A Stochastic Extension of Stateflow

Stefan Kaalen
kaalen@kth.se
KTH Royal Institute of Technology
Department of Machine Design
Stockholm, Sweden

Anton Hampus
ahampus@kth.se
KTH Royal Institute of Technology
Department of Machine Design
Stockholm, Sweden

Mattias Nyberg
matny@kth.se
KTH Royal Institute of Technology
Department of Machine Design
Stockholm, Sweden

Olle Mattsson
ollemat@kth
Scania CV
Södertälje, Sweden

ABSTRACT

Although commonly used in industry, a major drawback of Stateflow is that it lacks support for stochastic properties; properties that are often needed to build accurate models of real-world systems. In order to solve this problem, as the first contribution, Stochastic Stateflow (SSF) is presented as a stochastic extension of a subset of Stateflow models. As the second contribution, the tool SMP-tool is updated with support for SSF models specified in Stateflow. Finally, as the third contribution, an industrial case study is presented.

CCS CONCEPTS

• **Theory of computation** → **Quantitative automata**; • **Computer systems organization** → **Reliability**.

KEYWORDS

Stateflow, SSF, SMP-tool, stochastic, model-based

ACM Reference Format:
Stefan Kaalen, Anton Hampus, Mattias Nyberg, and Olle Mattsson. 2022. A Stochastic Extension of Stateflow. In *ICPE '22: Proceedings of the ACM/SPEC International Conference on Performance Engineering, April 09–13, 2022, Bejing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3489525.3511679

1 INTRODUCTION

Nowadays, research and development in industry to a large degree rely on *model-based development*. One model-based tool that is well spread in industry is Stateflow[1]. Stateflow supports modeling of finite-state machines and has in many areas become an industrial standard practice, e.g., within automotive [14]. Even though very popular, Stateflow has the limitation that it does not explicitly support modeling of stochastic properties. To overcome this limitation, the present paper proposes a *stochastic extension* of Stateflow.

[1]https://www.mathworks.com/products/stateflow.html

ICPE '22, April 09–13, 2022, Beijing, China
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9143-6/22/04.
https://doi.org/10.1145/3489525.3511679

One area where modeling of stochastic properties is vital is within *model-based safety analysis*. Model-based safety analysis is of particular importance to support development of complex safety-critical systems [2, 6, 11]. Stochastic properties are needed e.g. to model random faults and failures, uncertain human behavior, and uncertain behaviors of complex sensors such as radars and cameras.

Stateflow is based upon state transition diagrams for which much of the groundwork was laid in the introduction of Statecharts [9]. In previous work, Jansen [10] has presented Stocharts as a stochastic extension of Statecharts. However, while the syntax is similar, the semantics of Statecharts and Stateflow have major differences [2, 14]. Therefore, the stochastic extension in Stocharts can not be applied to Stateflow.

The stochastic extension of Stateflow, proposed in the present paper, is divided into three contributions. The first contribution is a language named *Stochastic StateFlow* (SSF), which is a stochastic extension of a subset of the Stateflow language. The extension is twofold: a stochastic time delay of transitions, and a probabilistic choice of the destination state of transitions. SSF is presented with both syntax and semantics, and is constructed such that models can be easily built in the standard graphical user interface of Stateflow.

As the second contribution, the Matlab application SMP-tool, presented in [12], has been updated to support analysis of SSF models. Analysis of SSF models in SMP-tool is performed by Monte Carlo simulations. The previous version of SMP-tool [12] had a syntax and semantics that, to a much lesser degree, matched the syntax and semantics of standard Stateflow. Most importantly, the previous version did not support the so-called AND states used in Stateflow to model parallelism.

As the final contribution, an industrial case study of a subsystem of a gearbox from the heavy vehicle manufacturer Scania is presented. This subsystem is modeled as an SSF model and analyzed using SMP-tool.

The outline of this paper is as follows. In Sec. 2 and 3, the first contribution, i.e., the syntax and semantics of SSF models, is presented. Then, Sec. 4, the second contribution, i.e., the updated SMP-tool, is presented. Moreover, in Sec. 5, the third contribution, containing the industrial case study, is presented. Finally, in Sec. 6, related work is discussed.

2 SYNTAX OF SSF MODELS

The syntax of SSF models will first be presented in terms of an abstract syntax. Then, an SSF model built in Stateflow will be utilized to explain the concrete syntax. The goal has been to align the syntax of SSF models with the syntax of a subset of Stateflow discussed further in the end of Sec. 2.1.

2.1 Abstract syntax

The abstract syntax is presented in the same formalism as similar work in the area [4, 10]. Let $\mathcal{P}(\mathcal{A})$ denote the power set of a set $\mathcal{A}$, i.e., the set of all subsets of $\mathcal{A}$. The abstract syntax of SSF models is defined as follows.

Definition 2.1. An SSF model is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{S}^{\text{down}}, \texttt{initial}, \mathcal{F}, \mathcal{E}, \mathcal{G}, \mathcal{T})$ where

- $\mathcal{S}$ is a finite, non-empty set of states organized as an ordered tree by the three functions `children`, `type`, and `order`. The function $\texttt{children} : \mathcal{S} \to \mathcal{P}(\mathcal{S})$ associates each state with a possible empty set of children states. The function $\texttt{type} : \mathcal{S} \to \{\texttt{BASIC}, \texttt{AND}, \texttt{OR}\}$ assigns a type to each state. Leafs of the state tree have type `BASIC` and all other nodes have type `AND` or `OR`. The function $\texttt{order} : \mathcal{S} \to \mathbb{N}^+$ associates each state sharing the same `AND` parent s_i with a non-negative integer such that $\texttt{order}(s_j) \neq \texttt{order}(s_k)$ if $s_j \neq s_k$ for all $s_j, s_k \in \texttt{children}(s_i)$.
- $\mathcal{S}^{\text{down}} \subseteq \mathcal{S}$ is a subset of states (representing system failure).
- `initial` is a function $\{s_i \in \mathcal{S} \mid \texttt{type}(s_i) = \texttt{OR}\} \to \mathcal{S}$ that assigns one of the children states of each `OR` state as the initial state.
- $\mathcal{F} = \{F_i(t)\}_i$ is a set of Cumulative Distribution Functions (CDFs), each non-decreasing and satisfying $\lim_{t\to 0^-} F_i(t) = 0$ and
$\lim_{t\to\infty} F_i(t) = 1$. Note that by including degenerate distributions, deterministic times can also be represented in $\mathcal{F}$.
- $\mathcal{G}$ is a finite set of guards specified in BNF as:
guard ::= *proposition* | ($\neg$*guard*) | (*guard*$\vee$*guard*) | (*guard*$\wedge$*guard*)
proposition ::= `in(` **state** `)` | `after(` **CDF** `)` | $\mathbb{T}$,
where **state** is a symbol representing a state in $\mathcal{S}$, **CDF** is a symbol representing a CDF in $\mathcal{F}$, and where $\mathbb{T}$ denotes a *proposition* that evaluates to `true`.
- $\mathcal{E}$ is a finite set of events including the non-event $\diamond$ with the intuitive interpretation that no event is broadcast.
- $\mathcal{T}$ is a set of transitions given as tuples
$\tau = (s_\tau, \mathcal{E}_\tau^{\text{in}}, g_\tau, (\mathcal{S} \times \mathcal{E}) \to_\tau [0,1], n_\tau)$, where $s_\tau \in \mathcal{S}$, $\mathcal{E}_\tau^{\text{in}} \in \mathcal{P}(\mathcal{E})$, $g_\tau \in \mathcal{G}$, $(\mathcal{S} \times \mathcal{E}) \to_\tau [0,1]$ is a probability vector, and n_τ is a non-negative integer assigned such that $n_{\tau_i} \neq n_{\tau_j}$ if $\tau_i \neq \tau_j$ for all $\tau_i, \tau_j \in \mathcal{T}$ with $s_{\tau_i} = s_{\tau_j}$. □

For any transition τ, s_τ is referred to as the *source state*, $\mathcal{E}_\tau^{\text{in}}$ is referred to as the set of *condition events*, each pair (s_i, e_i) with positive probability in the probability vector is referred to as a pair of *destination state* and *broadcast event*, and n_τ is referred to as the *transition order.*

The subset of Stateflow that SSF models are built upon, satisfies the following two points. Firstly, there are no variables. Secondly, in the spirit of providing a safe subset of Stateflow [3], only transition actions are allowed and not also condition actions.

2.2 Concrete syntax

The abstract syntax will now be mapped to a concrete syntax in Stateflow through the example model visualized graphically in Fig. 1. The states of the underlying SSF model of the graphical model give rise to the ordered tree visualized in Fig. 2.

In addition to the states explicitly visualized in Fig. 1, the underlying SSF model of the graphical model has a further state `root` satisfying `type(root) = AND` and `children(root) = {S1,S2}`. In the graphical model, for each state with parent of type `AND`, the value of the function `order` is given by the number in the upper right corner of the state. For instance, `order(S1) = 1` and `order(S2) = 2`.

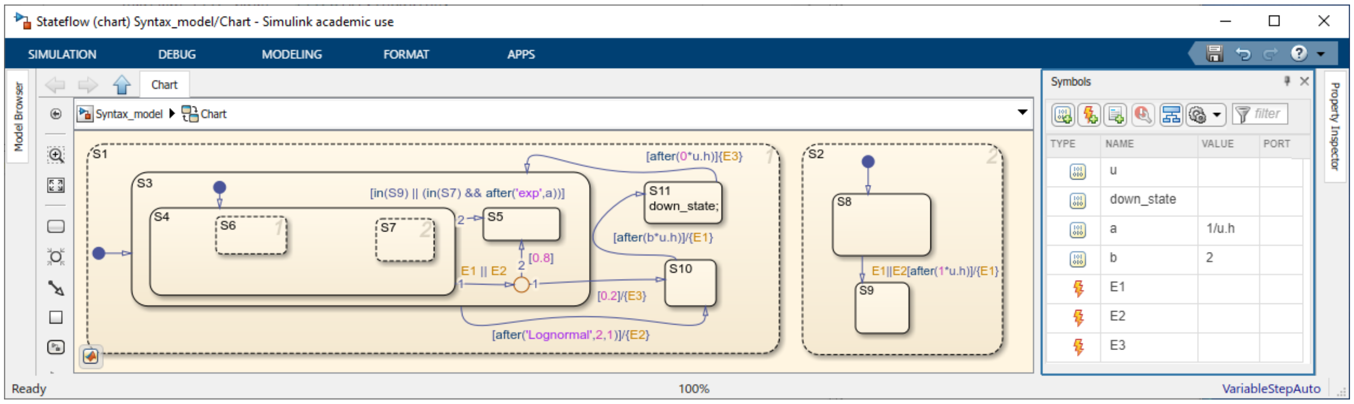
Down states are specified in the graphical model by giving them the label “down_state” beneath the state name. In Fig. 1, the only down state is $S11$.

For each state of type `OR`, the function `initial` is graphically specified by assigning a Stateflow *default transition* to one of its children. In the example, `initial(S1) = S3`, `initial(S2) = S8`, and `initial(S3) = S4`.

The transitions of the graphical model are represented by arrows leaving a state. In cases where there are several pairs of destination state and broadcast event for a transition in the underlying SSF model, the corresponding arrow from the source state in the graphical model has a Stateflow junction as destination and an arrow from the junction to the corresponding destination state for each pair. Each of these arrows from a junction is labeled by the probability in a square bracket and the broadcast event in curly brackets that are preceded by a slash. The right-most of the two transitions leaving S4 in the graphical model is an example of such a case. The set of condition events of a transition are given in the graphical model by labeling the corresponding arrow from the source state with the events separated by “||”, outside of any brackets. The guard is given in square brackets with “&&” representing conjunction, “||” representing disjunction, and “~” representing negation. If a transition τ has no such brackets attached to it, the interpretation is that the transition has the guard $g_\tau = \mathbb{T}$. Note that in the graphical model, square brackets might be interpreted differently depending on the source of the arrow. For an arrow originating from a state, it is seen as a guard. However, for an arrow originating from a junction, it is a probability. When there is only one pair of destination state and broadcast event for a transition, the event is given in curly brackets that are preceded by a slash in the label of the corresponding arrow from the source state to the destination state. Note that if all possible pairs of a destination state and a broadcast event from a transition share the same broadcast event then the event can be included as label only on the arrow to the junction instead of all arrows out from the junction. The transition order of a transition is given by the number written on the arrow leaving the source state of the corresponding transition.

The set $\mathcal{F}$ of CDFs is given in the graphical model by all CDFs that appear in any guard of the model. For Fig. 1, the set of CDFs is $\{\texttt{exp}(1h^{-1}), \texttt{lognormal}(2,1), \texttt{deg}(0h), \texttt{deg}(1h), \texttt{deg}(2h)\}$, containing an exponential, a lognormal, and several degenerate distributions.

Note that writing “after(0*u.h)” in the guard of a transition is interpreted equivalently to not writing the `after()` expression at all.



Symbols

TYPE	NAME	VALUE	PORT
	u		
	down_state		
	a	1/u.h	
	b	2	
	E1		
	E2		
	E3		

Figure 1: Model specified in Stateflow following the syntax of Def. 2.1.

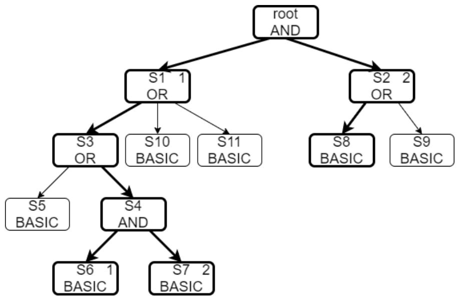


Figure 2: The set of states $\mathcal{S}$ of the model in Fig. 1 organized as an ordered tree. The bold subtree is the initial subtree discussed in Sec. 3.1.

The time units that can be assigned to parameters of CDFs in the concrete syntax is given by all time units in the Matlab Units List[2]. In the graphical model in Fig. 1, all parameters in all the CDFs have the unit *hours*.

The graphical model in Fig. 1 gives rise to seven symbols in the Stateflow symbol list that can be seen in the right hand side of the figure; u, down_state, a, b, E1, E2, E3. The events E1, E2, and E3 are set to the type "Local Event", u (which appears through the use of the Matlab unit list) and down_state are set to type "Input Data", and the added parameters a and b are set to type "Local Data". Furthermore, a and b have their values set in the value field of the symbols list and the unit is set in the value field of the symbols list for a and in each place the parameter is utilized in the model for b.

The concrete syntax of SSF models has been developed with the aim to be aligned to the concrete syntax of the subset of Stateflow considered. Yet the addition of stochastic properties give rise to some differences. In the `after()` expressions of SSF models, general CDFs can be specified and not just constants times as is the case in standard Stateflow. In SSF-models probabilities are assigned within square brackets labeling arrows out of junctions while these brackets are designated for guards in standard Stateflow.

[2]https://www.mathworks.com/help/symbolic/units-list.html

3 SEMANTICS OF SSF MODELS

The semantics of SSF models can be summarized as follows. Initially a subtree $\mathcal{S}_0$ of $\mathcal{S}$ is entered by the use of the `initial` and `type` functions. A random sample is then drawn from each CDF in each `after()` expression in each transition for which the source state is in $\mathcal{S}_0$. When the guard of a transition evaluates to `true`, and the transition has no condition events, the transition is triggered. When a transition is triggered, a random draw is made from the probabilities on the arrows from the corresponding junction to find which state will be entered and which event will be broadcast. The effects of broadcasting the event will first be evaluated, after which the transition will actually be performed. This results in a new subtree of $\mathcal{S}$ and the process then repeats itself. However, in detail, the semantics is considerably more intricate. The semantics of SSF models will therefore now be presented first informally by considering the model in Fig. 1 and then formally using an operational semantics. The idea has been to as far as possible align the semantics of SSF models with the semantics of the subset of Stateflow that SSF is based upon, motivated by the wide spread of Stateflow in industry. A study of a number of different example models has indicated that this goal has been satisfied.

3.1 Informal semantics

Consider the model in Fig. 1. At time $t = 0$ the state `root` (which is implicit in the model) is entered. Since `root` is an `AND` state, all its children states (S1 and S2) will be entered. Given their order, S1 will be entered first and will be considered next. S1 is an `OR` state and `initial(S1)` = S3 will thereby be the only state in `children(S1)` that is entered. Recursively, given that S3 is an `OR` state, the `AND` state S4 will be entered which yields that also S6 and S7 are entered. Next, since both S6 and S7 are of type `BASIC`, no more states can be entered from their branches. Now, S8 is entered since S2 is an `OR` state and `initial(S2)` = S8. This yields the complete initial subtree illustrated by the bold subtree in Fig. 2. The subtree contains the following states written in order of highest to lowest priority of consideration: `root`, S1, S3, S4, S6, S7, S2, S8. These states are referred to as the set of active states.

The transitions that are active, i.e., all transitions for which the source state is in the set of active states, are now considered. The active transitions are ordered firstly by the priority of the source states and secondly by the execution order for all transitions sharing a source state. The initially active transitions in order from the highest to lowest priority are:

- $\tau_1 = (\mathsf{S3}, \emptyset, \mathtt{after}(\mathtt{lognormal}(2,1)), ((\mathsf{S10}, \mathsf{E2}) \to 1), 1)$
- $\tau_2 = (\mathsf{S4}, \{\mathsf{E1}, \mathsf{E2}\}, \mathbb{T}, ((\mathsf{S10}, \mathsf{E3}) \to 0.2,\ (\mathsf{S5}, \diamond) \to 0.8), 1)$
- $\tau_3 = (\mathsf{S4}, \emptyset, \mathtt{in}(\mathsf{S9}) \mid\mid (\mathtt{in}(\mathsf{S7})\ \&\&\ \mathtt{after}(\mathtt{exp}(1/h))), ((\mathsf{S5}, \diamond) \to 1), 2)$
- $\tau_4 = (\mathsf{S8}, \{\mathsf{E1}, \mathsf{E2}\}, \mathtt{after}(\mathtt{deg}(1h)), ((\mathsf{S9}, \mathsf{E1}) \to 1), 1)$,

where $\mathtt{lognormal}(\mu, \sigma)$, $\mathtt{exp}(\lambda)$, and $\mathtt{deg}(\alpha)$ denotes the CDF of a lognormal, an exponential, and a degenerate distribution and where only the pairs of destination state and broadcast event are written out for the probability vector. Since for each active transition, either no condition event is currently broadcast or the guard is evaluated to false at time $t = 0$, no transition is triggered at this time.

A sample is now drawn from the CDF of each `after()` expression in each active transition. As an example, assume that the lognormal distribution yields $10.5h$, the exponential distribution yields $0.5h$, and trivially that the degenerate distribution for the `after` expression in τ_4 yields $1h$. The lowest time in all of the `after()` expressions in all four active transitions will now be considered. In the example, this is $0.5h$ yielded by the exponential distribution. The transitions are now considered (in order of priority) to see if a transition is triggered at time $t = 0.5h$. The result after checking τ_1, τ_2, and τ_3 in order is that transition τ_3 is triggered at this time since it has an empty set of condition events and its guard evaluates to `true` at time $t = 0.5h$. A sample is now drawn from the probability vector in τ_3 to find which destination state is entered and which event is broadcast when the transition is triggered. In the example, the result is trivially that the destination state is S5 and that the broadcast event is the event $\diamond$. The source state S4 and all descendants thereof are now removed from the set of active states together with all ancestors of S4 that are not shared by S5. In the example, S4, S6, and S7 are removed. Next, τ_3 is removed from the set of active transitions. Since the non-event $\diamond$ is broadcast by triggering τ_3, no effects of the broadcasting is considered. The next step is to set the destination state S5 and, as was done with the root state for finding the initial subtree, recursively setting children states as active. In a similar manner, all ancestors of S5 must be considered such that the resulting set of active states is a subtree of $\mathcal{S}$ satisfying that all children of AND states in the subtree are active, one child of each OR state in the subtree is active and each leaf node is a BASIC state. The new subtree of $\mathcal{S}$, in order of highest to lowest priority, is given by the states `root`, S1, S3, S5, S2 and S8.

All previously active transitions with a source state no longer in the set of active states are no longer active (τ_2 and τ_3 in the example) and all transitions (none for the example) with a newly entered state as source state are active. Next, a sample is drawn from all CDFs of the newly activated transitions (none for the example). Note that for each active transition that were not made active by the triggering of τ_3, the earlier samples for the CDFs are kept. In the example, this applies to the lognormal distribution in τ_1 and the degenerate distribution in τ_4. The updated list of active transitions is ordered in the same manner as for the active transitions for the initial subtree. Now, all active transitions are checked to see if they are triggered at the current time, in an order starting with the one with highest priority that was not made active in the latest step and that had not yet been checked before transition τ_3 was triggered. In the example, τ_4 will be checked followed by τ_1. This is repeated until no more transition are triggered at time $t = 0.5h$.

The process now repeats itself starting by finding the lowest remaining time in the samples of the `after()` expressions. For the example, this time is $0.5h$ for the `after()` expression in τ_4. However, looking through the active transitions at this updated time $t = 1h$, i.e., 0.5 hours after the latest considered time $t = 0.5h$, it is for the example found that no transition is triggered at this point in time since the `after()` expression in the guard of τ_1 is evaluated to false and since none of the condition events in τ_4 are being broadcast. The considered `after()` expression of τ_4 will now remain to evaluate to `true` until the transition is triggered.

The next time that is considered is $t = 10.5h$. At this point, τ_1 is triggered. After removing states from the set of active states as described earlier and inactivating τ_1, the broadcasting of E2 is considered. It is clear that this event will trigger transition τ_4 since τ_4 is active, E2 is in the set of condition events of τ_4, and the guard of τ_4 is evaluated to `true`. Now, as described earlier, states are removed from the set of active states for this transition τ_4. Since no active state has the event E1, which is broadcast by triggering τ_4, as condition event, the next step is to set the destination state S9 as active together with recursively setting children states and ancestor states as active as was done above for the transition τ_3. This is followed by repeating the procedure for τ_1 with the destination state S10. The resulting subtree of active states is given by the states `root`, S1, S10, S2 and S9.

No more transitions are triggered at the time and the next time to check is $t = 12.5h$ (from the expression $\mathtt{after}(\mathtt{deg}(2h))$ given in the transition $\tau_5 = (\mathsf{S10}, \emptyset, \mathtt{after}(\mathtt{deg}(2h)), ((\mathsf{S11}, \mathsf{E1}) \to 1), 1)$ that became active at time $t = 10.5h$). The transition broadcasts E1 after removing the source state S10, and transition τ_5 is no longer in the set of active transitions. No active transition is triggered by E1 at this time and the new subtree of active states is given by the states `root`, S1, S11, S2 and S9.

The process continues in the above fashion. A complete formal semantics of SSF models will now be presented.

3.2 Formal semantics

Consider an SSF model $\mathcal{M} = (\mathcal{S}, \mathcal{S}^{\text{down}}, \text{initial}, \mathcal{F}, \mathcal{E}, \mathcal{G}, \mathcal{T})$. Inspired by [8], the formal semantics of the model will be defined using an operational semantics. In order to provide such a definition, some notation will be introduced first.

- Timer: given a guard $g_i \in \mathcal{G}$, a *timer* of g_i corresponds to a specific instance of a CDF in g_i. For instance, consider g_1 to be an expression of the form $\mathtt{in}(s_1) \wedge \mathtt{after}(F_1) \wedge \neg\mathtt{after}(F_2)$, where $s_1 \in \mathcal{S}$ and $\{F_1, F_2\} \subseteq \mathcal{F}$. Then g_1 has two distinct timers corresponding to F_1 and F_2, respectively. Note that the two timers are distinct even in the case where the CDFs F_1 and F_2 are identical.
- $\mathtt{source}(\tau_i)$, where $\tau_i \in \mathcal{T}$: the unique source state of τ_i in $\mathcal{S}$.
- $\mathcal{D}$: the set of all timers, i.e., $\mathcal{D} = \{d_i \mid \exists g_j \in \mathcal{G} \text{ s.t. } d_i \text{ is a timer or } g_j\}$.
- $\mathcal{D}_{s_i}$, where $s_i \in \mathcal{S}$: the set of all timers d_j such that there exists a transition $\tau_k \in \mathcal{T}$ with $\mathtt{source}(\tau_k) = s_i$ such that d_j is a timer of g_{τ_k}.

- $\mathtt{source}(d_i)$, where $d_i \in \mathcal{D}$: the unique state $s_j \in \mathcal{S}$ such that $d_i \in \mathcal{D}_{s_j}$.
- $\mathtt{root}$: the root state of the tree given by $\mathcal{S}$.
- $\mathcal{T}_{s_i}$, where $s_i \in \mathcal{S}$: the set of all transitions with s_i as source state, i.e., $\mathcal{T}_{s_i} = \{\tau_j \in \mathcal{T} \mid \mathtt{source}(\tau_j) = s_i\}$.
- $\mathtt{parent}(s_i)$, where $s_i \in \mathcal{S}$: the unique state $s_j \in \mathcal{S}$ such that $s_i \in \mathtt{children}(s_j)$.
- $\mathtt{children}_Q(s_i)$, where $Q \subseteq \mathcal{S}$ and $s_i \in \mathcal{S}$: the set $\mathtt{children}(s_i) \cap Q$.
- $\mathtt{lca}(s_i, s_j)$, where $\{s_i, s_j\} \subseteq \mathcal{S}$: the lowest common ancestor of s_i and s_j.
- $\mathtt{first}(Q)$, where $Q \subseteq \mathtt{children}(s_i)$ for some state $s_i \in \mathcal{S}$: if s_i is of type AND, $\mathtt{first}(Q)$ is the state $s_j \in Q$ with minimal $\mathtt{order}(s_j)$. If s_i is of type OR and Q is a singleton, then $\mathtt{first}(Q)$ is the only element in Q. Otherwise, $\mathtt{first}(Q)$ is undefined.
- $\mathtt{first}(T)$, where $T \subseteq \mathcal{T}_{s_i}$ for some state $s_i \in \mathcal{S}$: the transition $\tau_j \in T$ with minimal transition order n_{τ_j}.
- $\pi_i \cdot \pi_j$, where π_i and π_j are sequences: the concatenation of π_i and π_j.
- ϵ: the empty sequence.
- $f[u \mapsto v]$, where f is a function: a function with the same domain as f such that $f[u \mapsto v](x) = \begin{cases} v & \text{if } x = u \\ f(x) & \text{otherwise} \end{cases}$.

In this notation, the reference to the specific SSF model is always implicit and assumed obvious from the context. Note also that the operators parent, source, lca, and first operate implicitly on the entire set $\mathcal{S}$ of states.

3.2.1 *Sampling future.*

In order to separately describe the stochastic component of an SSF model $\mathcal{M}$, the notion of *sampling future* of $\mathcal{M}$ will be used. Intuitively, a sampling future of a model is a function that maps each timer and each transition to an infinite sequence of future sample values. For a formal definition of a sampling future, let A^∞, where A is a set, denote the set of all infinite sequences consisting of elements in A.

Definition 3.1. A *sampling future* of an SSF model $\mathcal{M} = (\mathcal{S}, \mathcal{S}^{\text{down}}, \text{initial}, \mathcal{F}, \mathcal{E}, \mathcal{G}, \mathcal{T})$ is a tuple $\Phi = (\Phi^{\mathcal{D}}, \Phi^{\mathcal{T}})$ where

- $\Phi^{\mathcal{D}}$ is a function $\Phi^{\mathcal{D}} : \mathcal{D} \to \mathbb{R}^\infty_{\geq 0}$ mapping each timer $d_i \in \mathcal{D}$ to an infinite sequence of realizations of corresponding i.i.d. random variables sharing the distribution $\mathcal{F}_i$, and
- $\Phi^{\mathcal{T}}$ is a function $\Phi^{\mathcal{T}} : \mathcal{T} \to (\mathcal{S} \times \mathcal{E})^\infty$ mapping each transition $\tau_i \in \mathcal{T}$ to an infinite sequence of realizations of corresponding i.i.d. random variables sharing the distribution represented by the probability vector denoted by $\rightarrow_{\tau_i}$. □

3.2.2 *Configuration.*

Intuitively, a configuration of an SSF model $\mathcal{M}$ represents the currently active states of $\mathcal{M}$, a remaining time value for each timer contained in $\mathcal{M}$ and a sampling future of $\mathcal{M}$. Given a set Q of states, a timer d_i is said to be *active* in Q if $\mathtt{source}(d_i) \in Q$ and *inactive* otherwise. The convention used in this paper is that each timer that is inactive in the set of states of a configuration has remaining time value 0.

Definition 3.2. A *configuration* C of an SSF model $\mathcal{M} = (\mathcal{S}, \mathcal{S}^{\text{down}}, \text{initial}, \mathcal{F}, \mathcal{E}, \mathcal{G}, \mathcal{T})$ is a tuple $(\mathcal{S}_C, r_C, \Phi_C)$ where $\mathcal{S}_C$ is a subset of $\mathcal{S}$, $r_C : \mathcal{D} \to \mathbb{R}_{\geq 0}$ is a function, and Φ_C is a sampling future of $\mathcal{M}$ such that the following holds:

- The set $\mathcal{S}_C$ contains root. For each AND state $s_i \in \mathcal{S}_C$, each child in $\mathtt{children}(s_i)$ is an element of $\mathcal{S}_C$. For each OR state $s_j \in \mathcal{S}_C$, exactly one child in $\mathtt{children}(s_j)$ is an element of $\mathcal{S}_C$.
- For each timer $d_i \in \mathcal{D}$, if d_i is inactive in $\mathcal{S}_C$ then $r_C(d_i) = 0$. Otherwise, d_i is not greater than the maximum value of the support of the probability density function corresponding to d_i. □

As an example, consider the model depicted in Fig. 1. The model has five timers which will be denoted $d_1 = \mathtt{lognormal}(2, 1)$, $d_2 = \mathtt{exp}(1/h)$, $d_3 = \mathtt{deg}(0h)$, $d_4 = \mathtt{deg}(1h)$ and $d_5 = \mathtt{deg}(2h)$. A configuration $C = (\mathcal{S}_C, r_C, \Phi_C)$ representing the initial situation for this model may be the following. Examining the initial set of active states graphically represented in Fig. 2, it is clear that $S_C = \{\mathtt{root}, S1, S2, S3, S4, S6, S7, S8\}$ (written in no particular order). Following the same sampling pattern as in Sec. 3.1, timers d_1, d_2 and d_4 initially get values 10.5, 0.5 and 1, respectively, while timers d_3 and d_5 are inactive and get the value 0. Thus, r_C is the function $\{d_1 \mapsto 10.5,\ d_2 \mapsto 0.5,\ d_3 \mapsto 0,\ d_4 \mapsto 1,\ d_5 \mapsto 0\}$. Originating from the fact that the timers d_3, d_4 and d_5 are degenerate, it must be the case that the sampling future Φ_C gives $\Phi^{\mathcal{D}}_C(d_3) = 0, 0, 0\ldots$, $\Phi^{\mathcal{D}}_C(d_4) = 1, 1, 1\ldots$ and $\Phi^{\mathcal{D}}_C(d_5) = 2, 2, 2\ldots$. For the timers d_1 and d_2, which are non-degenerate, $\Phi^{\mathcal{D}}_C$ can yield any sequences of numbers as long as they are in the support of each respective timer. The case is similar for $\Phi^{\mathcal{T}}_C$.

The set of all configurations is denoted **C**. A configuration C is said to be *initial* if the following holds:

- for each state $s_i \in \mathcal{S}_C$ such that $\mathtt{type}(s_i) = \mathtt{OR}$, its initial child $\mathtt{initial}(s_i)$ is also a member of $\mathcal{S}_C$,
- for each active timer d_i in $\mathcal{S}_C$, the value $r_C(d_i)$ is in the support of the probability density function corresponding to d_i.

A transition $\tau_i \in \mathcal{T}$ is said to be *enabled* in a configuration C under an event e_j, written $\mathtt{enabled}(\tau_i, C, e_j)$, if the following holds:

- $\mathtt{source}(\tau_i) \in \mathcal{S}_C$,
- the set $\mathcal{E}^{\text{in}}_{\tau_i}$ of condition events is either empty or contains e_j,
- the guard of τ_i evaluates to true in C, written $(C, g_{\tau_i}) \vdash \mathtt{true}$.

3.2.3 *Guard evaluation.*

The evaluation of guards is defined using an operational semantics, see Fig. 15. In order to get a clear understanding of how the evaluation works, some of the rules will be examined more closely by means of a small example. Assume for instance that g_1 is a guard of the form $\mathtt{in}(s_1) \wedge \mathtt{after}(d_1)$, where s_1 is a state in $\mathcal{S}$ and d_1 is an instance of a CDF in $\mathcal{F}$. Assume also that C is a configuration where $s_1 \in \mathcal{S}_C$ and $r_C(d_1) = 2.5$. Now, in order to evaluate $g_1 = \mathtt{in}(s_1) \wedge \mathtt{after}(d_1)$, the rule Eval-6 will be used. The premises of this rule are evaluated on the sub-expressions $\mathtt{in}(s_1)$ and $\mathtt{after}(d_1)$, respectively. According to rules Eval-4 and Eval-3, the statements $(C, \mathtt{in}(s_1)) \vdash \mathtt{true}$ and $(C, \mathtt{after}(d_1)) \vdash \mathtt{false}$ holds, respectively. Thus applying Eval-6 gives $(C, g_1) \vdash \mathtt{false}$ since $\mathtt{true} \wedge \mathtt{false}$ is $\mathtt{false}$.

The semantics of determining a successor for a given configuration, e.g., by the triggering of some transitions or broadcasting of an event, is also defined using an operational semantics, see Fig. 11-9. The rules are organised into nine different levels of abstraction, each responsible for its own computational step. The following subsection is devoted to give an intuition for each level, by explaining how a subset of the rules work.

3.2.4 *Overview of the operational semantics.*

At the highest level, the relation $\xrightarrow[\texttt{Exec}]{}$ describes how a subtree of the set $\mathcal{S}$ of states is executed. The statement $(C, Q, T) \xrightarrow[\texttt{Exec}]{e} C'$ can be read "executing configuration C with remaining states Q and remaining transitions T to consider, under event e, produces configuration C'". It is assumed that the elements of Q and T all belong to the same source state, say s_i, so that the tuple (C, Q, T) can be viewed as a breakpoint in the execution of the model such that s_i is the currently considered state and Q and T are the children and transitions left to consider, respectively. As an example, the rule

$$\text{Exec-3: }\frac{T \neq \emptyset \quad \exists s_p \in \mathcal{S} \,.\, T \subseteq \mathcal{T}_{s_p} \quad \tau = \texttt{first}(T) \qquad \texttt{enabled}(\tau, C, e) \quad C \xrightarrow[\texttt{Trans}]{\tau} C'}{(C, Q, T) \xrightarrow[\texttt{Exec}]{e} C'}$$

says that if there are transitions left to consider, all of them belonging to the same source state and the prioritised one being τ, with τ being enabled and the result of taking τ is a configuration C', then executing the remaining subtree induced by (C, Q, T) yields C'. According to the rule Exec-5, a statement of the form $C \xrightarrow[\texttt{Exec}]{e} C'$ is simply shorthand for $(C, \texttt{children}_{\mathcal{S}_C}(\texttt{root}), \mathcal{T}_{\texttt{root}}) \xrightarrow[\texttt{Exec}]{e} C'$, which represents an execution of the entire tree of states $\mathcal{S}$ under event e.

One level deeper, triggering a transition is described by the relation $\xrightarrow[\texttt{Trans}]{}$. The statement $C \xrightarrow[\texttt{Trans}]{\tau} C'$ can be read as "triggering transition τ from configuration C produces configuration C'". As an example, the rule

$$\text{Trans-2: }\frac{\begin{array}{c}\texttt{source}(\tau) = s \quad \texttt{parent}(s) = s_p \quad (C, \tau) \xrightarrow[\texttt{Branch}]{} (C'', s', e) \\ \texttt{parent}(s') = s'_p \quad \texttt{lca}(s, s') = \widehat{s} \quad (C'', \{s, s'\} \cup \texttt{children}(\widehat{s})) \xrightarrow[\texttt{Exit}]{} C''' \\ C''' \xrightarrow[\texttt{Send}]{e} C'''' \quad \widehat{p} = \texttt{lca}(\widehat{s}, \texttt{lca}(s_p, s'_p)) \quad \widehat{p} \in \mathcal{S}_{C''''} \\ \texttt{type}(\widehat{p}) \neq \texttt{OR} \vee \texttt{children}_{\mathcal{S}_{C''''}}(\widehat{p}) = \emptyset \\ \texttt{type}(\widehat{p}) \neq \texttt{AND} \vee \texttt{children}_{\mathcal{S}_{C''''}}(\widehat{p}) \neq \texttt{children}(\widehat{p}) \quad (C'''', \{s'\}) \xrightarrow[\texttt{Enter}]{} C'\end{array}}{C \xrightarrow[\texttt{Trans}]{\tau} C'}$$

says that if τ has source s, the branching of τ is determined to go to state s' and broadcast event e, exiting the required states from C and broadcasting event e does not cause a transition from an ancestor of s, and lastly entering s' produces C', then the result of triggering transition τ from C is C'.

Sending an event $\diamond$ does nothing. However, sending any other event e consists of executing the entire tree of active states under e. These are exactly the cases captured by the two rules

$$\text{Send-1: }\frac{e = \diamond}{C \xrightarrow[\texttt{Send}]{e} C} \qquad \text{Send-2: }\frac{e \neq \diamond \quad C \xrightarrow[\texttt{Exec}]{e} C'}{C \xrightarrow[\texttt{Send}]{e} C'} \;.$$

Exiting a set of states is described by $\xrightarrow[\texttt{Exit}]{}$. Here, in order to keep the tree of active states connected, exiting some state s must imply that all descendants of s are also exited. The rule

$$\text{Exit-3: }\frac{(C, s) \xrightarrow[\texttt{Inactivate}]{} C'' \quad (C'', \texttt{children}_{\mathcal{S}_{C''}}(s)) \xrightarrow[\texttt{Exit}]{} C'}{(C, \{s\}) \xrightarrow[\texttt{Exit}]{} C'}$$

says that exiting the singleton $\{s\}$ consists of first inactivating s and then exiting all children of s recursively. Inactivating a state is described by the single rule

$$\text{Inactivate-1: }\frac{C' = (\mathcal{S}_C \setminus \{s\}, r_C[d \mapsto 0, \forall d \in \mathcal{D}_s], \Phi_C)}{(C, s) \xrightarrow[\texttt{Inactivate}]{} C'}$$

which states that inactivating s from C results in the configuration where s is removed from the active states and all timer values of s are remapped to 0. The case is similar for entering and activating states, but with two slight differences. First, in order to be able to enter a state s for which neither the ancestors or descendants are active, both the parent and the children of s must be recursively entered. Second, when activating a state, instead of resetting the timer values to 0, new values must be taken from the sampling future.

The stochastic aspect of the execution is described by the relations $\xrightarrow[\texttt{Restart}]{}$ and $\xrightarrow[\texttt{Branch}]{}$. The restart relation describes the behavior of restarting a set of timers and the branch relation describes the choice made among destination states and broadcast events in the probability vector of a transition. As an example, the rule

$$\text{Restart-3: }\frac{\Phi_C^{\mathcal{D}}(d) = v \cdot \phi \quad r' = r_C[d \mapsto v] \quad \Phi' = \left(\Phi_C^{\mathcal{D}}[d \mapsto \phi], \Phi_C^{\mathcal{T}}\right) \qquad C' = (\mathcal{S}_C, r', \Phi')}{(C, \{d\}) \xrightarrow[\texttt{Restart}]{} C'}$$

simply says that if the sampling future for a timer d begins with time value v, then restarting d consists of removing the first element v from the sampling future and remapping the timer value of d to v. Lastly, the rule

$$\text{Branch-1: }\frac{\Phi_C^{\mathcal{T}}(\tau) = (s, e) \cdot \phi \quad \Phi' = \left(\Phi_C^{\mathcal{D}}, \Phi_C^{\mathcal{T}}[\tau \mapsto \phi]\right) \quad C' = (\mathcal{S}_C, r_C, \Phi')}{(C, \tau) \xrightarrow[\texttt{Branch}]{} (C', s, e)}$$

states that if the sampling future of a transition τ begins with the state-event pair (s, e) then the pair is removed from the sampling future and the determined destination of τ is s with broadcast event e.

3.2.5 *Run.*

The semantics of an SSF model $\mathcal{M}$ is given in terms of so-called *runs* of $\mathcal{M}$ with respect to a given sampling future. A run is informally considered to be a timed sequence of configurations taking the form

$$C_0 \xrightarrow{t_1} C_1 \xrightarrow{t_2} C_2 \xrightarrow{t_3} \ldots$$

where C_0 is an initial configuration and any two successive configurations are distinguished by the fact that a transition has been triggered between them. If no transition is enabled in a configuration C_i, then time is advanced until a transition becomes enabled. This time advancement corresponds to the value t_{i+1}. When this has happened, the model is *executed* until once again no transitions are enabled. Execution of a model, which happens with no time delay, consists of triggering all enabled transitions in an order imposed by the state hierarchy together with the transition order and the order of children states. Enforcing such an ordering of transitions within a run consists of the following two parts. First, a relation is defined such that each configuration is related to that which results after executing it. This relation is denoted $\xrightarrow[\texttt{Exec}]{}$. Second, it is required that the configurations visited in a run without any passing of time are connected by $\xrightarrow[\texttt{Exec}]{}$ in such a way to ensure that whenever a transition is triggered, it is because it is a part of some ongoing execution. Thus, a run can be viewed at two levels. At the highest level, successive configurations are distinguished by the fact that the model has been executed between them. Meanwhile, at the lower level, successive configurations are distinguished by the fact that a single transition has been triggered between them.

Before a run can be defined formally, the following notions need to be introduced. Let $\mathtt{advance} : \mathbf{C} \times \mathbb{R}_{\geq 0} \to \mathbf{C}$ be a function such that $\mathtt{advance}(C, t)$ gives the configuration that results after advancing all timers in C with t, i.e., the configuration C' where $\mathcal{S}_{C'} = \mathcal{S}_C$, $\Phi_{C'} = \Phi_C$, and for each timer $d_i \in \mathcal{D}$, $r_{C'}(d_i) = \mathtt{max}(0, r_C(d_i) - t)$. Let moreover $\mathtt{ttt} : \mathbf{C} \to \mathbb{R}_{\geq 0}$ denote the time to trigger for a given configuration, i.e., the remaining time until a transition is enabled, such that for each $C \in \mathbf{C}$, $\mathtt{ttt}(C) = \mathtt{min}\{t \in \mathbb{R}_{\geq 0} \mid \exists \tau_i \in \mathcal{T} \text{ s.t. } \mathtt{enabled}(\tau_i, \mathtt{advance}(C, t), \diamond)\}$. A configuration C for which there exists no such time t, i.e., for which $\mathtt{ttt}(C)$ is undefined, is said to be a *trap configuration.*

Definition 3.3. A *run* of an SSF model $\mathcal{M}$ with respect to a sampling future Φ of $\mathcal{M}$ is a (finite or infinite) sequence

$$\omega = C_0 \xrightarrow{t_1} C_1 \xrightarrow{t_2} C_2 \xrightarrow{t_3} \ldots$$

such that

- C_0 is an initial configuration with $\Phi_{C_0} = \Phi$.
- For each $i \geq 0$ such that C_i is not a trap configuration, $t_{i+1} = \mathtt{ttt}(C_i)$.
- For each $i \geq 0$ such that C_i is not a trap configuration, the successor of C_i is the configuration C_{i+1} such that
$$\mathtt{advance}(C_i, t_{i+1}) \xrightarrow[\mathsf{Trans}]{\tau_i} C_{i+1}$$
for some enabled transition τ_i in $\mathtt{advance}(C_i, t_{i+1})$.
- For each i such that $t_i > 0$ or $i = 0$, if there exists some j such that ω contains the substring $C_i \xrightarrow{0} C_{i+1} \xrightarrow{0} C_{i+2} \xrightarrow{0} \ldots \xrightarrow{0} C_j$ and $t_{j+1} > 0$, then there also exist numbers $k_1, k_2, \ldots k_n$ with $i \leq k_l \leq k_m \leq j$ for each $1 \leq l \leq m \leq n$ such that $C_i \xrightarrow[\mathsf{Exec}]{\diamond} C_{k_1} \xrightarrow[\mathsf{Exec}]{\diamond} C_{k_2} \xrightarrow[\mathsf{Exec}]{\diamond} \ldots \xrightarrow[\mathsf{Exec}]{\diamond} C_{k_n} \xrightarrow[\mathsf{Exec}]{\diamond} C_j$.
- For each i such that $t_i > 0$ or $i = 0$, if ω contains an infinite substring $C_i \xrightarrow{0} C_{i+1} \xrightarrow{0} C_{i+2} \xrightarrow{0} \ldots$, then there exists a sequence $k_1, k_2, \ldots$ of infinite length with $i \leq k_l \leq k_m$ for each $1 \leq l \leq m$ such that $C_i \xrightarrow[\mathsf{Exec}]{\diamond} C_{k_1} \xrightarrow[\mathsf{Exec}]{\diamond} C_{k_2} \xrightarrow[\mathsf{Exec}]{\diamond} \ldots$. □

The relations $\xrightarrow[\mathsf{Trans}]{}$ and $\xrightarrow[\mathsf{Exec}]{}$ are defined formally in Fig. 11-9.

4 SMP-TOOL

Part of the Graphical User Interface (GUI) of SMP-tool is here presented together with the results of running some analyses. This is followed by a presentation of how the simulation for transient analysis of an SSF model $\mathcal{M}$ is performed in SMP-tool.

4.1 Graphical User Interface

Fig. 3 visualizes some of the GUI of SMP-tool together with the result of running some analyses. At the top left corner, the main GUI is visualized when an SSF model named "redundantSteering" has been loaded into the tool. Here the different types of analyses available in the tool are presented. Two examples are: 1) Transient analysis that finds the probability that the model has reached a down state by either a symbolic/numerical or a simulation engine for the times specified in "Analysis time" and then interpolates to visualize how the probability changes over time. 2) Sensitivity analysis that repeats the process of transient analysis for different values of the model parameters to find how much each of them affects the resulting probability.

The plot in the top right corner of Fig. 3 has been created by running a transient analysis using the simulation engine. Here it is visualized how the probability of reaching a down state changes over time with a confidence interval. The plot in the bottom left corner is the result of running a sensitivity analysis of a model with three parameters "Accident_rate", "FR_Primary", and "FR_Secondary". The result is a window with a tab for each parameter showing how the probability of reaching a down state differs when the value of the parameter is changed. The bottom right plot was generated by running "Display state transition" and visualizes some probability distributions, e.g., the distribution of the sojourn time, i.e., how much time is spent in the specified state.

All of these functionalities are available for a subset of Hierarchical Semi-Markov Process (HSMP) models as presented in [12]. The support for SSF models which are not also HSMP models currently applies only to transient and sensitivity analysis utilizing the simulation engine.

4.2 Simulation

SMP-tool can currently perform transient Monte Carlo simulation of SSF models given that the following six restrictions hold. (1) No `AND` state has an `OR` parent. (2) No guard has more than one `after()` expression. (3) Given any two children s_i and s_j of an `AND` state, no transition with s_i or any of its descendants as source state can have s_j or any of its descendants as destination state or vice versa. (4) All destination states of all transitions are of type `BASIC`. (5) No transition with a source state s_i can have a destination state s_j where s_i is an ancestor of s_j. (6) There exists no negation of an `after()` expression in any guard.

The simulation is performed in a two-step manner. Firstly, the model is transformed such that no `OR` states with an `OR` state as parent state remain, in detail described in Sec. 4.2.1. Secondly, a Monte Carlo simulation is performed on the new transformed model, in detail described in Sec. 4.2.2.

4.2.1 Model transformation. The model transformation works by, for each `OR` state s_i with an `OR` state s_j as parent state, moving all children states of s_i to be children of a complementary `OR` state $s_{i}\_$ with the `AND` state `root` as parent. Complementary transitions are then added within $s_{i}\_$ together with a complementary child state $s_{i}\_inactive$. Only the information about whether s_i is active or not is then contained in s_j while the more detailed information about which children is active is contained in $s_{i}\_$. The transformation can be repeated if there are several layers of `OR` states with an `OR` state as parent state until all children of `OR` states are of type `BASIC`. Figure 4 illustrates the transformation applied to a simple example model. `S3_` represents the complementary state and its child state S3_inactive that S3 is not currently active. Furthermore, `E1_` and `E2_` are two complementary events created during the transformation.

The model transformation is summarized in the following conjecture.

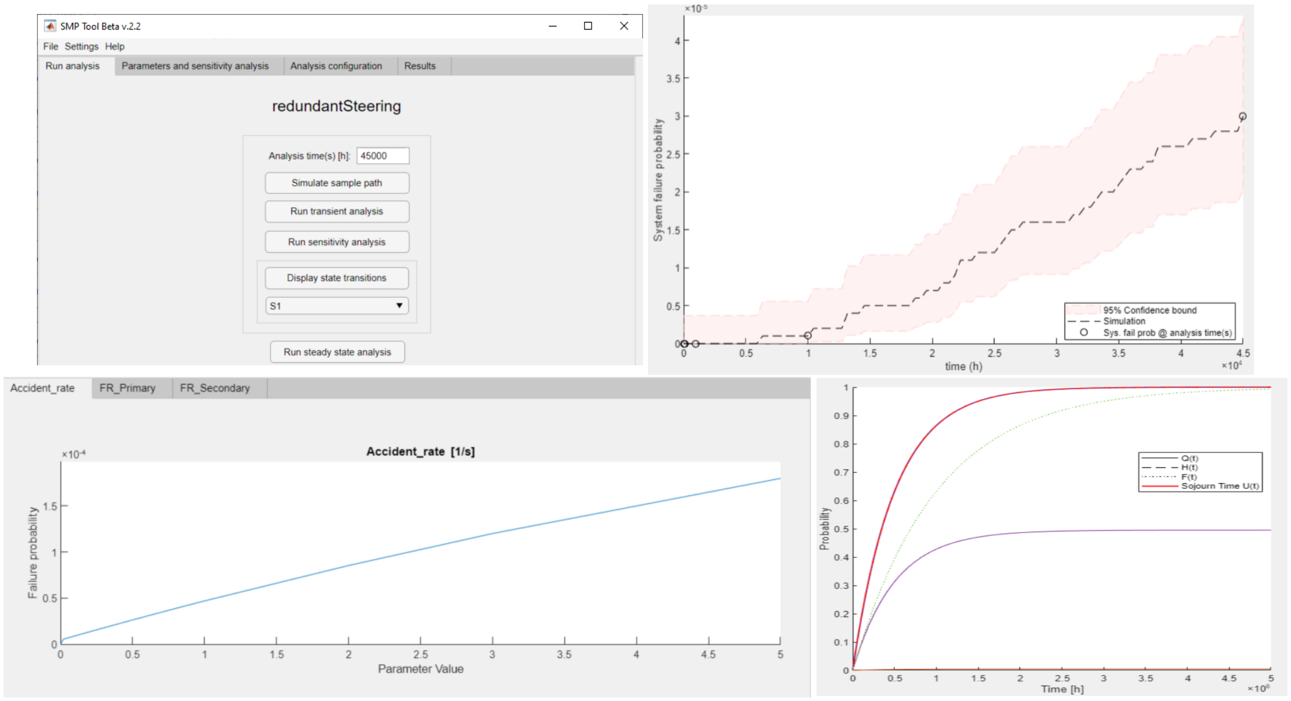
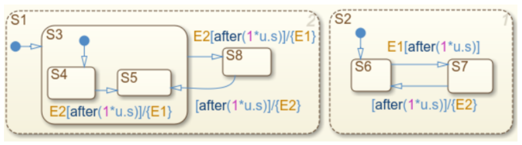
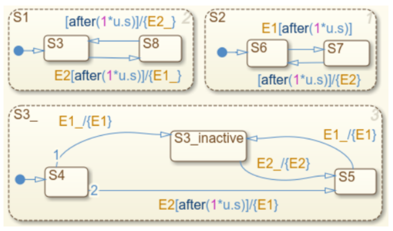


Figure 3: Subset of the GUI of SMP-tool.



(a)



(b)

Figure 4: a) A simple example SSF model before the model transformation has been applied. b) The model yielded from transforming the example model.

Conjecture 4.1. *Consider any SSF model $\mathcal{M}$ satisfying the restrictions listed in the beginning of Sec. 4.2 and the model $\mathcal{M}^*$ resulting of transforming $\mathcal{M}$ as described above. Consider a given sampling future Φ and a run ω of $\mathcal{M}$ with respect to the sampling future Φ. Let $(C_i)_i$ denote the sequence of configurations and $(t_i)_i$ denote the sequence of times of ω. Then the run ω^* of $\mathcal{M}^*$ given by the sampling future Φ satisfies $(t_i)_i = (t_i^*)_i$ and each active state in each configuration C_i is also an active state of C_i^*.*

The conjecture has been verified by transforming several example models and comparing the runs for some sampling futures.

4.2.2 Simulating a transformed model. The algorithm for simulating one lifetime of an SSF model $\mathcal{M}$ yielded by transforming an SSF model $\mathcal{M}'$ as described in Sec. 4.2.1 is presented in Alg. 1. The algorithm takes as input the SSF model $\mathcal{M}$ and the time t_{life} that the model will be simulated for. The output of the algorithm is a time t_{down}. If, and only if, $t_{down} \leq t_{life}$, a down state has been reached in the simulation. By repeating this process and seeing for how many simulations a down state was reached, the probability of reaching a down state can be estimated with a confidence interval.

5 CASE STUDY

A case study will now be presented. The case study is from a real industrial system from the heavy-vehicles manufacturer Scania but has been somewhat modified for confidentiality reasons. The system is modeled in Stateflow as an SSF model and analyzed using SMP-tool.

5.1 Gearbox wheel lock

The case study considers a gearbox of a truck and a dangerous failure of the gearbox causing the wheels of the vehicle to lock at their current position during driving. The cause considered for this failure is a short circuit in the internal electrical system of the gearbox. For a system failure to occur, two faults must be present at the same time; Short Circuit to Battery (SCB) and Short Circuit to Ground (SCG). It is assumed that the appearance of each of these faults have exponential distributions with rates $10^{-6}/h$. Moreover,

Algorithm 1: One simulation of an SSF model.

```
Input: An SSF model $\mathcal{M}$ and an analysis time $t_{life}$.
Output: $t_{down}$ - The time a down state was reached in the
        simulation (inf if it wasn't reached within $t_{life}$).
1  Set $S_{active}$ to a list of the initially active states according to
     their order of priority.
   while true do
2      if a state in $S_{active}$ is in $\mathcal{S}^{down}$ then
3          Set $t_{down}$ to the current time.
           break
       else
4          Sample a time from the timer of each transition in $\mathcal{T}$
             with source location in $S_{active}$ and for which no
             sampled time exists.
           if the lowest sampled time $t_{min}$ causes the total time
             to be above $t_{life}$ then
5              $t_{down} = \infty$.
               break
           else
6              for the transition $\tau$ with highest priority and
                 sampled time $t_{min}$ that satisfies $\mathcal{E}_{\tau}^{in} = \emptyset$ do
                   Sample a pair of destination state and
                     broadcast event $(s, e)$ from $\tau$.
7                  Remove $s_{\tau}$, all its children states and all
                     ancestors of $s_{\tau}$ that are not ancestors of $s$
                     from $S_{active}$.
8                  If $e$ triggers other transitions with source
                     state in $S_{active}$ and sampled times $\leq t_{min}$,
                     sample destination state and broadcast
                     event $(s', e')$ and remove states from
                     $S_{active}$ as above. Repeat this with $e'$ until
                     no more transition are triggered.
9                  Forget the sampled times of all triggered
                     transitions and all transitions with source
                     state not in $S_{active}$.
10                 Add the destination states of these triggered
                     transitions along with ancestors and
                     descendants such that the states in $S_{active}$
                     represent a proper subtree of $\mathcal{S}$ where each
                     active OR state has one child active, each
                     active AND state has all children active, and
                     initial is used for entering children of OR
                     states.
11                 Reduce the time of all sampled timers by
                     $t_{min}$.
               end
           end
       end
   end
```

there is a diagnosis that may detect each of these faults. The diagnosis is assumed to have a certain probability of detecting a fault and if a fault goes undetected it is assumed to remain that way until the vehicle visits a repair shop to get the gearbox fixed. The probability that a fault is detected by the diagnosis is 0.99. Furthermore, the diagnosis can in itself become unavailable. If this occurs, no faults will be detected. It is assumed that the appearance of failure in the diagnosis has an exponential distribution with rate $10^{-6}/h$. At every time the vehicle is started up, a separate diagnosis is performed to check the availability of the diagnosis of SCB and SCG faults. This start-up diagnosis has a certain probability of missing that the diagnosis of SCB and SCG is unavailable. The probability that an unavailability is detected in this start-up diagnosis is 0.99. It is conservatively assumed for both of the diagnosis procedures that if any fault goes undetected after a diagnosis it will continue to be undetected indefinitely. It is also conservatively assumed that each driving session of the vehicle is 16 hours long.

When a fault in SCB or SCG is detected, the driver will be warned and the vehicle will be locked in its current gear. It is assumed that the driver will stop the vehicle after 1 hour in this state after which the vehicle will be repaired before started again. When the diagnosis at the start-up of the vehicle finds that the diagnosis of SCB and SCG faults is unavailable, the driver will not be allowed to drive the vehicle and it is assumed that the fault is fixed in a repair shop before the vehicle will be driven again.

5.2 Modeling

An SSF model specified in Stateflow for the gearbox case study is presented in Fig. 5. For the resulting model, the root state is an AND state with 6 parallel children states. There are two parallel states named SCB_status and SCG_status tracking whether an SCB or SCG fault is present, respectively. The parallel state diagnosis_function tracks the result of the diagnostic procedure when an SCB or SCG fault becomes present. The availability of the diagnostic procedure together with the result of the start-up diagnosis is tracked in the parallel state diagnosis_status. Including both of these parts in diagnosis_status significantly lowers the expected number of triggered transitions during the vehicle life time, and thereby makes the simulation more efficient. Furthermore, there is a parallel state system_controller. This state contains the functionality of the safety procedure of the vehicle which is initialized when an SCB or SCG fault is detected. Finally, there is a parallel state system_availability which tracks whether the vehicle has reached a down state, i.e., if there has been a wheel lock, or not.

In the state SCB_status, a fault event is broadcast when a failure in SCB becomes present. Furthermore, when a failure in SCB is present and a repair event is received, SCB_status goes back to working condition. The state SCG_status works in the same manner for SCG faults.

In the state diagnosis_status, the feedback will either be unavailable but it is yet to be diagnosed (diagnosis_down), be unavailable but this was missed by the start-up diagnosis (diagnosis_dead), or be available (diagnosis_up).

In the state diagnosis_function, either the state fault_detected or the state diagnosis_monitoring will be in the set of active states. Initially, time is spent in diagnosis_monitoring. If the diagnosis is available and a fault event is received when in diagnosis_monitoring, a transition will be triggered either back into diagnosis_monitoring or to the state fault_detected in which case the event fault_detected will be broadcast. When in fault_detected and the repair event is received, a transition into diagnosis_monitoring is triggered.

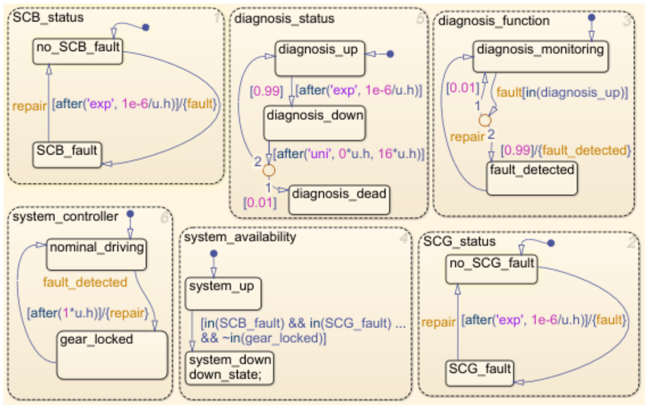


Figure 5: SSF model of the gearbox case study.

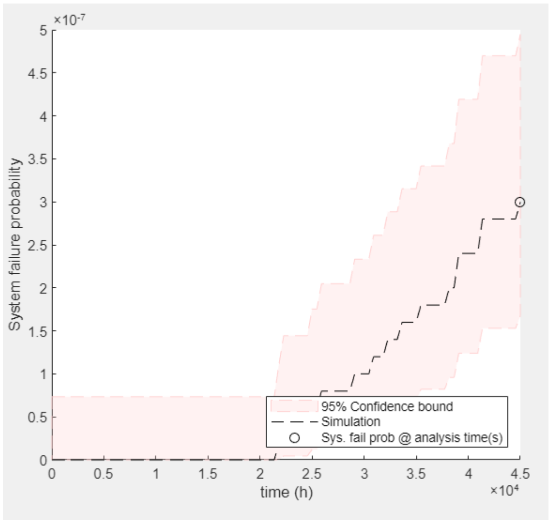


Figure 6: Result of simulating the system in the gearbox case study $5 \cdot 10^7$ times for the lifetime 45000 hours

In the state system_controller, the vehicle will either be in the state nominal_driving or in the state gear_locked. When in nominal_driving and the event fault_detected is received, a transition to gear_locked is triggered. After being in gear_locked for an hour, a transition into nominal_driving will be triggered and the event repair is broadcast.

In the state system_availability, the vehicle will either be in the state system_up representing that no wheel lock has occurred, or in the state system_down which represents the absorbing down state that a wheel lock has occurred. A transition from system_up to system_down is triggered when both an SCB and SCG fault are present and the gearbox is not locked in a gear.

5.3 Analysis

SMP-tool will now be used to find the probability of getting a wheel lock during the vehicle lifetime of 45000 hours. It is assumed that the vehicle should be designed to have a maximum failure probability of 10^{-6} during the lifetime of the system. By designing the system according to the chosen distributions, simulating the model $5 \cdot 10^7$ times yields the result illustrated in Fig 6. As can be seen, the entire 95% confidence bound has a system failure probability of below 10^{-6} after 45000 hours of driving.

Of course, there are several other possible configurations of parameter values that also satisfy the required maximum system failure probability. By utilizing the sensitivity analysis built into the SMP-tool, other configurations of parameter values can be found.

6 RELATED WORK

Here, some related work about models related to SSF models, formalization of the semantics of Stateflow, and extensions of Stateflow are presented.

A modeling formalism that syntactically resembles that of SSF models is StoCharts presented in [10]. However, there are essential semantic differences based on the differences between the semantics of Stateflow Models and Statecharts. Unlike Statecharts, Stateflow has no notion of true concurrency; only one event can be broadcast at once and only one transition can be considered at once. On the contrary, several events can be active at the same time and several transitions can be considered at the same time in Statecharts. These differences may cause non-determinism to appear in Stocharts on places where they would not appear in SSF-models [13–15]. While the use of non-determinism in practise is not necessarily a disadvantage, it goes against the purpose of the present paper to align the semantics of SSF models with the semantics of Stateflow.

When it comes to the semantics of Stateflow, several attempts have been made to formalize it for a subset of the language [1, 4, 7, 8, 15]. Like the semantics presented in the present paper, the semantics for a subset of Stateflow models presented in [8] is an operational semantics and some differences between the two semantics will therefore be discussed. Except that the semantics in [8] is of Stateflow models and thereby lacks the stochastic parts of the semantics of SSF models, the semantics is defined over different subsets of Stateflow models. Some restrictions which are made in the semantics of [8] that the semantics of the present paper does not adhere to are: (1) Each transition can only have one condition event. (2)Events can be broadcast only to parallel states. (3) There can be no recursion with events. (4) There can be no transitions out of parallel states. (5) Events can only be broadcast to already visited states. Point 2 refers to that there is no transition τ_i with a broadcast event e_j where there exists a transition τ_k with e_j in its set of condition events such that the source states of τ_i and τ_k do not have the same parent AND state. Point 3 refers to that the broadcasting of an event e_1 from a transition τ_i can never cause a chain of events that eventually triggers the transition τ_i again (and thus cause an infinite loops of recurring transitions). However, since the semantics of the present article only allow transition events, i.e., events where the source state is set as inactive and the destination state not yet set as active when evaluating the event, these infinite loops can not occur. Point 4 refers to that no transition can have a source state s_i such that the parent of s_i is of type AND. Point 5 refers to that there exists no run of an SSF model where a transition broadcasts an event e_i before the source state of each transitions with e_i in its set of condition events has been active earlier in the run.

There are also some restrictions on the subset of Stateflow that is utilized in SSF models and that are not made by [8]. The perhaps greatest restriction is that SSF models do not allow Stateflow variables. Furthermore, in the spirit of allowing only a safe predictable subset of Stateflow, SSF models only allow transition actions and not condition actions in accordance with the guideline [3].

Another stochastic extension of Stateflow has been previously presented in [2]. However, the extension limits the models to Continuous-Time Markov Chains (CTMC), i.e., only exponential CDFs are allowed while SSF models allow general CDFs. Even though methods have been presented to approximate non-exponential distributions with Phase-type distributions, i.e., a combination of exponential distributions, in for example [5], the number of exponential distributions needed for a good approximation may be very high, causing the state space to be greatly increased during the approximation. Furthermore, the methods does not approximate all probability distributions well, this is especially the case for distributions with heavy-tails that are not exponentially bounded. Moreover, in [2] the models are translated to PRISM[3] having its own semantics different from Stateflow, e.g., non-determinism exists while Stateflow is deterministic. Finally, the CDFs of SSF models are specified directly in the Stateflow chart while CDFs of the models referred to in [2] are specified as Simulink requirements outside of the Stateflow chart.

7 CONCLUSION

Stateflow is a common tool in industry for modeling systems as state-transition diagrams. However, Stateflow lacks explicit support for stochastic properties that are often important for building accurate models of real-world systems.

As a first contribution, a stochastic extension of a subset of Stateflow was therefore presented. For these new models, referred to as SSF models, a complete syntax and semantics were presented. As a second contribution, SMP-tool was extended with support for Monte Carlo simulation of SSF models. As the final contribution, a subsystem of a gearbox from an industrial case study was modeled as an SSF model and analyzed using SMP-tool.

Future work includes relaxing the restrictions of which SSF models can be simulated and prove that the simulation follows the semantics. Future work also includes complementing the Monte-Carlo simulation in SMP-tool with a numerical transient analysis of SSF models.

ACKNOWLEDGMENTS

The authors acknowledge the following agencies and projects for financial support: the European H2020 - ECSEL PRYSTINE project and Vinnova FFI through the SafeDim project.

REFERENCES

[1] Aditya Agrawal, Gyula Simon, and Gabor Karsai. 2004. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science* 109 (2004), 43—-56.

[2] Adrian Beer, Todor Georgiev, Florian Leitner-Fischer, and Stefan Leue. 2013. Model-Based Quantitative Safety Analysis of Matlab Simulink/Stateflow Models. In *MBEES*.

[3] MathWorks Advisory Board. 2020. *Control algorithm modeling guidelines using MATLAB, Simulink, and Stateflow.* Technical Report. SRI International. `https://www.mathworks.com/solutions/mab-guidelines.html`.

[4] Pontus Boström. 2006. *Mode-Automata in Simulink/Stateflow.* Technical report. Turku Centre for Computer Science.

[5] David Roxbee Cox. 1955. A use of complex probabilities in the theory of stochastic processes. *Mathematical Proceedings of the Cambridge Philosophical Society* 51(2) (1955), 313—-319.

[6] Matthias Gudemann and Frank Ortmeier. 2010. A framework for qualitative and quantitative formal model-based safety analysis. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering.* IEEE, 132–141.

[7] Grégoire Hamon. 2005. A Denotational Semantics for Stateflow. In *Proceedings of the 5st International Conference On Embedded Software (EMSOFT).* ACM press, New York, NY, 164–172.

[8] Grégoire Hamon and John Rushby. 2007. An Operational Semantics for Stateflow. *International Journal on Software Tools for Technology Transfer (STTT)* 9 (2007), 447—-456.

[9] David Harel. 1987. Statecharts: a visual formalism for complex systems. *Science of computer programming* 8 (1987), 231—-274.

[10] David N. Jansen. 2003. *Extensions of Statecharts with probability, time, and stochastic timing.* Ph.D. Dissertation. University of Twente.

[11] Anjali Joshi, Steven P. Miller, Michael Whalen, and Mats P.E. Heimdahl. 2005. A proposal for model-based safety analysis. In *24th Digital Avionics Systems Conference*, Vol. 2. IEEE, 13–.

[12] Stefan Kaalen, Mattias Nyberg, and Olle Mattsson. 2021. Transient Analysis of Hierarchical Semi-Markov Process Models with Tool Support in Stateflow. In *Quantitative Evaluation of Systems: 18th International Conference, Proceedings, Springer Nature , 2021, p. 105-126.* Springer nature, 105—-126.

[13] Paula J. Pingree, Erich Mikk, Gerard J. Holzmann, Margaret H. Smith, and Dennis Dams. 2002. Validation of mission critical software design and implementation using model checking. In *Proceedings of the 21st Digital Avionics Systems Conference (DASC).* IEEE.

[14] Norman Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, and Florence Maraninchi. 2004. Defining and translating a ´´safe" subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM international conference on Embedded software (EMSOFT).* ACM press, New York, NY, 259—-268.

[15] Ashish Tiwari. 2002. *Formal semantics and analysis methods for Simulink Stateflow models.* Technical Report. SRI International. `http://www.csl.sri.com/-~tiwari/stateflow.html`.

$$\text{Inactivate-1:}\ \frac{C' = (\mathcal{S}_C \setminus \{s\}, r_C[d \mapsto 0, \forall d \in \mathcal{D}_s], \Phi_C)}{(C, s) \xrightarrow[\texttt{Inactivate}]{} C'}$$

Figure 7: Rule for inactivating a state

$$\text{Activate-1:}\ \frac{(C, \mathcal{D}_s) \xrightarrow[\texttt{Restart}]{} C'' \qquad C' = (\mathcal{S}_{C''} \cup \{s\}, r_{C''}, \Phi_{C''})}{C \xrightarrow[\texttt{Activate}]{s} C',}$$

Figure 8: Rule for activating a state

$$\text{Branch-1:}\ \frac{\Phi_C^{\mathcal{T}}(\tau) = (s, e) \cdot \phi \qquad \Phi' = \left(\Phi_C^{\mathcal{D}}, \Phi_C^{\mathcal{T}}[\tau \mapsto \phi]\right) \qquad C' = (\mathcal{S}_C, r_C, \Phi')}{(C, \tau) \xrightarrow[\texttt{Branch}]{} (C', s, e)}$$

Figure 9: Rule for branching in a transition

[3]https://www.prismmodelchecker.org/

$$\text{Trans-1: } \frac{\begin{array}{c}\texttt{source}(\tau) = s \quad \texttt{parent}(s) = s_p \quad (C,\tau) \xrightarrow[\texttt{Branch}]{} (C'', s', e) \quad \texttt{parent}(s') = s'_p \quad \texttt{lca}(s,s') = \widehat{s} \quad (C'', \{s,s'\} \cup \texttt{children}(\widehat{s})) \xrightarrow[\texttt{Exit}]{} C''' \quad C''' \xrightarrow[\texttt{Send}]{e} C' \\ \widehat{p} = \texttt{lca}(\widehat{s}, \texttt{lca}(s_p, s'_p)) \quad \left(\widehat{p} \notin \mathcal{S}_{C'}\right) \vee \left(\texttt{type}(\widehat{p}) = \texttt{OR} \wedge \texttt{children}_{\mathcal{S}_{C'}}(\widehat{p}) \neq \emptyset\right) \vee \left(\texttt{type}(\widehat{p}) = \texttt{AND} \wedge \texttt{children}_{\mathcal{S}_{C'}}(\widehat{p}) = \texttt{children}(\widehat{p})\right)\end{array}}{C \xrightarrow[\texttt{Trans}]{\tau} C'}$$

$$\text{Trans-2: } \frac{\begin{array}{c}\texttt{source}(\tau) = s \quad \texttt{parent}(s) = s_p \quad (C,\tau) \xrightarrow[\texttt{Branch}]{} (C'', s', e) \quad \texttt{parent}(s') = s'_p \quad \texttt{lca}(s,s') = \widehat{s} \quad (C'', \{s,s'\} \cup \texttt{children}(\widehat{s})) \xrightarrow[\texttt{Exit}]{} C''' \quad C''' \xrightarrow[\texttt{Send}]{e} C'''' \\ \widehat{p} = \texttt{lca}(\widehat{s}, \texttt{lca}(s_p, s'_p)) \quad \widehat{p} \in \mathcal{S}_{C''''} \quad \texttt{type}(\widehat{p}) \neq \texttt{OR} \vee \texttt{children}_{\mathcal{S}_{C''''}}(\widehat{p}) = \emptyset \quad \texttt{type}(\widehat{p}) \neq \texttt{AND} \vee \texttt{children}_{\mathcal{S}_{C''''}}(\widehat{p}) \neq \texttt{children}(\widehat{p}) \quad (C'''', \{s'\}) \xrightarrow[\texttt{Enter}]{} C'\end{array}}{C \xrightarrow[\texttt{Trans}]{\tau} C'}$$

Figure 10: Rules for transitions

$$\text{Exec-1: } \frac{}{(C,\emptyset,\emptyset) \xrightarrow[\texttt{Exec}]{e} C}$$

$$\text{Exec-2: } \frac{\begin{array}{c} Q \neq \emptyset \quad \exists s_p \in \mathcal{S} \,.\, Q \subseteq \texttt{children}_{\mathcal{S}_C}(s_p) \quad s = \texttt{first}(Q) \\ (C, \texttt{children}_{\mathcal{S}_C}(s), \mathcal{T}_s) \xrightarrow[\texttt{Exec}]{e} C'' \quad (C'', (Q \setminus \{s\}) \cap \mathcal{S}_{C''}, \emptyset) \xrightarrow[\texttt{Exec}]{e} C' \end{array}}{(C, Q, \emptyset) \xrightarrow[\texttt{Exec}]{e} C'}$$

$$\text{Exec-3: } \frac{\begin{array}{c} T \neq \emptyset \quad \exists s_p \in \mathcal{S} \,.\, T \subseteq \mathcal{T}_{s_p} \quad \tau = \texttt{first}(T) \\ \texttt{enabled}(\tau, C, e) \quad C \xrightarrow[\texttt{Trans}]{\tau} C' \end{array}}{(C, Q, T) \xrightarrow[\texttt{Exec}]{e} C'}$$

$$\text{Exec-4: } \frac{\begin{array}{c} T \neq \emptyset \quad \exists s_p \in \mathcal{S} \,.\, T \subseteq \mathcal{T}_{s_p} \quad \tau = \texttt{first}(T) \\ \neg\texttt{enabled}(\tau, C, e) \quad (C, Q, T \setminus \{\tau\}) \xrightarrow[\texttt{Exec}]{e} C' \end{array}}{(C, Q, T) \xrightarrow[\texttt{Exec}]{e} C'}$$

$$\text{Exec-5: } \frac{(C, \texttt{children}_{\mathcal{S}_C}(\texttt{root}), \mathcal{T}_{\texttt{root}}) \xrightarrow[\texttt{Exec}]{e} C'}{C \xrightarrow[\texttt{Exec}]{e} C'}$$

Figure 11: Rules for executing an active subtree

$$\text{Exit-1: } \frac{}{(C,\emptyset) \xrightarrow[\texttt{Exit}]{} C} \qquad \text{Exit-2: } \frac{\begin{array}{c} Q_1 \cap Q_2 = \emptyset \quad Q_1 \neq \emptyset \quad Q_2 \neq \emptyset \\ (C, Q_1) \xrightarrow[\texttt{Exit}]{} C'' \quad (C'', Q_2) \xrightarrow[\texttt{Exit}]{} C' \end{array}}{(C, Q_1 \cup Q_2) \xrightarrow[\texttt{Exit}]{} C'}$$

$$\text{Exit-3: } \frac{(C, s) \xrightarrow[\texttt{Inactivate}]{} C'' \quad (C'', \texttt{children}_{\mathcal{S}_{C''}}(s)) \xrightarrow[\texttt{Exit}]{} C'}{(C, \{s\}) \xrightarrow[\texttt{Exit}]{} C'}$$

Figure 12: Rules for exiting a set of states

$$\text{Restart-1: } \frac{}{(C,\emptyset) \xrightarrow[\texttt{Restart}]{} C}$$

$$\text{Restart-2: } \frac{\begin{array}{c} R_1 \cap R_2 = \emptyset \quad R_1 \neq \emptyset \quad R_2 \neq \emptyset \\ (C, R_1) \xrightarrow[\texttt{Restart}]{} C'' \quad (C'', R_2) \xrightarrow[\texttt{Restart}]{} C' \end{array}}{(C, R_1 \cup R_2) \xrightarrow[\texttt{Restart}]{} C'}$$

$$\text{Restart-3: } \frac{\begin{array}{c} \Phi_C^{\mathcal{D}}(d) = v \cdot \phi \quad r' = r_C[d \mapsto v] \quad \Phi' = \left(\Phi_C^{\mathcal{D}}[d \mapsto \phi], \Phi_C^{\mathcal{T}}\right) \\ C' = (\mathcal{S}_C, r', \Phi') \end{array}}{(C, \{d\}) \xrightarrow[\texttt{Restart}]{} C'}$$

Figure 13: Rules for restarting timers

$$\text{Enter-1: } \frac{}{(C,\emptyset) \xrightarrow[\texttt{Enter}]{} C} \qquad \text{Enter-2: } \frac{\begin{array}{c} Q_1 \cap Q_2 = \emptyset \quad Q_1 \neq \emptyset \quad Q_2 \neq \emptyset \\ (C, Q_1) \xrightarrow[\texttt{Enter}]{} C'' \\ (C'', Q_2) \xrightarrow[\texttt{Enter}]{} C' \end{array}}{(C, Q_1 \cup Q_2) \xrightarrow[\texttt{Enter}]{} C'}$$

$$\text{Enter-3: } \frac{s \in \mathcal{S}_C}{(C, \{s\}) \xrightarrow[\texttt{Enter}]{} C} \qquad \text{Enter-4: } \frac{\begin{array}{c} s \notin \mathcal{S}_C \quad \texttt{type}(s) = \texttt{BASIC} \\ (C, s) \xrightarrow[\texttt{Activate}]{} C'' \\ (C'', \{\texttt{parent}(s)\}) \xrightarrow[\texttt{Enter}]{} C' \end{array}}{(C, \{s\}) \xrightarrow[\texttt{Enter}]{} C'}$$

$$\text{Enter-5: } \frac{\begin{array}{c} s \notin \mathcal{S}_C \quad \texttt{type}(s) = \texttt{AND} \quad Q = \texttt{children}_{\mathcal{S}}(s) \cup \{\texttt{parent}(s)\} \\ (C, s) \xrightarrow[\texttt{Activate}]{} C'' \quad (C'', Q) \xrightarrow[\texttt{Enter}]{} C' \end{array}}{(C, \{s\}) \xrightarrow[\texttt{Enter}]{} C'}$$

$$\text{Enter-6: } \frac{\begin{array}{c} s \notin \mathcal{S}_C \quad \texttt{type}(s) = \texttt{OR} \quad \texttt{children}_{\mathcal{S}_C}(s) \neq \emptyset \\ (C, s) \xrightarrow[\texttt{Activate}]{} C'' \quad (C'', \{\texttt{parent}(s)\}) \xrightarrow[\texttt{Enter}]{} C' \end{array}}{(C, \{s\}) \xrightarrow[\texttt{Enter}]{} C'}$$

$$\text{Enter-7: } \frac{\begin{array}{c} s \notin \mathcal{S}_C \quad \texttt{type}(s) = \texttt{OR} \quad \texttt{children}_{\mathcal{S}_C}(s) = \emptyset \\ (C, s) \xrightarrow[\texttt{Activate}]{} C'' \quad (C'', \{\texttt{initial}(s), \texttt{parent}(s)\}) \xrightarrow[\texttt{Enter}]{} C' \end{array}}{(C, \{s\}) \xrightarrow[\texttt{Enter}]{} C'}$$

Figure 14: Rules for entering states

$$\text{Eval-1: } \frac{}{(C, \mathbb{T}) \vdash \texttt{true}} \qquad \text{Eval-2: } \frac{r_C(d) = 0}{(C, \texttt{after}(d)) \vdash \texttt{true}}$$

$$\text{Eval-3: } \frac{r_C(r) \neq 0}{(C, \texttt{after}(d)) \vdash \texttt{false}} \qquad \text{Eval-4: } \frac{s \in \mathcal{S}_C}{(C, \texttt{in}(s)) \vdash \texttt{true}}$$

$$\text{Eval-5: } \frac{s \notin \mathcal{S}_C}{(C, \texttt{in}(s)) \vdash \texttt{false}} \qquad \text{Eval-6: } \frac{(C, g_1) \vdash v_1 \quad (C, g_2) \vdash v_2}{(C, g_1 \wedge g_2) \vdash v_1 \wedge v_2}$$

$$\text{Eval-7: } \frac{(C, g_1) \vdash v_1 \quad (C, g_2) \vdash v_2}{(C, g_1 \vee g_2) \vdash v_1 \vee v_2} \qquad \text{Eval-8: } \frac{(C, g) \vdash v}{(C, \neg g) \vdash \neg v}$$

Figure 15: Rules for evaluating a guard

$$\text{Send-1: } \frac{e = \diamond}{C \xrightarrow[\texttt{Send}]{e} C} \qquad \text{Send-2: } \frac{e \neq \diamond \quad C \xrightarrow[\texttt{Exec}]{e} C'}{C \xrightarrow[\texttt{Send}]{e} C'}$$

Figure 16: Rules for sending an event