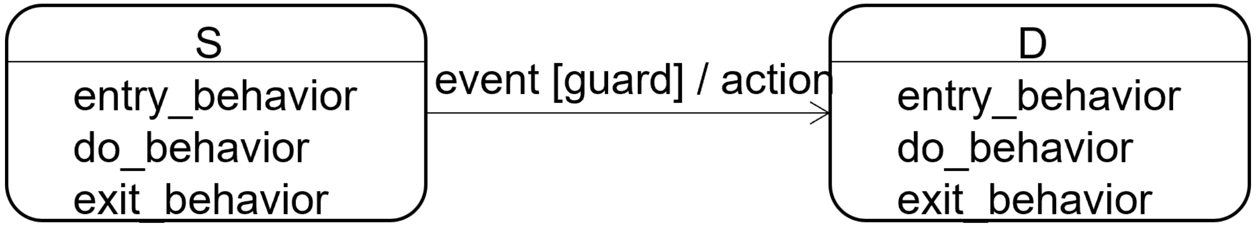
Declarative Representation of UML State Machines for Querying and Simulation

FAACS 2023

Authors: Zohreh Mehrafrooz, Ali Jannatpour, and Constantinos Constantinides

Present by: Zohreh Mehrafrooz

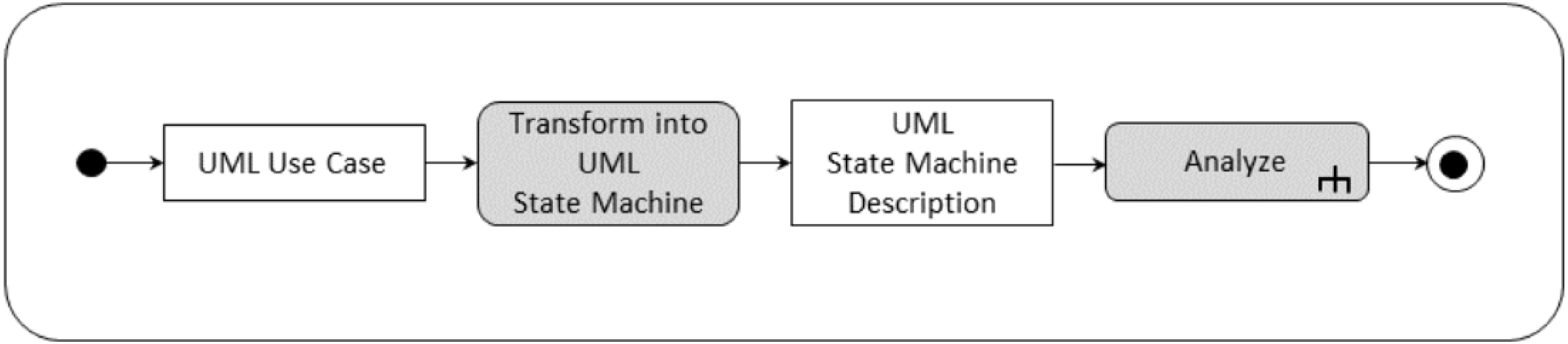
Background: UML state machines

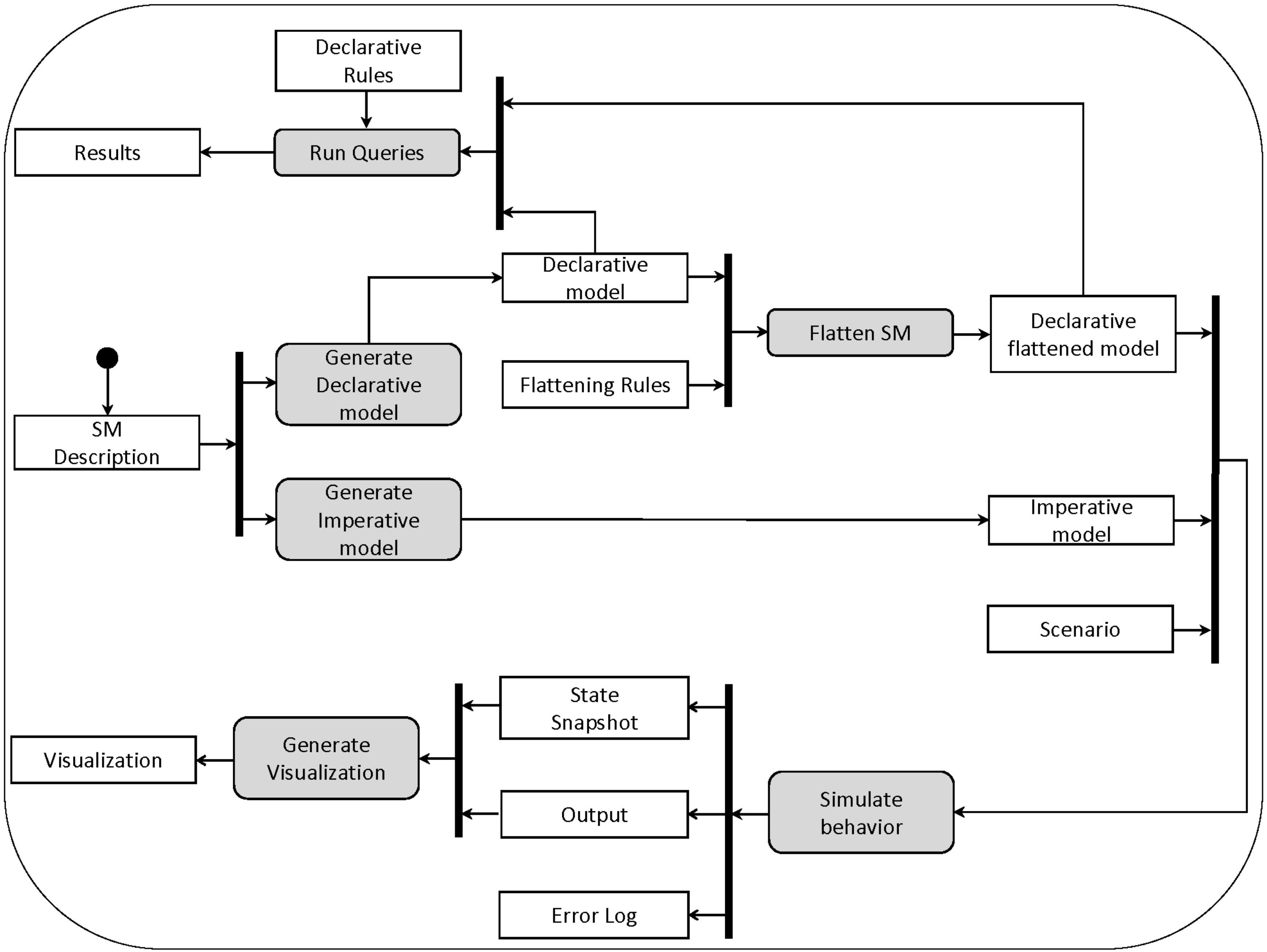


$(Q, \Sigma_1, \Sigma_2, q_0, V, \Gamma, \Lambda)$, where

Q is a finite set of *states*,
Σ_1 is a non-empty finite set of *events*,
Σ_2 is a finite set of *actions*,
$q_0 \in Q$ is the *starting state*,
V is a finite set of *mutable* global *variables*,
Γ is a finite set of *guards*,
$\Lambda = \{\lambda_i : q \xrightarrow{e[g]/a} q'\}$, is a finite set of transitions

Our approach: Top-level activity diagram

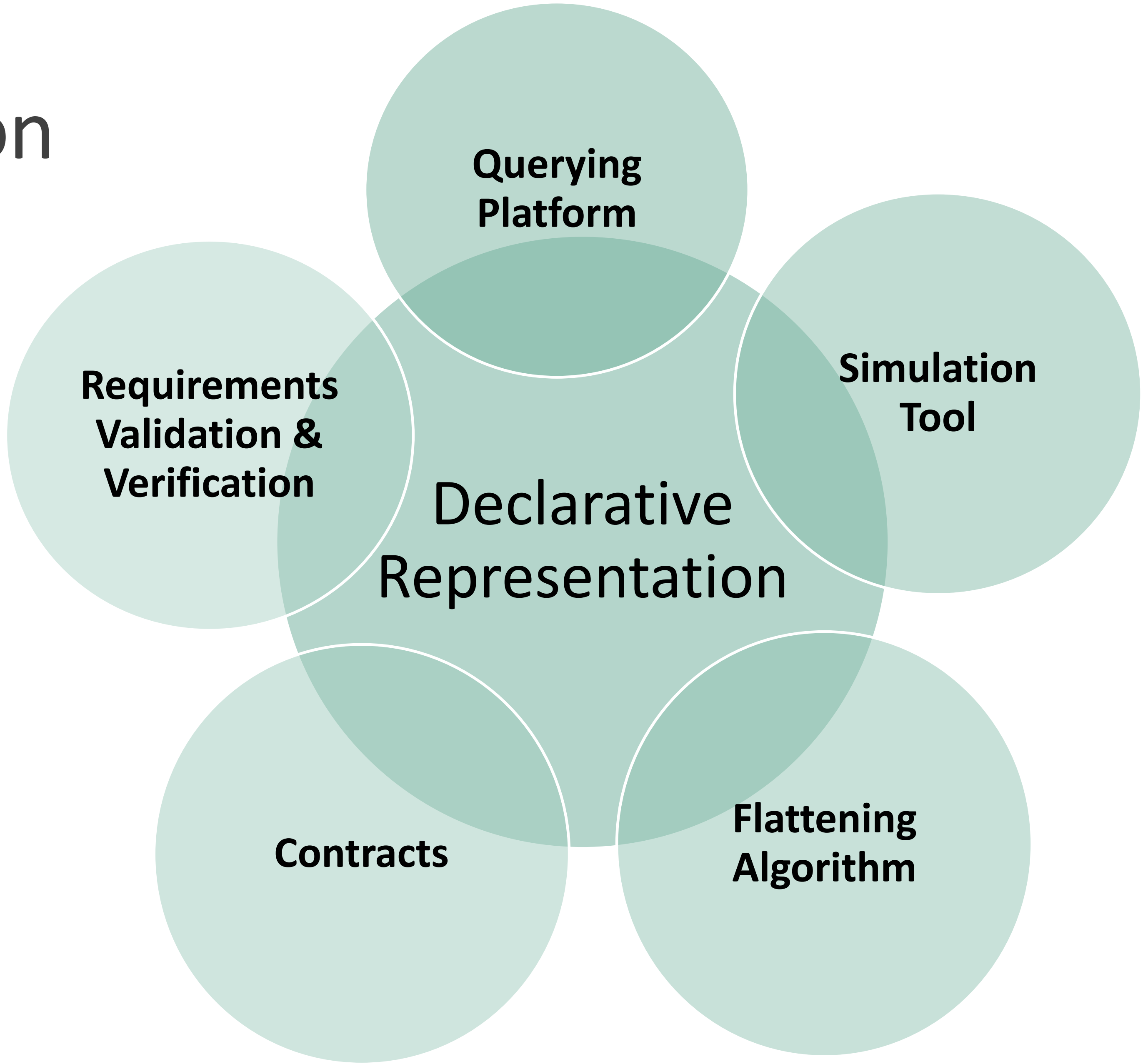



Declarative Rules
Results
Run Queries
Declarative model
Generate Declarative model
Flattening Rules
Flatten SM
Declarative flattened model
SM Description
Generate Imperative model
Imperative model
Scenario
State Snapshot
Visualization
Generate Visualization
Output
Simulate behavior
Error Log

Our motivation and previous works

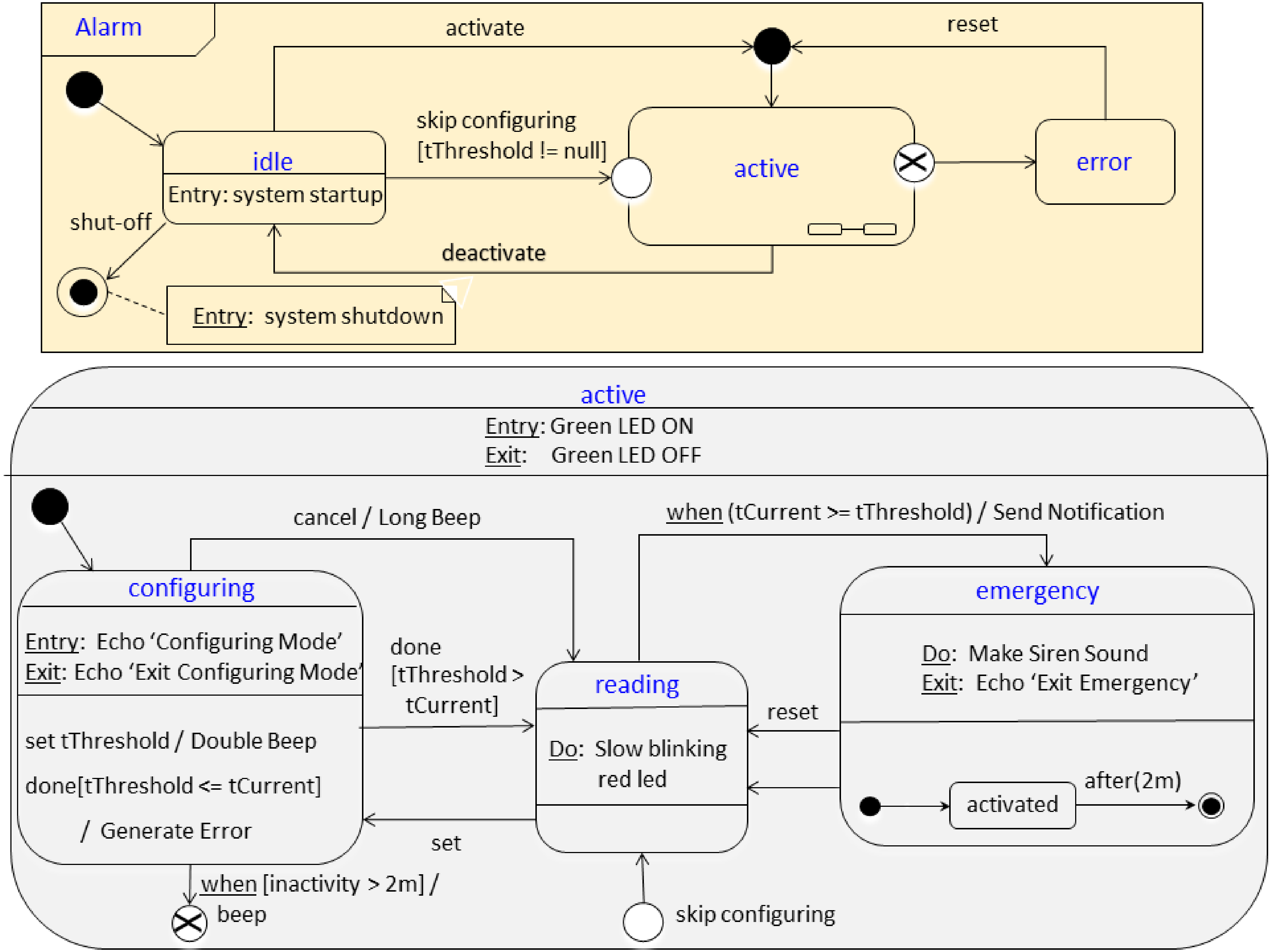
- Sheng et al. [2019] present a Prolog-based consistency checking for UML class and object diagrams.
- Khai et al. [2011] propose a Prolog-based approach for consistency checking of class and sequence diagrams.
- Mens et al. [2020] introduce a technique to improve statechart design by a modular Python library, Sismic.
- Mierlo and Vangheluwe [2019] a present approach for modeling, simulating, testing, and deploying statecharts.
- Balasubramanian et al. [2013] introduce Polyglot, a comprehensive framework for analyzing models described using multiple statechart formalisms.
- E. V. and Samuel [2019] describe a technique to transform hierarchical, concurrent, and history states into Java code using a design pattern-based methodology.

Our contribution



Case study: an alarm system

UML state machine representation



Declarative representation

Model transformation:
Initial declarative representation

```
state/1
initial/1
alias/2
final/1
```

FACT	DESCRIPTION
entry_pseudostate/2	entry_pseudostate(?Entry, ?Substate) implies that ?Substate is the target inner-state whose superstate is already defined by superstate(?Superstate, ?Substate).
exit_pseudostate/2	exit_pseudostate(?Exit, ?Superstate) implies that ?Exit is an exit state within the superstate ?Superstate.
superstate/2	superstate(?Superstate, ?Substate) implies that ?Superstate is a composite state with ?Substate being a nested state.
onentry_action/2	onentry_action(?Name, ?Action) implies that ?Name defines ?Action as an entry behavior.
onexit_action/2	onexit_action(?Name, ?Action) implies that ?Name defines ?Action as an exit behavior.
do_action/2	do_action(?Name, ?Proc) implies that ?Name defines ?Proc as a do behavior.
transition/5	transition(?Source, ?Destination, ?Event, ?Guard, ?Action) indicates that while the system is in state ?Source, should ?Event occur and with ?Guard being true, the system performs a transition to state ?Destination while performing ?Action. All elements of the triple (?Event, ?Guard, ?Action) are optional, and the absence of an element is codified as nil.
internal_transition/4	internal_transition(?State, ?Event, ?Guard, ?Action) indicates that while the system is in ?State, should ?Event occur and with ?Guard being true, the system performs ?Action. In the triple (?Event, ?Guard, ?Action), only ?Guard is optional, the absence of which is codified as nil.
event/2	event(?Type, ?Argument) indicates an event where ?Type shows event type and ?Argument is a literal.
action/2	action(?Type, ?Argument) indicates an action where ?Type shows action type and ?Argument is a literal.

Model transformation: Event and action types

```
event/2

event(?Type, Argument)
```

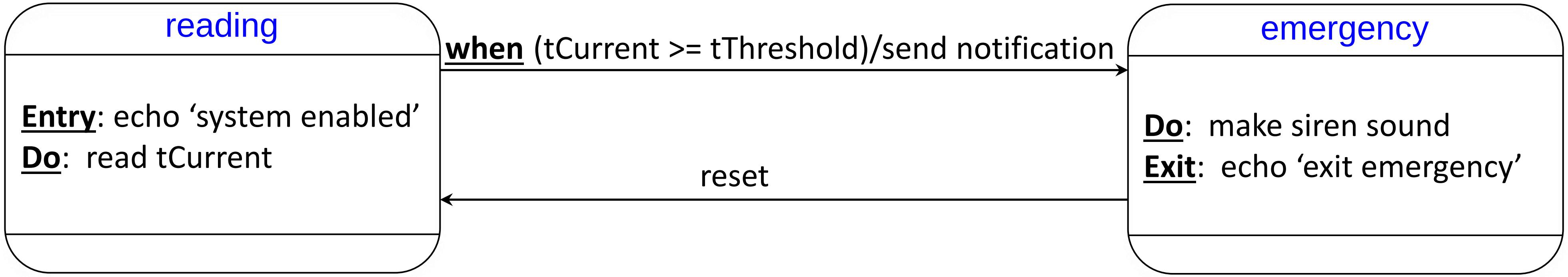
Event types
- call
- signal
- time
- change

- inactivity
- update
- completion

```
action/2

action(?Type, ?Argument)
```

Action types
- exec
- log

Model transformation of a state machine into a declarative model: An example

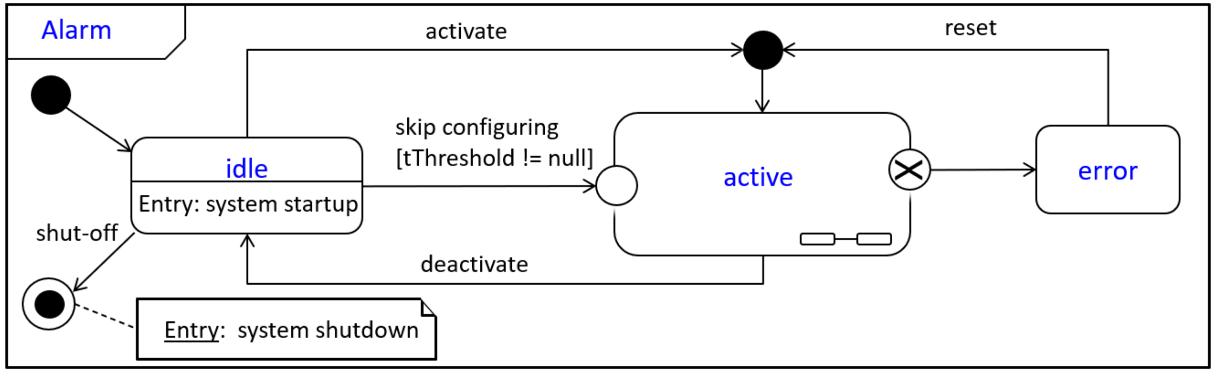


- The clause `transition/5` is codified as

```
transition(?Source, ?Target, ?Event, ?Guard, ?Action).
```

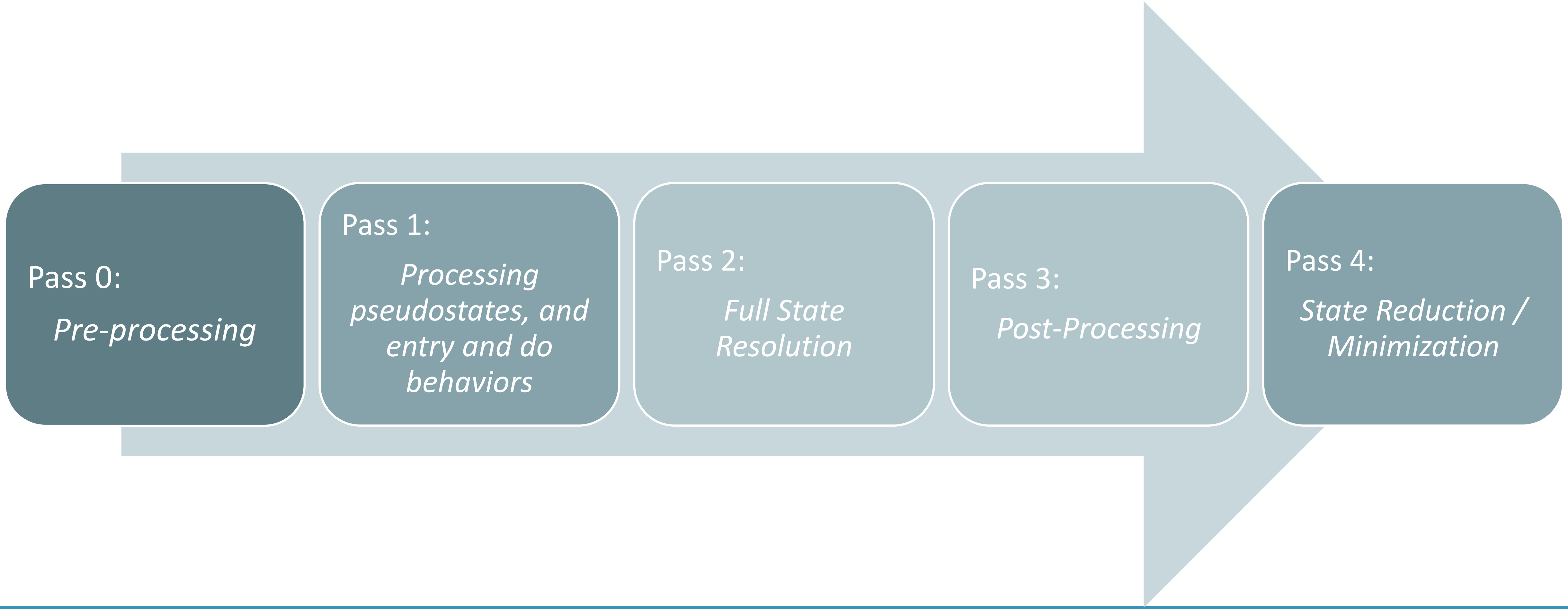
```
transition(reading, emergency, event(when, “tCurrent >= tThreshold”), nil,
                                                   action(exec, “send notification“)).
transition(emergency, reading, event(call, reset), nil, nil).
```

Model transformation:
Initial declarative representation

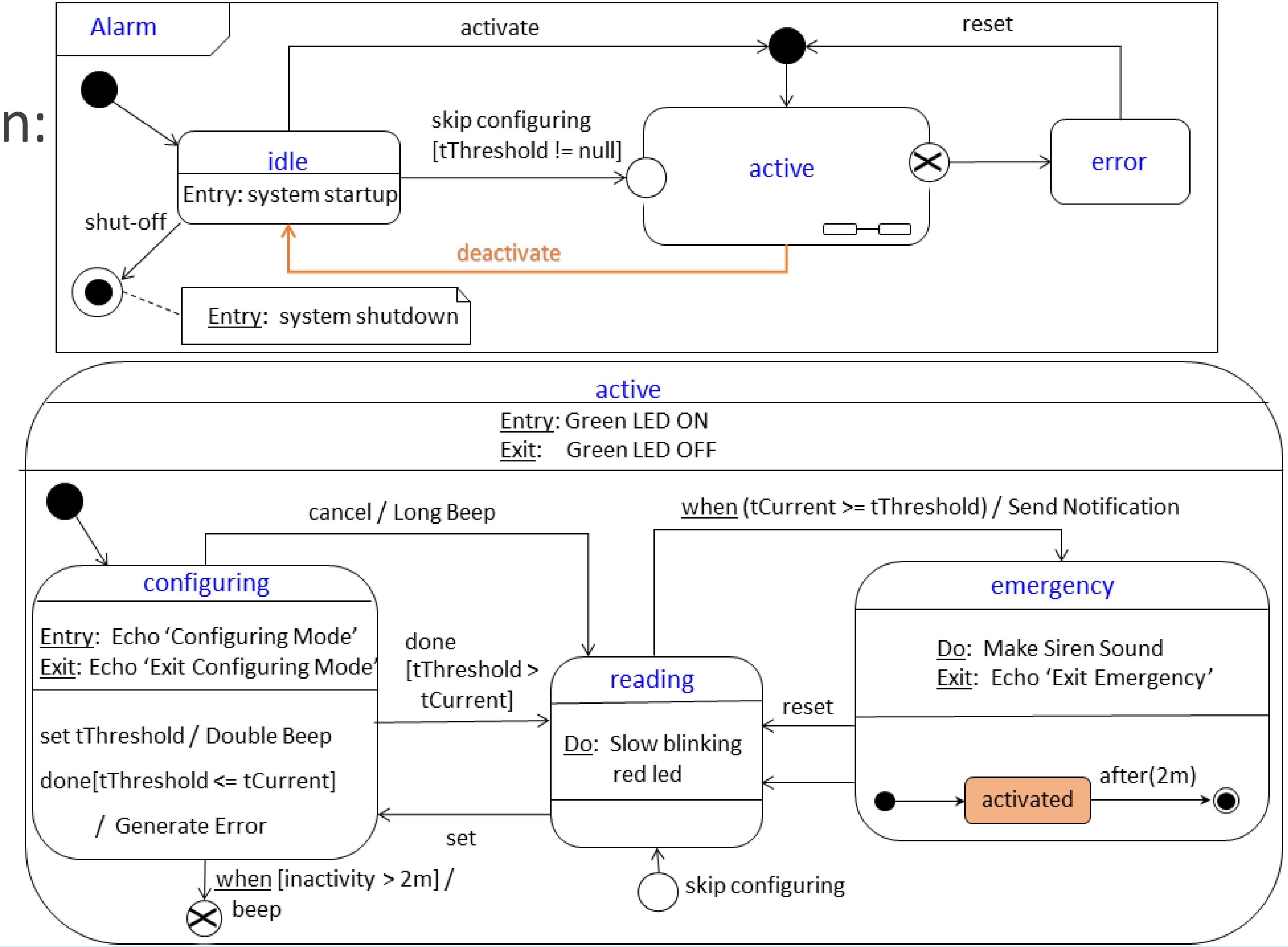



```
% top level
state(idle).
state(active).
state(error).
state(final).
initial(idle).
final(final).
alias(final, "").
entry_pseudostate(active_skip_config_entry, reading). % active superstate is implied
exit_pseudostate(active_exit, active).
transition(idle, active, event(call, activate), nil, nil).
transition(idle, active_skip_config_entry, event(call, "skip configuring"), nil, nil).
transition(error, active, event(call, reset), nil, nil).
transition(active, idle, event(call, deactivate), nil, nil).
transition(idle, final, event(call, shutoff), nil, nil).
transition(active_exit, error, nil, nil, nil). % see exit_pseudostate
onentry_action(idle, action(log, "System Startup")).
onentry_action(final, action(log, "System Shutdown")).
```

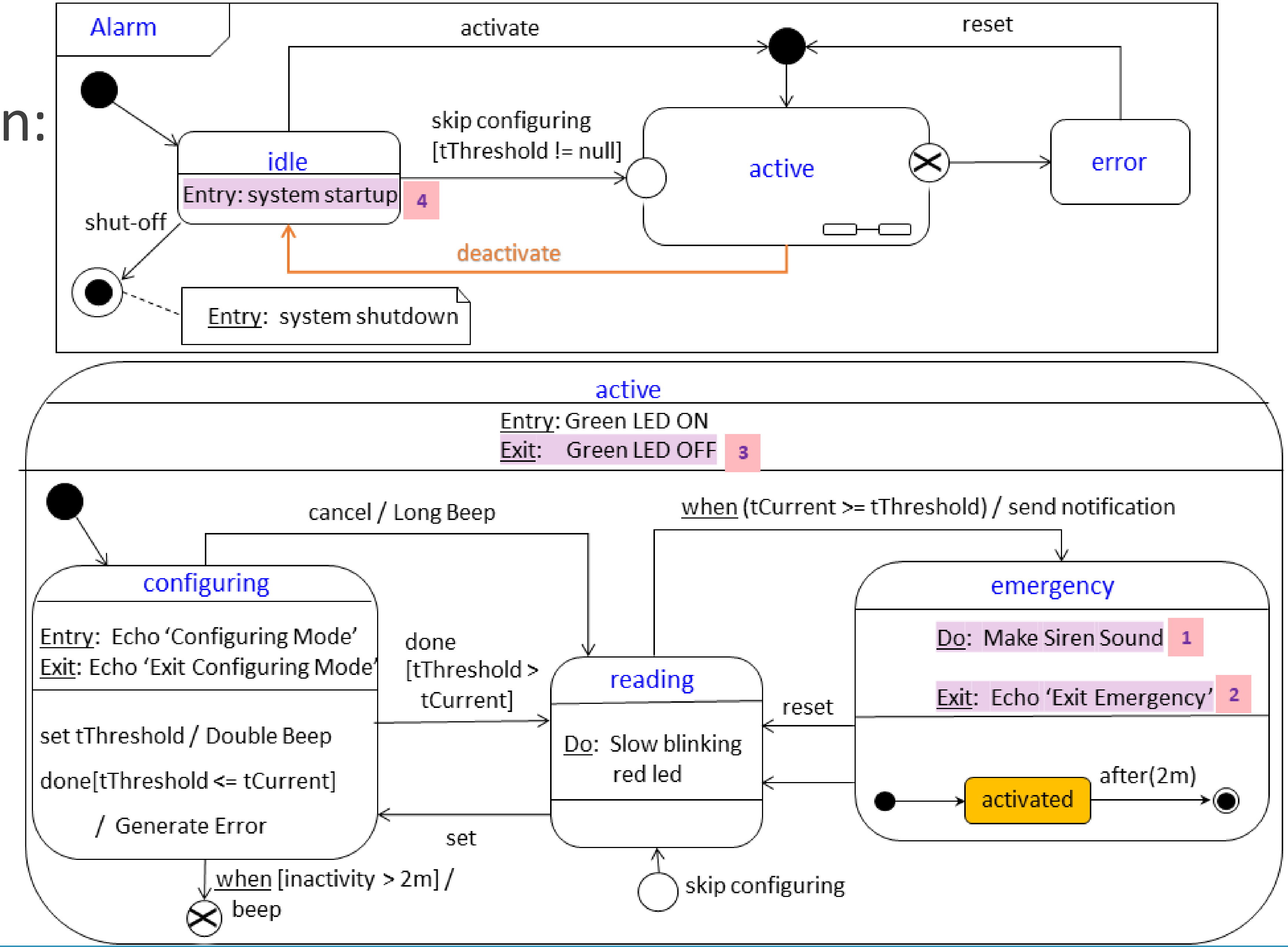
Flattening a UML state machine: Overview of the algorithm



Flattened representation: An example



Flattened representation: An example



Flattened declarative representation: An example

```
transition(activated, s21, event(call, deactivate), nil,
                action(log, "ABORT 'Make Siren Sound'")).
transition(s21, s22, nil, nil, action(exec, "echo('Exit Emergency');")).
transition(s22, pre_idle, nil, nil, action(log, "Green LED OFF")).
transition(pre_idle, idle, nil, nil, action(log, "System Startup")).
```

Model transformation: Flattened declarative representation

- state/1
- initial/1
- transition/5
- event/2
- action/2
- final/1

```
state(idle).        state(error).        state(final).
state(pre_idle).    state(configuring). state(reading).
state(active_exit). state(activated).    state(efinal).
state(s11).         state(s12).          state(s21).
state(s22).         state(s31).          state(s41).
state(s71).         state(s91).          state(s92).
initial(pre_idle).  final(final).
alias(final, "").   alias(efinal, "").
transition(activated, s11, event(call, reset), nil, action(log, "ABORT 'Make Siren Sound'")).
transition(s11, s12, nil, nil, action(exec, "echo('Exit Emergency');")).
transition(s12, reading, nil, nil, action(log, "START 'Slow blinking red LED'")).
transition(activated, s21, event(call, deactivate), nil, action(log, "ABORT 'Make Siren Sound'")).
transition(s21, s22, nil, nil, action(exec, "echo('Exit Emergency');")).
transition(configuring, s31, event(timeout, "2:00"), nil,
    action(exec, "echo('Exit configuring mode');")).
transition(s31, active_exit, nil, nil, action(exec, "beep();")).
transition(configuring, s41, event(call, cancel), nil,
    action(exec, "echo('Exit configuring mode');")).
transition(reading, s71, event(call, set), nil, action(log, "ABORT 'Slow blinking red LED'")).
transition(s71, configuring, nil, nil, action(exec, "echo('Configuring mode');")).
transition(reading, s91, event(when, "tCurrent >= tThreshold"), nil,
    action(log, "ABORT 'Slow blinking red LED'")).
transition(s91, s92, nil, nil, action(exec, "sendNotification();")).
transition(s92, activated, nil, nil, action(log, "START 'Make Siren Sound'")).
transition(idle, final, event(call, shutoff), nil, action(log, "System Shutdown")).
transition(activated, efinal, event(after, "2:00"), nil, nil).
transition(pre_idle, idle, nil, nil, action(log, "System Startup")).
transition(active_exit, error, nil, nil, action(log, "Green LED OFF")).
transition(configuring, configuring, event(set, tThreshold), nil, action(exec, "doubleBeep();")).
transition(configuring, configuring, event(call, done),
    "tThreshold <= tCurrent", action(exec, "generateError();")).
transition(s41, s12, nil, nil, action(exec, "longBeep();")).
transition(configuring, s12, event(call, done),
    "tThreshold > tCurrent", action(exec, "echo('Exit configuring mode');")).
transition(idle, s12, event(call, "skip configuring"), nil, action(log, "Green LED ON")).
transition(idle, s71, event(call, activate), nil, action(log, "Green LED ON")).
transition(error, s71, event(call, reset), nil, action(log, "Green LED ON")).
transition(s22, pre_idle, nil, nil, action(log, "Green LED OFF")).
transition(activated, s11, event(completed, emergency), nil,
    action(log, "STOP 'Make Siren Sound'")).
transition(efinal, s11, nil, nil, action(log, "STOP 'Make Siren Sound'")).
transition(configuring, s22, event(call, deactivate), nil,
    action(exec, "echo('Exit configuring mode');")).
transition(reading, s22, event(call, deactivate), nil,
    action(log, "ABORT 'Slow blinking red LED'")).
```

Querying

Building a query platform

Study the behavior
- exposed interface
- legal events at a state

Study quality attributes
- rooted, connectivity
- order, size, degree

Study the well-formedness
- dead ends, infinite loops
- non mutually exclusive guards

Studying the behavior

```
get_interface(Interface) :-
   findall(Event,
           (transition(_, _, event(call, Event), _, _) ;
            transition(_, _, event(set, Event), _, _)),
            EventList),
```

```
is_legal(State, Event) :-
           transition(State, _, event(_, Event), _, _);
           internal_transition(State, event(_, Event), _, _).
```

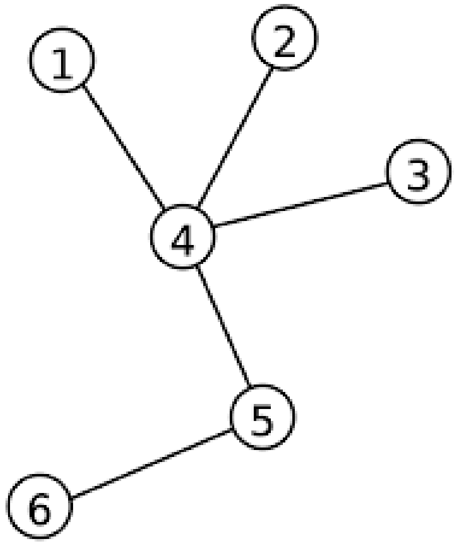
Studying quality attributes

```
order(N) :-
   findall(
      State,
      (state(State); superstate(_, State)),
      StateList
    ),
   list_to_set(StateList, States),
   length(States, N).
```

Initial Model

Flattened Model

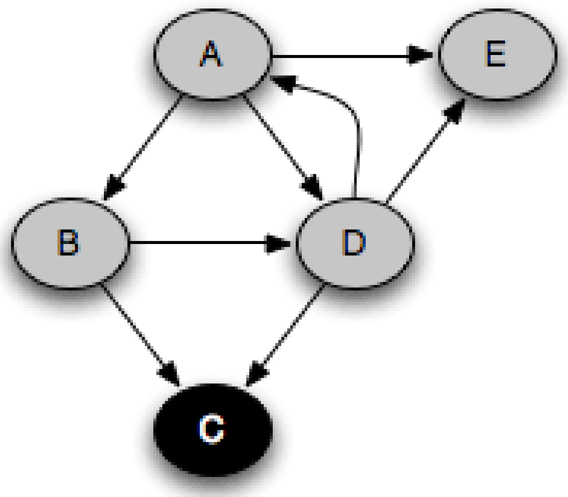
```
order(N):-
   findall(S,
      state(S),
      Length),
   length(Length, N).
```



Studying the well-formedness

```
path(X, Y) :- path(X, Y, [X]).

path(X, Y, V) :- transition(X, Y, _, _, _), \+ member(Y, V).

path(X, Y, V) :- transition(X, Z, _, _, _),

   \+ member(Z, V), path(Z, Y, [Z|V]).
```

```
dead_end :-

   findall(

      State,

      \+path(State, final),

      L),

   L \= [].
```



Simulation

Technologies

Prolog

- Query engine
- Pattern matching

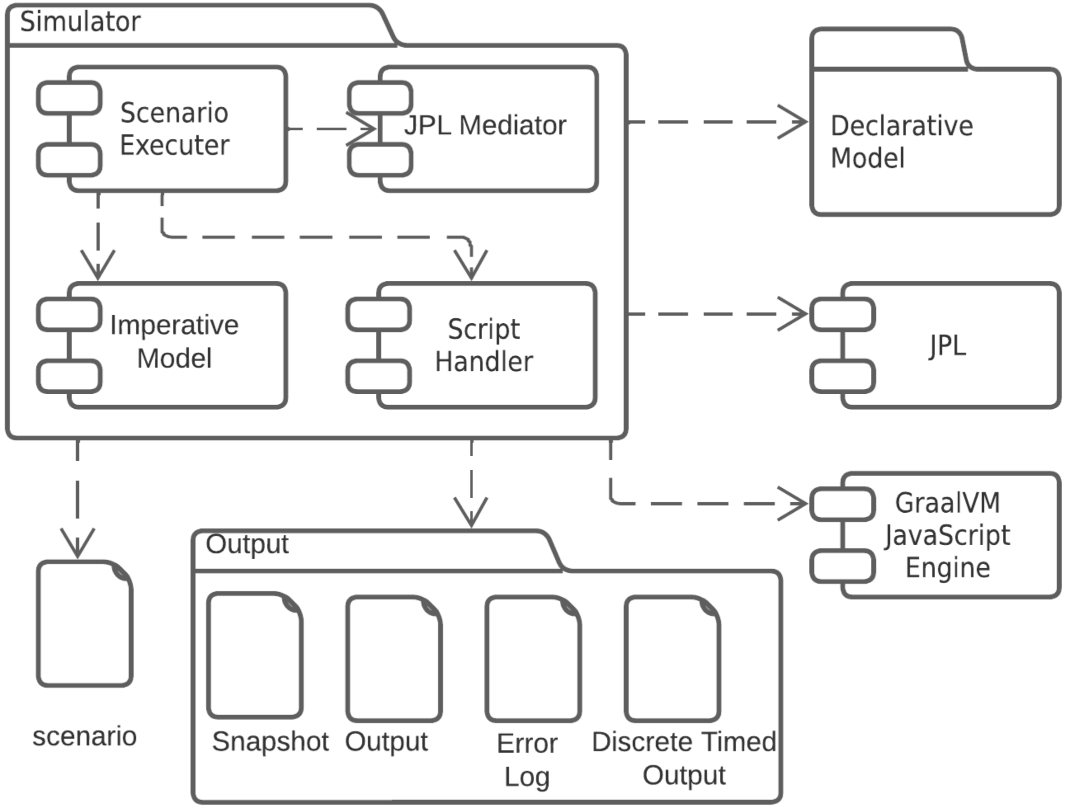
Java

- Iterative code to trace the state machine
- Maintain the snapshots of variables

JavaScript

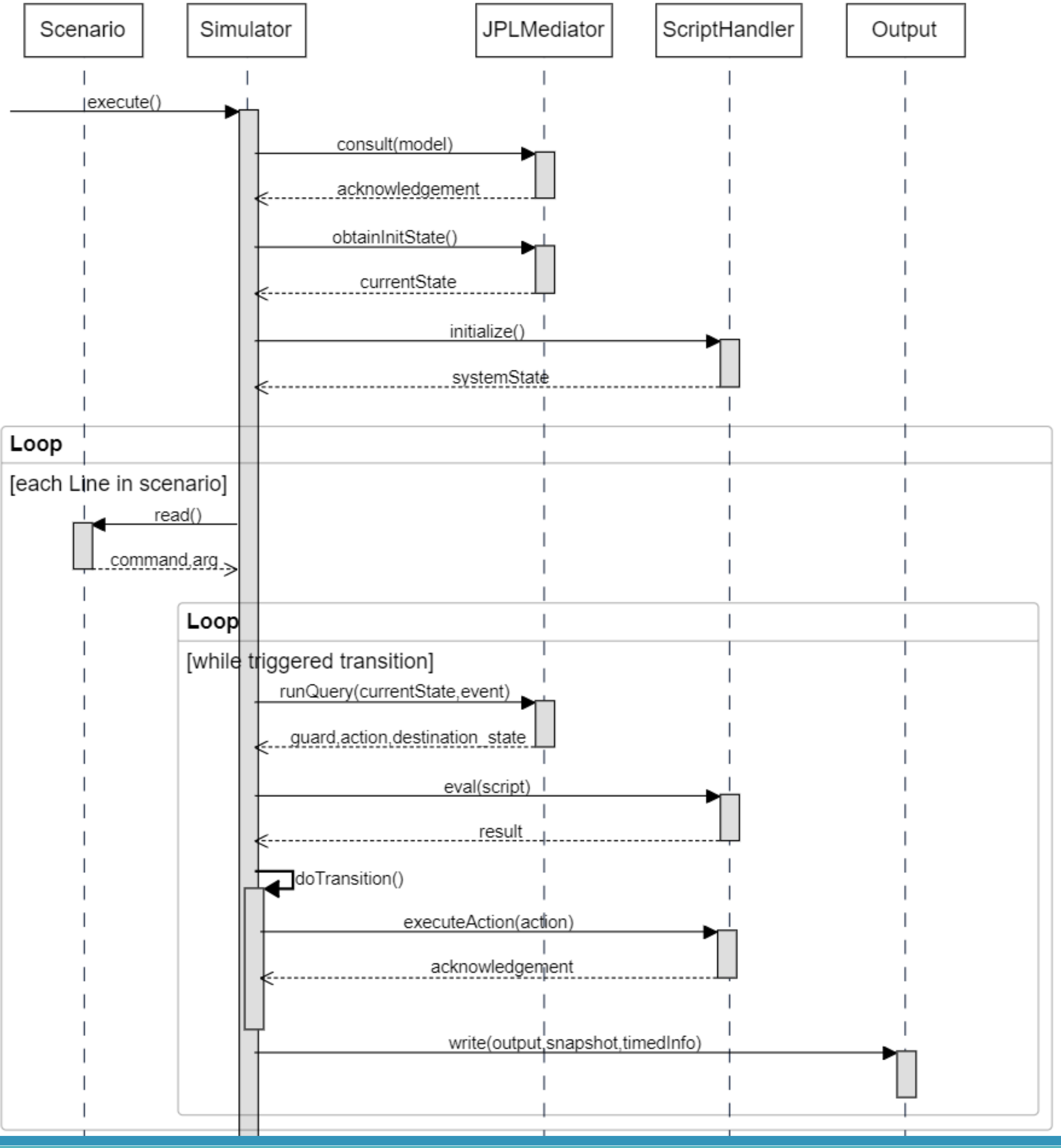
- Evaluate scripts in guards, events, actions
- Direct manipulation of variables in actions

Simulator architecture: UML component diagram

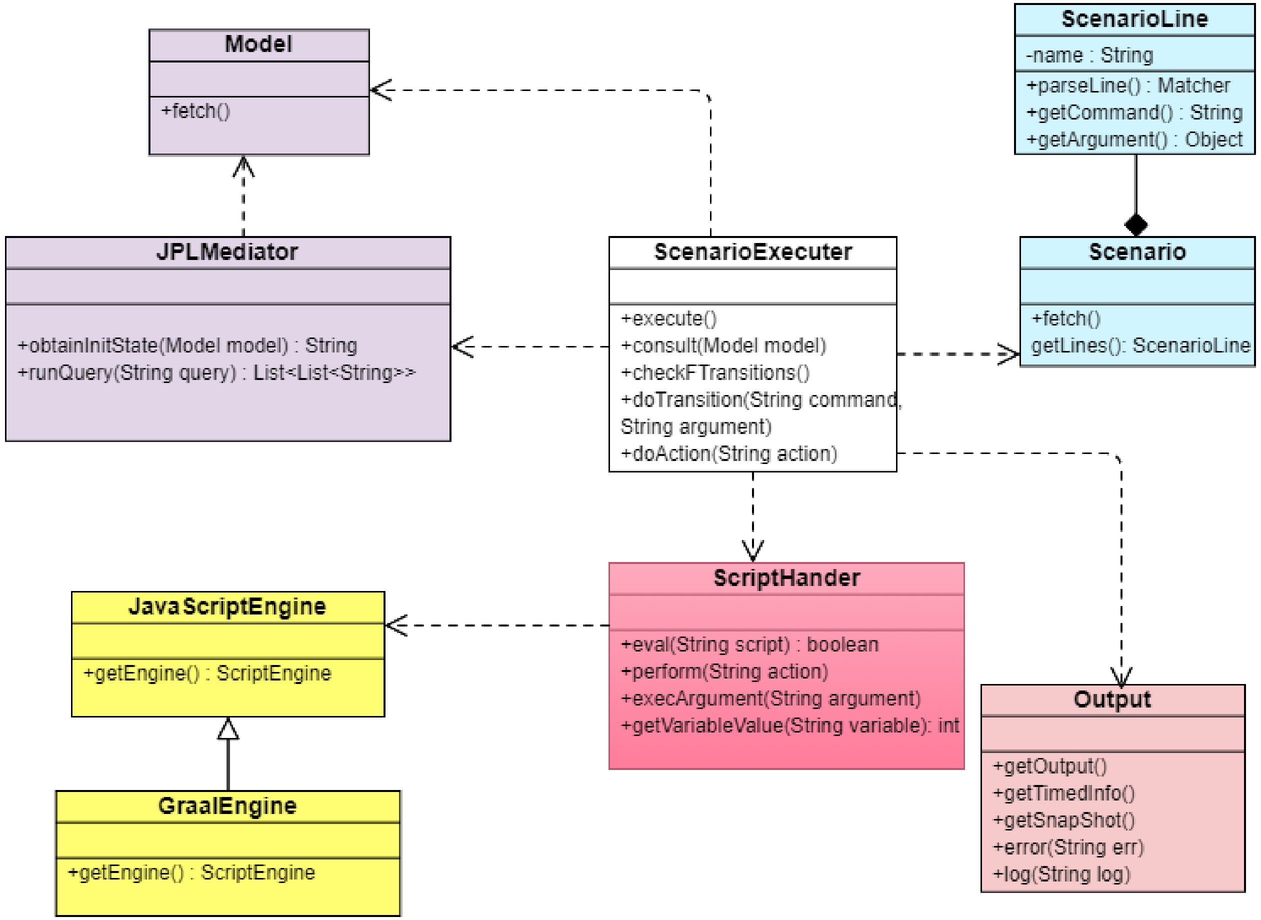


Read-Evaluate-Execute cycle

```
(set current state to initial state)
while (scenario not exhausted)
do
   (read line)
   (transform line into declarative query)
   if (evaluate query) {
      (execute query)
      (obtain results)
   }
   else Error
end
```



Simulator design: UML class diagram



A sample scenario and results of simulation

EVENT
EXECUTE
AT
AFTER

```
1  EVENT call activate
2  EVENT set tThreshold=30
3  AFTER '3:00'
4  EVENT call reset
5  EXECUTE tCurrent=20
6  EVENT call done
7  EXECUTE tCurrent=40
8  EVENT call deactivate
9  EVENT call shutoff
```

timeId,lineNo,type,arguments

```
1,1,call,"activate"
1,1,transition,"pre_configuring"
1,1,actions,"Green LED ON"
1,1,transition,"configuring"
1,1,actions,"EXEC echo('Configuring mode');"
2,2,set,"tThreshold=30"
2,2,transition,"configuring"
2,2,actions,"EXEC doubleBeep();"
3,3,transition,"s3"
3,3,actions,"EXEC echo('Exit configuring mode');"
3,3,transition,"active_exit"
....
```

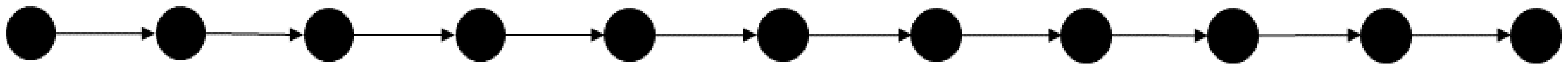
timeId,tThreshold,tCurrent

```
1,30,5
2,30,5
3,30,5
4,30,5
5,30,5
6,30,20
7,30,20
8,30,40
9,30,40
10,30,44
```

timeId,absoluteTime,duration

```
timeId,absoluteTime,duration
1,null,PT0S
2,null,PT0S
3,null,PT3M
4,null,PT3M
5,null,PT0S
6,null,PT0S
7,null,PT0S
8,null,PT0S
9,null,PT0S
10,null,PT0S
```

A sample scenario \cont.: Visualizing the results through a model of behavior



timeId:	0	1	2	3	4	5	6	7	8	9	10
State:	idle	configuring	configuring	error	configuring	configuring	reading	reading	activated	idle	final
tThreshold:	30	30	30	30	30	30	30	30	30	30	30
tCurrent:	5	5	5	5	5	5	20	20	40	40	40

Conclusion

Provide a powerful tool for analyzing the two aspects of the state machines.

- Declarative analysis: query platform.
- Imperative analysis: simulation.

Future work: Expand the model to include contract considerations, and additional UML features such as history pseudostates and orthogonal regions.

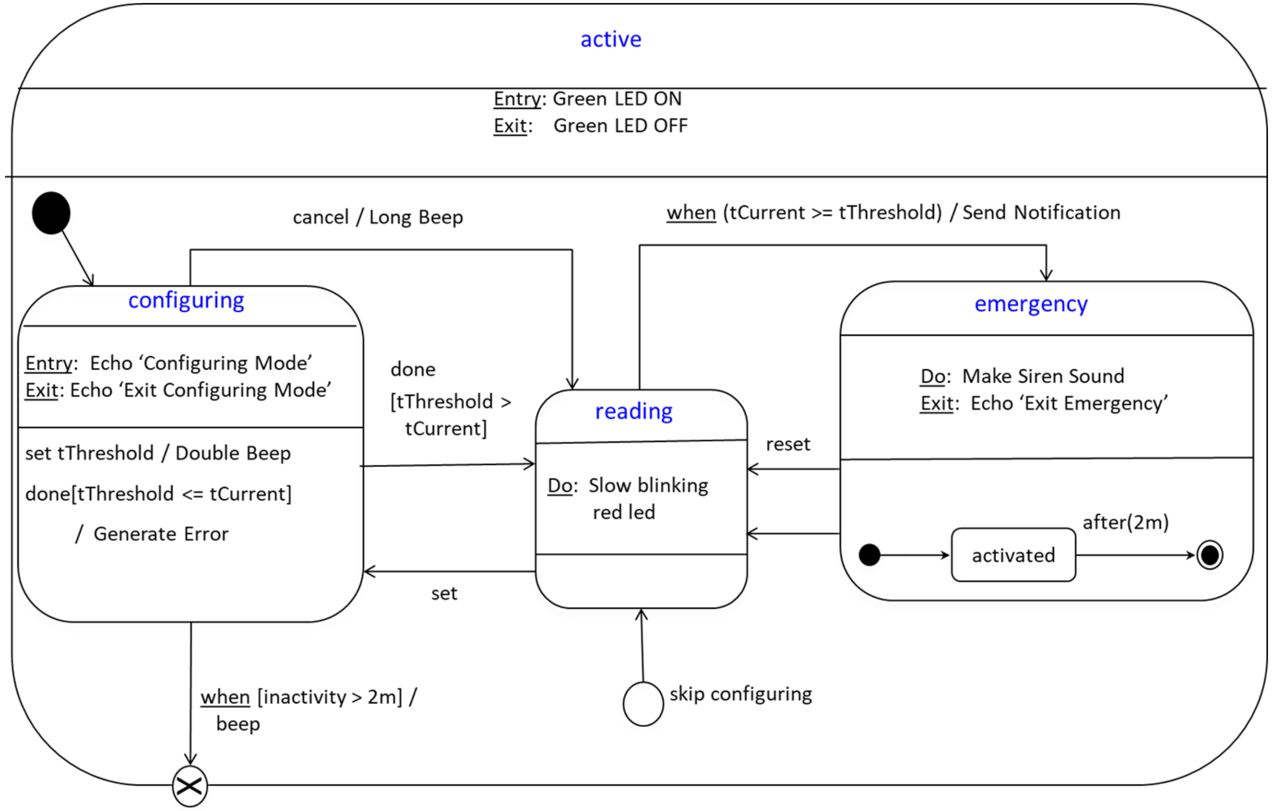
Thank you!

A second scenario and results of simulation

```
1 EVENT call 'skip configuring'
2 EXECUTE tCurrent=0
3 AT '10:00'
4 EXECUTE tCurrent=50
5 EVENT call reset
6 AT '11:00'
7 EXECUTE tCurrent=40
8 EVENT call reset
9 AFTER '3:00'
10 EXECUTE tCurrent=30
11 EVENT call reset
12 EVENT call set
13 EVENT call deactivate
14 EVENT call shutoff
```

```
timeId,LineNo,type,arguments
1,1,EVENT,call,"skip configuring"
1,1,transition,active_skip_config_entry
1,1,action,"Green LED ON"
1,1,transition,reading
1,1,action,"START 'Slow blinking red LED'"
2,2,EXECUTE,"tCurrent=0"
3,4,EXECUTE,"tCurrent=50"
4,4,transition,s91
4,4,action,"ABORT 'Slow blinking red LED'"
4,4,transition,s92
4,4,action,"EXEC sendNotification();"
4,4,transition,activated
4,4,action,"START 'Make Siren Sound'"
5,5,EVENT,call,reset
5,5,transition,s61
…
```

```
Jul. 10, 2023 11:02:44 P.M. ScenarioExecuter doTransition
SEVERE: Error: Number of transitions with true guards is zero!

Process finished with exit code 0
```



Contract Consideration

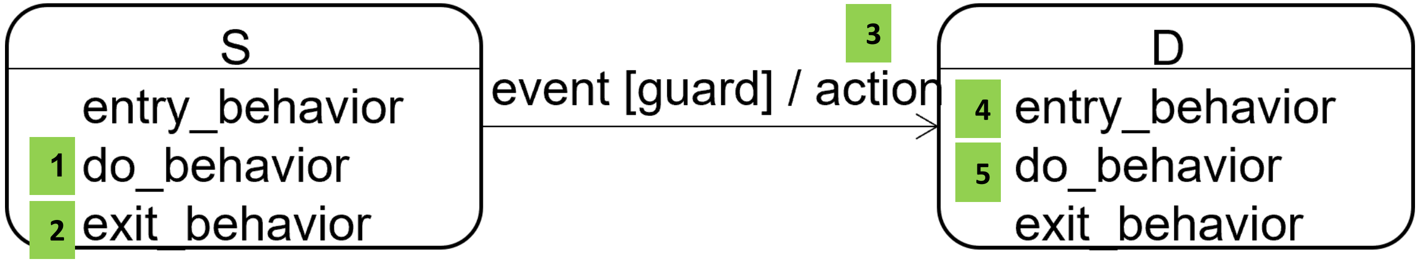
Contract consideration

Assertions on actions

Orthogonality of actions

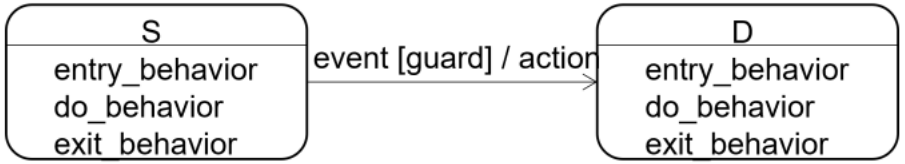
Global and state invariant properties

$$\lambda \in \Lambda : q \xrightarrow{\mathrm{e\ [g]\ /\ \{P\}\ a\ \{Q\}}} q'$$



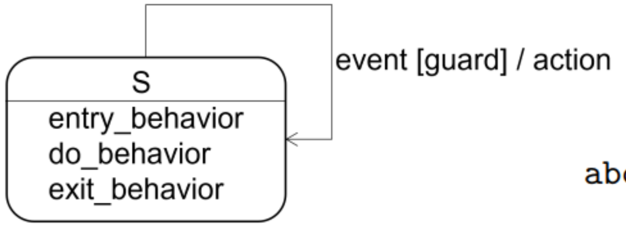
External transition

```
abort S.do_behavior
evaluate S.invariant + global invariant
evaluate S.exit_behavior.pre-condition
execute S.exit_behavior
evaluate S.exit_behavior.post-condition
evaluate global invariant
evaluate action.pre-condition
execute action
evaluate action.post-condition
evaluate global invariant
evaluate D.entry_behavior.pre-condition
execute D.entry_behavior
evaluate D.entry_behavior.post-condition
evaluate D.invariant + global invariant
evaluate D.do_behavior.pre-condition
execute D.do_behavior
evaluate D.invariant + global invariant
```

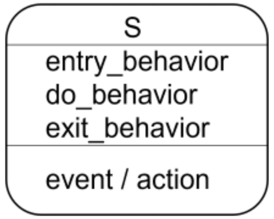


Recursive transition – Internal transition

```
abort do_behavior
evaluate S.invariant + global invariant
evaluate exit_behavior.pre-condition
execute exit_behavior
evaluate exit_behavior.post-condition
evaluate global Invariant
evaluate action.pre-condition
execute action
evaluate action.post-condition
evaluate global invariant
evaluate entry_behavior.pre-condition
execute entry_behavior
evaluate entry_behavior.post-condition
evaluate S.invariant + global invariant
evaluate do_behavior.pre-condition
execute do_behavior
evaluate S.invariant + global invariant
```




```
abort do_behavior
evaluate S.invariant + global invariant
evaluate action.pre-condition
execute action
evaluate action.post-condition
evaluate S.invariant + global invariant
evaluate do_behavior.pre-condition
execute do_behavior
evaluate S.invariant + global invariant
```



Incorporating contracts in declarative model

assert(?State, ?Invariant)

assert(globalSM, “tThreshold>= 0”).
assert(reading, “tThreshold!=null”).
assert(emergency, “tThreshold<=tCurrent”).

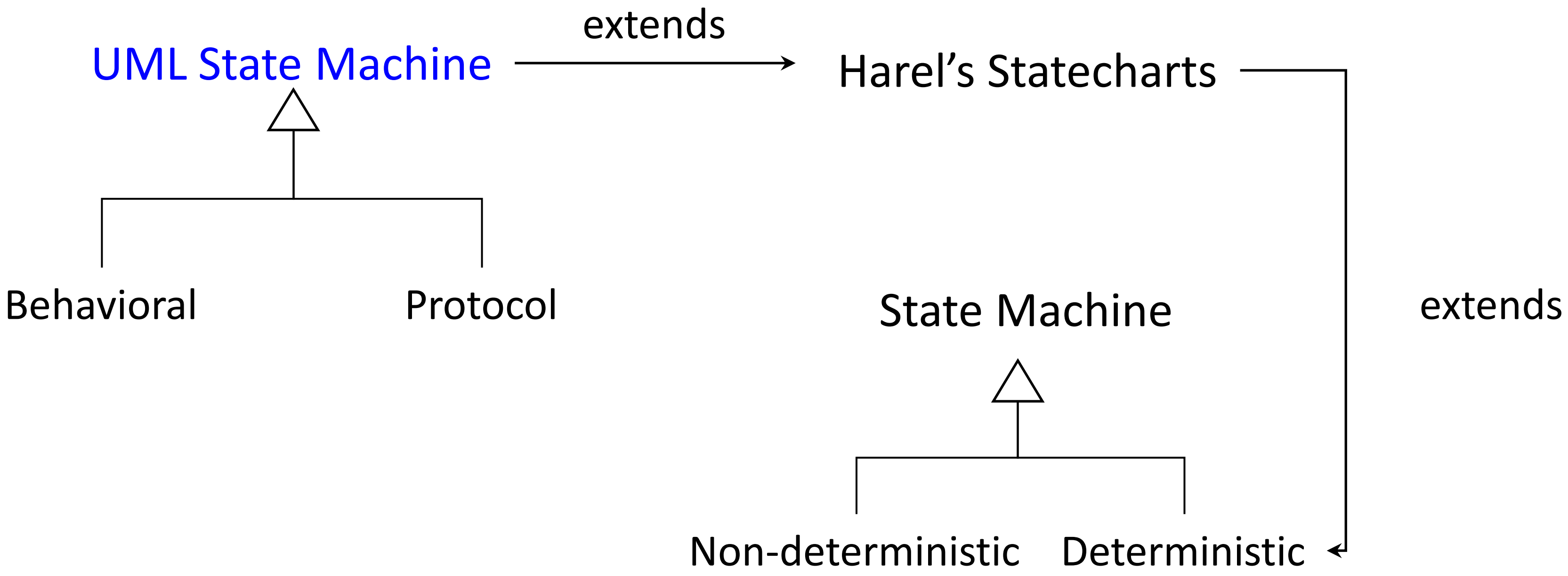
action(?Type, ?Name, ?Pre-condotion, ?Post-condition)

action(exec, "generateError();", "tThreshold!=0", "tThreshold=40").

From Harel's statecharts to (UML) state machines

- Originally introduced by Gill (1962) and later proposed by Harel in 1984 as a significant extension over traditional (deterministic) finite state machines, a statechart is a formalism to model the dynamic behavior of a component at any level of abstraction like e.g. an object, a system unit, a use case, or the entire system itself.
- The Unified Modeling Language adopted Harel's statecharts in its specification and extended them.
- A state transition diagram is the visual counterpart of a state machine.

Evolution of state machines



- This study is on the extended statechart model which is part of the OMG UML specification, referred to in the literature as UML state machine (or UML statechart) and referred to throughout the presentation as state machine.

Overloaded terminology: An attempt to find order

- A state machine (also: finite-state machine, finite (state) automaton), is a mathematical model of computation, and it has two representations:
  - Mathematical representation (see next), and
  - Visual representation.
- The visual representation is captured by a state transition diagram (also: State [machine] diagram, statechart [diagram]).