A Common Declarative Language for UML State Machine Representation, Model Transformation, and Interoperability of Visualization Tools

ALI JANNATPOUR AND CONSTANTINOS CONSTANTINIDES

DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING, CONCORDIA UNIVERSITY, MONTREAL, CANADA

THE 25TH INTERNATIONAL CONFERENCE ON FORMAL ENGINEERING METHODS

2~6 DECEMBER 2024 - HIROSHIMA, JAPAN

{ALI.JANNATPOUR | CONSTANTINOS.CONSTANTINIDES}@CONCORDIA.CA

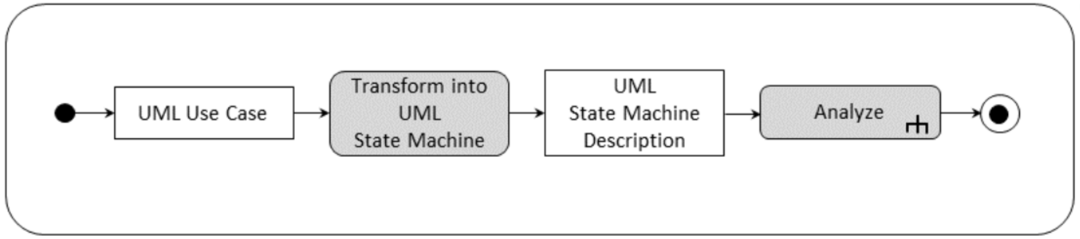
Motivation

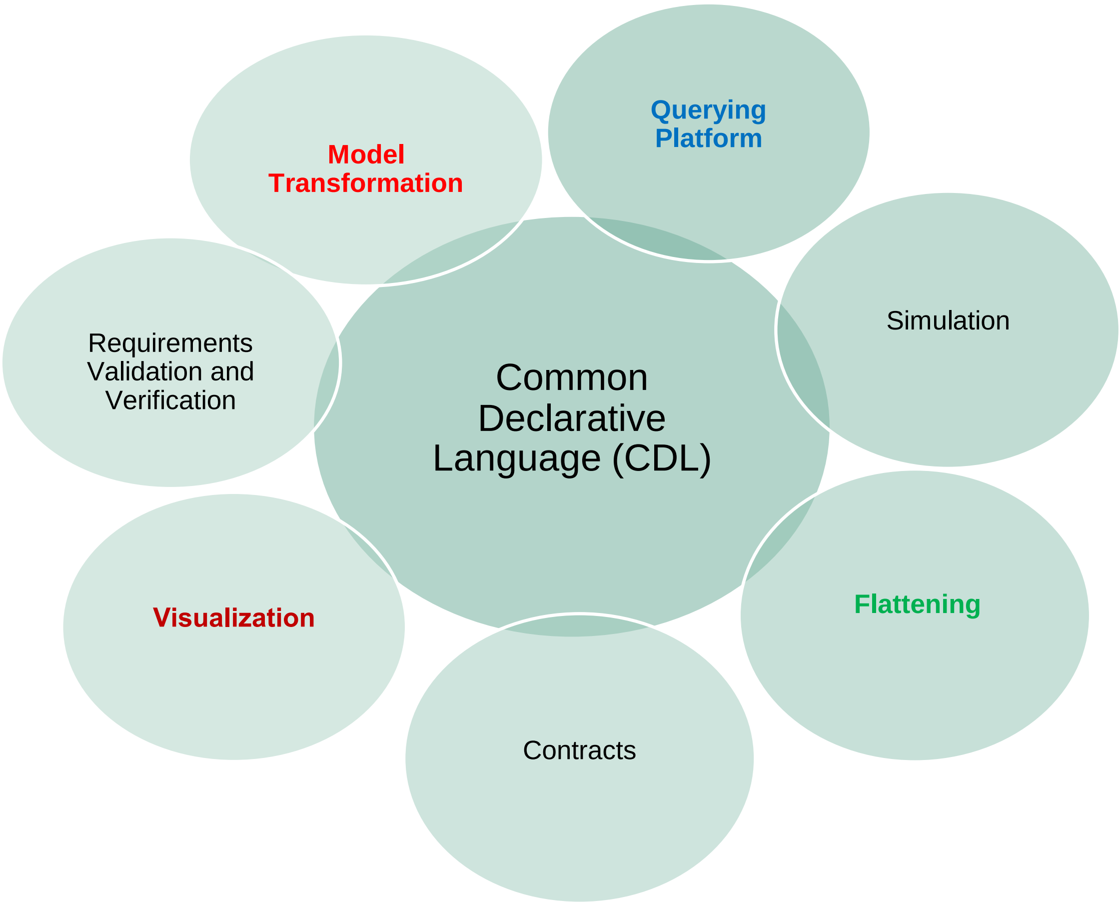
State Machines	Widely used in the field of Software Engineering, including System Modeling, Requirements Specification, Software Testing, etc.
Industry support	Commercial: IBM Rational Rhapsody, The MathWorks Stateflow, … Text-based / Open-source: PlantUML, Mermaid, …
Declarative and Queryable	Prolog seems to be a good fit to be used as the declarative language for UML representation

Related Work

- 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.

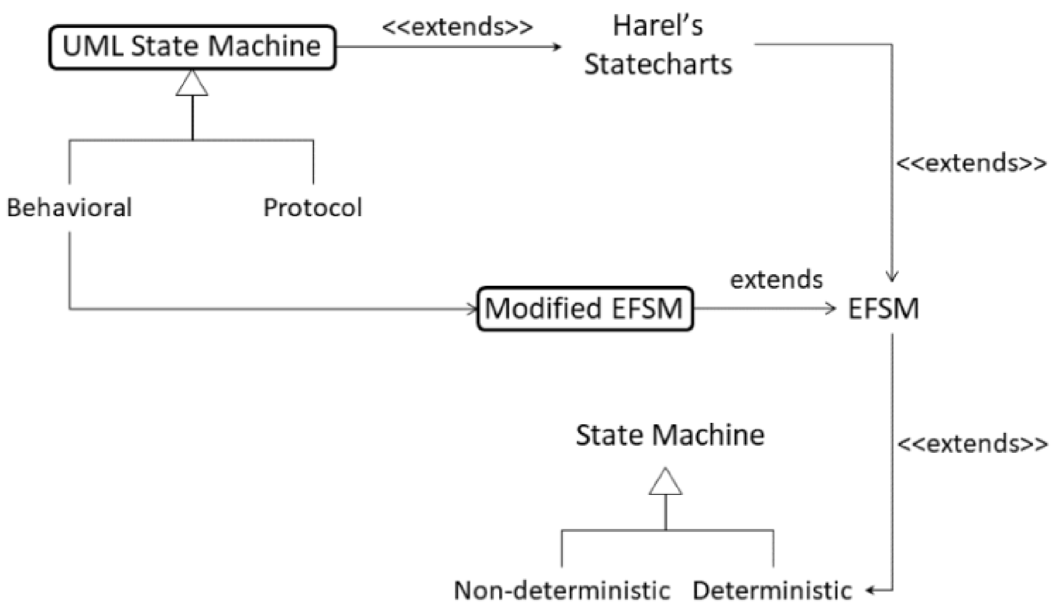
The Common Declarative Language (CDL) as a Platform





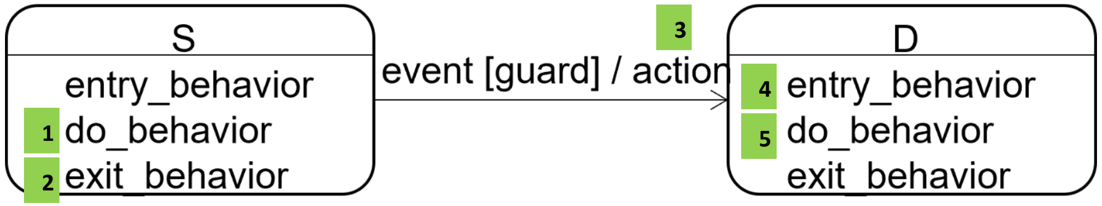
Introduction and Background

- Originally introduced by Gill (1962) and later proposed by **Harel** in 1984 as an 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
- Implemented as Higraph to extend mathematical graphs by including notions of depth and orthogonality.
- Statecharts = state diagrams + depth + orthogonality + broadcast
  - depth / Hierarchy (XOR)
  - orthogonality concurrency (AND)
  - broadcast (events visibility), application in reactive systems
- UML 2.5.1
  - providing numerous complex features, such as composite and nested states; entry and exit pseudostates; entry, exit, and do state behaviors; implicit region completion transition.
- Major Incompatibilities
  - EFSM does not support state behaviors, composite states, and pseudostates.
  - In UML, completion events are not explicitly defined.

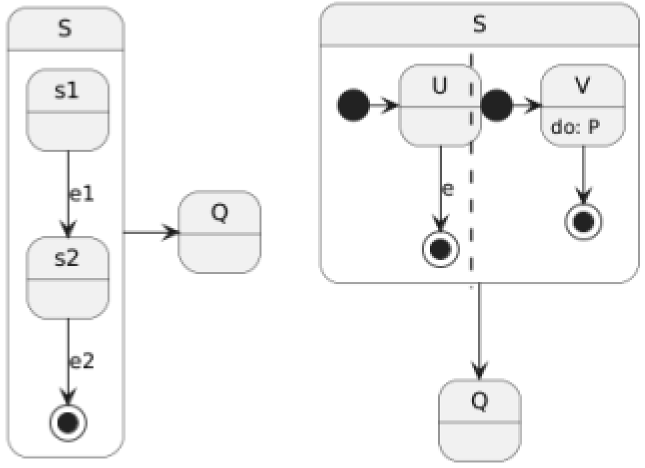


State transition examples

- *A transition between states with behavior*



State transition examples



(a) completion by substate (b) completion by sub-regions

Various types of completion events

event types
- call
- signal
- time
- change

- inactivity
- update
- completion*

Modified EFSM

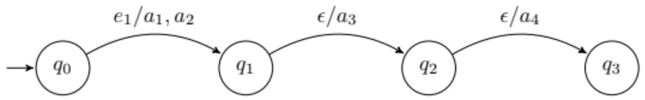
2.1 A Modified Definition of an Extended Finite State Machine

We define an EFSM M, as a 7-tuple $\langle Q, \Sigma_1, \Sigma_2, q_0, V, \Gamma, \rangle$, where

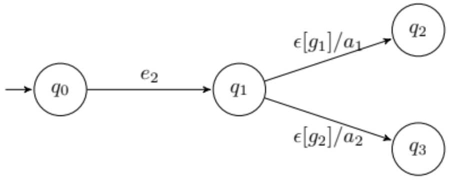
Q is a finite set of *states*.
$\Sigma_1 = \{e_i : i \in \mathbb{Z}\}$, is a non-empty finite set of *events*.
$\Sigma_2 = \{a_i : i \in \mathbb{Z}\}$, is a finite set of *actions*.
$q_0 \in Q$, is the *initial state*.
$V = \{v_i : i \in \mathbb{Z}\}$, is a finite set of *mutable* global *variables*.
$\Gamma = \{g_i : i \in \mathbb{Z}\}$, is a finite set of *guards*.
$\Lambda = \{\lambda : q \xrightarrow{e_i[g_j]/a_k} q', \text{ where } i, j, k \in \mathbb{Z}\}$, is a finite set of *deterministic* transitions defined on $Q \times (\{\epsilon\} \cup \Sigma_1) \times 2^{\Gamma} \to Q \times {\Sigma_2}^{\star}$, where ϵ denotes *null*, $q, q' \in Q$, $e_i \in \{\epsilon\} \cup \Sigma_1$, $g_j \subseteq \Gamma$, is a set of guards, and $a_k \in {\Sigma_2}^{\star}$ (the Kleene closure of Σ_2), is a sequence of actions.

A guarded ϵ-transition is represented by $\lambda : q \xrightarrow{\epsilon[g_j]/a_k} q'$.

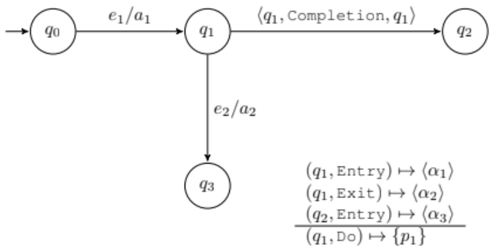
EFSM in action



Sequence of ϵ-transitions in an EFSM



Guarded ϵ-transitions, modeling a choice pseudostate



An equivalent EFSM, demonstrating state behaviors

The Case Study

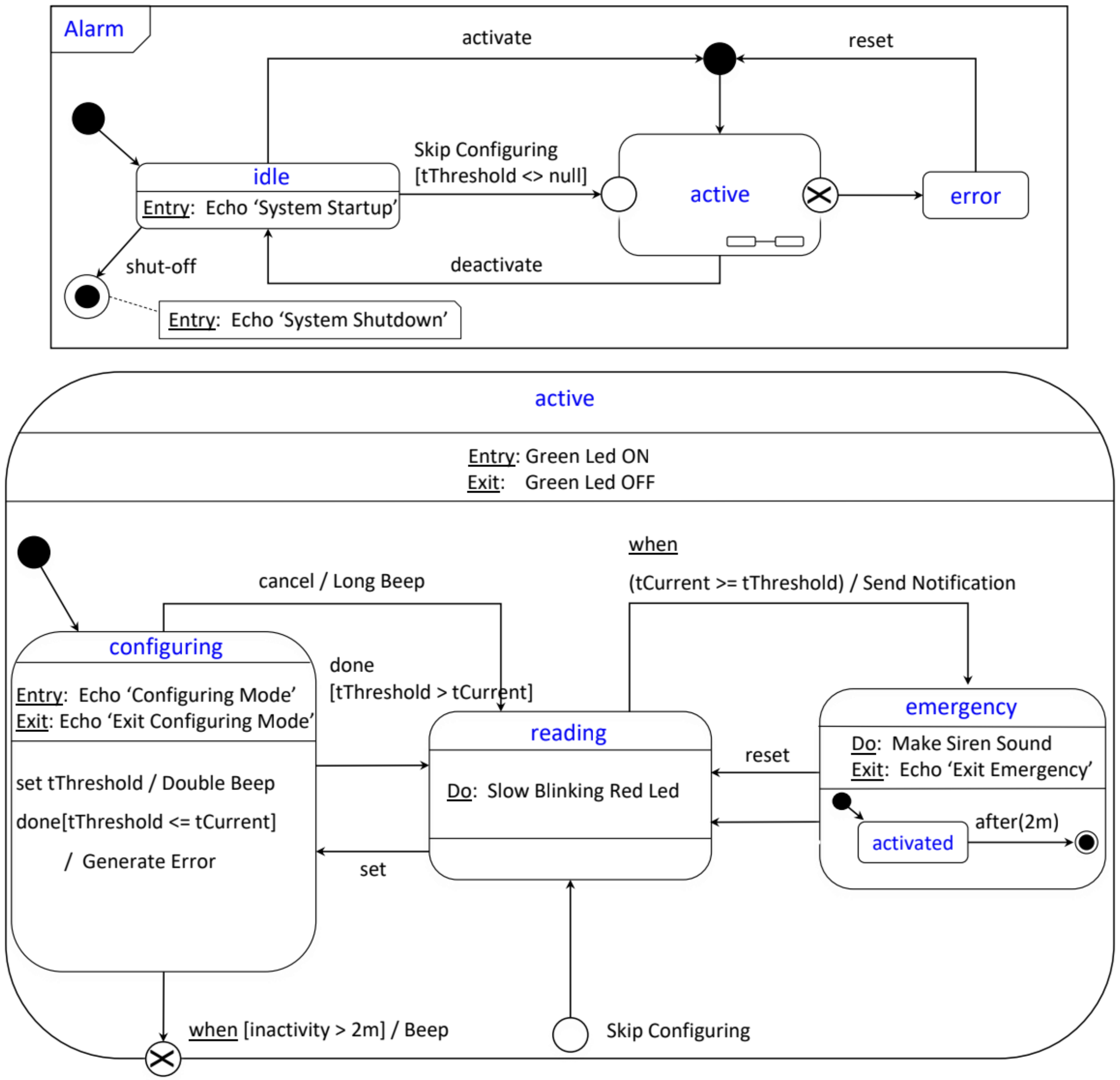


Fig. 1: A sample case-study representing complex UML features.

- Major Features:
  - 3 levels of nestedness
  - Complex nested state behavior
  - Internal and external transitions
  - Entry and exit pseudostates
  - Various events types including implicit completion events

Detailed Case Study Coverage

UML Feature	Coverage in case study
composite state	*active*, *emergency* (nested)
entry behaviour	considered for both simple and composite states in *active*, *configuring*
exit behaviour	considered for both simple and composite states in *active*, *configuring*, *emergency*
do behaviour	considered for both simple and composite states in *reading*, *emergency*
entry point pseudostate	"skip configuring" event in *idle*
exit point pseudostate	when "inactivity > 2m" event in *configuring*
final state (nested)	in *emergency* region
internal transition	"set tThreshold" event in *configuring*
call event	"shut-off", "activate", "deactivate", "skip configuring", "reset" in the highest level of FSM; "done", "set", "cancel", "reset" in *active*
set event	"set tThreshold" in *configuring*
time event	"after 2m" in *emergency* region
completion event	it is covered for both cases. Case 1 is completion of do behaviours in the model. Case 2 is conclusion of *emergency* region
timeout event	"inactivity > 2m" in *active* region
change event	"when [tCurrent > tThreshold]" in *active* region

The Common Declarative Language (CDL)

Features

- Implemented in Prolog
- Queryable & Verifiable
- Extensible

Types

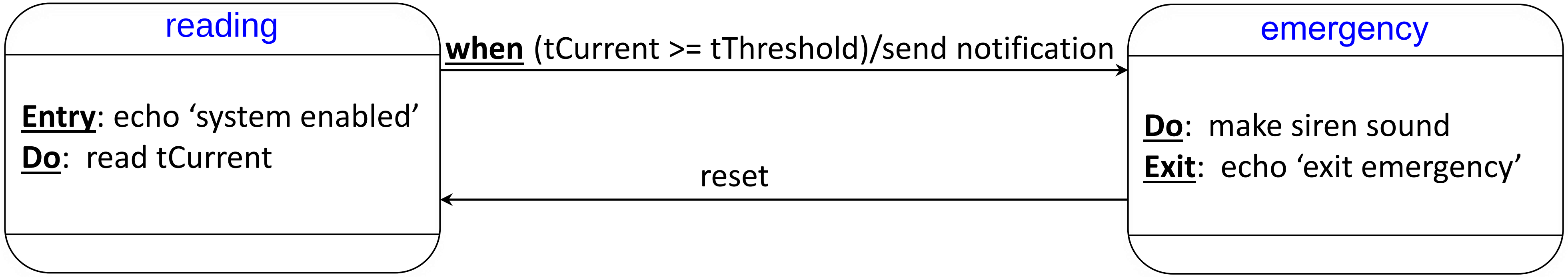
- Simple / Flat (EFSM): simple states and transitions (to be covered by core clauses)
- Complex (UML): composite states, state behaviors, and pseudostates

The Common Declarative Language (CDL)

CLAUSE	DESCRIPTION
state/1	state(?Name) implies that ?Name is a state.
alias/2	alias(?Name, ?Alias) implies that ?Alias is a new name for ?Name.
initial/1	initial(?Name) implies that ?Name is the initial state of the state machine.
final/1	final(?Name) implies that ?Name is the exit (final) state of the state machine.
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.
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.

Table 1: Core common clause signatures for UML state machines / EFSMs.

Model Transformation - State Machine into CDL: 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).
```

The Common Declarative Language (CDL)

CLAUSE	DESCRIPTION
substate/2	substate(?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.
proc/1	proc(?Procedure) implies that ?Procedure is a process in do behavior.
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 triplet (?Event, ?Guard, ?Action), only ?Guard is optional, the absence of which is codified as nil.

(a) Clause signatures for composite states and state behaviors.

The Common Declarative Language (CDL)

CLAUSE	DESCRIPTION
entry_pseudostate/2	entry_pseudostate(?Entry, ?Substate) implies that ?Substate is the target inner-state whose superstate is already defined by substate(?Superstate, ?Substate).
exit_pseudostate/2	exit_pseudostate(?Exit, ?Superstate) implies that ?Exit is an exit state within the superstate ?Superstate.
choice/1	choice(?Name) defines a choice pseudostate.
junction/1	junction(?Name) defines a junction pseudostate.
history/1	history(?State) implies that history of the incoming transitions to state ?State is captured.
deep_history/1	deep_history(?State) implies that history of the incoming transitions to state ?State as well as all its substates are captured.

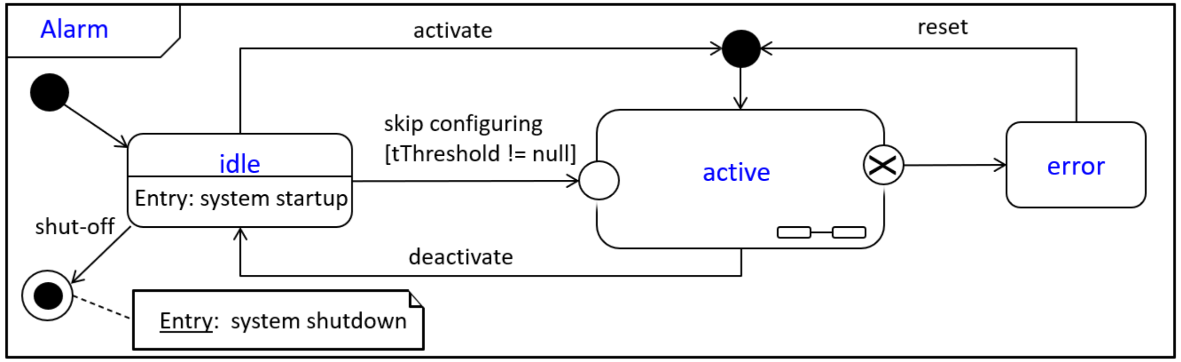
(b) Clause signatures for pseudostates.

The Common Declarative Language (CDL)

CLAUSE	DESCRIPTION
region/2	region(?State, ?Region) implies that ?State contains a autonomous region ?Region with substates, defined by substate(?Region, ?Substate).
fork/1	fork(?State) implies that ?State is a fork pseudostate.
join/1	join(?State) implies that ?State is a join pseudostate.
forking/2	forking(?Fork, ?State) implies a forked-transition to the ?State.
joining/2	joining(?Join, ?State) implies a joining-transition from the ?State to the join point ?Join.
par/2	par(?PState, ?List), used in the flattening process, keeps the list of all corresponding parallel [sub]-states that are handled by the state ?PState.

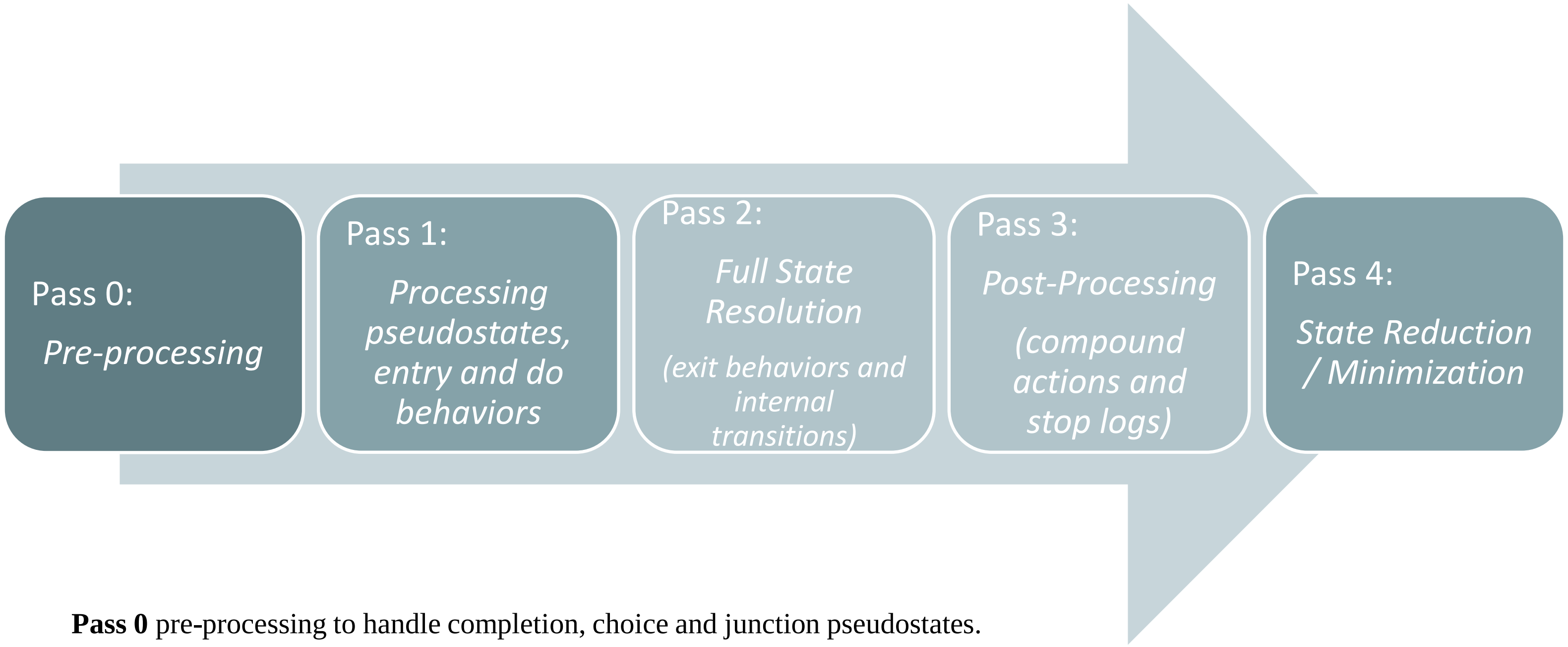
(c) Clause signatures for parallel regions and parallel states.

Model Transformation - State Machine into CDL: An example




```
% 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")).
```

The Flattening Process



Pass 0 pre-processing to handle completion, choice and junction pseudostates.
Pass 1 resoling pseudostates and entry behaviors + expanding the do behaviors.
Pass 2 full top-to-bottom state resolution

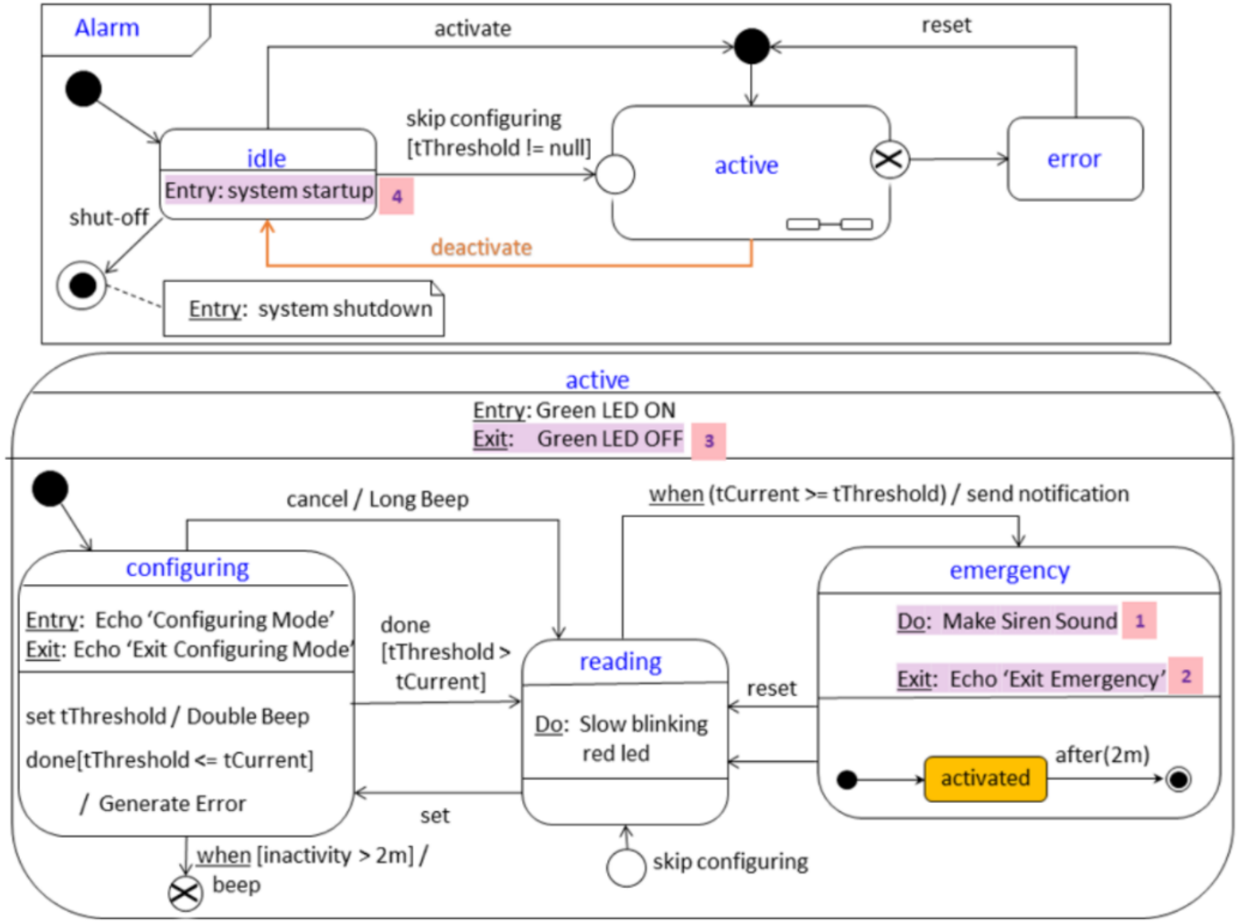
gradually resolving level states,
processing exit behaviors, and
handling internal transitions.

Pass 3 post-processing

correcting stop logs (do processes) + handling compound actions
The stop event logs are produced in two phases: book-marked in Pass 1 and resolved in Pass 3.

Pass 4, state minimization.

Order of Actions and State Behaviors



A transition and its corresponding order of actions.

The Flattened Output

a1: exec doubleBeep();
a2: exec echo('Exit configuring mode');
a3: exec longBeep();
a4: log Green LED OFF
e4: reset
a5: log Green LED ON
a6: exec generateError();
a7: log ABORT 'Make Siren Sound'
a8: log ABORT 'Slow blinking red LED'
a9: log START 'Make Siren Sound'
a10: exec beep();
a11: log STOP 'Make Siren Sound'
a12: exec echo('Configuring mode');
a13: log START 'Slow blinking red LED'
a14: exec echo('Exit Emergency');
a15: log System Startup
a16: exec sendNotification();
a17: log System Shutdown

e1: set tThreshold
e2: done
e3: deactivate
e4: set
e5: timeout 2:00
e6: completed emergency*
e7: when tCurrent >= tThreshold
e8: shutoff
e9: after 2:00
e10: skip configuring
e11: activate
e12: e12

g1: tThreshold > tCurrent
g2: tThreshold <= tCurrent

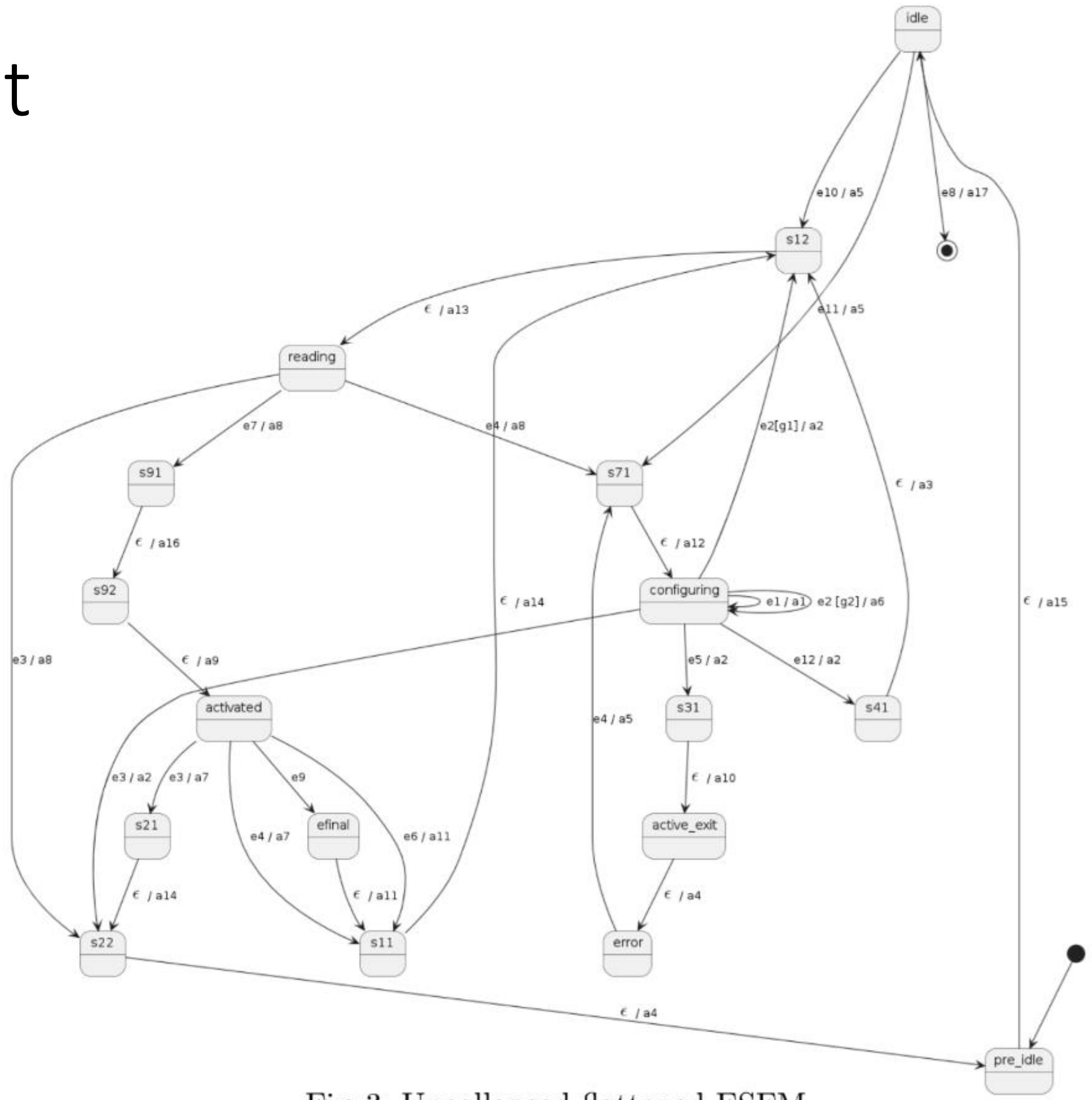


Fig. 3: Uncollapsed flattened ESFM.

The Flattened Output

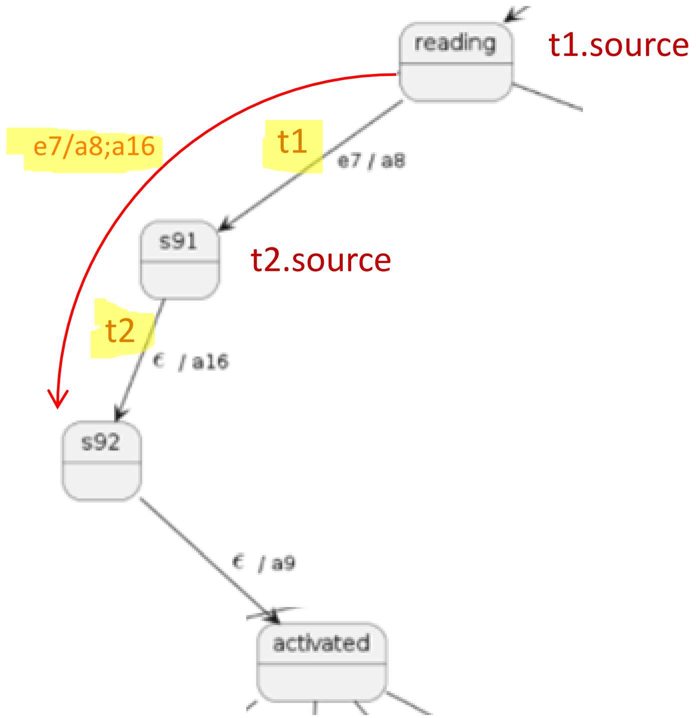
MEASURE	INITIAL MODEL	FLATTENED MODEL
number of states and substates	9	18
number of nested states	5	0
number of internal initial states	2	0
number of transitions	16	29
number of internal transitions	2	0
number of entry pseudo states	1	0
number of exit pseudo states	1	0
number of entry behavior	2	0
number of do behavior	2	0
number of exit behavior	3	0
number of guards	2	2
number of actions	10	26
number of nil transitions	2	11
number of levels	3	1

Minimizing the nil-transitions

Procedure ***Collapse***
Input: The EFSM machine in CDL.
Output: The EFSM machine in CDL.

1. Set $l_s \leftarrow \emptyset$.
   Set $l_t \leftarrow$ all t in $match(t, \texttt{transition/5}, t.\texttt{event} \neq \textbf{nil})$.
2. For each t_1 in l_t do:
2.1. $bind(q, t_1.\text{destination})$; remove$(t_1, l_t)$.
2.2. While $exits(t_2, \texttt{transition/5},$
$t_2[.\texttt{source},.\texttt{event},.\texttt{guard}] = \langle q, \textbf{nil}, \textbf{nil} \rangle$:
2.2.1. $match(t, \texttt{transition/5}, t.\texttt{source} = t_2.\texttt{source}$ and $t.\texttt{event} \neq \textbf{nil})$;
If $exists(t)$ return **ERR**.
2.2.2. $replace(t_1, \langle\, t_1.\texttt{source}, t_2.\texttt{destination}, t_1.\texttt{event}, t_1.\texttt{guard},$
$concat(t_1.\texttt{action}, t_2.\texttt{action}) \,\rangle)$.
2.2.3. $append(t_1, l_t)$.
2.2.4. $match(m, \texttt{initial/1}, m.\texttt{state} = q)$; If **not** $exists(m)$: $append(q, l_s)$.
3. For each s in l_s do:
3.1. $match(t, \texttt{transition/5}, t.\texttt{destination} = s)$; If $exists(t)$ return **ERR**.
3.2. $remove(t)$; $remove(s)$;

END Collapse.



The Minimized Flattened Output

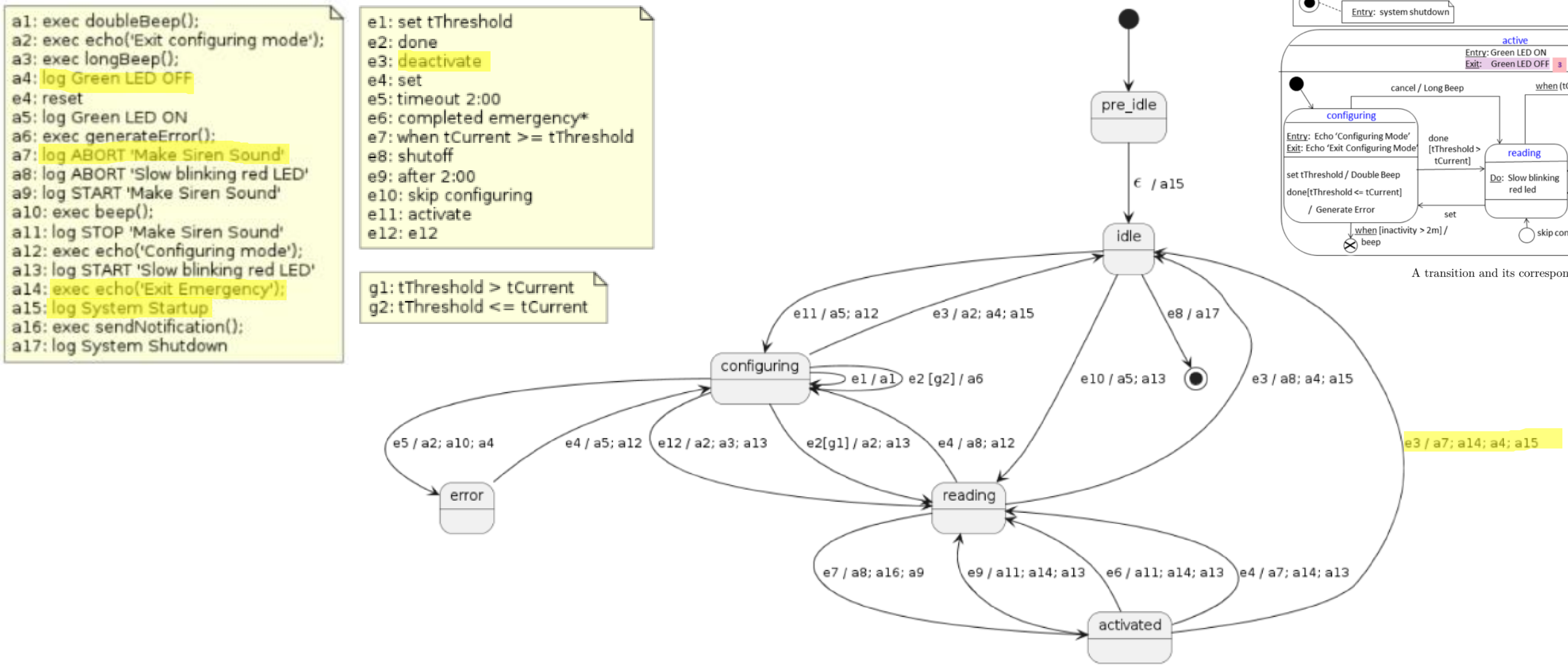
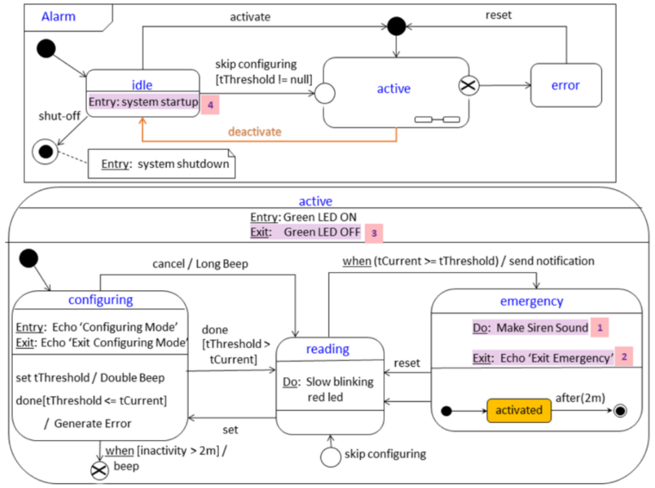


Fig. 4: Minimized collapsed flattened ESFM.



A transition and its corresponding order of actions.

The Flattened Output

MEASURE	INITIAL MODEL	FLATTENED MODEL
number of states and substates	9	~~18~~ → 7
number of nested states	5	0
number of internal initial states	2	0
number of transitions	16	~~29~~ → 18
number of internal transitions	2	0
number of entry pseudo states	1	0
number of exit pseudo states	1	0
number of entry behavior	2	0
number of do behavior	2	0
number of exit behavior	3	0
number of guards	2	2
number of actions	10	~~26~~ → 17
number of nil transitions	2	~~11~~ → 1
number of levels	3	1

Querying the CDL - Primitives

- ***new-id***([prefix]): creates and returns a new global unique identifier.
- ***match***(s, clause/arity [, condition = **true**]): selects all clauses matching clause/arity in s that satisfies given condition.
- ***add***(clause/arity, args): adds a new clause to the database.
- ***remove***(s): removes clause(es) denoted by the selector s from the database.
- ***replace***(s, args): replaces a single clause denoted by selector s with new arguments.
- ***select***(s, condition = **true**): selects all items from selector s that satisfy a given condition.
- ***exists***(s [, condition = **true**]): returns **true** if selector s contains elements that satisfy condition, otherwise **false**.
- ***exists***(s, clause/arity, [, condition = **true**]): ≡ ***match***(x, clause/arity); return ***exists***(x, condition); x may be referenced in condition.
- ***bind***(x, selector): binds x to the selector.
- ***insert***(e, place): inserts element e to the beginning of the list represented by place. If place is **nil**, a new list containing e is created, where place is pointing to. If place is singular, it is converted to a list that contains the element place.
- ***append***(e, place): same as ***insert***(), except e is appended to the end of the list represented by place.
- ***remove***(e, col): removes e from the collection represented by col. If col is singular, it is converted to a list that contains the element col itself.
- ***pop***(col): removes and returns the first element of col.
- ***diff***(s1, s2): returns the set difference s1 − s2. Both s1 and s2 are converted to a set if they are not.
- ***concat***(l1, l2): concatenates / appends l1 and l2 in a newly constructed list, as return value. If either arguments are singular they are converted to lists.

Querying the CDL - Example

- Using primitives

  featuring ***match***, ***add***, ***remove***

```
# Python

print("\nBefore:")
p.dumpall("state/1", "transition/3")
p.dynamic("transition/3", "transition/5")

m = p.matchall("transition/3")
for x in m:
   p.remove("transition", x)
   p.add("transition", x[0:3] + ['guard', 'action'])

print("\nAfter:")
p.dumpall("state/1", "transition/5")
```

```
%% Prolog Database
state(s).
state(t).
state(x).
transition(s, t, nil).
transition(s, x, e, g, a).
transition(s, x, e2, g2, "exec: v2 = v2 + 1;").
```

```
Before:
state('s').
state('t').
state('x').
transition('s','t','nil').

After:
state('s').
state('t').
state('x').
transition('s','x','e','g','a').
transition('s','x','e2','g2','exec: v2 = v2 + 1;').
transition('s','t','nil','guard','action').
```

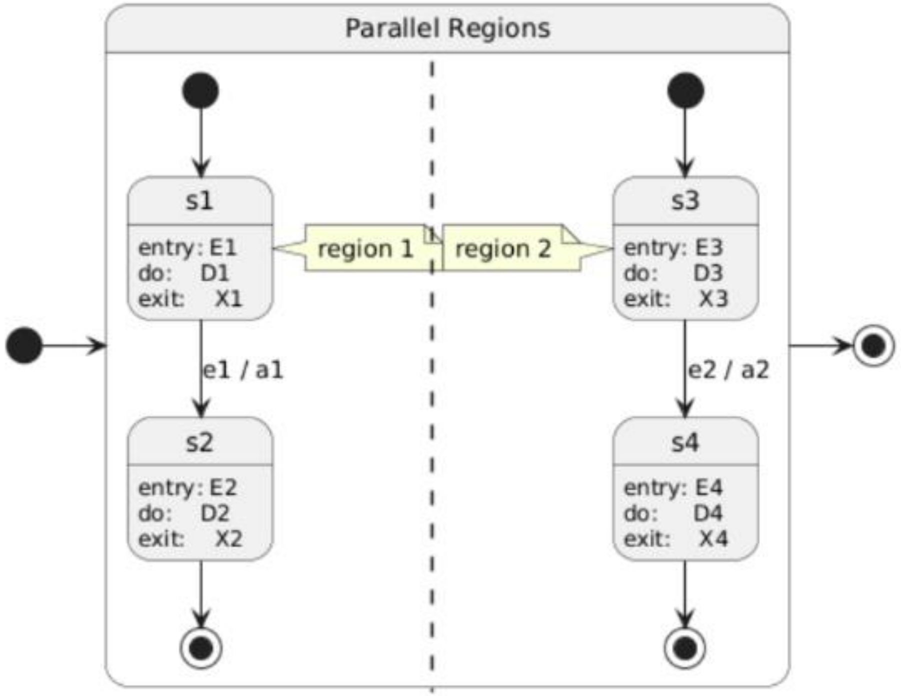
Flattening Orthogonal Regions

Procedure ***PExpand***
Input: The UML machine in CDL.
Output: The expanded UML machine in CDL.

0. For all t in `match`(t, `transition/5`, t.`event` = **nil**):
   Set t.`event` = '`event(completed, {`t`.source})`'.
1. Execute PCartesian.
2. Execute PStateBahavior.

END PExpand.

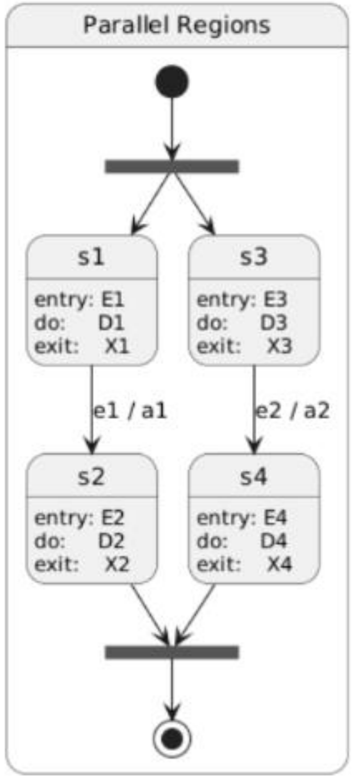
Parallel States / Orthogonal Regions




```
state(pr).            alias(pr, "Parallel Regions").
state(f).             alias(f, "");
initial(pr).          final(f).
region(pr, r1).       alias(r1, "Region 1").
region(pr, r2).       alias(r2, "Region 2").
superstate(r1, s1). superstate(r1, s2). superstate(r1, f1).
superstate(r2, s3). superstate(r2, s4). superstate(r2, f2).
initial(s1).          initial(s3).    final(f1).   final(f2).
transition(pr, f, nil, nil, nil).
transition(s1, s2, event(call, e1), nil, action(log, "a1")).
transition(s2, f1, nil, nil, nil).
transition(s3, s4, event(call, e2), nil, action(log, "a2")).
transition(s4, f2, nil, nil, nil).
transition(pr, s5, e5, nil, a5).
onentry_action(s1, action(log, "E1")).
onentry_action(s2, action(log, "E2")).
onentry_action(s3, action(log, "E3")).
onentry_action(s4, action(log, "E4")).
onexit_action(s1, action(log, "X1")).
onexit_action(s2, action(log, "X2")).
onexit_action(s3, action(log, "X3")).
onexit_action(s4, action(log, "X4")).
do_action(s1, proc("D1")).   do_action(s2, proc("D2")).
do_action(s3, proc("D3")).   do_action(s4, proc("D4")).
```

Fig. 5: An abstract UML state machine with parallel regions.

Parallel States / Orthogonal Regions




```
...
state(s1).
state(s2).
state(s3).
state(s4).

fork(f1). % special pseudostate
join(j1). % special pseudostate

forking(f1, s1).    % no event
forking(f1, s3).
joining(j1, s2).
joining(j1, s4).

initial(f1).
...
```

Fig. 6: Equivalent UML inner-states using join/fork pseudostates.

Flattening Orthogonal Regions

Subroutine ***PCartesian***

For each s_{top} in $match(s,$ state/1, $exists(r,$ region/1, r.state $= s))$ do:

1. $s_{new} = new\text{-}id(\text{'s'})$; $add($substate/2, $\langle s_{top}, s_{new}\rangle)$; $add($par/2, $\langle s_{new}, \{\}\rangle)$.

2. For each r in $match(r,$ region/2, r.state $= s_{top}$ and
    $exists(x,$ substate/2, x.superstate $= r$.state and
        $exists(y,$ initial/1, y.state $= x$.substate$))$ do:
    $match(\ell,$ par/2, ℓ.state $= s_{new})$; $append(y$.state, ℓ.list$)$.

3. Set $l \leftarrow \{s_{new}\}$.

4. While l is not empty do:

4.1. $s \leftarrow pop(l)$.

4.2. $match(x,$ par/2, x.state $= s)$; $bind(p, x$.list$)$.

4.3. For each t in $match(t,$ transition/5, t.source $\in p)$:

4.3.1. Set $p' \leftarrow p - \{ t$.source $\} + \{ t$.destination $\}$.

4.3.2. If not $exists(x,$ state/1, x.list $= p')$:
    $s_{new} = new\text{-}id(\text{'s'})$; $add($substate/2, $\langle s_{top}, s_{new}\rangle)$;
    $add($par/2, $\langle s_{new}, p'\rangle)$; $append(s_{new}, l)$.

4.3.3. $match(x,$ state/1, x.list $= p')$.

4.3.4. If $\forall x_i \in p' : exists(f,$ final/1, f.state $= x_i)$, then
    $add($final/1, $\langle s_{new}\rangle)$.

4.3.5. $add($transition/5, $\langle s, x$.state,t.event,t.guard,t.action$\rangle)$.

5. For all q in $match(r,$ region/2, r.state $= s_{top})$,
    $match(q,$ substate/2, q.region $= r$.region$)$ do:

5.1. $remove(t)$ in $match(t,$ transition/5, t.source $= q$ or
    t.destination $= q)$;

5.2. $remove(x)$ in $match(x,$ substate/2, x.substate $= q)$.

5.3. $remove(x)$ in $match(x,$ initial/1, x.state $= q)$, if any.

5.4. $remove(x)$ in $match(x,$ final/1, x.state $= q)$, if any.

6. $remove(r)$ in $match(r,$ region/2, r.state $= s_{top})$.

END PCartesian.

Subroutine ***PStateBehavior***

0. Set $\ell \leftarrow$ all ℓ.state in $match(x,$ par/2$)$s.
    Set $s \leftarrow x$.state in $match(x,$ initial/1, $x \in \ell)$.

1. For each $x \in \ell_2$.list in $match(\ell_2,$ par/2, ℓ_2.state $= s)$ do:

1.1. $match(e,$ onentry_action/2, e.name $= s)$;

1.2 $match(\alpha,$ onentry_action/2, e.name $= x)$;
    if $exits(\alpha)$ $append(\alpha$.action, e.action$)$.

1.3 $match(\alpha,$ do_action/2, e.name $= x)$;
    if $exits(\alpha)$ $append($'action(log, "START $\{\alpha$.name$\}$"))', e.action$)$.

2. Save ℓ in ℓ_{save}.

3. While ℓ is not empty do:

3.1. remove(s, ℓ).

3.2. $l_{from} \leftarrow x$.list where match$(x,$ par/2, x.state $= s)$.

3.3. For each t in $match(t,$ transition, t.source $= s)$ do:

3.3.1. $\ell_{to} \leftarrow p$.list where $match(p,$ par/2, $p.state = t$.destination$)$.

3.3.2. $s_{leave} \leftarrow diff(\ell_{to}, \ell_{from})$; $s_{enter} \leftarrow diff(\ell_{from}, \ell_{to})$.

3.3.3. $match(\alpha,$ onentry_action/2, α.name $= s_{enter})$;
    if $exits(\alpha)$ $append(\alpha$.action, t.action$)$.

3.3.4. $match(\alpha,$ onexit_action/2, α.name $= s_{leave})$;
    if $exits(\alpha)$ $insert(\alpha$.action, t.action$)$.

3.3.5. $match(\alpha,$ do_action/2, α.name $= s_{leave})$;
    if $exits(\alpha)$ and t.event = 'event(completed, $\{s_{leave}\}$)'
        $insert($'action(log, "STOP $\{\alpha$.name$\}$"))', t.action$)$,
    otherwise $insert($'action(log, "ABORT $\{\alpha$.name$\}$"))', t.action$)$.

3.3.6. $match(\alpha,$ do_action/2, α.name $= s_{enter})$;
    if $exits(\alpha)$ $append($'action(log, "START $\{\alpha$.name$\}$"))', t.action$)$.

4. Restore ℓ from ℓ_{save}.

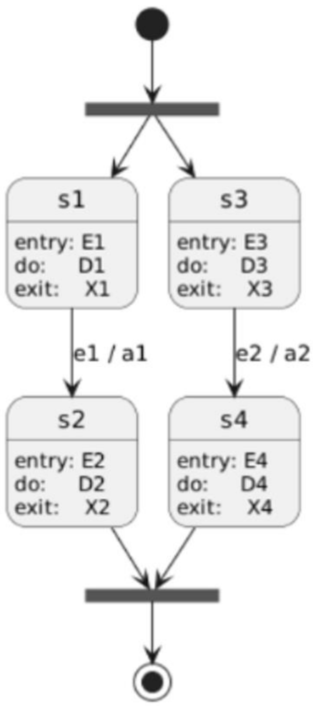
5. For all $p \in \ell$, For all x in $match(x,$ par/2, x.state $= p)$,
    For all s in x.list do:

5.1. $remove(e)$ where $match(e,$ onentry_action/2, e.name $= s)$.

5.2. $remove(e)$ where $match(e,$ do_action/2, e.name $= s)$.

5.3. $remove(e)$ where $match(e,$ onexit_action/2, e.name $= s)$.

END PStateBehavior.



Flattening Orthogonal Regions

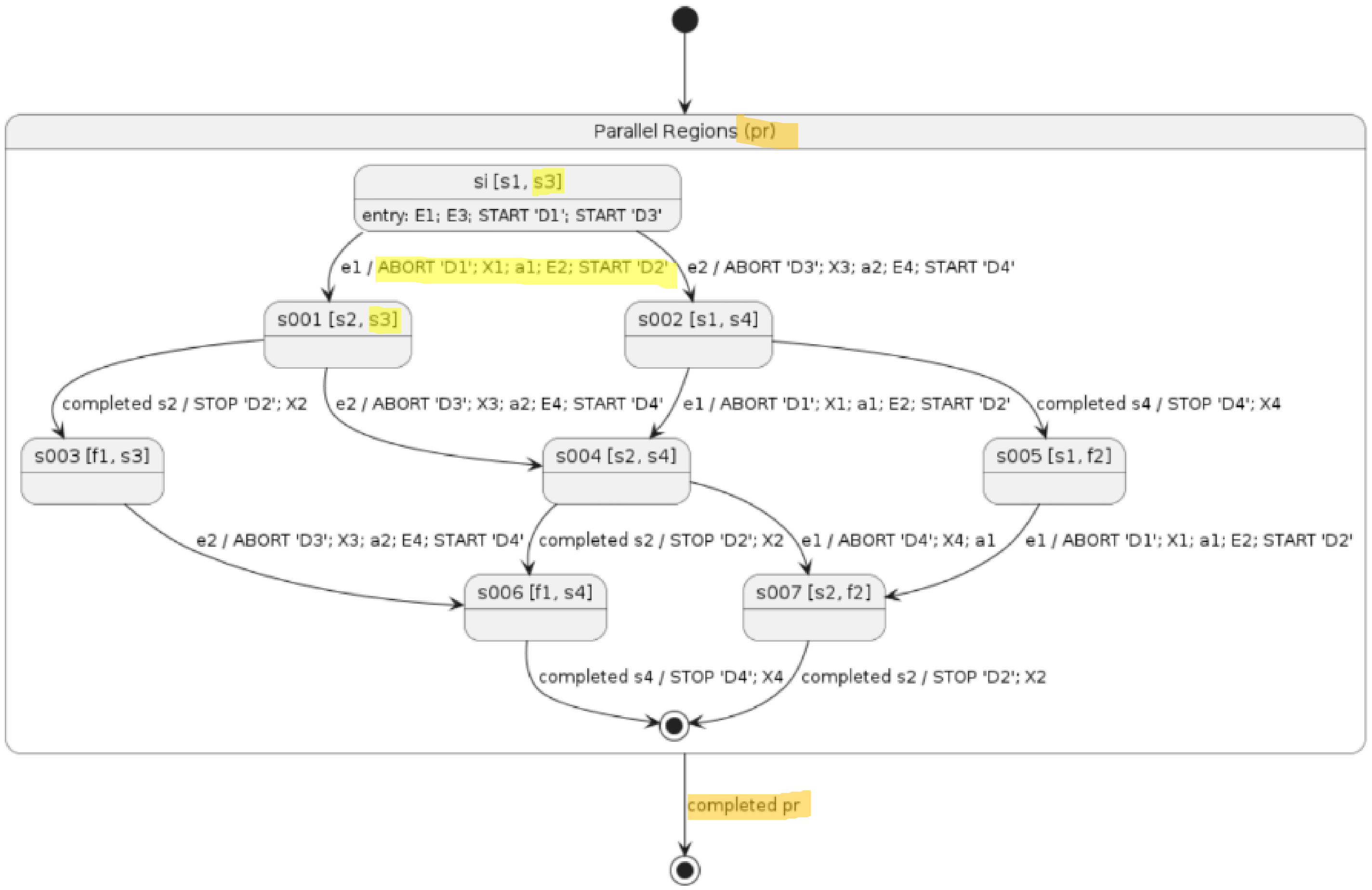
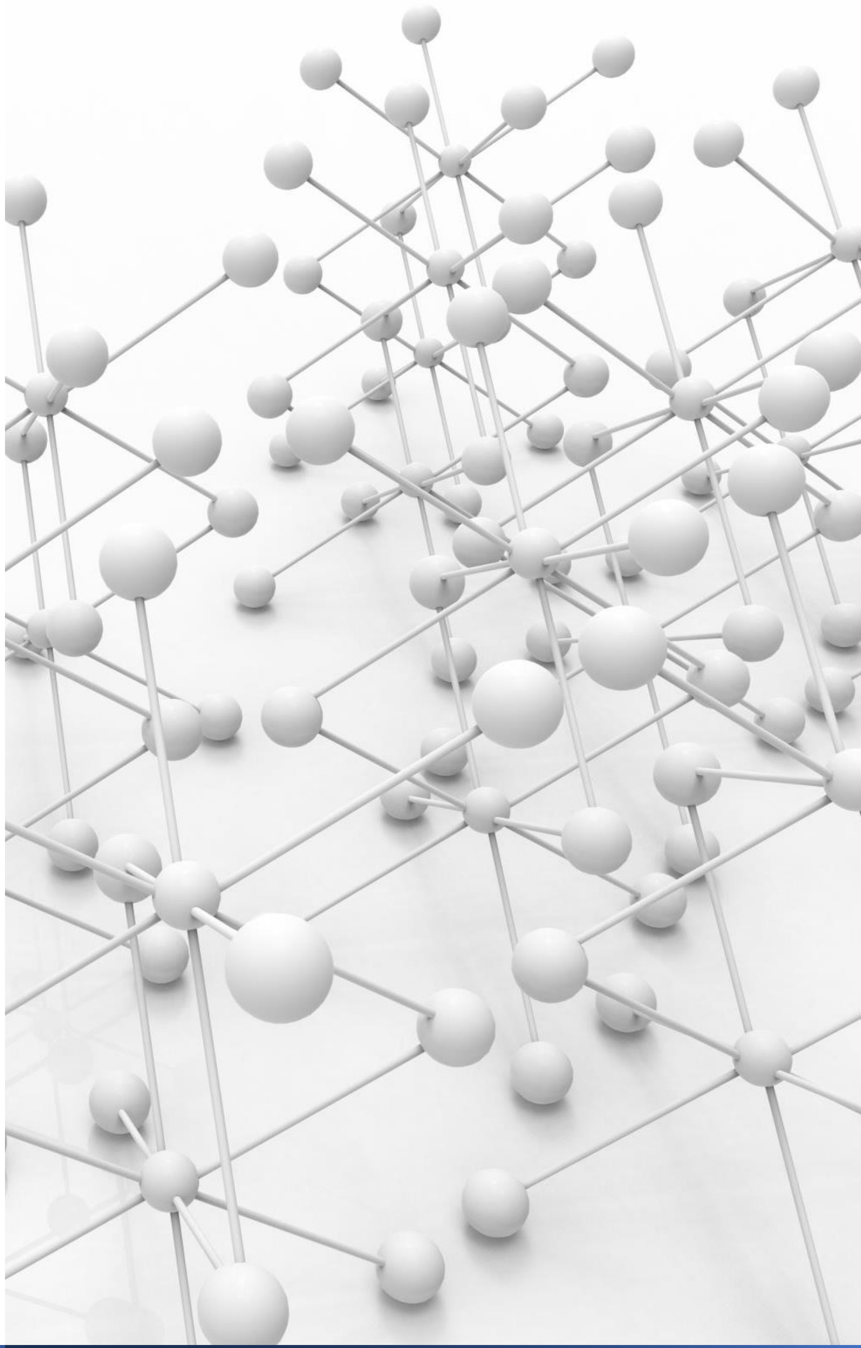


Fig. 7: Generated equivalent expanded machine without parallel regions.

Complexity and Correctness

- Using Flattened EFSM
  - more vertices may be produced all transitions are made explicitly.
  - can aid in behavior analysis of the initial machine
    (correctness, complexity, and welformedness)

- Verified though case-studies, using
  - nested composite states, with both implicit and explicit events, and
  - complex behaviors to verify the resulting sequence of actions.
  - We did not include **external event** in the complex region.
  - The formal proof of correctness may be provided by using formal definition of UML state machines. We plan to address this in future.



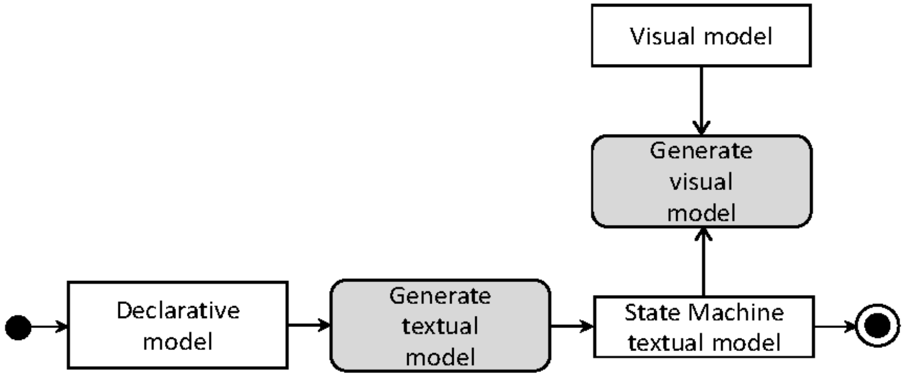
Interoperability among text-to-UML Drawing Tools

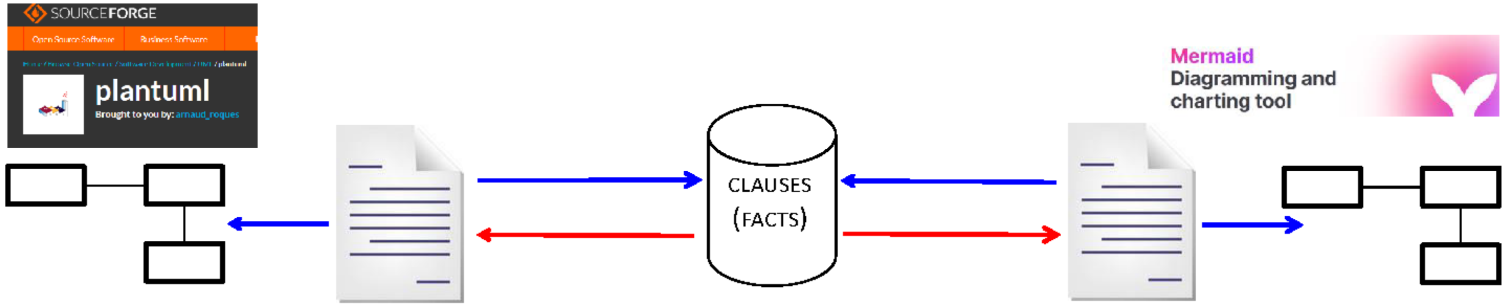
- A comprehensive database of of text-to-UML may be found at:
  https://modeling-languages.com/text-uml-tools-complete-list/

- Common issues:
  - Not all features are supported
    Examples: history annotation, state behaviors, composite state annotation, junction

- **We use CDL as a common interoperable platform**

CDL as a common interoperable platform





Interoperability among text-to-UML Drawing Tools

- We used an extensible template-based code-generation for conversion from CDL to target platform

- We used suggestive parsing for unsupported features
    - state behaviors as prefixes
    - default event types (all as call events)
    - UML stereotypes (i.e. composite, choice)

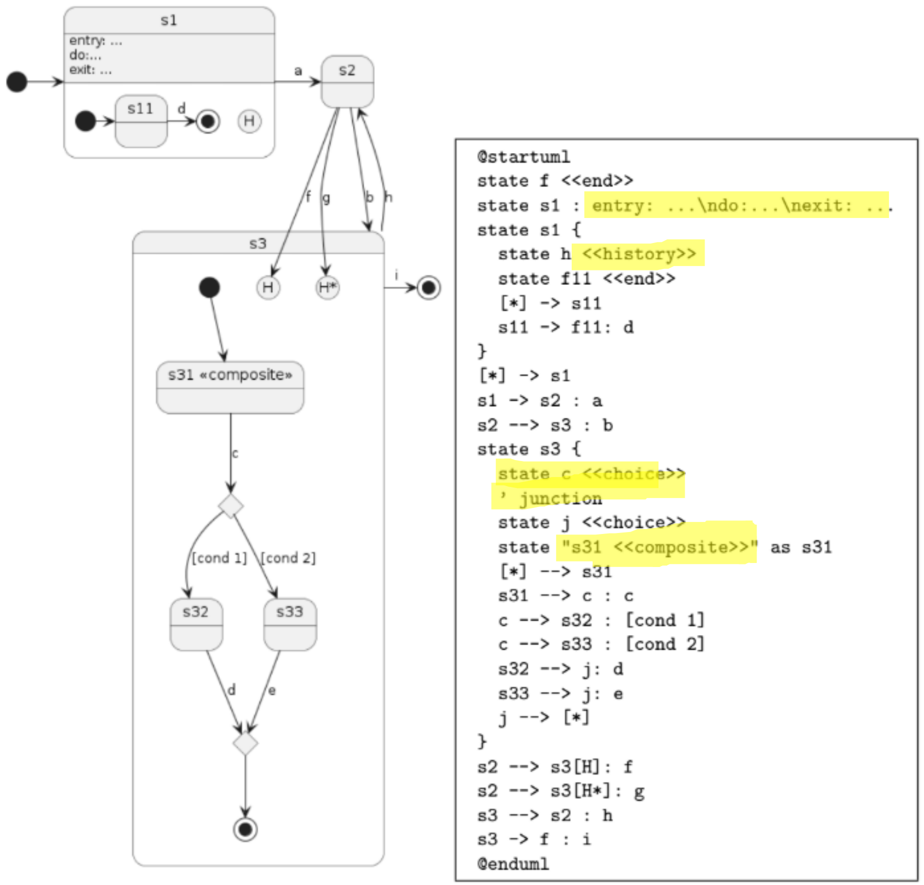


Fig. 9: A sample diagram in PlantUML, illustrating various UML features.

The Common Declarative Language as a Database

- Prolog enables us performing rule-based queries for complexity analysis and correctness

```
in_degree(State, N) :-
        findall([Source, State],
                (initial(State); transition(Source, State, _, _, _);
                (entry_pseudostate(Entry, Substate),
                transition(_, Entry, _, _, _),
                superstate(State, Substate));
                entry_pseudostate(Source, State)), Lst),
        length(Lst, N).
```

```
?- in_degree(configuring, N). %% N = 2
?- in_degree(reading, N). %% N = 5
?- in_degree(active, N). %% N = 3
```

The Common Declarative Language as a Database

- Prolog enables us performing rule-based queries for complexity analysis and correctness

```
get_all_internals(Lst) :-
        findall([Source, [EType, Event], [AType, Action]],
                internal_transition(Source, event(EType, Event), _,
                action(AType, Action)), Lst).


?- get_all_internals(Lst).
%% Lst = [[configuring, [set, tThreshold], [exec, "doubleBeep();"]],
%% [configuring, [call, done], [exec, "generateError();"]]]
```

Conclusion and Future Work

- The CDL serves as a textual representation of initial UML state machine, as well as the flattened model.
- Text-to-UML drawing tools can deploy CDL in model transformation.
  - CDL may be used to create a repository of representation as well as to support tool interoperability.
  - A machine produced by one tool can then be represented declaratively and read by another tool.
- Text-to-UML drawing tools may not support exact same set of UML elements
  - compatibility may not always be full

Conclusion and Future Work

- Our EFSM definition allows a UML state machine to be flattened, whereby composite and orthogonal states collapse into a single level of abstraction.
- In previous work we deployed the flattened model as the basis of simulation.
- Our previous work concentrated on the fundamental features of the UML, where the CDL was used as the basis for simulation. In this paper, we addressed major *advanced features* of the UML, including presence of **orthogonality**, while complementing previous work on **representation**, **model transformation**, and **visualization tool interoperability**.
- Future work will address the second major advanced feature of a UML state machine: the *History pseudostate*.

References

1. Daniel Balasubramanian, Corina S. Pasareanu, Gabor Karsai and Michael R.Lowry. Polyglot: systematic analysis for multiple statechart formalisms. In: Nir Piterman, and Scott A. Smolka, (Eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2013. Lecture Notes in Computer Science, vol 7795. Springer, Berlin, Heidelberg.
2. Bernhard Beckert, UML State Machines, Lecture notes, Universitat Koblenz-Landau.
3. Feng Sheng, Huibiao Zhu, Zongyuan Yang, Jiaqi Yin and Gang Lu. Verifying static aspects of UML models using Prolog. In Proceedings of the 31st international conference on software engineering and knowledge Engineering, SEKE 2019, Portugal.
4. Zohaib Khai, Aamer Nadeem and Gang-soo Lee. A Prolog based approach to consistency checking of UML class and sequence diagrams. In: Kim, Th., et al. Software Engineering, Business Continuity, and Education. Communications in Computer and Information Science, vol 257. Springer, Berlin, Heidelberg, 2011.
5. Tom Mens, Alexandre Decan and Nikolaos I. Spanoudakis. A method for testing and validating executable statechart models. Software and Systems Modeling, Volume 18, pp. 837–863, Springer-Verlag, 2019.
6. Kwang-Ting Cheng and A. S. Krishnakumar. Automatic generation of functional vectors using the extended finite state machine model. ACM Transactions on Design Automation of Electronic Systems, Volume 1, Issue 1, pp. 57 -59.
7. Sanford Friedenthal, Alan Moore and Rick Steiner, A Practical Guide to SysML (Third Edition), Morgan Kaufmann, 2015.
8. Object Management Group, Unified Modeling Language (UML) Version 2.5.1, Dec. 2017.
9. Vangalur S. Alagar and K. Periyasamy. Specification of Software Systems. Springer, 2011.
10. Andreas Podeski, Hierarchical State Machines, Lecture notes, Albert-Ludwigs-Universit ̈at Freiburg, 2015.
11. https://modeling-languages.com/text-uml-tools-complete-list/