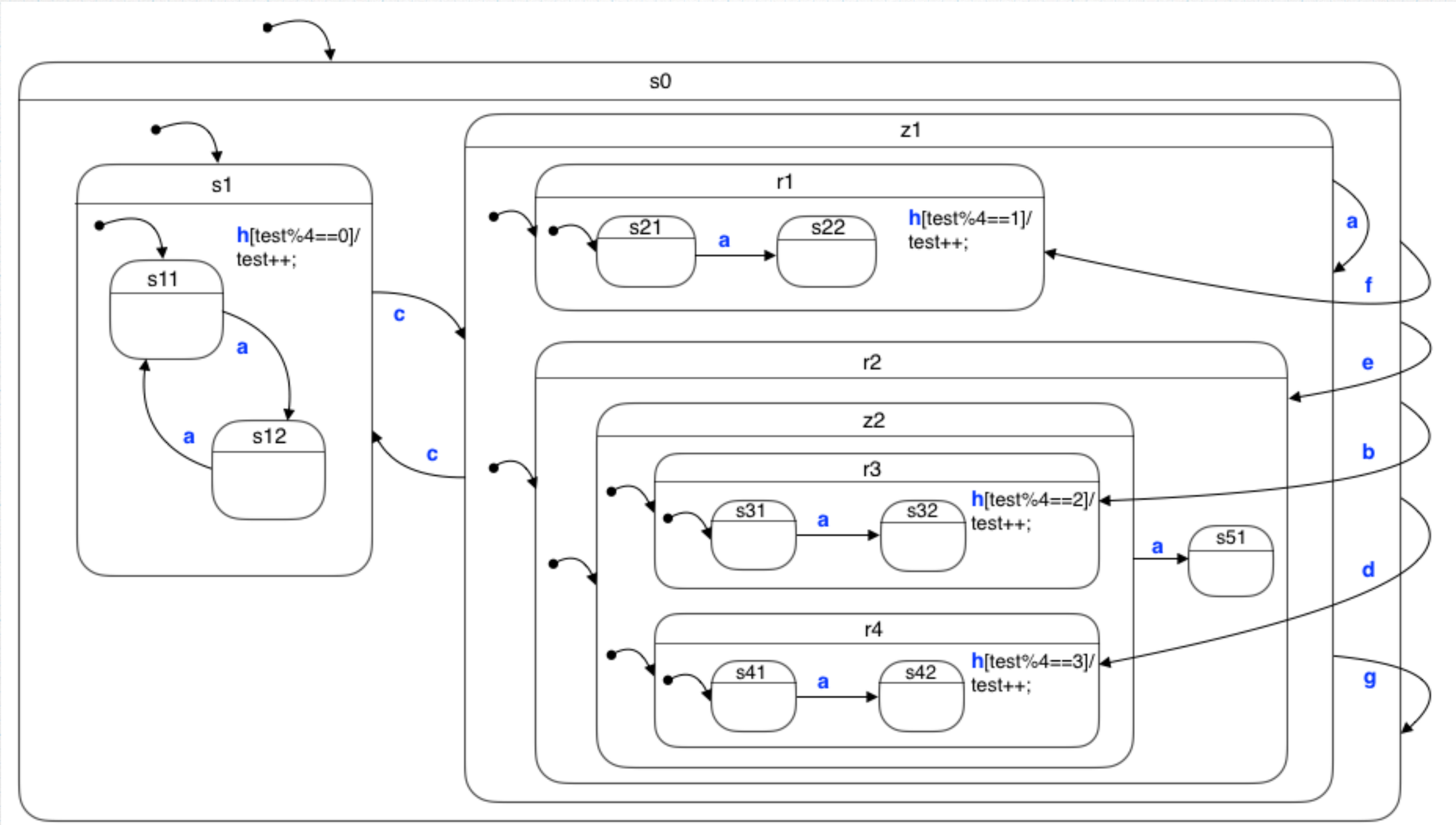
Region State

* Each sub state of a zone state is a region;
* It might be a single leaf state, or
* A super state which contains one or more sub states.
* Each region inside a zone is independent

Independence

* A region state can not have transition which goes to other region states;
* A region state can not have transition which goes to outside of the region;
* Transition to outside is made on it's zone state which will cause all region states to be gone;
* A zone state cannot local transition to it's sub state.

Region sample



Run To History

* Start to run the state machine from a state which
  * is saved in history;
  * It will not invoke start transition, and
  * Directly set the current state to the history state.

Resources

* UML State Machine @wikipedia
* [https://en.wikipedia.org/wiki/UML_state_machine]
* UML Hierarchical State Machine
* Intuitive Hierarchical State Machine Programming Specification