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
{ali.jannatpour | constantinos.constantinides}@concordia.ca

Abstract. Originally presented in previous work to capture the set of fundamental elements of the UML state machine specification, Common Declarative Language (CDL) provides a model that can aid in the validation and verification of requirements. In this paper we target two objectives: First, we extend CDL by addressing one of the advanced concepts of the UML state machine specification, namely the notion of orthogonality which allows complex machine behavior through parallel state configurations. Second, we complement previous work by focusing on how CDL can serve as a platform for the representation of a state machine, how the language can be deployed for a model transformation where the initial machine (containing composite and/or orthogonal states) can be flattened into a model whose formal definition we provide, and finally how the CDL can be deployed to support interoperability among text-to-UML drawing tools [11].

Keywords: UML, State Machine, EFSM, Declarative Representation.

1 Introduction and Background

The Unified Modeling Language (UML) adopted and extended Harel's statechart specification to provide an elaborated state machine specification. A UML state machine is part of the dynamic model of the UML specification and it can be deployed to model the behavior of a software system at any level of abstraction. We can view the state machine as a cyclic directed multigraph where the elements in the tuple representation of the state machine map to an ordered pair that represents the equivalent graph, namely $Graph(G) = (V, A)$, where V is a set of nodes, and A is a set of ordered pairs of nodes called (directed) edges.

In discrete mathematics, a state machine may be represented as a graph, consisting of labelled edges, where each label represents an event that triggers the transition. In an extended form, a state machine may be represented using an Extended Finite State Machine (EFSM) where each label is a triplet $\langle e, g, a \rangle$, representing the corresponding event, guard, and the post-transition action. It must be noted that EFSM representation does not allow the inclusion of most

UML features such as nested states, parallel regions, state behaviors, and pseudostates. Hence, only a *flat* UML state machine, that does not include such features, can be represented using an EFSM.

Various textual representations of UML diagrams may be found in the literature. While UML is a graphical language, elements of a UML diagram may be represented in a textual format. Balasubramanian et al. [1] introduce Polyglot, a comprehensive framework for analyzing models described using multiple statechart formalisms. Their approach involves translating statechart models into Java and analyzing them using pluggable semantics for different variants. The translation process captures the structure of the statechart model, while behavior is defined in separate Java modules. They also provide an implementation of their framework and present a case study where interacting components are modeled using different statechart formalisms. Sheng et al. [3] present a Prolog-based consistency checking for UML class diagram and object diagram. They formalize the elements of a model and then convert the model into Prolog clauses (facts). Consistency rules are also defined in Prolog, along with interfaces that enable querying of properties, elements, and submodels of the model. Khai et al. [4] propose an Prolog-based approach for consistency checking of class and sequence diagrams. Mens et al. [5] introduce a technique to improve statechart design supported by tools for test-driven development, behavior-driven development, design by contract, and property statecharts to facilitate the testing and validation process.

1.1 Motivation

In previous work, we presented the definition of Common Declarative Language (CDL) over the set of fundamental elements of the UML state machine specification while focusing on two applicabilities: 1) the provision of a query platform and 2) the simulation of state machine behavior. In this paper, we extend the CDL definition to capture one of the advanced elements of the UML specification, namely orthogonal states, while complementing the previous discussion by focusing on 1) how CDL can serve as a platform for the representation of a state machine, 2) how the language can be used for model transformation where the initial machine (with composite and orthogonal states) can be flattened into a model which we define, and finally 3) how the CDL can be deployed in an additional applicability: the interoperability among text-to-UML drawing tools[11].

The Common Declarative Language serves as a platform that can support a number of activities in the aid of requirements analysis and verification. It has several applicabilities: State machine presentation, UML state machine representation, a queryable database for model verification, transformation, and simulation. We map the graph's elements into a set of clauses, maintained in a declarative model as a set of facts in the Prolog language. We refer to such a set of clauses as the *Common Declarative Language as a Database*. While there exist many platform independent descriptive languages i.e. JSON, YAML, XML, etc. We view the Prolog representation as a common queryable and language-

independent platform, since Prolog's querying platform can efficiently used in model transformation.

1.2 Organization of the Rest of the Paper

The formalism of the EFSM and the state machine presentation including the clause signatures is discussed in Section 2. The transformation of the (initial) state machine model into a flattened model into an EFSM format is discussed in Section 3. The deployment of the visualization tools (text-to-UML drawing tools) and their interoperabilty issues are discussed in Section 4. Both the initial and the flattened models can function as databases where developers can execute queries to obtain knowledge on three aspects: Behavior, Complexity and Measurements, and Wellformedness (to ensure the validation of the machine). Even though the focus of the paper is not on the query system (as discussed in previous works), in Section 5, we discuss how CDL can be deployed in such a platform.

2 A Formalism for an Extended Finite State Machine

An EFSM is formally defined as a 7-tuple [6]. Our redefinition of an EFSM adopts this 7-tuple, with a slight modification on the transition inputs to address ϵ-transitions, *guard lists* and *action lists*. The formal definition of EFSM is given in seciton 2.1. The Common Declarative Language (CDL) clauses signatures are listed in section 2.2. The set of *selection* and *manipulation* primitives for model transformation are defined in section 2.3. These operations are applied on CDL clauses.

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*.
$\Lambda = \{\lambda : q \xrightarrow{e_i[g_j]/a_k} q'$, where $i, j, k \in \mathbb{Z}\}$, is a finite set of *deterministic* transitions defined on $Q \times (\{\epsilon\} \cup \Sigma_1) \times 2^{\Gamma} \rightarrow Q \times {\Sigma_2}^{\star}$, 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 $\lambda : q \xrightarrow{\epsilon[g_j]/a_k} q'$. In the case $g_j = \emptyset$, the transition becomes ϵ-transition. In order for Λ to be deterministic, for every state $q \in Q$, at most one possible transition must exist. While this property holds for all EFSMs, we enforce the following additional restrictions: i) If state q has an outgoing ϵ-transition, no other outgoing transitions are allowed on q. ii) If state q has an outgoing guarded ϵ-transition, all other transitions on the state must also be guarded ϵ-transitions.

2.2 The Common Declarative Language

The clause signatures of the common declarative language are compatible with Unified Modeling Language (UML) 2.5 [8] (currently the most recent version) and are shown in Tables 1 and 2. Table 1 lists the core (common clauses in both the initial and the flattened model) while tables 2a to 2c list the clause signatures to support UML features related to the composite states and state behaviors, pseudostates, and parallel regions. The core signatures essentially describe an EFSM model, where the events (including their types) are explicitly specified.

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.

2.3 Signatures of Transformation Operations

EFSMs do not support advanced UML features. To transform a UML state machine into an EFSM, we use tagging to attach and detach attributes to states and transitions. Examples of tagging are: attached by a 'do' tag to implement a *do* behavior, linking a sub-graph to a particular state to implement a *composite* state, etc. The following primitives are used in the transformations:

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**]): $\equiv$ *match*(x, clause/arity); return *exists*(x, condition); x may be referenced in `condition`.

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.

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.

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.

Table 2: The Common Declarative Language: Additional Clause Signatures.

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(s_1, s_2): returns the set difference $s_1 - s_2$. Both s_1 and s_2 are converted to a set if they are not.

concat(ℓ_1, ℓ_2): concatenates / appends ℓ_1 and ℓ_2 in a newly constructed list, as return value. If either arguments are singular they are converted to lists.

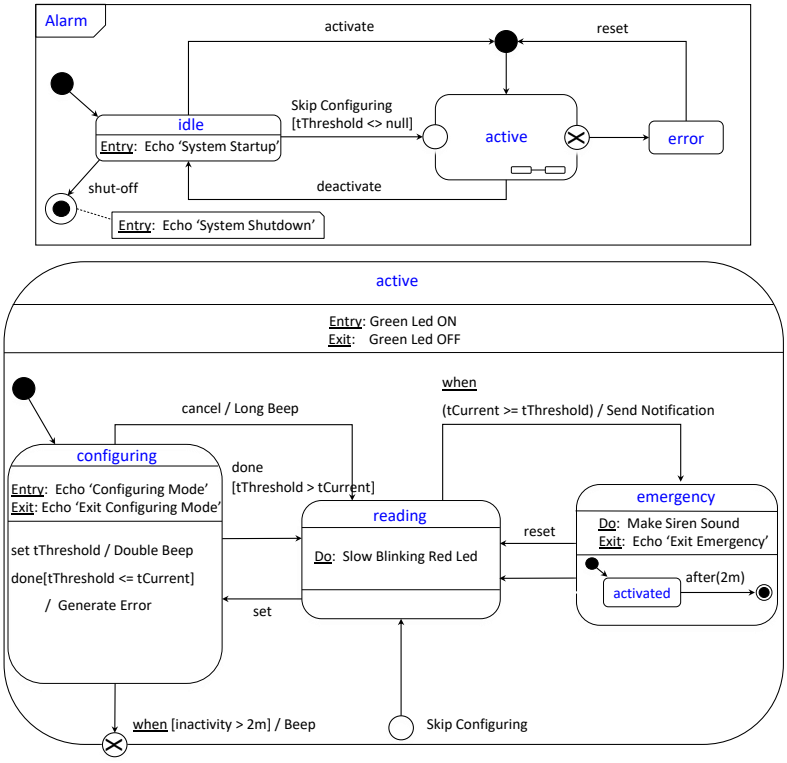


Fig. 1: A sample case-study representing complex UML features.

3 Flattening of a UML State Machine

In previous work, we presented an algorithm that flattens a state machine which is defined at a high level of abstraction (i.e. with superstates). The algorithm takes the UML representation of the machine and produces a flat state machine (with no composite states) which is a version of an EFSM (see Figure 2). In this paper, we extend the algorithm to support parallelism through the provision of orthogonal states. The algorithm covers the following UML features: parallel regions, join and fork pseudostates, and nested state behaviors. It also

addresses the implicit completion transitions in parallel regions, that are triggered by reaching inner final states in all branches.
In UML, completion events are represented as ϵ-transitions. An ϵ-transition is a transition whose *event* and *guard* are empty. Other examples of ϵ-transitions are those in pseudostates (i.e. entry and exit), as well as region completion (i.e. in the case of completion of a do action, or reaching a final substate). The flattened model is analogous to a bytecode platform for languages such as Java and Clojure, which is a seamless virtual machine. We believe that such a model can provide a platform for deeper analysis as well as a simulation of behavior. The flattening procedure is fully automated. The specification of the EFSM and the flattening procedure is given in the subsequent sections.

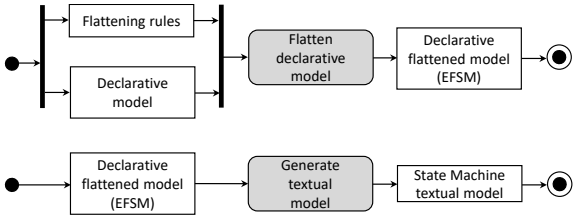


Fig. 2: Flattening activities.

3.1 The Flattening Process

In previous work, we presented a flattening algorithm that converts an input UML state machine into an EFSM. It was demonstrated that the process simplifies the complexity of a general UML diagram by resolving major features such composite states, state behaviors, exit- and entry-point pseudostates, choice and merge pseudostates into a flat EFSM machine with simple event-action-guard labels. The outline of the Flattening algorithm, namely procedure *Flatten* is given in the following.

Procedure *Flatten* (outline)
Input: The UML machine in CDL.
Output: The EFSM machine in CDL.

Pass 0 involves pre-processing that handles *completion*, *choice* and *junction* pseudostates.

Pass 1 resolves the pseudostates, *entry* behaviors. It also expands the *do* behaviors.

Pass 2 performs full top-to-bottom *full state resolution*, by which top level states are removed and their *exit* behaviors are handled. It also processes the *exit* behaviors for non-composite states as well as the internal transitions.

Pass 3 involves post processing, in which a) *stop* logs for *do* processes are resolved, and b) compound actions are converted into separate transitions. The *stop* event logs are produced in two phases: book-marked in Pass 1 and resolved in Pass 3.

In **Pass** 4, the resulting EFSM is minimized.

END Flatten.

A sample case-study (see Figure 1) is used to demonstrate the conversion process. The output is illustrated in Figure 3.

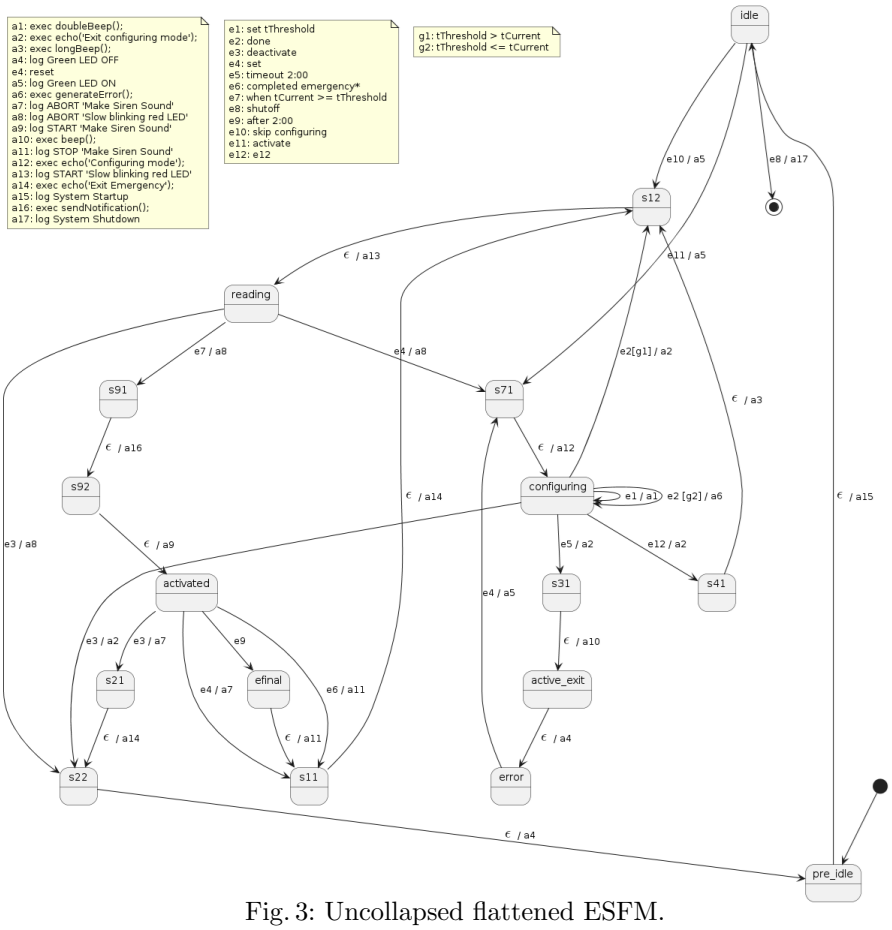


Fig. 3: Uncollapsed flattened ESFM.

In UML, the completion events are presented using an unlabeled transitions which should not be confused with ϵ-transitions in the EFSM model. As a result, all unlabeled transitions are converted into explicit completion event, and remain as such.

Each pass in the flattening procdure, processes the input using clause selectors and incrementally transforms the input model into a resolved model by pipelining the output to the next pass. Pass 0 essentially ensures that the input UML does not contain any implicit completion event. All `nil`-transitions implying a region completion must therefore be converted into an explicit completion event. Hence, we assume all `nil`-transitions imply region completion, except for the outgoing transitions in *choice* and *junction* pseudostates, which are essentially event-free. In our model, regions are represented using state names (see state `pr` and `completed pr` event in Figures 5 vs. 7, for instance). Passes 1-3

resolve state behaviors. We convert all state behaviors into actions. Handling the do behaviors in general is challenging. We treat do behaviors as processes that are being executed while the machine is *in* the corresponding state. Such process may be normally finished, in which case, it triggers a region completion even. We use the state name as a reference to the region. Alternatively, the process may be aborted, if there is an external event that triggers a transition from the state with the do behavior to another state. Such behavior may be analogous to the concept of processes and sub-processes in operating systems, where a process may be finished normally, or aborted by an external event. Hence, a do behavior representing a process 'P' is expanded into pairs of *start-stop* or *start-abort* action logs depending on whether the state/region is completed or aborted by an external event. We extend Pass 4 to further minimize the number of states of the resulting graph by collapsing the `nil`-transitions, as explained section 3.2. The original algorithm did not address the resolution of the orthogonal states and their behaviors. This is discussed in section 3.3.

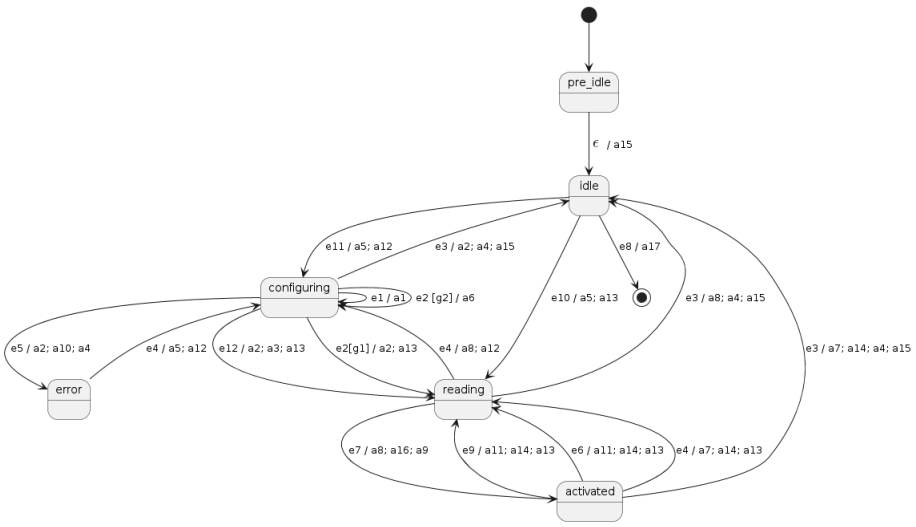


Fig. 4: Minimized collapsed flattened ESFM.

3.2 Minimization of `nil`-Transitions

The minimization of the flatten algorithm reduces the number of states by merging equivalent states and transitions. The original EFSM model did not support array of actions. That resulted in creating many (as well as) reduntant ϵ-transitions. By redefining the EFSM in section 2.1, we suggest that a transition can contain a sequence of actions. Thus, we add a post-process minimization step by which, all sequences of ϵ-transitions are followed and merged into a single transition and the intermediary states are removed. Compare the resulting machines in Figures 3 and 4. The post-processing step is implemented by procedure *Collapse*.

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

Note that in step 2, the list l_t includes all transitions whose source is a candidate state that is to be eliminated. The list is dynamic and newly added items are revisited in the iteration (see step 2.2.3.).

3.3 Parallel State Configurations Through Parallel Regions

Our declarative model is extended to support UML parallel regions using fork and join pseudostates. A sample abstract case study with corresponding declarative models are given in Figure 5. An equivalent join-fork version of the above model is given in Figure 6 with the corresponding changes in the declarative model. The two models are automatically convertible to one another.

3.4 Extending the Flattening Process

We extend the flattening algorithm to resolve the parallel states by simulating the parallel regions in an equivalent machine. The simulation is done by generating an equivalent UML state machine whose states are the Cartesian product of the set of states in parallel regions. The algorithm detail is given in procedure *PExpand*. The output of the algorithm on the input model in Figure 5 is given in the Figure 7 which illustrates that the flattened model correctly captures the handling of state behaviors in parallel states, in particular with parallel do behaviors, and with regards to the normal completion or aborted externally. The result is a UML state machine without parallel states. As illustrated, the initial entry behaviors are kept to keep the number of states to minimum. Once the parallel states are eliminated, the resulting machine may be fully flattened by applying the process Flatten to it. Note that the state entry and exit are covered in steps 3.3.3 to 3.3.6 in subroutine PStateBahavior.

Here we assume that the states in the parallel regions are not composite. All junctions and choice pseudostates are converted into regular states whose outgoing events are `nil`. The process expects explicit events. Hence, all `nil`-transitions are collapsed using procedure *Collapse*, as specified in section 3.2.

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.

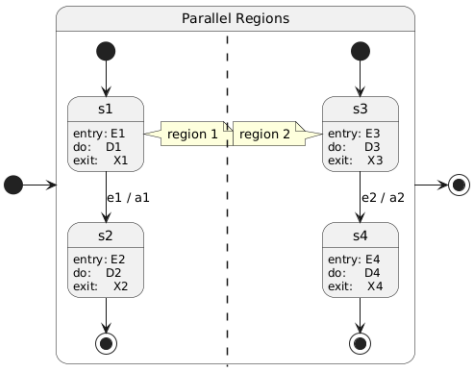
Subroutine *PStateBehavior*
0. Set $\ell \leftarrow$ all ℓ.`state` in *match*(x, `par/2`)s.
   Set $s \leftarrow x$.`state` in *match*(x, `initial/1`, $x \in \ell$).
1. For each $x \in \ell_2$.`list` in *match*(ℓ_2, `par/2`, ℓ_2.`state` = s) do:
1.1. *match*(e, `onentry_action/2`, e.`name` = s);
1.2 *match*(α, `onentry_action/2`, e.`name` = x);
   if *exits*(α) *append*(α.`action`, e.`action`).
1.3 *match*(α, `do_action/2`, e.`name` = x);
   if *exits*(α) *append*('`action(log, "START {`α.`name}"))`', e.`action`).
2. Save ℓ in ℓ_{save}.
3. While ℓ is not empty do:
3.1. `remove`(s, ℓ).
3.2. $l_{\text{from}} \leftarrow x$.`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. $\ell_{\text{to}} \leftarrow p$.`list` where *match*($p$, `par/2`, p.*state* = t.`destination`).
3.3.2. $s_{\text{leave}} \leftarrow$ *diff*($\ell_{\text{to}}, \ell_{\text{from}}$); $s_{\text{enter}} \leftarrow$ *diff*($\ell_{\text{from}}, \ell_{\text{to}}$).
3.3.3. *match*(α, `onentry_action/2`, α.`name` = s_{enter});
   if *exits*(α) *append*(α.`action`, t.`action`).
3.3.4. *match*(α, `onexit_action/2`, α.`name` = s_{leave});
   if *exits*(α) *insert*(α.`action`, t.`action`).
3.3.5. *match*(α, `do_action/2`, α.`name` = s_{leave});
   if *exits*(α) and t.`event` = '`event(completed, {`$s_{\texttt{leave}}$`})`'
      insert('`action(log, "STOP {`α.`name}"))`', t.`action`),
   otherwise *insert*('`action(log, "ABORT {`α.`name}"))`', t.`action`).
3.3.6. *match*(α, `do_action/2`, α.`name` = s_{enter});
   if *exits*(α) *append*('`action(log, "START {`α.`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.

Subroutine *PCartesian*

For each s_{top} in *match*(s, `state/1`, *exists*(r, `region/1`, r.`state` = s)) do:
1. s_{new} = *new-id*('s'); *add*(`substate/2`, $\langle s_{\text{top}}, s_{\text{new}} \rangle$); *add*(`par/2`, $\langle s_{\text{new}}, \{\} \rangle$).
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(ℓ, `par/2`, ℓ.`state` = s_{new}); *append*(y.`state`, ℓ.`list`).
3. Set $l \leftarrow \{s_{\text{new}}\}$.
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-id*('s'); *add*(`substate/2`, $\langle s_{\text{top}}, s_{\text{new}} \rangle$);
   add(`par/2`, $\langle s_{\text{new}}, p' \rangle$); *append*($s_{\text{new}}, l$).
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`, $\langle s_{\text{new}} \rangle$).
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.




```
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.

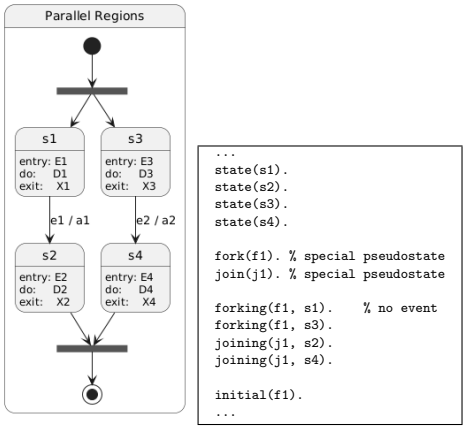


Fig. 6: Equivalent UML inner-states using join/fork pseudostates.

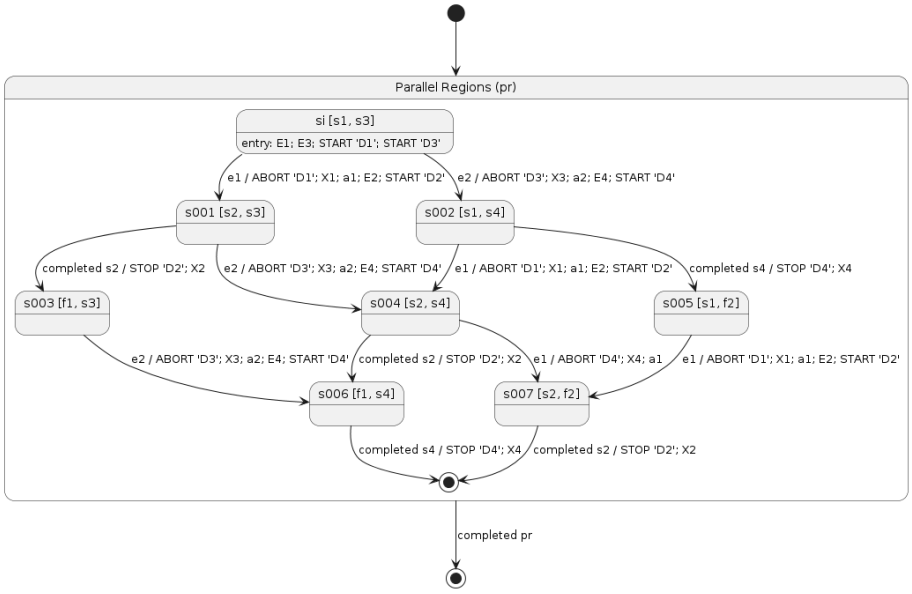


Fig. 7: Generated equivalent expanded machine without parallel regions.

3.5 Complexity and Correctness of the Flattening Process

We verified the correctness of the algorithm by using a database of case-studies with nested composite states, with both implicit and explicit events. We used complex behaviors to verify the resulting sequence of actions [10]. For simplicity, we did not include an external event in the complex region.

The flattened model though at a low level of abstraction, serves as a tool to aid in the behavior analysis of the state machines. While the resulting EFSM

includes more vertices compared to the number of states in the original UML state machine, it makes all transitions explicit. This can aid in behavior analysis of the initial machine (correctness, complexity, and welformedness). The formal proof of correctness may be provided by using formal definition of UML state machines. We plan to address this in future.

4 Interoperability among text-to-UML Drawing Tools

There exist, currently, a number of industrial tools that allow developers to provide a textual description of UML diagrams, which are subsequently visualized. Examples include PlantUML, Mermaid, and others, that normally support a number of different diagrams [11]. One of the problems we have identified is that there does not exist any interoperability between these tools, e.g. a visual representation created by one cannot be backward (i.e. visual-to-text) mapped to a different specification (and subsequently extended or modified). The lack of interoperability is mainly caused by missing and/or custom implementation of certain UML features (i.e. pseudostates, junctions, and region completion). Our approach to resolving the interoperability issue is to use our declarative model as a common descriptive language among the different visualization tools. There are two types of model transformation, as shown in Figure 8.

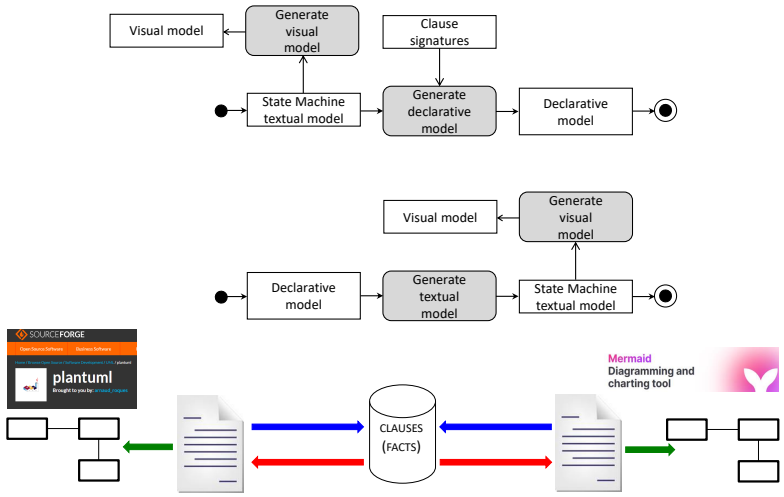


Fig. 8: Transformation activities to support visualization tool interoperability.

A state machine is initially represented as a textual model in some text-to-UML drawing package like e.g. PlantUML. The package can provide a visualization of the machine as a state transition diagram. Our first model transformation is to represent the machine in a declarative model, as shown in Figure 8. Utilizing Prolog to establish a declarative representation of state machines offers an simple and powerful method to depict the diverse constituents of a state machine,

encompassing states and transitions. State machines often encompass intricate and diverging behavior, introducing complexity in ensuring exhaustive testing of all conceivable paths. Prolog's functionalities such as pattern matching and backtracking render it particularly apt for simulating the intricate behavior of complex systems. Furthermore, Prolog offers the valuable assets of a query engine and a query interface, which play a pivotal role in streamlining the process of flattening a state machine. This technology enables us to seamlessly navigate the intricacies of state machines by formulating and executing queries that extract essential information about states and transitions. Additionally, Prolog's declarative nature provides the flexibility to expand the model's capabilities. By introducing custom Prolog rules, we gain the ability to delve into the study of behavioral patterns, complexity analysis, and overall design intricacies inherent in the underlying state machine. This strategic incorporation of Prolog not only facilitates our immediate goals but also lays a solid foundation for comprehensive exploration and understanding of the state machine's behavior and structure.

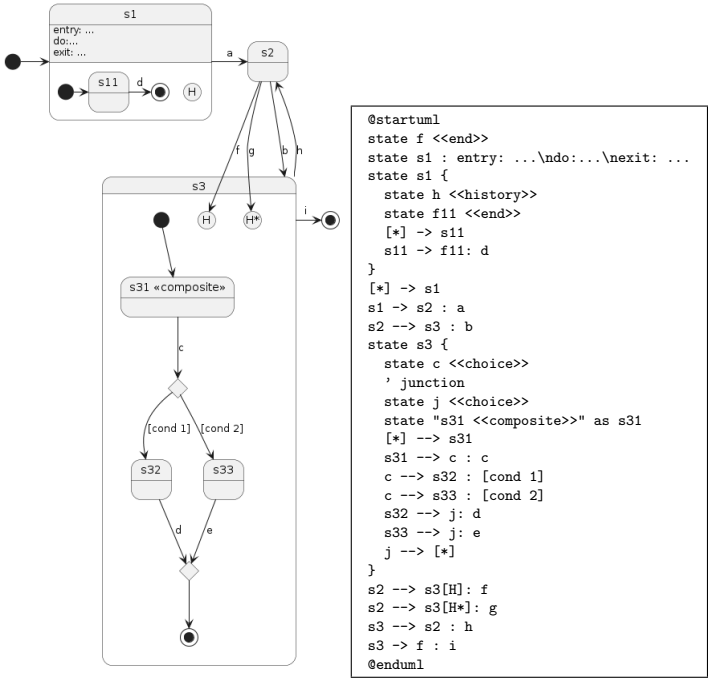


Fig. 9: A sample diagram in PlantUML, illustrating various UML features.

The transformation from the declarative model to a visualization tool deploys a template-based approach. Every clause is mapped into certain annotated code-blocks with placeholders, in which the element name, id, and underlying attributes are codified. The full diagram is then embedded in a top level template with an appropriate header and footer. Visualization tools would not necessarily support the most recent of the UML specifications in their entirety.

A sample diagram in PlantUML highlighting the above features is given in Figure 9. Indicatively, we can refer to PlantUML where a number of elements are not supported such as History annotated state (see state `s1`, as opposed to the History entry pseudostate in `s3`), the composite annotated state (state `s31`), state behavior (state `s1`), junction pseudostates (state `s3`). As illustrated, missing features are implemented using mock states, as alternative notations, to be replaced by proper notation when the feature is supported by the tool. The transformation from the visualization tool to the common declarative model is challenging, as we need to identify and detect any and all alternative notations which will have to be properly codified in the declarative model. To achieve this, our UML parser searches for certain keywords in the textual description and when it finds a match, it automatically puts the detected element (i.e. the state behavior) in the declarative model. Furthermore, since the declarative model requires specific event types, certain assumptions are made. For instance, we assume all event types are call events, all do behaviors are processes, and all actions (including the actions in the state behaviors) are log actions. In general, it is desirable to have unique element IDs in the diagram. However, this is not enforced by the tools. To address this, we define an alias clause (see Table 1). Another example of using the alias clause is in the process of flattening parallel states, in which, the link between the newly generated states and the expanded states is maintained (see Figure 7).

5 The Common Declarative Language as a Database

State machine can be deployed during requirements analysis to capture functional requirements such as a use case and thus can provide a helpful tool for the validation of the requirements. The machine can also serve as a tool further down the line of the development during testing for the verification of the requirements. With the declarative model as is, we can execute simple (ground and non-ground) queries that can give us some basic knowledge of the machine. We extend the database with rules that reason about graph navigation and graph complexity. These two aspects would correspond to the observable behavior and the properties of the underlying machine, which can be subsequently mapped to the functional and non-functional requirements respectively. Available rules include `in_degree/2` that succeeds by returning the in-degree of a state, and `get_all_internals/1` that succeeds by finding all state-event-action triplets over internal transitions whose definitions and sample execution are shown below:

```
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).
```

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

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

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

When we study the observable behavior of the machine, we want rules that reason about elements such as the exposed interface and legal event sequences. When we study the properties of the machine, we want rules that reason about elements such as connectivity and measurements. A third aspect of analysis is the well-formedness of the machine. Example issues include the presence of infinite loops, dead ends, or conflicts with the UML specification, such as e.g. the existence of an internal transition without an action association. If an issue is present in a machine, then there are two issues to consider: If the machine faithfully maps requirements, then the conflict originates in requirements and the discovery of such conflict aids in requirements validation where developers pose the following question: "Are we building the right product?" Otherwise, if the machine does not faithfully map requirements, then the discovery of such conflict aids in the proper construction of the machine.

6 Conclusion and Future Work

The common declarative language of a UML state machine serves initially as a textual representation. Text-to-UML drawing tools can deploy this language in a model transformation to create a repository of representation as well as to support tool interoperability, as a machine produced by one tool can then be represented declaratively and read by another tool. A possible challenge and limitation to this is the fact that not participants (UML specification, visualization tools and declarative language) may support the exact same set of UML elements, so compatibility may not always be full, while at the same time it is never static as all participants constantly evolve. Our extended finite state machine 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. Both the initial and the flattened representations can serve as declarative databases, where one can execute queries in order to extract more knowledge about the machine and consequently on the corresponding functional and non-functional requirements of the component that the machine represents, whether the system in its entirety,

a case study, or other. In previous work, we concentrated on the fundamental features of the UML specification, where the common declarative language was deployed mainly as a database and the basis for simulation. In this paper we addressed one of the major advanced features of the UML specification, namely the presence of orthogonality, while complementing on previous work concentrating on representation, model transformation (flattening) and visualization tool interoperability. Future work should address the second major advanced feature of a UML state machine, namely the presence of the History pseudostate.

References

1. Daniel Balasubramanian, Corina S. Păsăreanu, Gábor 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, Universität 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-Universität Freiburg, 2015.
11. `https://modeling-languages.com/text-uml-tools-complete-list/`