Declarative Representation of UML State Machines for Querying and Simulation

Zohreh Mehrafrooz(✉), Ali Jannatpour, and Constantinos Constantinides

Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada
zohreh.mehrafroozmayvan@mail.concordia.ca,
{ali.jannatpour,constantinos.constantinides}@concordia.ca

Abstract. Among the various aspects of the Unified Modeling Language, state machines are utilized to model the dynamic behavior of reactive systems. In this paper we present a platform where we transform a state machine into a declarative model, implemented as a database of clauses in Prolog. To tackle the complexity of composite states, we propose an algorithm for flattening the state machine's representation. Both initial and flattened declarative models allow for querying on the quality attributes, the behavior and the well-formedness of the underlying machine. To complement the query-based analysis, we present a simulation process and we describe its automation and tool support. We demonstrate the analysis through a case study. The approach can assist software developers while performing validation of requirements.

Keywords: UML state machines · Model transformation · Declarative modeling · Simulation · Automation

1 Introduction and Motivation

Originally introduced by Gill in 1962 [7] and later proposed by Harel in 1987 [8] as a significant extension over traditional finite state machines, statecharts are a visual formalism for modeling the dynamic behavior of components at various levels of abstraction. The Unified Modeling Language (UML), an industrial de facto standard that supports software modeling, adopted Harel's statecharts in its specification and extended them. This study is based on the extended statechart model, referred to in the literature as "UML state machine" (or "UML statechart"). A state machine can model the behavior of a reactive system at any level of abstraction. In this study, we define a declarative representation of a state machine, and construct a platform to analyze the machine through queries and simulation. The objective is to assist in the validation of system requirements captured by the machine and the methodology entails the study of quality attributes, behavior and well-formedness of the machine as well as simulation.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
B. Tekinerdoğan et al. (Eds.): ECSA 2023, LNCS 14590, pp. 133–150, 2024.
https://doi.org/10.1007/978-3-031-66326-0_9

A declarative model is a powerful and intuitive way to represent state machines, offering numerous advantages in terms of maintainability, scalability, and analysis. In fact, declarative representation expresses the behavior and transitions of the state machine using logical clauses and rules. This model can be implemented in Prolog, which provides capabilities like pattern matching and backtracking, making it well-suited for modeling complex behavior in state machines [16]. Sheng et al. [15] present a Prolog-based consistency checking for UML class diagrams and object diagrams. They formalize the elements of the model and then convert the model into Prolog facts along with some consistency rules that enable querying of the properties, elements, and subsequent parts of the model. Similarly, Khai et al. [12] propose a Prolog-based approach for consistency checking of class and sequence diagrams. State machines are widely utilized in software testing to evaluate performance and quality against predefined requirements. Hashim and Dawood [11] conduct a review of test case generation methods that use UML statecharts. Chen and Lin [3] propose a test case generation strategy that enhances efficiency and guarantees high test coverage and accuracy. Aktaş and Ovatman [1] discuss statechart anti-patterns which may occur in software development process.

Using a declarative model, the static behavior of a system can be studied and the system requirements can be validated. Additionally, statecharts are a widely-used notation for representing the dynamic and executable behavior of complex systems [5]. This highlights the significance of having tools for visualizing and simulating statecharts. Mens et al. [13] introduce a technique to improve statechart design using specialized tools including a modular Python library called Sismic [5]. Van Mierlo and Vangheluwe [17] present an approach for modeling, simulating, testing, and deploying statecharts. Balasubramanian et al. [2] introduce Polyglot, a framework for analyzing models described using multiple statechart formalisms. Their approach involves translating the structure and behavior of statechart models into Java and analyzing them using pluggable semantics. Modeling state machines with nested composite states and flattening the model has been a challenge. One major issue is the potential occurrence of unwanted non-determinism which has also been studied in the literature [9,10], and [17]. E. V. and Samuel [6] describe a technique to transform hierarchical, concurrent, and history states into Java code using a design pattern-based methodology.

We structure the remainder of this paper as follows: We provide a background to the mathematical specification of a state machine in Sect. 2. We present an overview of our approach and the case study in Sect. 3. We present our initial declarative model in Sect. 4 and describe our query system in Sect. 6. We present our flattened declarative model in Sect. 5; and the simulation process in Sect. 7, together with a discussion on the results of a given scenario. We finally present our conclusion.

2 Background and Assumptions

UML 2.5.1 [14] provides numerous complex features, such as composite and nested states; entry and exit pseudostates; entry, exit, and do state behavior;

as well as implicit region completion transitions. These features lead to a complex behavioral analysis. We simplify the machine by converting it into a modified Extended Finite State Machine EXTENDED FINITE STATE MACHINE (EFSM), as specified in the subsequent section. Moreover, the standard UML does not allow ϵ-transitions. An ϵ-transition is a transition whose *event* and *guard* are empty. Observe that ϵ-transitions are only allowed in pseudostates (i.e. entry and exit), as well as region completion (i.e. in the case of the completion of a *do* behavior, or reaching a final substate).

2.1 Modified Extended Finite State Machine (EFSM)

The EFSM is formally defined as a 7-tuple [4]. Our definition of EFSMs adapts this 7-tuple, with a slight modification on the inputs of the transition. An EFSM M, is defined as a 7-tuple $(Q, \Sigma_1, \Sigma_2, q_0, V, \Gamma, \Lambda)$, 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 *starting 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*,
$\Lambda = \{\lambda : q \xrightarrow{e_i[g_i]/a_i} q', i \in \mathbb{Z}\}$, is a finite set of *deterministic* transitions defined on $Q \times \mathring{\Sigma}_1 \times \mathring{\Gamma} \rightarrow Q \times \mathring{\Sigma}_2$, where $\mathring{\Sigma}_1 = \{\epsilon\} \cup \Sigma_1$, $\mathring{\Gamma} = \{\epsilon\} \cup \Gamma$, $\mathring{\Sigma}_2 = \{\epsilon\} \cup \Sigma_2$, ϵ denotes *null*, $q, q' \in Q$, $e \in \mathring{\Sigma}_1$, $g_i \in \mathring{\Gamma}$, and $a_i \in \mathring{\Sigma}_2$ are all *bindable* string literals.

A guarded ϵ-transition is represented by $\lambda : q \xrightarrow{e_i[g_i]/a_i} q'$ where $e_i = \epsilon$. In the case where $g = \epsilon$, the transition is referred to as ϵ-transition. In order for Λ to be deterministic, for every state $q \in Q$, at most one possible transition must exist. In other words, $\forall q \forall \lambda_i : q \xrightarrow{e_i[g_i]/a_i} q'$, the satisfiability of (e_i, g_i) must be exclusive. While this property holds for all EFSMs, we enforce the following restrictions:

1. If state q has an outgoing ϵ-transition, no other outgoing transitions are allowed on q.
2. If state q has an outgoing guarded ϵ-transition, only other guarded ϵ-transitions are allowed on the state. Let $\{g_i\}$ be the set of all guards for all guarded ϵ-transitions on state q. i) $\cup g_i = \texttt{True}$; ii) $\forall i \forall j \neq i \; (\neg(g_i \wedge g_j))$.

3 Overview of the Approach and Case Study

An overview of our approach is illustrated in the UML activity diagram of Fig. 1, and the various aspects of the diagram will be discussed in the subsequent sections through a case study that models an alarm system, shown in Fig. 2 and Fig. 3.

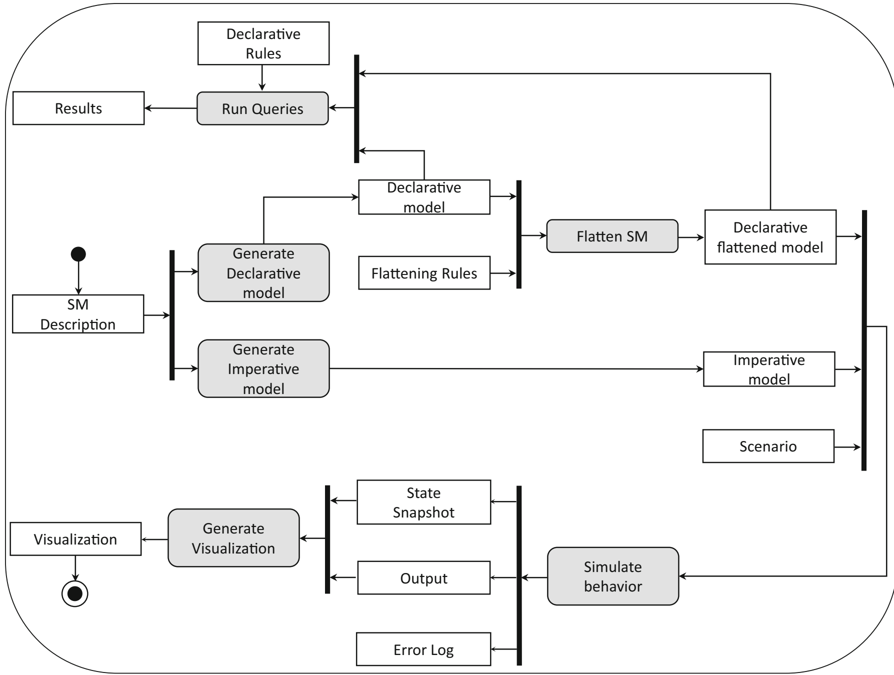

Fig. 1. UML activity diagram of the approach.

4 Transformation of the State Machine into a Declarative Model

The first part of this task is to provide a platform that can serve as a *virtual machine* for analysis of a state machine. The model consists of a declarative representation of a machine, following a defined structure of clauses, implemented as Prolog facts, that represent the state machine as a cyclic directed multigraph, where states are modeled as nodes and where transitions are modeled as edges. Unary clauses such as `state/1`, `pseudostate/1`, `initial/1`, `final/1` model their respective language element and `proc/1` defines a *do* behavior. Binary and multi-arity clauses are defined in Table 1.

4.1 Modeling Events

In this declarative model, events are represented by the `event/2` clause, implemented as `event(type, argument)`. The supported event types in accordance with the UML specification include *call*, *signal*, *time* and *change*. Additionally, we introduce three new event types: *inactivity*, *update* and *completion*. A brief description of all event types is shown below:

call: An external event that triggers a transition. Makes use of keyword `call`.

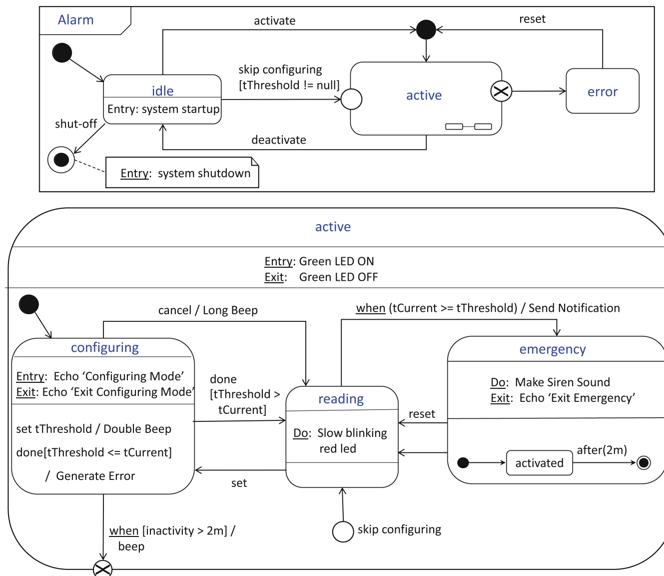

Fig. 2. Case study: Alarm.

```
% 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")).
% level 2
initial(configuring).            superstate(active, configuring).
superstate(active, reading).     superstate(active, emergency).
onentry_action(active, action(log, "Green LED ON")).
onexit_action(active, action(log, "Green LED OFF")).
onentry_action(configuring, action(exec, "echo('Configuring mode');")).
onexit_action(configuring, action(exec, "echo('Exit configuring mode');")).
do_action(reading, proc("Slow blinking red LED")).
transition(configuring, reading, event(call, cancel), nil, action(exec, "longBeep();")).
transition(configuring, active_exit, event(timeout, "2:00"), nil, action(exec, "beep();")).
transition(reading, emergency, event(when, "tCurrent >= tThreshold"),
    nil, action(exec, "sendNotification();")).
transition(reading, configuring, event(call, set), nil, nil).
transition(emergency, reading, event(call, reset), nil, nil).
transition(emergency, reading, nil, nil, nil). % completed emergency
transition(configuring, reading, event(call, done), "tThreshold > tCurrent", nil).
internal_transition(configuring, event(set, tThreshold), nil, action(exec, "doubleBeep();")).
internal_transition(configuring, event(call, done),
    "tThreshold <= tCurrent", action(exec, "generateError();")).
% level 3
initial(activated).    final(efinal).    alias(efinal, "").
superstate(emergency, activated).    superstate(emergency, efinal).
do_action(emergency, proc("Make Siren Sound")).
onexit_action(emergency, action(exec,"echo('Exit Emergency');")).
transition(activated, efinal, event(after, "2:00"), nil, nil).
```

Fig. 3. The initial declarative model of the alarm case study.

signal: Triggered by an internal or external clock, which indicates a specific time for triggering a transition. Makes use of keyword `at`.

time: When the source state has been active for a specified length of time, the transition occurs if its guard evaluates to true. If no guard is present (nil), a transition occurs automatically. Makes use of keyword `after`.

change: Triggered by a constantly evaluated condition once true. Makes use of keyword `when`.

inactivity: The system is expected to be inactive over a given amount of time, specified by the argument. Though treated as a time event, it makes use of keyword `timeout`.

update: Updates the value of a variable or attribute, which may subsequently trigger a transition if the new value satisfies the conditions for the transition. Makes use of keyword `set`.

completion: Occurs when a region concludes or a `do` behavior completes, modeled as `event(completed, ?state)`, where `?state` represents the current state (or region). Makes use of keyword `completed`.

4.2 Modeling Actions

We classify actions into `EXEC` and `LOG`. This classification provides the means to manage each action type differently, allowing for greater flexibility in the model. This classification is particularly useful when we need to flatten the model (see Sect. 5), as it allows us to easily identify and apply the appropriate processing to each type. Finally, the model introduces `action/2` to codify actions. The case study illustrates actions that are executed by the script engine (e.g. invoking the `echo()` method) as well as actions that are logged by the system (e.g. `Green LED OFF`). Note that a *do* behavior is a *process* that is *started* when the machine enters a state and may be *stopped* (upon successful termination) or *aborted* (triggered by an *exit* event). Finally, in Fig. 2, `system shutdown` is implemented as an entry behavior of the final state, since a final state cannot have an exit behavior.

5 Flattened Representation of UML State Machines

We extend the initial declarative model and develop an algorithm that flattens the machine. We believe that a flattened model can provide a platform for deeper analysis as well as a simulation of behavior (see Sect. 7). The flattened model provides the same semantic model as the initial model, though at a lower level of abstraction, being analogous to the bytecode platform for languages such as Java and Clojure, which is a seamless virtual machine. The flattened model can also be extended with rules that target the three aspects of our analysis (quality attributes, behavior, and well-formedness).

Table 1. Major clause signatures of the initial declarative model.

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

5.1 The Flattening Process

In a complex UML machine, transitions can trigger various sequences of actions. For example, when transitioning from *idle* to *active*, while the transition itself has no action, the *activate* event triggers the *entry* action on *active* before transitioning into *configuring*. Similarly, when transitioning from *activated* (substate of *active*) to *idle*, a sequence of actions is executed: *aborting* 'Make Siren Sound', *executing* echo('Exit Emergency'), and *logging* 'Green LED OFF'.

To analyze the behavior of the UML state machine, we convert it into a *flattened* EFSM by chaining the subsequent actions using ϵ-transitions. Our flattening algorithm consists of 4+1 passes, progressively eliminating complex UML features such as composite states, pseudostates, state behaviors, and inter-

nal transitions. Each pass involves multiple steps, modifying facts and reducing complexity until the machine is fully flattened. Finally, the resulting machine is minimized by reducing the number of states and combining equivalent transitions. Prolog queries are used as selectors to process the working database. An outline of the flattening algorithm is presented on the next page:

Procedure *Flatten*(Input: UML in decl. DB, **Output:** EFSM in decl. DB**)**
Pass 0: *Preprocessing*
1: Convert all outgoing `nil`-events form state *s* to *event(completed, s)*.
2: Convert all actions to action-lists.
Pass 1: *Processing pseudostates, entry, exit, and do behaviors*
1: *Resolving do behaviors*: For each state *s* with *do* behavior with process *p*: i) Append "start *p*", insert "abort *p*" notification actions to the *entry* and *exit* actions of state *s*, respectively; ii) For every *completed* event on state *s*, insert "stop *p*" notification action to transition's actions; iii) Remove the *do* behavior from *s*.
2: *Resolving entry/exit pseudostates:* i) Replace all `entry_pseudostate(s, t)` clauses with `transition(s, t, nil, nil, [])` and `superstate(p, s)` where `superstate(p, t)`; ii) Change all `exit_pseudostate(s, p)` clauses to `superstate(p, s)`.
3: *Resolving entry behaviors:* Starting from top to bottom, for every state with *entry* behavior: i) Find `onentry_action(s, a)` and remove it; ii) For each incoming transition from an external state *x* to *s*: append *s* to the transition's action list; iii) For each incoming transition from an external state *x* to a substate *b* of *s*, append *a* to the transition's action list; iv) If *s* is a top-level initial state, create a new state *ps*, add `state(ps)`; change `initial(s)` to `initial(ps)`, and add `transition(ps, s, nil, nil, a)`; v) Otherwise if *s* is a non-top-level initial state, find *p* where `superstate(p, s)`; add `superstate(p, ps)`; change `initial(s)` to `initial(ps)`, and add `transition(ps, s, nil, nil, a)`.
Pass 2: *Full State Resolution*
1: For each composite state *p* do the following: i) Obtain the list of immediate substates of *p* into *l*; Obtain the exit behavior of *p* into *ea*; ii) Change the target state of all incoming transitions to *p*, to the initial substate of *p*; iii) For each non-final substate *s* of *p* repeat: a) Inherit all outgoing `nil`-transitions from the superstate, if the child state does not contain a `nil`-transition; b) For every outgoing transition from the state *s* to a state that is not in *l*, including the above; insert *ea* to the transition's action list, if $ea \neq$ `nil`; c) Replace `superstate(p, s)` with `state(s)`.

iv) Find inner final state *f* (if applicable); remove both `superstate(p, f)` and `final(f)`; add `state(f)`; for each `transition(p, t, e, g, a)` from *p* to the target state *t* where *e* is a *region completion* event on *p*: add `transition(s, t, nil, g, a)`; insert *ea* to *a*, if $ea \neq$ `nil`; v) Remove the composite state *p*, its behaviors, and all its outgoing transitions.
2: For each remaining state *s* with *exit* behavior *e*, insert *e* to all outgoing transitions' actions list and remove the *exit* behavior clause.

3: For each internal transition on state s, convert `internal_transition` to `transition` to self.

Pass 3: *Post-Processing*

1: For all action lists containing "stop p", find corresponding "abort p" in the list; remove "stop p", and change "abort p" to "stop p".

2: For all `transition(`s, t, e, g, l`)`, where length(l) $>$ 1, create intermediary state i, replace the original transition with `transition(`s, i, e, g, `head`(l)`)` and `transition(`i, t, `nil, nil, tail`(l)`)`; Resolve `transition(`i, t, `nil, nil, tail(`l`))`, recursively. 3: Replace all `transition(`s, t, e, g, `[])` with `transition(`s, t, e, g, `nil)`.

Pass 4: *State Reduction/Minimization*

For each `transition(`s, t, e, g, a`)`: Find all `transition(`s_2, t, e, g, a`)` where s_2 is not initial and $s_2 \neq s$. Replace all `transition(`x, s_2, e_2, g_2, a_2`)` with `transition(`x, s, e_2, g_2, a_2`)`. Remove all instances of `state(`e_2`)` and `transition(`e_2, t, e, g, a`)`. Repeat until no more transitions can merge.

Having produced a flattened model, we perform a model transformation into a (new) declarative representation, deploying only the clause structures `state/1`, `initial/1`, `final/1`, `transition/5`, `event/2`, and `action/2`.

```
state(pre_idle).    state(idle).        state(configuring).
state(error).       state(active_exit).
final(final).       alias(final, "").
state(s71).         state(s41).         state(s31).
state(s12).
. . .
transition(pre_idle, idle, nil, nil, action(log, "System Startup")).
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(s71, configuring, nil, nil, action(exec, "echo('Configuring mode');")).
transition(configuring, configuring, event(set, tThreshold), nil, action(exec, "doubleBeep()
transition(configuring, configuring, event(call, done),
    "tThreshold <= tCurrent", action(exec, "generateError();")).
transition(configuring, s31, event(timeout, "2:00"), nil,
    action(exec, "echo('Exit configuring mode');")).
transition(s31, active_exit, nil, nil, action(exec, "beep();")).
transition(active_exit, error, nil, nil, action(log, "Green LED OFF")).
transition(error, s71, event(call, reset), nil, action(log, "Green LED ON")).
transition(configuring, s41, event(call, cancel), nil,
    action(exec, "echo('Exit configuring mode');")).
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(s12, reading, nil, nil, action(log, "START 'Slow blinking red LED'")).
transition(reading, s71, event(call, set), nil, action(log, "ABORT 'Slow blinking red LED'")
. . .
```

Fig. 4. Partial flattened declarative model of the alarm case study.

Figure 4 includes a partial model capturing transitions from states *idle* and *configuring* to *reading*. Consider the transition from *idle* to *configuring* in Fig. 2. Such transition causes `system startup` notification upon entry to *idle*. The reception of the event `activate` causes a transition to the *active* superstate which is now collapsed. Upon reaching *active*, the transition causes `echo`

`configuring mode` upon entry to the *configuring* substate. Such sequence of actions are implemented in the flattened model by sequence of transitions starting from *initial*, to *pre_idle*, *idle*, *s71*, and finally to *configuring*. Note that one may extend the model to support transition with multiple actions, in which case, an extra step in pass 4 may reduce the total number of states by following and merging all outgoing `nil`-transitions into a single transition. We intentionally avoided this to make the model compatible with the definition of EFSMs (Fig. 5).

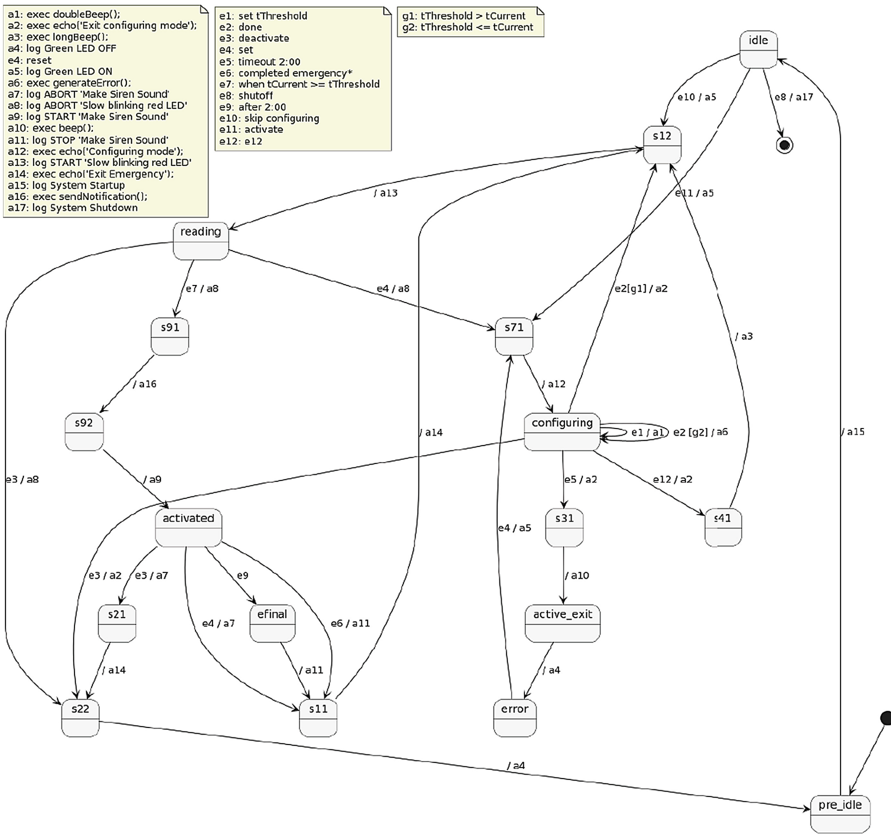

Fig. 5. The flattened UML diagram

6 Building a Query Platform

With the declarative model as is, we can execute simple ground queries that can give us some basic knowledge of the machine such as *"Is there a transition from state idle to state configuring?"*

```
? transition(idle, configuring, _, _, _).
Yes.
```

We can also execute non-ground queries such as *"Under what conditions, if any, would the state machine perform a transition to the emergency state?"* This would entail capturing any and all `state-event-guard` triples that can cause such a transition.

```
? transition(State, emergency, event(_,Event), Guard, _).
Event = "tCurrent >= tThreshold",  Guard = null,   State = reading
```

6.1 Extending the Declarative Model with Rules

We can extend the declarative model by introducing rules. We can identify three types of rules: (1) We have rules that reason about the *behavior* of the state machine by examining the traversal of the underlying graph under various different conditions. When we study behavior, we want rules that reason about elements such as the exposed interface and legal event sequences. (2) We have rules that reason about the *quality attributes* of the state machine by examining the properties and measurements of the underlying directed graph. When we study graph (machine) complexity we want rules that provide knowledge about aspects such as connectivity and (global and nodal) measurements. We argue that the above two types of rules roughly correspond to the state machine's functional and non-functional requirements. (3) We have rules that reason about the well-formedness of the machine, such as the presence of infinite loops, dead ends, or conflicts with the UML specification e.g. the existence of an internal transition without an action association.

6.2 Studying Behavior

Exposed Interface: The *call* and *set* events correspond to messages sent to the system and they collectively constitute the exposed interface of the system. Rule `get_interface/1` succeeds by collecting any and all such events.

```
get_interface(Interface) :- %% Consults: Initial model.
   findall(E, (transition(_, _, E, _, _),
            (E = event(call, _); E = event(set, _)));
        (internal_transition(_, E, _, _),
            (E = event(call, _); E = event(set, _))),
        EventList), list_to_set(EventList, Interface).
```

Legal Events at a Given State: Given the system exposed interface, it is important to note that not all events can be acted upon unconditionally. An event can be accepted based on the system's current state. It will be acted upon provided the associated guard (if one is present) evaluates to true.

```
is_legal(State, Event) :- %% Consults:  Initial model.
  transition(State, _, event(_, Event), _, _);
  internal_transition(State, event(_, Event), _, _).
```

6.3 Studying Complexity

We provide rules for properties and measurements. Measurements in graphs can be global or nodal. Global measurements refer to global properties of the graph and consist of a single number for any given graph. Nodal measurements refer to properties of the nodes and consist of a number for each node for any given graph.

Order of Graph: This measurement refers to the number of nodes in a graph. In the context of state machines, we believe that the initial model may not give us an accurate picture due to the presence of composite states. The flattened model would be more accurate for this measurement. For the initial and flattened models the corresponding rules are shown below:

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

%% Consults:  Flattened model.
order(N):- findall(S, state(S), Length), length(Length, N).
```

Number of `nil` Transitions: The number of *nil* transitions in a flattened model can be a measure of the complexity of a state machine. The following rule succeeds by returning the number of *nil* transitions:

```
nil_transition(N) :- %% Consults: Flattened model.
  findall(Nilevents,
    (transition(_, _, Nilevents, _, _), Nilevents=nil), Transitions),
  length(Transitions, N).
```

Size (or Length) of Graph: This measurement refers to the number of edges in a graph. In the context of state machines, we believe that the initial model may not give us an accurate picture due to the fact that in the presence of composite states, their nested states inherit the transitions of their superstate. The flattened model would be more accurate for this measurement.

```
size(N):- %% Consults: Flattened model.
  findall(S, transition(S,_,_,_,_), Length), length(Length, N).
```

6.4 Studying the Well-Formedness of the State Machine

We define rules to study the design of the state machine and find cases such as dead ends, conflicts, or inconsistencies among the state machine's elements, considering issues such as (1) Dead ends and infinite loops, (2) Internal transition without an action, (3) Multiple change events originating from the same state, (4) Non mutually exclusive guards originating from the same state, (5) The absence of a do behavior in the presence of an external transition with no event, and (6) As the previous item for a composite state, in the absence of an exit substate.

Dead Ends: We are interested in finding out if the machine can enter a state from which the final state is not reachable. Rule `dead_end/0` succeeds by obtaining a non-empty list of states from each of which there is no path to state `final`.

```
%% Consults:  Initial model.
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 \= [].
```

7 Simulating State Machine Behavior

The query system provides a level of analysis that is complemented with a simulation of the machine. The flattened model serves as the platform for simulation. A simulation reads in a machine representation and a scenario under which the machine is traversed and its state and behavior is monitored and recorded. The question we ask here is "*Is the Machine behaving according to its specification?*" During simulation, we need to be able to identify issues perhaps not having been identified by the query system, e.g. "*Has the simulator encountered an ambiguous transition?*", in which case we need to report such issues.

Structure of Scenario: A scenario is a sequence of commands consisting of three types of tags: `EVENT`, `EXECUTE`, and `TIME`. `EVENT` tags can be of type `call`, `set`, or `completion`, and must trigger the corresponding transition. `EXECUTE` tags contain expressions that modify variable values, and may trigger a transition. `TIME` tags can be either `after` or `at`, which update time variables `duration` and `absoluteTime` (if applicable) and may trigger a transition.

Read-Evaluate-Execute Cycle: In UML, it is assumed that a state machine processes one event at a time and finishes all the consequences of that event before processing next event [14]. At the highest level of abstraction, and given a scenario, the simulation would be performed using a Read-Evaluate-Execute Cycle. When a command in a scenario is `EVENT` e, where $e \in \Sigma_1$, given the current state and the event, the simulator would construct a `transition` query and consult the declarative model. We query the database and find all transitions $\lambda_i \in \Lambda$ with event e. The result of the query is a set of λ_i, associated with tuples $\{(q, g, a)_i\}$ where $q \in Q$ is the target state, $g \in \Lambda$ is a guard, and $a \in \Sigma_2$ is an action. Each tuple is also associated with a set of $v_i \subset V$, containing all variables used in g_i and a_i. The query is successful only if one transition is possible. This is achieved by instantiating all variables in v_i and evaluating g_i. Upon success, a single transition is fired. The simulator consequently checks if any additional transitions can be triggered, following the most recent transition. The process continues until no further possible transition is applicable.

Simulator Architecture: To perform a simulation, we need to provide storage of all variables (machine and environment) while keeping track of any changes.

We also need to provide storage and keep track of the machine's current state. To support these requirements, we provide an imperative model in Java while deploying Java Prolog Library (JPL). We use Javascript to maintain system variables, and we deploy the GraalVM engine to evaluate events and guards, and finally to execute actions. We illustrate the architecture of the simulator in the UML component diagram shown in Fig. 6. We illustrate the interaction of the various components during simulation with the UML sequence diagram of Fig. 7. The diagram illustrates the interactions among high-level objects, including `SimulatorExecuter`, `JPLMediator` (facilitating the communication with the declarative model), `ScriptHandler` (responsible for evaluating guards, actions, and modifying variables), a `Scenario` defined as a text file containing a sequence of events for simulation, and the `Output` generated by the tool. The outer loop in the sequence diagram illustrates the `Read-Evaluate-Execute` cycle and the inner loop mostly covers ϵ-transitions in our flattened model.

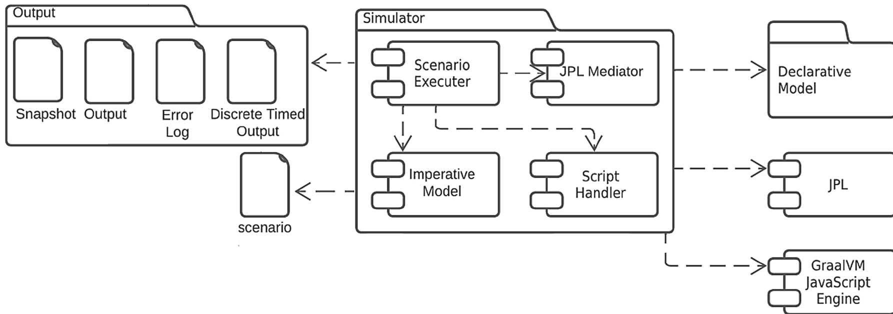

Fig. 6. UML component diagram of the simulator.

Results of Simulation in the Case Study: We applied the flattening algorithm to the declarative representation of our case study, and the resulting minimized flattened model is shown in Fig. 4. Also, Fig. 8 presents a sample scenario (top-left) along with the corresponding simulation output (top-right).

Visualization of Results: we visualize the results of simulating the scenario as the model of behavior which is shown in Fig. 8 (bottom). This diagram shows the current state of the state machine as well as state of the system in each time id.

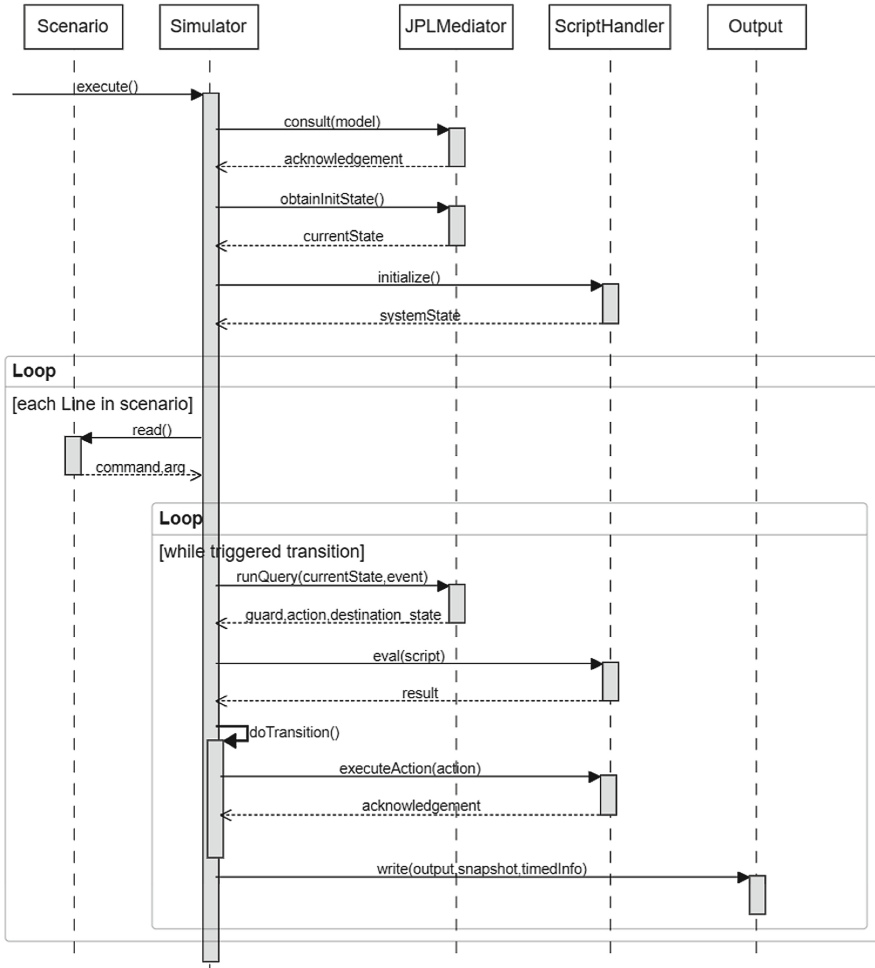

Fig. 7. UML sequence diagram of the simulation process.

Table 2. Complexity Metrics: Original vs Flattened Models

Metric	Original	Flattened
states and substates	9	18
internal initial states	2	0
transitions(+ internal)	16+2	29
entry/exit pseudostates	2	0
entry/exit (+do behaviors)	5+2	0
ϵ-transitions	2	11
actions	10	26

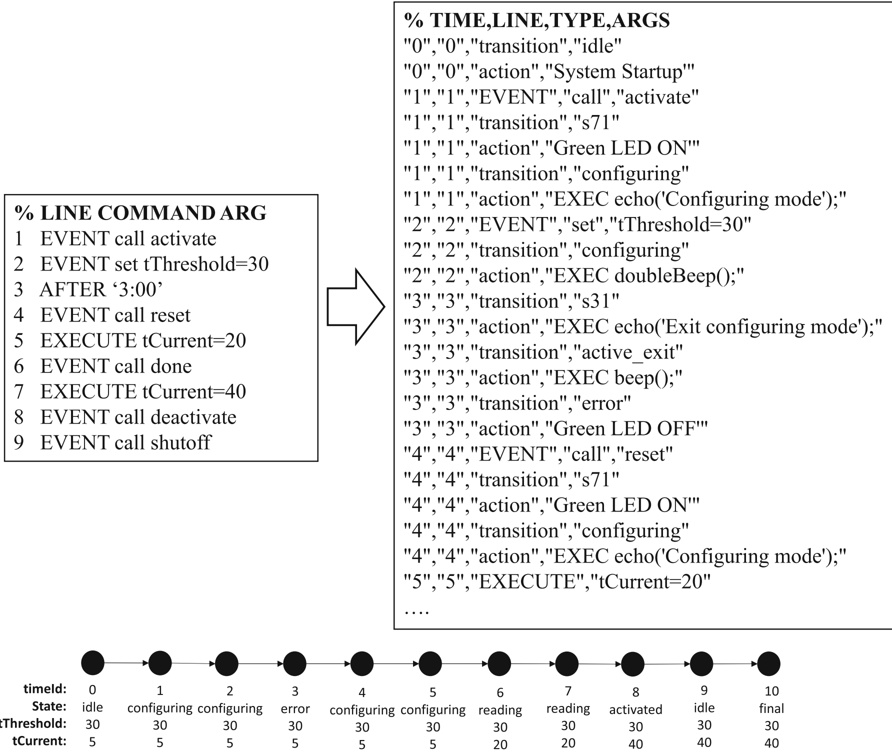

Fig. 8. Input scenario and the corresponding simulation output (top), and its model of behavior (bottom)

Conclusion

In this paper, we presented a declarative model to represent UML state machines. The model is used to study the dynamic behavior of the underlying machine. The simulation results provide insights into the machine's behavior under specific scenarios. We developed a simulation tool and a query engine that use the model in Prolog environment and run scenarios in an imperative platform. We deployed JPL for Java-Prolog interoperability. Our platform supports codified actions in JavaScript, by which developers may set or update system variables, in both the model, as well as in scenarios.

We introduced an algorithm to flatten the UML state machine and convert it into an extended finite state machine. Our algorithm supports major UML 2.5.1 features including single and composite states; exit and entry pseudostates; state behaviors including entry, do, and exit; in addition to the UML events including call, signal, time, change, as well as three newly introduced events namely inactivity, update, and completion. Table 2 lists some metrics that may be used to measure the complexity of the UML diagrams in both original and flattened models.

We used a modified version of the extended finite state machine to support guarded and unguarded ϵ-transitions that are required for handling complex sequences of actions and notifications in a non-flattened model. Future work may involve expanding the model to include contract considerations as well as other UML features such as history pseudostates and orthogonal regions.

Acknowledgments. The authors would like to thank Robin Laliberté-Beaupré and Simon Foo for their contributions to the automation and tool support for this project.

References

1. Aktaş, M., Ovatman, T.: UML statechart anti-patterns. In: 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 413–414 (2022)
2. Balasubramanian, D., Păsăreanu, C.S., Karsai, G., Lowry, M.R.: Polyglot: systematic analysis for multiple statechart formalisms. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 523–529. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_36
3. Chen, C., Lin, W.: Research of software testing technology based on statechart diagram. In: Pan, J.-S., Li, J., Namsrai, O.-E., Meng, Z., Savić, M. (eds.) Advances in Intelligent Information Hiding and Multimedia Signal Processing. SIST, vol. 211, pp. 314–322. Springer, Singapore (2021). https://doi.org/10.1007/978-981-33-6420-2_39
4. Cheng, K.T.T., Krishnakumar, A.: Automatic generation of functional vectors using the extended finite state machine model. ACM Trans. Des. Automation Electron. Syst. **1**, May 1999. https://doi.org/10.1145/225871.225880
5. Decan, A., Mens, T.: Sismic - A Python library for statechart execution and testing. SoftwareX **12**, 100590 (2020). https://doi.org/10.1016/j.softx.2020.100590. https://www.sciencedirect.com/science/article/pii/S2352711020303034
6. Sunitha, E.V., Samuel, P.: Automatic code generation from UML state chart diagrams. IEEE Access **7**, 8591–8608 (2019). https://doi.org/10.1109/ACCESS.2018.2890791
7. Gill, A.: Introduction to the Theory of Finite-State Machines. McGraw-Hill, Electronic Science Series (1962)
8. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9. https://www.sciencedirect.com/science/article/pii/0167642387900359
9. Harel, D., Kugler, H.: The Rhapsody semantics of statecharts (or, on the executable core of the UML). In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) Integration of Software Specification Techniques for Applications in Engineering. LNCS, vol. 3147, pp. 325–354. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27863-4_19
10. Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Trans. Softw. Eng. Methodol. **5**(4), 293–333 (1996). https://doi.org/10.1145/235321.235322. https://doi.org/10.1145/235321.235322
11. Hashim, N.L., Dawood, Y.S.: A review on test case generation methods using UML statechart. In: 2019 4th International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE), pp. 1–5 (2019). https://doi.org/10.1109/ICRAIE47735.2019.9037786

12. Khai, Z., Nadeem, A., Lee, G.: A prolog based approach to consistency checking of UML class and sequence diagrams. In: Kim, T., Adeli, H., Kim, H., Kang, H., Kim, K.J., Kiumi, A., Kang, B.-H. (eds.) ASEA 2011. CCIS, vol. 257, pp. 85–96. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-27207-3_10
13. Mens, T., Decan, A., Spanoudakis, N.I.: A method for testing and validating executable statechart models. Softw. Syst. Model. **18**(2), 837–863 (2019). https://doi.org/10.1007/s10270-018-0676-3. https://doi.org/10.1007/s10270-018-0676-3
14. Object Management Group: UML® 2.5.1 (2017). https://www.omg.org/spec/UML/2.5.1/
15. Sheng, F., Zhu, H., Yang, Z., Yin, J., Lu, G.: Verifying static aspects of UML models using Prolog. In: Perkusich, A. (ed.) The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10-12, 2019, pp. 259–342. KSI Research Inc. and Knowledge Systems Institute Graduate School (2019). https://doi.org/10.18293/SEKE2019-175, https://doi.org/10.18293/SEKE2019-175
16. Sterling, L., Shapiro, E.: The Art of Prolog: Advanced Programming Techniques, vol. 2. MIT Press, Cambridge (1994)
17. Van Mierlo, S., Vangheluwe, H.: Introduction to statecharts modeling, simulation, testing, and deployment. In: 2019 Winter Simulation Conference (WSC), pp. 1504–1518 (2019). https://doi.org/10.1109/WSC40007.2019.9004771