Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

BACHELOR OF SCIENCE THESIS

COMPARING THE MODELING CAPABILITIES OF UPPAAL AND REALYST FOR STOCHASTIC HYBRID SYSTEMS

Seraphim Zaytsev

Examiners:
Prof. Dr. Dr.h.c. Erika Ábrahám
Apl. Prof. Dr. Thomas Noll

Additional Advisor: Lisa Willemsen

Aachen, 30.09.2025

Abstract

We compare the expressivity of UPPAAL and RealySt for the modeling of stochastic hybrid systems through the CAMELS classification [WRÁ25]. We align rectangular-automata formalisms with CAMELS and show that the random-clock semantics in RealySt realizes the decomposed eager non-predictive corner [DSSR24, DRÁ+25]. For UPPAAL we give constructive encodings for the CAMELS variants we use, based on three reusable patterns: countdown clocks for random delays, window refinement to make enabledness piecewise constant, and urgent expiry. These patterns realize CAMELS semantics in UPPAAL when native support is absent. RealySt supports explicit random clocks and computes maximal time-bounded reachability by exact reachability analysis and numerical integration. To support the comparison we formalize a translation from decomposed eager non-predictive specifications to rectangular automata with random clocks. We evaluate on the ARCH-COMP'22 stochastic minimal examples [ABD+22] and on a small CAMELS "Sisyphus" example. We observe agreement when semantics align and divergence where approximations are required, which shows how semantics influences the reported probabilities.

Contents

Chapter 1

Introduction

Models for stochastic hybrid systems combine continuous time evolution, discrete jumps, and stochasticity. This modeling lets us examine whole sets of possible evolutions to pinpoint stress hot spots, anticipate wear and tear over time, and highlight subsystems that merit extra attention. Different semantics for how random delays and non-deterministic choices interact can yield different probabilities, even for models that look the same on paper. Tool choice then becomes dependent on the specification. Our goal is to make these semantic differences explicit and to provide modeling patterns that let modelers pick and reproduce the intended semantics without surprises.

We study two families of tools. UPPAAL offers statistical model checking on networks of stochastic timed automata with races on sampled delays. It provides quantitative answers by Monte Carlo estimation with confidence-style guarantees [DLL$^+$15]. RealySt targets stochastic hybrid systems with random clocks on a decidable hybrid backbone and computes time-bounded maximal reachability probabilities by combining exact reachability with integration under prophetic schedulers [DSSR24]. A recent formal bridge via rectangular automata with random clocks shows how to maximize reachability for decomposed eager non-predictive semantics and when such results are preserved under translation [DSSR24]. The CAMELS classification organizes the space of composed versus decomposed choices and lazy versus eager realizability and it lets us state tool expressivity in a uniform way [WRÁ25].

1.1 Objectives and scope

We delimit the subset of the UPPAAL language and its stochastic semantics that we rely on later, giving a theoretical account of what our automata can express and how races on sampled delays are interpreted. Independently of UPPAAL, we provide a translation from decomposed eager non-predictive specifications to rectangular automata with random clocks together with the assumptions under which time-bounded reachability is preserved. We then implement some ARCH-COMP 2022 benchmarks in UPPAAL, report time-bounded probability estimates, and note required approximations such as ticked noise in flows. Finally we place UPPAAL and RealySt in the CAMELS landscape and compare expressivity and guarantees, aligning cases where semantics coincide and discussing divergences where support differs. Our scope is time-bounded reachability. We do not address unbounded properties or expected

reward objectives, and continuous stochastic noise is handled only via the stated approximations where needed.

1.2 Related work

Work on stochastic extensions of hybrid automata has been organized by the CAMELS framework, which contrasts decomposed and composed scheduling and distinguishes lazy and eager modeling [WRÁ23, WRÁ25]. UPPAAL SMC provides simulation driven verification for networks of timed and hybrid automata with a stochastic semantics and query primitives for probability and expectation estimation [DLL+15, BDL04]. On top of this semantics Uppaal Stratego synthesizes strategies for quantitative objectives and has been applied to control problems [DJL+15, LMT15]. RealySt targets rectangular automata with random events and random clocks and computes optimal time bounded reachability by exact flowpipe construction combined with multi dimensional integration [DRÁ+25, DSSR24]. The transformation from random events to random clocks preserves time and jump bounded reachability and enables reuse of rectangular automata techniques [DRÁ+25]. Surveys and benchmarks from the ARCH community summarize capabilities and performance trends of tools for stochastic models and provide context for our evaluation [ABD+22].

1.3 Method snapshot

We use UPPAAL to estimate time-bounded reachability by Monte Carlo queries with fixed sample size and report estimates with confidence information [DLL+15]. We use RealySt to compute time-bounded maximal reachability under prophetic schedulers on models in the random clock view and integrate over the stochastic domain [DSSR24, DRÁ+25]. As datasets we use the ARCH-COMP 2022 stochastic minimal examples [ABD+22] and a small Sisyphus style example that exercises repeated enablement patterns.

1.4 Contributions

First, we align the UPPAAL modeling primitives with CAMELS and contrast them with the random clock semantics used by RealySt, which clarifies what each tool expresses natively and what needs approximation. Second, we give an intuitive and practical translation from decomposed eager non-predictive specifications to rectangular automata with random clocks and state the assumptions under which time-bounded reachability properties are preserved [DRÁ+25]. Third, we carry out an empirical study on the ARCH-COMP minimal examples by implementing reusable UPPAAL patterns for composed and decomposed behavior and by running RealySt on matching random clock models, then we compare probabilities and explain agreements and divergences.

1.5 Findings in brief

UPPAAL realizes multiple CAMELS corners through modeling patterns, while RealySt natively targets the decomposed eager non-predictive setting. On ARCH-COMP

2022 the tools agree when the semantics align and they diverge exactly where support differs, for example when noise in flows must be tick approximated in UPPAAL but is captured through random clocks and exact reachability on the RealySt side [DLL+15, DSSR24, WRÁ25].

Chapter 2

Preliminaries

2.1 Hybrid automata

The following definitions and notations follow the formalization given in [WRÁ25].

Hybrid automata (HA) are a formal model for systems exhibiting both discrete and continuous behavior. They extend finite automata or transition systems by incorporating real-valued variables whose evolution over time is determined by differential equations.

Definition 2.1.1 (HA Syntax)**.** *A HA is as a tuple* $\mathcal{H} = (\text{Loc}, \text{Var}, \text{Flow}, \text{Inv}, \text{Lab}, \text{Edge}, \text{Init})$ *with the following components:*

- Loc *is a non-empty finite set of locations.*
- $\text{Var} = \{x_1,...,x_d\}$ *is a finite ordered set of real-valued variables with* d *indicating the dimension of* $\mathcal{H}$*.*
- $\text{Flow} : \text{Loc} \to (\mathbb{R}^d \to \mathbb{R}^d)$ *specifies the continuous dynamics (derivatives) in each location.*
- $\text{Inv} : \text{Loc} \to 2^{\mathbb{R}^d}$ *specifies an invariant for each location.*
- $\text{Lab} = \{a_1,...,a_k\}$ *is a non-empty finite ordered set of labels*
- $\text{Edge} \subseteq \text{Loc} \times \text{Lab} \times 2^{\mathbb{R}^d} \times 2^{\mathbb{R}^d \times \mathbb{R}^d} \times \text{Loc}$ *is a finite set of discrete transitions or jumps. A jump* $(\ell,a,g,R,\ell') \in \text{Edge}$ *has source* ℓ*, target* ℓ'*, label* a*, guard* g*, and a reset relation* $R \subseteq \mathbb{R}^d \times \mathbb{R}^d$*. Guards for jumps with the same source and label are disjoint. Resets may be non-deterministic. A reset is deterministic when there exists a function* r *on* $\mathbb{R}^d$ *with values in* $\mathbb{R}^d$ *such that* $R = \{(\nu, r(\nu)) \mid \nu \in \mathbb{R}^d\}$*.*
- $\text{Init} : \text{Loc} \to 2^{\mathbb{R}^d}$ *defines the initial valuations in every location.*

HA can also be composed in parallel, allowing multiple components to interact via synchronization. We omit the formal definition, as we do not make use of it in this thesis.

A state of a d-dimensional HA with locations Loc is a pair $(\ell, \nu) \in \text{Loc} \times \mathbb{R}^d$. In hybrid systems, the system state evolves through two types of behavior: continuous flows and discrete jumps. Flows describe how variables change over time within a

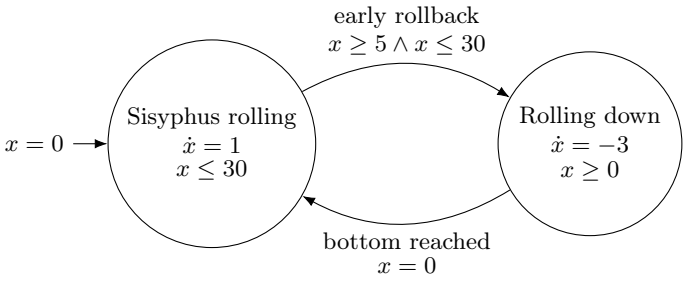

Figure 2.1: HA for Sisyphus in a low-memory simulation, where he must roll the boulder back before the top, let it fall under gravity, and start pushing again from the bottom.

location, potentially using constant derivatives such as $\dot{x} = 1$, which results in linear time progression. A derivative of $\dot{x} = 0$ models a variable that remains constant (i.e., does not evolve with time). More complex dynamics can also be modeled using non-constant derivatives, such as linear or polynomial expressions, leading to nonlinear behavior. Time may only progress in a location as long as the current valuation satisfies the location's invariant. Before the invariant gets violated, the system must take a jump (if one is enabled), or a deadlock occurs.

Jumps, on the other hand, represent discrete transitions between locations. A jump can be taken if its guard condition is satisfied in the current valuation, and it may update variable values via a reset function. After the reset, the new valuation must satisfy the invariant of the target location.

A HA can be defined by an operational semantics that combines continuous behavior and discrete jumps:

Definition 2.1.2 (HA Semantics)**.** *Let* $\mathcal{H} = (\mathrm{Loc}, \mathrm{Var}, \mathrm{Flow}, \mathrm{Inv}, \mathrm{Lab}, \mathrm{Edge}, \mathrm{Init})$ *be a d-dimensional HA. The operational semantics of* $\mathcal{H}$ *is defined by the following rules:*

$$\frac{\begin{array}{c}\ell \in \mathrm{Loc} \quad \nu, \nu' \in \mathbb{R}^d \quad t \in \mathbb{R}_{\geq 0} \quad f : [0,t] \to \mathbb{R}^d \quad \frac{df}{dt} = \dot{f} : (0,t) \to \mathbb{R}^d \\ f(0) = \nu \quad f(t) = \nu' \quad \forall t' \in (0,t).\, \dot{f}(t') = \mathrm{Flow}(\ell)(f(t')) \\ \forall t' \in [0,t].\, f(t') \in \mathrm{Inv}(\ell)\end{array}}{(\ell, \nu) \xrightarrow{t} (\ell, \nu')} \textsc{Flow}$$

$$\frac{(\ell, a, g, r, \ell') \in \mathrm{Edge} \quad \nu, \nu' \in \mathbb{R}^d \quad \nu \in g, \quad \nu' = r(\nu) \quad \nu' \in \mathrm{Inv}(\ell')}{(\ell, \nu) \xrightarrow{a} (\ell', \nu')} \textsc{Jump}$$

Let $\mathcal{H} = (\mathrm{Loc}, \mathrm{Var}, \mathrm{Flow}, \mathrm{Inv}, \mathrm{Lab}, \mathrm{Edge}, \mathrm{Init})$ be a HA with $\mathrm{Var} = \{x_1, \ldots, x_d\}$ and $\mathrm{Lab} = \{a_1, \ldots, a_k\}$. A run (or path) of $\mathcal{H}$ is a sequence of states connected by alternating time steps and discrete transitions $\pi = \sigma_0 \xrightarrow{t_0} \sigma_0' \xrightarrow{a_{w(0)}} \sigma_1 \xrightarrow{t_1} \cdots$ either infinite or ending with a time step, where each $\sigma_i = (\ell_i, \nu_i)$ and $\sigma_i' = (\ell_i, \nu_i')$ is a state consisting of a location and a valuation, $t_i \in \mathbb{R}_{\geq 0}$ denotes the duration of a flow, and $a_{w(i)} \in \mathrm{Lab}$ is the label of the i-th jump.

A run is initial if σ_0 belongs to the initial state set, i.e., if $\nu_0 \in \mathrm{Init}(\ell_0)$. Its length, $\mathrm{len}(\pi)$, is the number of jumps in it, and its duration, $\mathrm{dur}(\pi)$, is the total time $\mathrm{len}(\pi) = \sum_{i=0}^{n} t_i$ spent in flows. A state σ is reachable if it occurs in some initial run.

2.2 Rectangular automata

Rectangular automata (RA) are a subclass of HA obtained by imposing syntactic restrictions on the form of their continuous dynamics, invariants, and guards. They generalize timed automata by allowing variables to evolve with different constant rates or within constant rate intervals, but remain less expressive than unrestricted HA.

1. Rectangular constraints on guards and invariants:
   In RA, each invariant and guard must be a rectangle, meaning, a Cartesian product of one-dimensional intervals $x \in [l, u] \in \mathbb{R}$, where each bound l (lower) or u (upper) can be a rational constant or infinite, representing an unbounded direction. This restriction excludes diagonal constraints such as $x_1 + x_2 \leq 5$ or $x_1 \leq x_2$ [HKPV98].
2. For each reset $r : \mathbb{R}^d \to \mathbb{R}^d$ with $r(x) = (r_1(x), \ldots, r_d(x))$ and each $i \in \{1, \ldots, d\}$ either $r_i(x) = x$ or $r_i(x) = c_i$ for all $x \in \mathbb{R}^d$ and some $c_i \in \mathbb{R}$.
3. Constant or interval-bounded rates:
   In each location, the derivative of each variable is constant or belongs to a fixed interval $\dot{x}_i \in [a_i, b_i]$, with $a_i = b_i$ giving a deterministic rate and $a_i < b_i$ allowing bounded variability. Each variable's rate interval is specified independently, so there are no constraints coupling the derivatives of different variables.

These restrictions give RA a state space geometry that is well-suited to certain exact analysis techniques, while still being expressive enough to model systems with multiple clocks, resource variables, and bounded-rate uncertainties. Timed automata appear as a special case where all variables have $\dot{x} = 1$ and only finite upper bounds are used in constraints.

2.2.1 Rectangular automata with random events

We extend a rectangular automaton by random events [DRÁ$^+$25]. Each random event labels a set of jumps and the next occurrence time of that event is sampled from a continuous distribution. Different events are sampled independently. A jump is called stochastic if it carries a random event label and nonstochastic otherwise.

Definition 2.2.1 (RAE syntax)**.** *A rectangular automaton with random events is a tuple* $\mathcal{E} = (\mathcal{R}, \text{Lab}, \text{Distr}, \text{Event})$ *with the following components:*

- $\mathcal{R}$ *is a rectangular automaton with jump set* $Jump$*.*
- Lab *is a finite ordered set of random events. We write* $d_R = |\text{Lab}|$*.*
- Distr *has domain* Lab *and codomain the set* $\mathbb{F}$ *of continuous probability distributions on* $\mathbb{R}_{\geq 0}$*.*
- Event *maps each* $e \in Jump$ *to an element of* $\text{Lab} \cup \{\perp\}$*. If two different jumps have the same source location and the same label in* Lab *then their guards are disjoint.*

The semantics augments the RA state by two vectors. The vector μ stores for each event the duration of enabledness. The vector R stores the sampled expiration times [DRÁ$^+$25]. At a stochastic jump the corresponding expiration is resampled.

Time can elapse while no guard is crossed and while no expiration is reached. We require maximal time steps so a time step cannot be prolonged without crossing a guard or hitting an expiration [DRÁ+25].

Definition 2.2.2 (RAE semantics)**.** *A state is a tuple* (ℓ,c,μ,s) *with location* ℓ*, valuation* $c \in \mathbb{R}^d$*, durations* $\mu \in \mathbb{R}^{d_R}_{\geq 0}$ *and expirations* $R \in \mathbb{R}^{d_R}_{\geq 0}$*. Initially* $c \in \mathrm{Init}(\ell)$ *and* $\mu = 0$*. For every* $r \in \mathrm{Lab}$ *the component* R_r *is sampled from* $\mathrm{Distr}(r)$*.*

An event r *is enabled in* (ℓ,c) *if there exists a jump* $e = (\ell,a,g,R,\ell')$ *with* $\mathrm{Event}(e) = r$ *and* $c \in g$*.*

Transitions are as follows.

- *Flow (time elapse): Choose* $t \in \mathbb{R}_{\geq 0}$ *and a rate vector admitted by the flow of* ℓ*. Set* $c' = c + t \cdot \mathrm{rate}$*. All intermediate valuations satisfy* $\mathrm{Inv}(\ell)$*. For each* r *set* $\mu'_r = \mu_r + t$ *if* r *is enabled at all intermediate valuations and set* $\mu'_r = \mu_r$ *otherwise. Require* $\mu' \leq s$ *componentwise. The step is maximal with respect to guards and expirations.*
- *Nonstochastic jump: For* $e = (\ell,a,g,R,\ell')$ *with* $\mathrm{Event}(e) = \bot$ *and* $c \in g$ *pick* c' *with* $(c,c') \in R$ *and* $c' \in \mathrm{Inv}(\ell')$*. Keep* $(\mu',s') = (\mu,s)$*.*
- *Stochastic jump: If* $\mu_r = R_r$ *for some* $r \in \mathrm{Lab}$ *then any jump* $e = (\ell,a,g,R,\ell')$ *with* $\mathrm{Event}(e) = r$ *and* $c \in g$ *may be taken. Pick* c' *with* $(c,c') \in R$ *and* $c' \in \mathrm{Inv}(\ell')$*. Set* $\mu'_r = 0$ *and resample* R'_r *from* $\mathrm{Distr}(r)$*. For* $r' \neq r$ *keep* $\mu'_{r'} = \mu_{r'}$ *and* $R'_{r'} = R_{r'}$*.*

Non-determinism appears in the choice of the initial state, in the choice of time elapse while some jump remains enabled, in rate intervals and in the choice among enabled jumps. We resolve it by schedulers that may use the sampled expirations stored in the state [DRÁ+25]. A scheduler observes the current location and valuation and the sampled expirations and resolves choices. It can let time elapse while the invariant holds and before any guard or an expiration is crossed. It can select one enabled nonstochastic jump. It can select among enabled stochastic jumps once an expiration has occurred. It can pick a concrete rate inside a rate interval when this is permitted by the location. Stochastic jumps do not occur before their expiration.

2.2.2 Rectangular automata with random clocks

RAC makes the durations of enabledness explicit by variables called random clocks. This turns part of the semantics into syntax [DRÁ+25]. It is convenient for reachability analysis since the values of these clocks appear in the state space directly. RAC is understood as a stochastically nonguarded RAE. For each event $r \in \mathrm{Lab}$ we identify r with a clock variable of the same name and require $\mathrm{Lab} \subseteq \mathrm{Var}$ and for every reachable state (ℓ,c,μ,s) that $c_r = \mu_r$ [DRÁ+25]. In every location the derivative of r equals 1 exactly while a jump labelled r with this location as source is enabled and equals 0 otherwise. Guards and invariants do not restrict these variables and resets leave them unchanged except that at a stochastic jump labelled r we set $r := 0$.

Definition 2.2.3 (RAC)**.** *A rectangular automaton with random clocks is a stochastically nonguarded RAE in which for every* $r \in \mathrm{Lab}$ *a variable with the same name is added to* Var*. In every location the derivative of* r *equals* 1 *exactly when some* r *labelled jump with that location as source is enabled and equals* 0 *otherwise. Guards and invariants do not restrict these variables and resets leave them unchanged except that*

at a stochastic jump with label r the value of r is set to 0. For every reachable state the valuation of the variable r equals the semantic duration μ_r [DRÁ+25]. Optionally a global time variable t with derivative 1 is added.

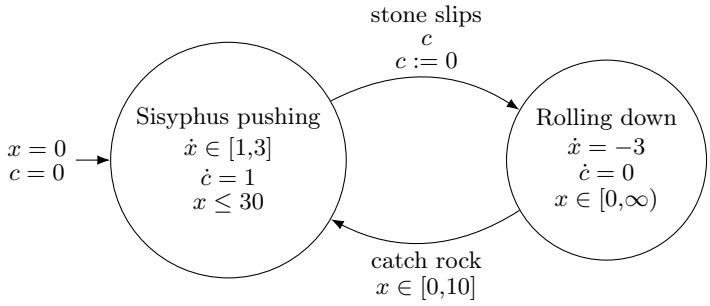

Figure 2.2: RAC version of the previous case. While pushing, the random clock c runs until the slip occurs when $c = R_c$ with $R_c \sim$ Uniform[0,6]. The nonstochastic "catch rock" jump is enabled for $x \in [0,10]$ and leaves x unchanged. Hence x never exceeds 28 and the invariant $x \leq 30$ is satisfied.

Figure 2.2 illustrates a RAC. In Sisyphus pushing the variable x increases with a rate in [1,3] and the random clock c grows with unit rate. At entry a sample R_c is drawn from [0,6] and the slip occurs when $c = R_c$. The c labelled jump to Rolling down resets $c := 0$. In Rolling down the nonstochastic jump back to Sisyphus pushing is enabled for $x \in [0,10]$ and leaves x unchanged. The model is non-deterministic because of the rate choice and the choice of when to take the nonstochastic jump while its guard holds. Since $R_c \leq 6$ and the pushing rate is at most 3, x never exceeds 18 before a slip when starting from 0. After a catch we re-enter with $x \in [0,10]$ and the next pushing phase can add at most 18, hence x never exceeds 28. The invariant $x \leq 30$ is respected and deadlocks are avoided.

2.3 CAMELS

The "(de-)Composed And More: Eager and Lazy Specifications (CAMELS)" [WRÁ25] classification was introduced to systematically distinguish between different semantic variants of stochastic HA [WRÁ23, WRÁ25]. Over the years, a variety of extensions to HA have been proposed to capture stochastic behavior, but their expressiveness and semantic assumptions were often incomparable or implicit. The CAMELS framework formalizes and classifies these variants based on how stochasticity and non-determinism are resolved during execution.

The classification is based on two orthogonal modeling dimensions

- Scheduling granularity:
  Distinguishes whether stochastic delays are sampled globally (composed) or individually per transition (decomposed).

- Realizability semantics:
  Distinguishes how the model ensures that randomly selected delays correspond to executable transitions. Here three variants are distinguished

- Lazy: delays may lead to disabled transitions and require resampling
- Eager predictive: only realizable delays are sampled which requires global knowledge
- Eager non-predictive: samples durations of enabledness rather than absolute delays

While the combination of scheduling and realizability dimensions yields six theoretical variants, the decomposed eager predictive setting is excluded from the classification as it is not semantically well defined [WRÁ25]. As a result, CAMELS identifies five meaningful and distinct model classes. This classification provides a unified framework for comparing the semantics of various stochastic hybrid system formalisms and tools [WRÁ23, WRÁ25].

Although in the paper [WRÁ25] we get a detailed semantic characterization for each variant, we restrict ourselves here to summarizing the key structural distinctions of the five well defined model classes. Our goal is to convey the core modeling ideas relevant to tool comparison.

In the following we consider stochastic extensions of simple HA only, which have a single initial state and $\mathbb{R}^d$ as invariant in each location [WRÁ25]. We first fix measurability and resampling conventions used throughout, adapted from [WRÁ25].

Definition 2.3.1 (Stochastic conventions, adapted from [WRÁ25])**.** *We write $\mathcal{B}(X)$ for the Borel σ-algebra on a topological space X. For a finite set S we set $\mathcal{B}(S) = 2^S$. For products we use the product σ-algebra. The hybrid state space is $\Sigma = \mathrm{Loc} \times \mathbb{R}^d$ with $\mathcal{B}(\Sigma) = 2^{\mathrm{Loc}} \otimes \mathcal{B}(\mathbb{R}^d)$.*

A stochastic kernel from $(X,\mathcal{A})$ to $(Y,\mathcal{B})$ is a map κ with type $\mathcal{B} \times X \to [0,1]$ such that for each $x \in X$ the function $B \mapsto \kappa(B,x)$ is a probability measure on $(Y,\mathcal{B})$ and for each $B \in \mathcal{B}$ the function $x \mapsto \kappa(B,x)$ is $\mathcal{A}$-measurable. We write Dist_x^κ for the probability measure induced by κ at x [WRÁ25].

For a probability measure μ on Y we write $\mathrm{supp}(\mu) = \{y \in Y \mid \mu(U) > 0$ for every open $U \ni y\}$. If μ has a density f we also use $\mathrm{supp}(f) = \{y \in Y \mid f(y) > 0\}$.

For $\sigma = (\ell,x) \in \Sigma$ the set of enabled labels is

$$E_\Sigma(\sigma) = \{\, a \in \mathrm{Lab} \mid \exists(\ell,g,a,r,\ell') \in \mathrm{Edge} \text{ with } x \in g \,\}\,. \text{ [WRÁ25]}$$

Definition 2.3.2 (Resampling extensions, adapted from [WRÁ25])**.** *Let $\mathcal{H}_{\mathrm{in}} = (\mathrm{Loc},\mathrm{Var},\mathrm{Flow},\top,\mathrm{Lab},\mathrm{Edge}_{\mathrm{in}},\mathrm{Init})$ be a simple HA.*

Composed resampling extension: add for each location $\ell \in \mathrm{Loc}$ one self loop $e_\ell = (\ell,\epsilon,g_\ell,\mathrm{id},\ell)$ with a fresh label $\epsilon \notin \mathrm{Lab}$, identity reset, and a guard g_ℓ that holds exactly when no original outgoing jump from ℓ is enabled [WRÁ25].

Decomposed resampling extension: add for each location $\ell \in \mathrm{Loc}$ and each label $a \in \mathrm{Lab}$ one self loop $e_{\ell,a} = (\ell,a,g_{\ell,a},\mathrm{id},\ell)$ with identity reset and a guard $g_{\ell,a}$ that holds exactly when among the original jumps leaving ℓ with label a exactly one is enabled in the current state [WRÁ25].

We write $\mathcal{H}_{\mathrm{comp}}$ and $\mathcal{H}_{\mathrm{decomp}}$ for the resulting extensions. Their clock augmented versions are denoted $\mathcal{H}^*_{\mathrm{comp}}$ and $\mathcal{H}^*_{\mathrm{decomp}}$ and add a fresh clock c with $\dot{c} = 1$ in every location and reset c to 0 on every discrete jump [WRÁ25].

2.3.1 Composed lazy (CL) hybrid automata

In CL scheduling stochastic choices happen in two stages. First a random delay until the next jump is sampled. Then once the delay has elapsed a jump is selected

at random among the enabled transitions. We model this by adding two stochastic kernels Ψ_c for delays and Ψ_d for labels.

Definition 2.3.3 (Kernels used in CL)**.** *We use two stochastic kernels as in the conventions.*

- Ψ_c *delay kernel. For each hybrid state it gives a distribution over non-negative delays. A sample from this distribution is written* R
- Ψ_d *label kernel. For each hybrid state it gives a distribution over jump labels with probability zero on disabled labels*

We call a kernel discrete when it induces a discrete distribution on a finite outcome set and continuous when the induced measure admits a density on $\mathbb{R}^n$*.*

Let $\mathcal{C} = (\mathcal{H},\Psi)$ be a Composed Hybrid Automaton with lazy specification where $\mathcal{H} = \mathcal{H}_{\text{comp}}$ is the composed resampling extension of a simple HA and $\Psi = (\Psi_c,\Psi_d)$ are the kernels defined above. To track time a fresh clock c is introduced and the state is extended to store the sampled delay R.

After each jump a delay R is drawn using Ψ_c. The clock c increases with rate one. When $c = R$ the current state is read and a label a is drawn using Ψ_d. A jump with label a is taken and c is reset. If no jump is enabled at time R a new delay is drawn and the evolution continues.

We now give the syntax and semantics of a CHA with lazy specification.

Definition 2.3.4 (Composed Lazy Syntax, adapted from [WRÁ25])**.** *A CHA with lazy specification is a tuple* $\mathcal{C} = (\mathcal{H}, \Psi)$ *with* $\Psi = (\Psi_c,\Psi_d)$ *where*

- $\mathcal{H} = (\text{Loc}, \text{Var}, \text{Flow}, \top, \text{Lab}, \text{Edge}, \text{Init})$[1] *is the composed resampling extension of a simple HA with state space* Σ
- Ψ_c *is a continuous stochastic kernel from* $(\Sigma,\mathcal{B}(\Sigma))$ *to* $(\mathbb{R}_{\geq 0},\mathcal{B}(\mathbb{R}_{\geq 0}))$
- Ψ_d *is a discrete stochastic kernel from* $(\Sigma,\mathcal{B}(\Sigma))$ *to* $(\text{Lab},2^{\text{Lab}})$ *such that* $\text{supp}(\text{Dist}_\sigma^{\Psi_d}) \subseteq E_\Sigma(\sigma)$ *for all* $\sigma \in \Sigma$

Definition 2.3.5 (Composed Lazy Semantics, adapted from [WRÁ25])**.** *Let* $\mathcal{C} = (\mathcal{H}, (\Psi_c, \Psi_d))$ *be a CHA with lazy specification. A run of* $\mathcal{C}$ *is a sequence* $\pi = (\sigma_0, R_0) \xrightarrow{t_0} (\sigma_0', R_0) \xrightarrow{a_{w(0)}} (\sigma_1, R_1) \xrightarrow{t_1} \cdots$ *where each* R_i *is a delay drawn from* Ψ_c *and* $a_{w(i)}$ *is a label drawn from* Ψ_d*.*

The sequence $\pi' = \sigma_0 \xrightarrow{t_0} \sigma_0' \xrightarrow{a_{w(0)}} \sigma_1 \xrightarrow{t_1} \cdots$ *must be a run of the extended automaton* $\mathcal{H}^*_{\text{comp}}$ *and the following conditions hold for all time steps*

- R_i *belongs to the support of* Ψ_c *at state* σ_i *that is* $R_i \in \text{supp}(\text{Dist}_{\sigma_i}^{\Psi_c})$
- *For each* $0 \leq i < \text{len}(\pi')$ *the clock value satisfies* $\sigma_i'(c) = R_i$ *and the chosen label* $a_{w(i)}$ *is supported by* Ψ_d *at* σ_i' *i.e.* $a_{w(i)} \in \text{supp}(\text{Dist}_{\sigma_i'}^{\Psi_d})$
- *If the run ends in a final time step* $(\sigma_i, R_i) \xrightarrow{t_i} (\sigma_i', R_i)$ *then it must hold that* $\sigma_i'(c) \leq R_i$

[1] Here $\top$ denotes the trivial invariant function, i.e. $\top(\ell) = \mathbb{R}^d$ for all $\ell \in \text{Loc}$.

To ensure semantic soundness the support of Ψ_d at any state is restricted to labels of enabled transitions so that the system never attempts invalid jumps. While the sampled delay R_i could be encoded as a countdown state variable r with $\dot{r} = -1$ that is reset to R_i and triggers a jump when $r = 0$, we instead use a clock c that increases with rate one and check $c = R_i$. This keeps the original flows unchanged and aligns the construction with the other scheduling variants.

2.3.2 Composed eager predictive (CEP) stochastic hybrid automata

CEP scheduling restricts the behavior of composed lazy scheduling by precomputing, for each state, the set of time delays that allow at least one jump to be enabled. The delay kernel Ψ_c is constrained to only sample from these so-called enabling delays, ensuring that a jump will always be possible when the sampled delay elapses. As a result, no resampling is needed at the time of jumping.

Definition 2.3.6 (Composed Eager Predictive Syntax, adapted from [WRÁ25])**.** *A CHA with eager predictive specification is a tuple $\mathcal{C}^{\mathcal{E}P} = (\mathcal{H}, \Psi)$ with $\Psi = (\Psi_c, \Psi_d)$, such that:*

- *$\mathcal{H}$ is a simple HA with state space Σ and $\Psi_d : 2^{\mathrm{Lab}} \times \Sigma \to [0,1]$ as in CL,*
- *the support of Ψ_c is restricted to the enabling time set, i.e., $\mathrm{supp}(\mathrm{Dist}_{\sigma}^{\Psi_c}) \subseteq T_{\mathrm{in}}(\sigma)$ for all $\sigma \in \Sigma$,*

where $T_{\mathrm{in}}(\sigma)$ denotes the set of delays after which at least one jump is enabled from state σ.

The semantics of $\mathcal{C}^{\mathcal{E}P}$ are the same as those of CL scheduling except that there is no resampling step and runs are taken in $\mathcal{H}^*$ since $\mathcal{H}$ is a simple HA [WRÁ25].

2.3.3 Composed eager non-predictive (CENP) stochastic hybrid automata

In CENP scheduling, a delay is sampled only once the system enters a state where at least one jump is enabled. At that point, a stopwatch clock c is initialized to zero and begins tracking the elapsed time. The clock evolves with $\dot{c} = 1$ as long as the system is in a state where any regular (non-resampling) jump is enabled, and remains frozen with $\dot{c} = 0$ otherwise.

This can be formalized as:

$$\dot{c} = \begin{cases} 1, & \text{if } E_{\Sigma}(\sigma) \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

where $E_{\Sigma}(\sigma)$ denotes the set of enabled transitions in state σ. As soon as c reaches the sampled delay, a transition is selected via Ψ_d, executed, and c is reset.

Definition 2.3.7 (Composed Eager Non-predictive Syntax, adapted from [WRÁ25])**.** *A CHA with eager non-predictive specification is a tuple $\mathcal{C}^{\mathcal{E}_N} = (\mathcal{H}, \Psi)$ with $\Psi = (\Psi_c, \Psi_d)$, such that $(\mathcal{H}, \Psi)$ satisfies the syntax of a CHA with lazy specification.*

The semantics of $\mathcal{C}^{\mathcal{E}_N}$ follows that of composed lazy scheduling, but is modified to include the stopwatch clock c.

2.3.4 Decomposed lazy (DL) stochastic hybrid automata

Whereas CL scheduling samples a single delay for all transitions, DL scheduling introduces a separate delay for each label. In this setting, the system executes the jump whose associated delay expires first, effectively creating a race condition between labels. The structure is otherwise similar to the composed variant, but requires multiple delay clocks.

Let $\mathcal{D} = (\mathcal{H}, \Psi)$ denote a Decomposed Hybrid Automaton (DHA) with lazy specification, where $\Psi = (\Psi_1, \ldots, \Psi_k)$ consists of one delay kernel Ψ_i for each label $a_i \in \text{Lab} = \{a_1, \ldots, a_k\}$. The state space is extended to include a delay vector $R \in \mathbb{R}^k_{\geq 0}$, with one component per label. A fresh clock c_j is introduced for each label a_j to track its delay independently.

Definition 2.3.8 (Decomposed Lazy Syntax, adapted from [WRÁ25])**.** *A DHA with lazy specification is a tuple $\mathcal{D} = (\mathcal{H}, \Psi)$ where:*

- $\mathcal{H} = (\text{Loc}, \text{Var}, \text{Flow}, \top, \text{Lab}, \text{Edge}, \text{Init})$ *is the decomposed resampling extension of a simple HA, with state space Σ.*
- *For each $i \in \{1, \ldots, k\}$, $\Psi_i : \mathcal{B}(\mathbb{R}_{\geq 0}) \times \Sigma \to [0,1]$ is a continuous stochastic kernel from $(\Sigma, \mathcal{B}(\Sigma))$ to $(\mathbb{R}_{\geq 0}, \mathcal{B}(\mathbb{R}_{\geq 0}))$.*

Definition 2.3.9 (Decomposed Lazy Semantics, adapted from [WRÁ25])**.** *Let $\mathcal{D} = (\mathcal{H}, \Psi)$ be a DHA with lazy specification. A run of $\mathcal{D}$ is a sequence $\pi = (\sigma_0, R_0) \xrightarrow{t_0} (\sigma'_0, R_0) \xrightarrow{a_{w(0)}} (\sigma_1, R_1) \xrightarrow{t_1} \cdots$ such that $\pi' = \sigma_0 \xrightarrow{t_0} \sigma'_0 \xrightarrow{a_{w(0)}} \sigma_1 \xrightarrow{t_1} \cdots$ is a run in the extended automaton $\mathcal{H}^*_{\text{dec}}$, and:*

- *For every time step i, the delay vector R_i contains one delay per label, i.e., $R_i \in \mathbb{R}^k_{\geq 0}$ for all $0 \leq i \leq \text{len}(\pi')$.*
- *At the start of the run, each component $R_0[j]$ is sampled from the corresponding delay distribution Ψ_j at σ_0, so $R_0[j] \in \text{supp}(\text{Dist}^{\Psi_j}_{\sigma_0})$ for all $j \in \{1, \ldots, k\}$.*
- *For each transition step, the clock $c_{w(i)}$ of the chosen label reaches its sampled delay, i.e., $\nu'_i(c_{w(i)}) = R_i[w(i)]$, and all clocks satisfy $\nu'_i(c_j) \leq R_i[j]$ for every j. After the jump, a new delay $R_{i+1}[w(i)]$ is drawn from $\Psi_{w(i)}$ at the new state σ_{i+1}, while delays for all other labels remain unchanged: $R_{i+1}[j] = R_i[j]$ for $j \neq w(i)$.*
- *If the run ends after a final time step $(\sigma_i, R_i) \xrightarrow{t_i} (\sigma'_i, R_i)$, then all clocks must still be within their delays, i.e., $\sigma'_i(c_j) \leq R_i[j]$ for every j.*

2.3.5 Decomposed eager predictive stochastic hybrid automata

As previously mentioned, this setting is semantically problematic because eager predictive scheduling requires the set of possible delays to be precomputed for each state. However, in decomposed scheduling, delays are sampled per label and retained across steps unless that label wins the race. Ensuring that all sampled delays lead to valid enabled transitions, without knowing which label will win, would require conditioning on the outcomes of other random variables, which is not generally possible. As a result, the decomposed eager predictive variant cannot be defined in a sound and deterministic way, and is therefore excluded from the CAMELS framework [WRÁ25].

2.3.6 Decomposed eager non-predictive (DENP) stochastic hybrid automata

In this model each label is assigned its own stopwatch to measure how long the corresponding jump has been enabled. Unlike predictive semantics no precomputation of enabling times is required. Instead each delay is interpreted as the time the corresponding jump must be enabled before it can be triggered. The winner of the race is the first label whose stopwatch reaches the sampled delay.

A DHA with eager non-predictive specification is a tuple $\mathcal{D}^{\mathcal{E}_N} = (\mathcal{H}, \Psi)$ where $\Psi = (\Psi_1, \ldots, \Psi_k)$ is a tuple of per label delay kernels. The syntax uses a simple HA $\mathcal{H}$ with state space Σ and the kernels Ψ_i as in DL. The semantics reuse the structure of DL scheduling extended with stopwatch clocks c_i for each label $a_i \in \text{Lab}$.

Each stopwatch c_i evolves according to

$$\dot{c}_i = \begin{cases} 1, & \text{if } E_\Sigma^{a_i}(\sigma) \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \qquad \text{where} \qquad E_\Sigma^{a_i}(\sigma) = \{\, e = (\ell, a_i, g, r, \ell') \mid e \in E_\Sigma(\sigma) \,\}.$$

That is c_i increases only when a regular (non-resampling) jump with label a_i is enabled. Otherwise it remains constant.

As a compact illustration, Fig. 2.3 shows a DENP instance ("DENP Sisyphus"). In location ℓ_0 two stopwatches c_1 (stone slips) and c_0 (reach top) run exactly when their corresponding labels are enabled. The event whose stopwatch first meets its sampled enabling duration fires. A slip resets the stone to $x = 0$ and returns to ℓ_0. We use $c_1 \sim \text{Uniform}[0,18]$ and $c_0 \sim \text{Uniform}[0,10]$.

Definition 2.3.10 (Decomposed Eager Non-predictive Syntax, adapted from [WRÁ25])**.** *A DHA with eager non-predictive specification is a tuple $\mathcal{D}^{\mathcal{E}_N} = (\mathcal{H}, \Psi)$ such that $\mathcal{H}$ is a simple HA with state space Σ and $\Psi = (\Psi_1, \ldots, \Psi_k)$ is as in DL.*

The semantics of $\mathcal{D}^{\mathcal{E}_N}$ follow those of DL scheduling but without a resampling step and runs are taken in $\mathcal{H}^*$ with per label stopwatch clocks c_i as defined above.

2.4 RA, RAE, and RAC in the CAMELS view

Rectangular Automata (RA) have no stochastic delays and therefore sit outside CAMELS, which classifies only stochastic hybrid formalisms [WRÁ25]. Rectangular Automata with Random Events (RAE) model per-label stochastic delays as enabling durations. When a label is enabled, a duration is sampled and the jump fires once the accumulated enabledness reaches that duration. Semantically, this corresponds to decomposed, eager non-predictive (DENP) CAMELS semantics [WRÁ25]. Rectangular Automata with Random Clocks (RAC) make those per-label enabledness timers explicit as continuous variables (*random clocks*). RAC is a syntactic subclass of RAE with the same CAMELS corner, namely DENP. The explicit clocks are useful for geometric reachability and optimization [DRÁ+25, DSSR24].

Formalism	CAMELS class
RA (no stochastic delays)	not classified (outside CAMELS)
RAE	DENP (semantics)
RAC	DENP (semantics, explicit clocks)

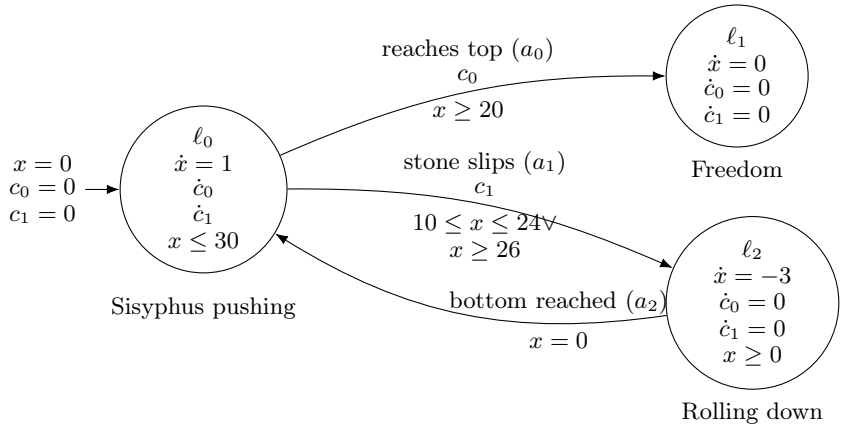

Figure 2.3: DENP Sisyphus. In pushing (ℓ_0), two stopwatches run: c_1 (stone slips) is enabled on $10 \leq x \leq 24 \vee 26 \leq x \leq 30$ and pauses in the gap $24 < x < 26$ (a cave where slipping cannot occur), while c_0 (reach top) is enabled for $x \geq 20$. The event whose stopwatch first reaches its sampled enabling duration fires. If slip occurs, the stone rolls to $x = 0$ and the system returns to ℓ_0. We use $c_1 \sim$ Uniform[0,18] and $c_0 \sim$ Uniform[0,10].

2.5 UPPAAL

UPPAAL is a tool suite for modeling, simulating, and verifying real-time systems, based on timed automata. It supports networks of automata communicating via synchronization channels, where transitions are guarded by clock constraints and may include variable and clock resets. Clocks are declared with the data type `clock`, they are real valued timers. They can be reset in updates.

The user models systems using graphical automaton templates. Each template contains locations with invariants, edges with guards and synchronizations, and updates to variables or clocks. Invariants bound how long the automaton may stay in a location, for example $c \leq 10$. Guards specify when an edge may fire, for example $c \geq 5$, and resets like $c = 0$ start a clock anew. Multiple templates can be composed into a system via parallel composition.

UPPAAL includes different analysis modes:

- The **Symbolic Simulator** explores system behavior with symbolic zone based techniques, where a zone is a convex set of clock valuations described by constraints like $x,y \in \mathbb{R}\ x \leq c$ and $x - y \leq c$ and represented internally by difference bound matrices, letting one symbolic step cover many concrete states.
- The **Concrete Simulator** executes one run with concrete numeric values starting from the initial state with clocks 0 and variable initializers and picks delays and edges interactively or at random to produce a single trace.
- The **Verifier** evaluates queries such as reachability or safety properties over the state space.

Figure 2.5 shows a run of the Concrete Simulator for the model in Figure 2.4, which approximately models Figure 2.2, with random sampling and linear evolution. Simulation steps and clock valuations are updated concretely at runtime.

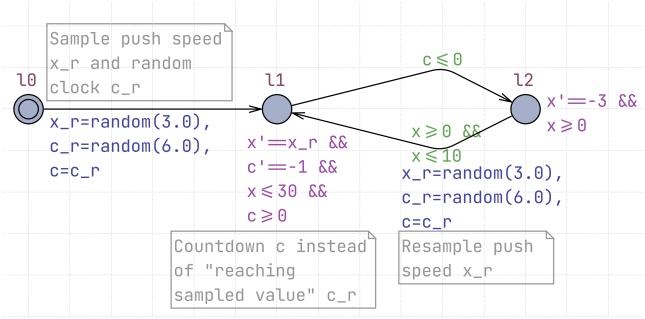

Figure 2.4: UPPAAL model of Sisyphus with stochastic uphill speed approximately modelling Figure 2.2. We use two clocks `x` and `c` and the corresponding double variables `x_r` and `c_r`. Sampling is only possible for doubles so we draw `x_r` and `c_r` and then assign the clocks, initially `c=c_r` and during ascent `x'=x_r` and `c'=-1` to count down, see subsection 2.5.1. At the start Sisyphus samples a random pushing speed `x_r` and ascends until Zeus forces an early rollback as the random event `c` with uniform distribution in [0,6]. When rolling down the speed is fixed at -3. Upon returning near the origin we allow the jump back to the pushing mode whenever $x \in [0,10]$ where a new random speed is sampled for the next push.

To improve readability, all textual elements inside our UPPAAL models follow a consistent color scheme. Grey translucent boxes denote comments. The weights on transitions come from SMC (see next subsection).

- Location name
- Invariant (location)
- Guard (transition)
- Update / assignment (transition)
- Probability / weight (transition)

2.5.1 SMC

UPPAAL SMC (Statistical Model Checking) is an extension of UPPAAL that enables the analysis of stochastic timed systems by combining simulation with statistical inference. Instead of exhaustively exploring the entire state space, UPPAAL SMC generates multiple runs of the system under randomized timing behavior and uses statistical tests to estimate or bound the probability that a given property holds [DLL+15].

Stochasticity in UPPAAL SMC is introduced in two ways:

- Random delays specified using `random(n)` sample uniformly from $[0,n)$ with $n \in \mathbb{R}$ as specified in the UPPAAL docs [JMG+20].
- Stochastic race semantics, where concurrently enabled components compete to execute their transitions, with probabilities derived from their timing distributions.

In the editor screenshot (Figure 2.4), both features are demonstrated: the doubles `x_r` and `c_r` are assigned values sampled via `random(3.0)` and `random(6.0)`,

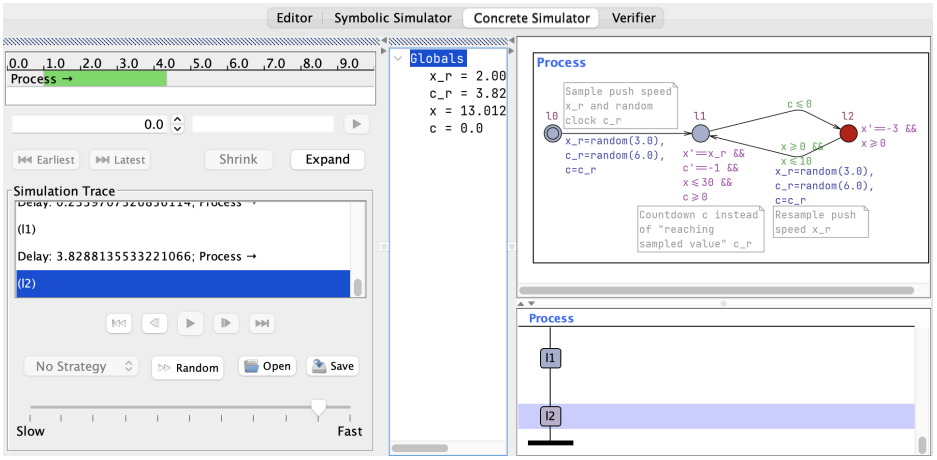

Figure 2.5: UPPAAL's Concrete Simulator with time steps and transitions with random delay realizations.

and the location `l1` uses the notation `x'==x_r` and `c'==-1` to indicate that `x` increases with the sampled speed while `c` counts down. More generally, UPPAAL SMC supports derivative expressions such as `x'==z` where `z` is any arithmetic expression over current variables for example a double variable like `x_r`. This differential behavior combined with guards and invariants defines the timing dynamics of the system during simulation or model checking.

UPPAAL SMC supports several query types [DLL+15], such as:

- Probability estimation:
  `Pr[<=T](<> goal)` estimates the probability that a goal location is reached within time T.
- Hypothesis testing:
  `Pr[<=T](<> goal) >= p` tests whether the goal is reached within time T with probability at least p.
- Expected value:
  `E[<=T](max: x)` computes the maximum expected value of x during runs up to time T.

By running many randomized simulations, UPPAAL SMC can estimate these quantities with configurable statistical confidence (e.g., 95%). This makes it suitable for verifying performance, safety, and reliability properties in systems with uncertainty or variability in timing behavior [DLL+15].

2.5.2 Stratego

UPPAAL stratego is an extension of UPPAAL that supports controller synthesis and strategy optimization for non-deterministic stochastic systems [DJL+15]. While UPPAAL SMC focuses on estimating the probability of property satisfaction under randomized behaviors, stratego in the presence of non-determinism aims to automatically construct a strategy (or controller) that optimizes a given quantitative objective.

A strategy in this context is a function that resolves non-determinism by selecting transitions based on the current state. At the same time timing remains under control of the built-in stochastic race. The goal is to synthesize strategies that maximize (or minimize) metrics such as the probability of reaching a goal, expected cost, or time to completion.

UPPAAL stratego operates on models where certain choices are left non-deterministic, for instance, which action to take when multiple transitions are enabled. These are interpreted as decision points under the control of a scheduler. The tool explores the space of possible strategies and uses reinforcement learning techniques (such as statistical policy iteration) to iteratively improve candidate strategies based on simulation results [DJL+15].

The resulting strategy can then be simulated or exported for deployment. Users can define queries such as:

- `strategy optimal = minE (time) [<= T]: goal;`
  to find a strategy that minimizes expected time to reach the goal within time T.
- `simulate 1 under optimal;`
  to simulate the system under the synthesized strategy.

This approach is illustrated in practical scenarios such as adaptive cruise control [LMT15], where the synthesized controller ensures both safety and performance. UPPAAL stratego thus bridges formal verification and control synthesis. It supports a class of stochastic hybrid games and can be seen as a lightweight alternative to full-blown game-theoretic solvers.

2.6 RealySt

RealySt is a C++ tool for modeling and analyzing stochastic hybrid systems within the formalism of RAC called RAR in the RealySt paper and their singular variant SAR [DSSR24]. In these models invariants and guards are rectangular sets, flows are specified per variable as constant or interval bounded rates, and stochastic delays are represented explicitly through random clock variables [DRÁ+25]. This explicit timing representation enables the use of geometric reachability techniques to compute maximum reachability probabilities under prophetic schedulers which resolve non-determinism with full knowledge of future random event times [DSSR24].

From a modeling perspective, a RealySt model consists of a set of locations connected by transitions. Each location defines its invariant, continuous flows, and any random clocks that track the duration of enabledness for stochastic events. Transitions carry guards, resets, and labels for deterministic or stochastic events, the latter being associated with probability distributions from which delays are sampled when enabled. The main modeling elements are:

- Locations with rectangular invariants and constant or interval-bounded rates per variable.
- Transitions with guards, resets, and labels for deterministic or stochastic events.
- Random clocks for events that evolve while the corresponding stochastic event remains enabled.

The analysis in RealySt combines geometric reachability computation with probabilistic evaluation [DSSR24]. It begins by determining all states that can be reached within the specified time and jump bounds, taking into account both the continuous dynamics and the evolution of the random clocks. From this set, a backward analysis identifies the states from which the goal is still attainable, resolving non-deterministic choices so as to maximize the probability of success. The resulting set is then projected onto the dimensions representing the random clock values, which describe all possible combinations of stochastic delays that lead to the goal. Finally, RealySt integrates over this domain to obtain the overall probability, using Monte Carlo sampling to provide both an estimate and a confidence bound [DRÁ+25].

From a usage perspective, RealySt is a command-line tool for Linux, with support for execution under the Windows Subsystem for Linux as well as via a prebuilt Docker container [DSSR24]. Models are defined directly in C++ source files, which allows them to be constructed programmatically and tailored to specific experiments. The tool is invoked as `./realyst [OPTIONS]`, where for example we use:

```
./realyst -t 30 -d 100 -b SISYPHUS -m A
          --plotDimensions 0 1 -l trace
```

The parameters are:

- `-t 30` sets the global time horizon for the reachability analysis to 30 time units,
- `-d 100` bounds the maximal number of jumps analyzed by 100,
- `-b SISYPHUS` selects the benchmark model,
- `-m A` chooses the model variant,
- `-plotDimensions 0 1` invokes plotting using variables 0 and 1 as dimensions,
- `-l trace` sets the logging verbosity to trace, other choices are `off`, `critical`, `err`, `warn`, `info`, `debug`.

During execution, RealySt outputs the chosen parameter settings and the computed reachability probability, with optional intermediate information depending on the configured logging level.

By integrating exact geometric reasoning with statistical integration, RealySt delivers precise probability estimates for bounded-time reachability problems in stochastic hybrid systems [DSSR24, DRÁ+25]. Its focus on prophetic schedulers and explicit stochastic timing makes it well suited for scenarios where both non-deterministic control decisions and probabilistic event timing must be analyzed within the same framework.

2.6.1 Encoding goals in RealySt

We encode the Sisyphus model shown in Figure 4.1, which was obtained by converting the discrete-event net representation in Figure 2.3. The full C++ model is listed in Appendix A. RealySt allows us to specify the reachability goal directly in code, either as one target (not more) location or as a state constraint or a combination of both.

Goal as location:

```
librealyst::Specification S{};
S.goalLocation = L2_ROLL;  // reach location "L2_ROLL"
spec.addSpecification(S);
```

This creates a specification object S sets its goal location to the identifier L2_ROLL and registers it in the global problem specification spec.

Goal in matrix notation:

```
// Encode x >= 30 as  -x <= -30
hypro::matrix_t<Number> M=hypro::matrix_t<Number>::Zero(1,4);
hypro::vector_t<Number> b(1);
M(0, 1) = static_cast<Number>(-1);  // -x
b(0)    = static_cast<Number>(-30); // -x<=-30  <=>  x>=30

hypro::ConstraintSetT<Number> cset(M, b);
librealyst::Specification S{};
S.constraints = hypro::Condition<Number>(cset);
spec.addSpecification(S);
```

Here constraints have the form $Av \leq b$ over the variable vector (t,x,c_0,c_1). We encode $x \geq 30$ as $-x \leq -30$. The single row puts -1 in the column for x and zeros elsewhere and sets $b = -30$. The constraint set is wrapped into a condition assigned to S and added to spec. If both a goal location and constraints are given the target is their intersection.

Corresponding matrix:

$$\begin{array}{rcccccc} & & \text{Variables:} & & & & \text{Vector:} \\ & t & x & c_0 & c_1 & & \\ \text{Constraint}_0: & (0 & -1 & 0 & 0 & \mid & -30\ ) \end{array}$$

A run produces a reachability graph of explored locations and transitions used for the backward analysis and a probability estimate of reaching the given specification within the bounds together with confidence information. Figure 2.6 shows an example of such a reachability graph for the automaton in Appendix A obtained with the command above.

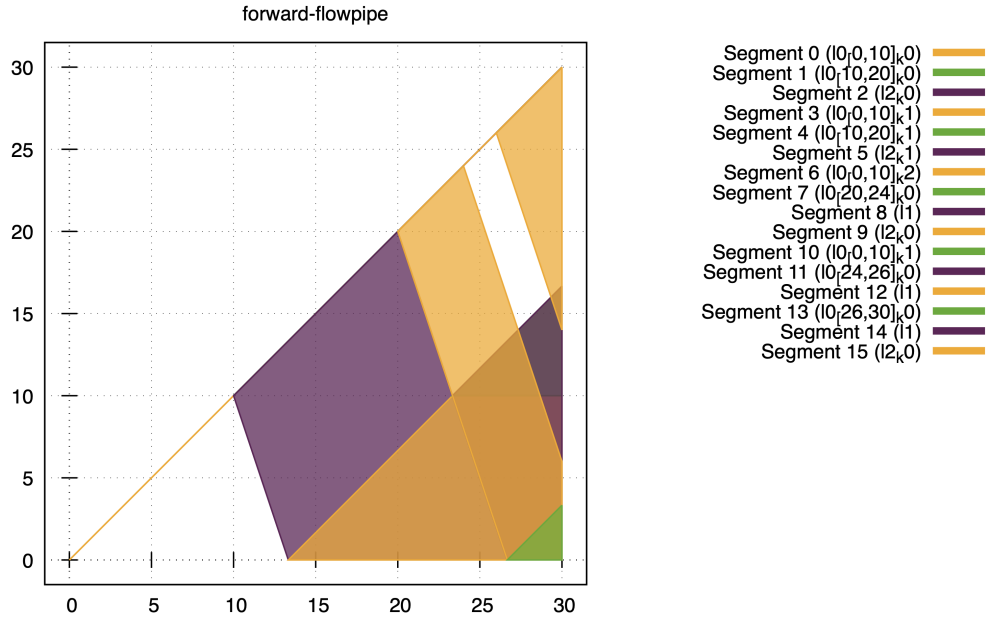

Figure 2.6: Reachability graph produced by RealySt for the Sisyphus automaton from Appendix A obtained with `./realyst -t 30 -d 100 -b SISYPHUS -m A -plotDimensions 0 1 -l trace`.

In the legend a Segment is one convex polytope of the forward flowpipe computed for one location within one time slab under one fixed choice of rates. Segments are numbered in the order in which the forward exploration creates them.

The label `l0[0,10]_k0` is the name of a location in the transformed RAC model. The base `l0` comes from the original model. During the transformation in Chapter 4 we split `l0` into several locations and encode their provenance in the name using the bracketed time interval and the suffix index. The whole token is a single location name in the graph.

Chapter 3

UPPAAL: Core Model and Stochastic Semantics

This chapter presents the UPPAAL model we use and its stochastic execution semantics. We first give the component syntax (locations, clocks, variables, flows, edges) and then the timing used in SMC-style analysis: bounded stays are uniform, unbounded stays use exponential exit rates, components race on their sampled delays, and discrete ties are resolved by edge weights [DLL+15].

Definition 3.1 (Component syntax with SMC attributes)**.** *A component is an extended timed or hybrid automaton* $A = (Loc, Clk, Var, Inv, Flow, E, \ell_0, v_0)$.

- *Locations Loc:*
  Each location has a kind in $\{normal, urgent, committed\}$. *Urgent and committed both forbid time elapse. Committed additionally requires that the next discrete step includes at least one transition from a component currently in a committed location. After each discrete step the status is re-evaluated from the new control state, if some component is in a committed location the restriction applies, else if some component is in an urgent location time may not pass, otherwise time may pass. Each location may also carry an SMC exit rate* $\lambda(\ell) \in \mathbb{R}_{\geq 0}$, *used only when the stay in* ℓ *is not bounded by its invariant.*

- *Clocks Clk:*
  Real valued variables declared with type `clock` *that are used in guards and invariants. They evolve according to location based rate equations written as* $c' == e$ *and they can be reset on edges. In classic symbolic analysis clocks advance with unit rate* 1 *while in SMC analysis they are driven by the declared rate equations. The right hand side* e *is an expression that only reads the current state and returns a real number, it may use numeric constants, clocks, double or integer variables, the operators* $+ - * /$ *and parentheses and built in math functions such as* sin, cos, log, exp, $\sqrt{}$ *as well as user defined pure functions.*

- *Variables Var:*
  Bounded integers, booleans, arrays and records. In SMC `double` *variables may appear in rates resets weights guards and invariants. If doubles occur in guards or invariants the symbolic simulator is disabled and analysis is done with SMC.*

- *Invariants Inv:*
  Each location ℓ carries an invariant $Inv(\ell)$. We write B for an integer valued bound expression (literals `const int` parameters or integer expressions over bounded `ints`) and use B_u and B_l for upper and lower bounds respectively. In normal form invariants constrain maximal remaining time by upper bounds on clocks, conjunctions of $c \leq B_u$ or $c < B_u$ with $c \in Clk$. The editor also permits lower bounds such as $c \geq B_l$. For classic timed automata where clocks are nondecreasing these do not restrict time progress once inside the location and we treat them as entry conditions and move them to guards on incoming edges. In SMC rate equations may make a clock decrease for example $c' == -1$, then a lower bound like $c \geq B_l$ does restrict time progress and is enforced as an invariant.

- *Flows Flow : $Loc \to \dot{Clk}$:*
  Each location ℓ declares derivative equations using $c' == expr$ inside the location. In SMC analysis the right hand side may be a general state-dependent arithmetic expression that can read integers and `doubles`. In classic symbolic timed automata there are no general rate equations and clocks have unit rate. In the UPPAAL editor, derivatives $c' == \cdots$ are written in the same location field as the invariant (next to constraints like $c \leq B_u$).

- *Edges E:*
  Each edge has the form $(\ell, select, guard, sync, update, w, \ell')$, where ℓ is the source and ℓ' the target. "select" declares local binders for this edge, each ranging over a finite type. The bound names may be used in the guard, sync, and update, and they do not persist after firing but the update might copy their values into the state. "guard" is a conjunction of clock-difference constraints and booleans (e.g. $c_1 - c_2 \leq B_u$ and predicates over $x \in Var$). "sync" uses broadcast channels with $a!$ for one sender and zero or more $a?$ receivers. Urgent broadcasts are declared in the global section with `urgent broadcast chan a` and used in the same way. No delay may occur when a synchronization on an urgent channel is enabled. A broadcast send is never blocking and if any receivers are enabled they must participate. "update" performs assignments and clock resets, and may include SMC sampling such as `random(n)`. The optional weight $w \in \mathbb{R}_{\geq 0}$ resolves discrete choices by normalizing over the enabled edges. A weight may be a state-dependent expression.

- *Initials ℓ_0, v_0:*
  ℓ_0 is the initial location. v_0 is the initial valuation over $Clk \cup Var$ at time 0, produced by declarations and initializers and any immediate updates before time can pass.

A system is a tuple $S = (Comp, Chan, Var^g)$ with $Comp = (A_1, \ldots, A_n)$ a finite sequence of components, Chan the set of channel names with a distinguished subset of urgent broadcast channels, and Var^g the set of shared global variables. The notation $A_1 \parallel \cdots \parallel A_n$ denotes the parallel execution of the components that communicate by broadcast over Chan and read and write the shared variables Var^g.

Definition 3.2 (SMC timing as part of the meaning)**.** *The stochastic timing used by SMC is attached to locations and edges.*

- *Delay distribution in a location:*
  Let s be a state in location ℓ. We write $c(s)$ for the current value of clock c in s. Let $b(s)$ be the maximal time for which the invariant of ℓ remains true along the declared derivatives from s, with $b(s) = +\infty$ if it never becomes false. In classic unit rate timed automata with upper bounds $c \leq B_u$ this is $b(s) = \min_c\big(B_u - c(s)\big)$. If $0 < b(s) < \infty$ then SMC samples a uniform delay $U[0,b(s)]$ and attempts to leave ℓ when that delay expires. If $b(s) = +\infty$ and the location declares a positive exit rate $\lambda(\ell) > 0$ then SMC samples an exponential delay with rate $\lambda(\ell)$. If $b(s) = +\infty$ and no positive rate is declared then the timing is not well defined for SMC queries, the concrete simulator still runs. When the sampled delay elapses a discrete step is taken if some outgoing edge is enabled, if no edge is enabled and no further delay is allowed by the invariant the run deadlocks, otherwise time continues and the component keeps delaying.

- *Random assignments:*
  Updates may sample values by SMC primitives such as uniform draws for reals. The sampled values are stored in variables and can influence later flows and guards [DLL+15].

- *Race between components:*
  In a network each component that can delay samples its own delay as above. The smallest sampled delay determines the next global action. Ties have probability zero [DLL+15]. A component is non blocked if it can let time pass from the current state, that is it has positive admissible delay $b_i(s) > 0$ or an exponential rate and it is not in an urgent or committed location and no urgent broadcast on one of its edges is enabled. Only non blocked components sample and participate in the race. If a component is blocked because $b_i(s) = 0$ or an urgent synchronization is enabled it must take a discrete step immediately if some edge is enabled otherwise the run deadlocks.

- *Discrete resolution at jump time:*
  When a component's delay expires, it may have multiple enabled outgoing edges. If multiple outgoing edges are enabled, UPPAAL reads optional non-negative weight expressions on those edges and chooses proportionally after normalizing so the probabilities sum to 1. Weights may depend on the current state, including sampled `double` values. For example, sample r and set an edge's weight to a function of r, such as $w := r$. If no weights are given, UPPAAL chooses uniformly among the enabled edges. To block an alternative in a choice situation, set its weight to 0. Weights are only consulted when at least two edges are enabled. If an edge is the only enabled option, it remains available and is taken deterministically.

Chapter 4

From DENP Semantics to RAC Syntax

We work in DENP and compile models that use RAE-style enabledness semantics into RAC, which is the syntax that RealySt analyzes. In RAC stochastic edges are unguarded. Enabled and disabled periods are encoded by random clock rates inside flows and invariants. If enabledness changes inside a location, we split the location into windows so enabledness is constant and each random clock has a constant rate in each window. For each label a we write c_a for its random clock. In a window W the rate is $\dot{c}_a = 1$ if a is enabled in W and $\dot{c}_a = 0$ otherwise. The original continuous dynamics of the plant variables and the location invariants stay as they are. Only the random clocks are added.

The compilation is as follows.

1. Windowing
   Cut each location along every hyperplane from each atomic guard literal of any stochastic edge and at every other enablement change point. This yields windows in which each literal has a fixed truth value so enabledness is constant. The invariant of each window is then an intersection of half spaces and stays convex and rectangular as required by RAC, see also the transformation figures in [DRÁ+25]. A hyperplane is the set of points where an affine equality $a^\top v = b$ holds for the continuous variables for example $x = 10$ and the literals $x \leq 10$ and $x \geq 10$ are the two half spaces cut by that hyperplane.

2. Clock rates:
   For each label a, set $\dot{c}_a = 1$ in windows where a is enabled and $\dot{c}_a = 0$ otherwise, while keeping the plant flows and invariants unchanged.

3. Unguarding:
   Replace each guarded stochastic edge for label a by unguarded edges from exactly those windows where its guard holds.

4. Deterministic switches:
   At every cutpoint insert a non-stochastic edge to the next window with a boundary guard, for example $x = 10$, and keep an invariant in the source window that prevents crossing the boundary, for example $x \leq 10$. Keep existing resets from the source model unchanged and we do not introduce new resets here.

This transformation keeps DENP behavior for time bounded and jump bounded reachability [DRÁ+25]. We only move enabledness accounting from semantics to explicit clocks. The number of windows equals the number of cells induced by the cuts. With rectangular guards the cuts are axis aligned. If variable x_i has k_i change points we get at most $k_i + 1$ windows along that dimension and in total at most $\prod_i (k_i + 1)$ windows. It is linear when all change points lie on one variable as in our Sisyphus example but in general it grows multiplicatively and can be exponential in the number of variables. The added deterministic switches are of the same order. Random clocks are the only new continuous variables. Boundary cuts lie on equalities like $x = 10$. With continuous sampling the chance to hit exactly that value is zero so including the equality on one side or the other does not change probabilities. The transformation can expose or create Zeno switching loops at cut points because of the added nonstochastic switches. These loops have probability zero under continuous sampling but they exist syntactically so we either restrict to time divergent runs or use time and jump bounds in the analysis. Figure 4.1 shows one RAC automaton with the windowed copies $\ell_0[\cdot,\cdot]$ and the locations ℓ_1 and ℓ_2 where the random events a_0 and a_1 use clocks c_0 and c_1 and the deterministic edge a_2 resets x to 0.

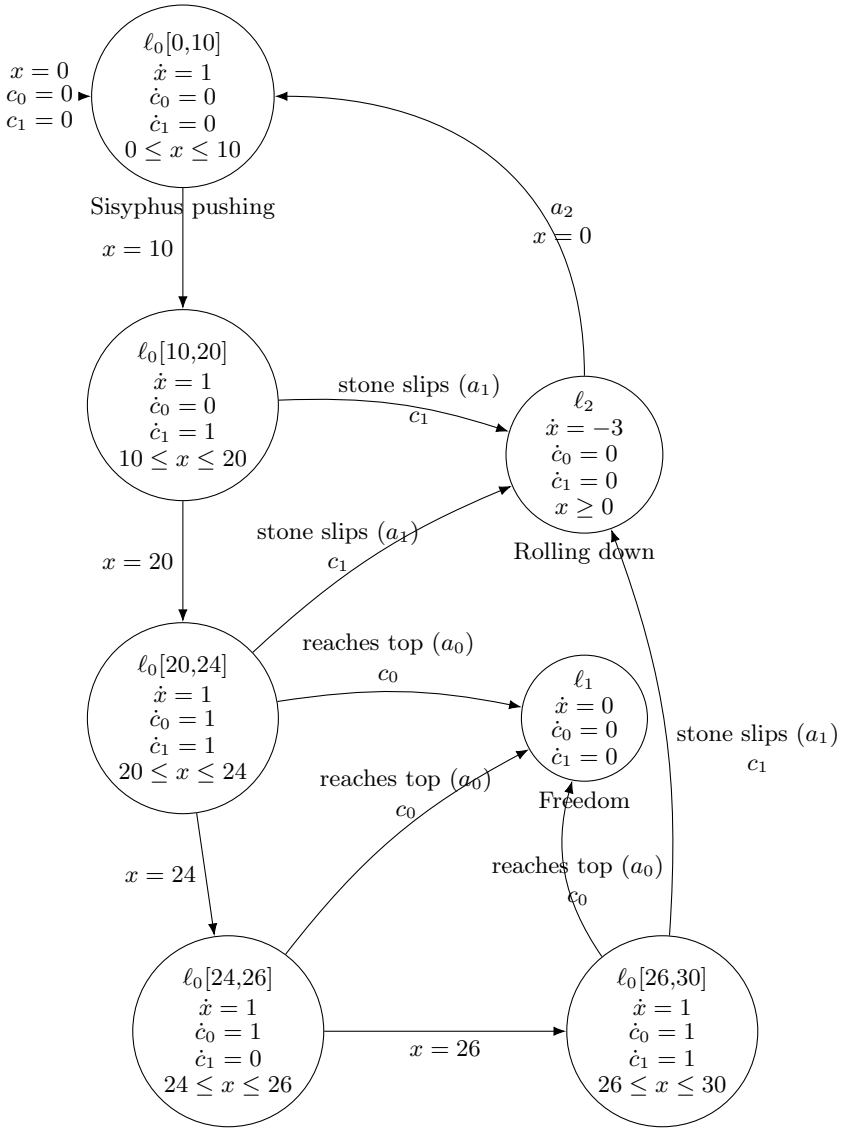

Figure 4.1: RAC encoding of the DENP Sisyphus in Figure 2.3. We split ℓ_0 at $x = \{10,20,24,26\}$ so enabledness is constant in each window. Random clocks run with rate 1 where their labels are enabled. Stochastic edges are unguarded. Deterministic switches at the cut points and the reset at $x = 0$ complete the model with $c_1 \sim U[0,18]$ and $c_0 \sim U[0,10]$.

Chapter 5

ARCH-COMP'22 UPPAAL Case Study

5.1 Aim and setup

We implement the four Minimal Examples from the ARCH-COMP 22 [ABD+22] Stochastic Models category in UPPAAL and compare outcomes. We model networks of stochastic timed automata. Unbounded stays in a location use an exponential delay set by the location rate. Bounded stays are interpreted as uniform on the admissible interval. Components communicate via broadcast channels. We explain the resulting race between components later when we present the automata.

For each property we estimate the probability by Monte Carlo with a fixed sample size N using the E operator in the form `E[<=T; N] (max: (Goal ? 1 : 0))`. The inner expression is a run indicator that returns 1 if the event happened at some time up to T and 0 otherwise. The E operator averages this indicator over N independent runs which yields the estimate $\hat{p}$.

For background on Bernoulli estimators and confidence intervals we follow [CK14]. We estimate a reachability probability by running N independent simulations and recording $Y_i \in \{0,1\}$ with $Y_i = 1$ if the goal is reached. The estimator is $\hat{p} = \frac{1}{N}\sum_{i=1}^{N} Y_i$. Each Y_i has mean p and variance $p(1-p)$, hence $\mathrm{Var}(\hat{p}) = p(1-p)/N$. For large N the central limit theorem says that $\hat{p}$ is approximately normal with mean p and standard deviation $\sqrt{p(1-p)/N}$. About 95% of a standard normal lies between -1.96 and 1.96, so p lies near $\hat{p} \pm 1.96\sqrt{p(1-p)/N}$. Replacing the unknown p by $\hat{p}$ in the standard deviation gives the simple 95% band $\hat{p} \pm 1.96\sqrt{\hat{p}(1-\hat{p})/N}$. This is the Wald interval and works well when N is large and $\hat{p}$ is not extremely close to 0 or 1.

We report $\hat{p}$ together with a simple 95% confidence band $\hat{p} \pm 1.96\sqrt{\hat{p}(1-\hat{p})/N}$. This plus minus band describes the random variation one should expect when repeating the experiment and it shrinks roughly like $1/\sqrt{N}$. If a target half width ε at 95% confidence is desired we choose $N \approx p(1-p)\left(\frac{1.96}{\varepsilon}\right)^2$. When p is unknown we either use a small pilot to plug in $p \approx \hat{p}$ or the conservative choice $p = 0.5$.

5.2 Case A: two exponential races

5.2.1 Exponential baseline

We target the property $\phi = F^{\leq 10}(x \leq -1)$. The system component has one initial location ℓ_0 with flow $\dot{x} = 2$. On $b1?$ it moves to ℓ_1 with $\dot{x} = 0$. On $b2?$ it moves to ℓ_2 with $\dot{x} = -3$. Two event components run in parallel. X_1 waits in a location with exponential rate $\lambda_1 = 0.1$ and emits $b1!$. X_2 waits with exponential rate $\lambda_2 = 0.08$ and emits $b2!$. Channels are broadcast so the system component receives the winning event and the losing timer resets. Sink locations after the events are non-blocking and use a tiny rate so time can progress, see Figure 5.1.

If $b1!$ wins then x stops at a non-negative value so ϕ cannot hold. If $b2!$ wins at time t then x equals $2t$ at that moment and then decreases with rate -3. The condition $x(10) \leq -1$ is equivalent to $2t - 3(10 - t) \leq -1$, hence $t \leq T^\star = 29/5 = 5.8$. With independent exponentials the winner is X_2 with probability $\lambda_2/(\lambda_1 + \lambda_2)$ and the first event time is exponential with rate $\lambda_1 + \lambda_2$. Therefore

$$\Pr(\phi) = \frac{\lambda_2}{\lambda_1 + \lambda_2}\left(1 - e^{-(\lambda_1+\lambda_2)\cdot 5.8}\right) \approx 0.288 \quad (\lambda_1 = 0.1,\ \lambda_2 = 0.08).$$

We estimate $\Pr(\phi)$ with a fixed N Monte Carlo run indicator using `E[<=10; N](max: (x <= -1))`. With $N = 800000$ we obtain $\hat{p} = 0.288719$ with a 95% confidence band $\hat{p} \pm 0.000993$. This is consistent with the RealySt value 0.288236 [ABD+22] whose stated statistical error is $4.651 \cdot 10^{-4}$.

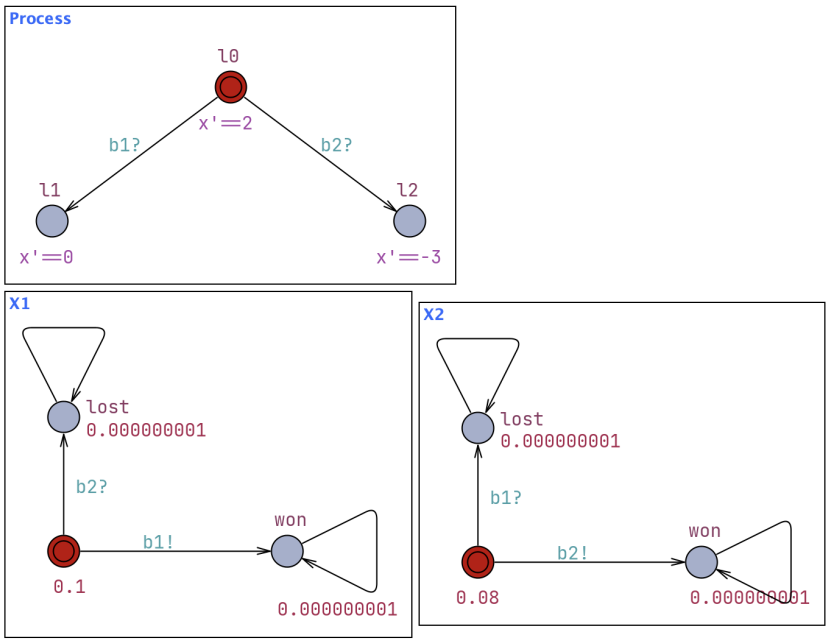

Figure 5.1: UPPAAL model for Case A. The system component on the top left races between two event components X_1 and X_2. X_1 emits $b1!$ after an exponential delay with rate 0.1 and X_2 emits $b2!$ with rate 0.08. Broadcast channels deliver the winning event to the system component and the losing timer moves to a sink location.

5.2.2 Normal delay for X_2

We keep $X_1 \sim \mathrm{Exp}(0.1)$ and change X_2 to sample a real $r \sim N(5,2)$ then set $n = |r|$ as a non-negative countdown. The dynamics in the race location are $n' = -1$ with invariant $n \geq 0$ and X_2 fires $b2!$ when $n \leq 0$. The corresponding UPPAAL model is shown in Figure 5.2. The property remains $\phi = F^{\leq 10}(x \leq -1)$. Conditioning on the sampled delay n of X_2 gives

$$\Pr(\phi) = \int_0^{5.8} f_{X_2}(n)\, e^{-0.1\,n}\, \mathrm{d}n \approx 0.448,$$

where f_{X_2} is the density of the chosen non-negative normal variant. For the folded normal with mean μ and standard deviation σ this density on $n \geq 0$ is

$$f_{\text{fold}}(n) = \frac{1}{\sigma}\varphi\!\left(\tfrac{n-\mu}{\sigma}\right) + \frac{1}{\sigma}\varphi\!\left(\tfrac{n+\mu}{\sigma}\right)$$

with φ the standard normal pdf, and in our model $\mu = 5$ and $\sigma^2 = 2$ which matches `random_normal(5,2)` in Figure 5.2. We evaluate the property with the same queries `E[<=10; N] (max: (x <= -1))` and `Pr[<=10] (<> x <= -1)`. For a target half width $\varepsilon = 10^{-3}$ at 95% confidence and p around 0.45 we use $N \approx p(1-p)(1.96/\varepsilon)^2 \approx 9.5 \cdot 10^5$ and round to $N = 1000000$. We obtain $\hat{p} = 0.44869$ with a 95% confidence band $\hat{p} \pm 0.000975$. RealySt reports 0.448211 with statistical error $8.292 \cdot 10^{-5}$ [ABD+22]. Both match our expected magnitude.

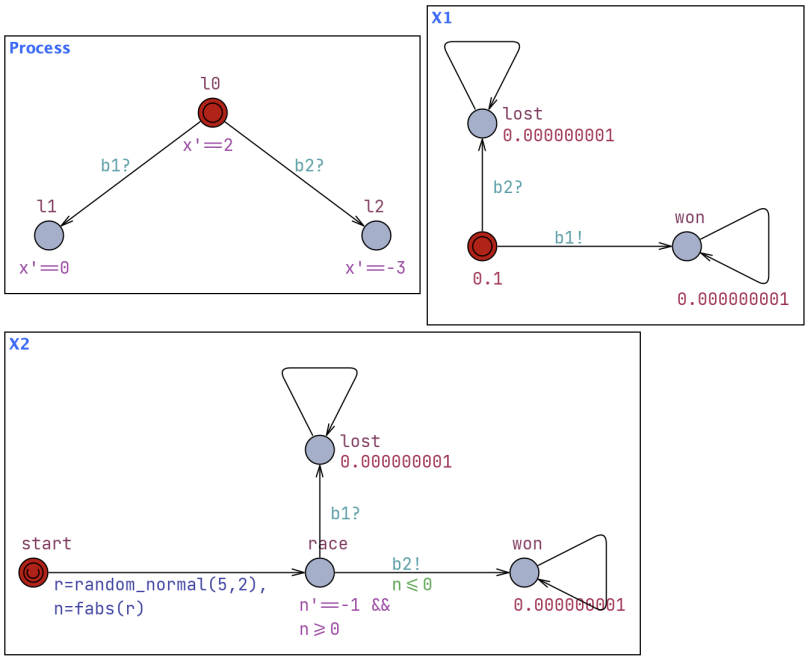

Figure 5.2: Normal variant of Case A. X_1 keeps an exponential delay with rate 0.1. X_2 samples $r \sim N(5,2)$ then sets $n = |r|$ and counts down with $n' = -1$ firing $b2!$ when $n \leq 0$. In the screenshot `random_normal(5,2)` is used which corresponds to $\sigma = 2$.

5.3 Case B: urgent split after a deterministic delay

5.3.1 Exponential baseline

We target $\phi = F^{\leq 10}(x \leq -1)$ and $\phi' = F^{\leq 12}(x \leq -1)$. The system component starts in ℓ_0 with $\dot{x} = 2$ and moves on $b1?$ to ℓ_1 with $\dot{x} = 0$ or on $b2?$ to ℓ_2 where x is frozen and an auxiliary variable y evolves with $\dot{y} = 1$ under the invariant $y \leq 2$. At $y = 2$ two urgent edges lead to ℓ_3 with $\dot{x} = 0$ and to ℓ_4 with $\dot{x} = -3$. We resolve this discrete conflict probabilistically with weights 1 and 1. The event components X_1 and X_2 keep exponential rates $\lambda_1 = 0.1$ and $\lambda_2 = 0.08$. See Figure 5.3.

If $b1!$ wins then x cannot become negative. If $b2!$ wins at time s the system stays for exactly 2 time units in ℓ_2 and then goes with probability $1/2$ to ℓ_4. For ϕ we need $2s - 3(10 - s - 2) \leq -1$ which gives $s \leq 23/5 = 4.6$. For ϕ' the bound is $s \leq 29/5 = 5.8$. With independent exponentials the winner is X_2 with probability $\lambda_2/(\lambda_1 + \lambda_2)$ and the first event time is exponential with rate $\lambda_1 + \lambda_2$. Hence

$$\Pr(\phi) = \tfrac{1}{2} \cdot \frac{\lambda_2}{\lambda_1 + \lambda_2} \left(1 - e^{-(\lambda_1+\lambda_2)\cdot 4.6}\right) \approx 0.125,$$

$$\Pr(\phi') = \tfrac{1}{2} \cdot \frac{\lambda_2}{\lambda_1 + \lambda_2} \left(1 - e^{-(\lambda_1+\lambda_2)\cdot 5.8}\right) \approx 0.144.$$

We evaluate both properties with `E[<=T; N] (max: (x <= -1))`. With $N = 500000$ we obtain for ϕ the estimate 0.125192 ± 0.000917 and for ϕ' the estimate 0.144624 ± 0.000975 at 95% confidence. The ARCH'22 table lists RealySt under maximizing semantics for the discrete conflict, namely 0.250016 for ϕ and 0.288236 for ϕ' [ABD+22], so our probabilistic split is roughly half of those values as expected. It also aligns with the probabilistic SMC rows of other tools in the same table (e.g., HYPEG and modes) [ABD+22].

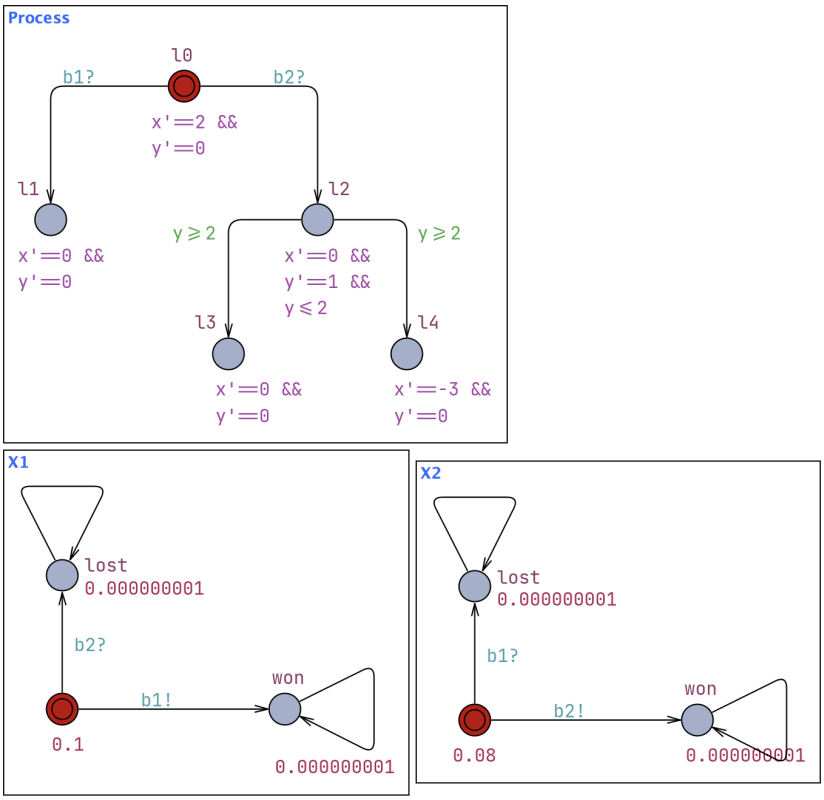

Figure 5.3: UPPAAL model for Case B. After $b2?$ the system waits exactly 2 time units in ℓ_2 via $\dot{y} = 1$ and $y \leq 2$. At $y = 2$ two urgent edges to ℓ_3 and ℓ_4 are resolved probabilistically with weights 1 and 1.

5.3.2 Normal delay for X_2

We keep the system component of Case B and only change the trigger for $b2!$: $X_1 \sim \text{Exp}(0.1)$ as before, while X_2 samples a single delay from the folded normal $|N(5,2)|$ and fires $b2!$ when a local clock reaches that sample. After $b2?$ the system waits exactly 2 time units in ℓ_2 via $\dot{y} = 1$ and $y \leq 2$, and at $y = 2$ the two urgent edges to ℓ_3 and ℓ_4 are both enabled. Without weights, UPPAAL resolves this discrete conflict uniformly at random. See Figure 5.4.

Conditioning on the sampled delay y of X_2 and on the exponential competitor gives

$$\Pr(\phi) = \tfrac{1}{2} \int_0^{4.6} f_{X_2}(y)\, e^{-0.1\,y}\, \mathrm{d}y \approx 0.154,$$

$$\Pr(\phi') = \tfrac{1}{2} \int_0^{5.8} f_{X_2}(y)\, e^{-0.1\,y}\, \mathrm{d}y \approx 0.224,$$

where f_{X_2} is the density of the folded normal $|N(5,2)|$ on $y \geq 0$.

We evaluate both properties with `E[<=T; N] (max: (x <= -1))`. With $N = 700000$ we obtain 0.154144 ± 0.000846 for ϕ and 0.224051 ± 0.000977 for ϕ' at 95% confidence. These values are consistent with the probabilistic SMC rows

of other tools in the ARCH'22 table (e.g., HYPEG and modes) [ABD+22] and, as expected, differ from the RealySt "max" row which applies maximizing semantics to the discrete conflict.

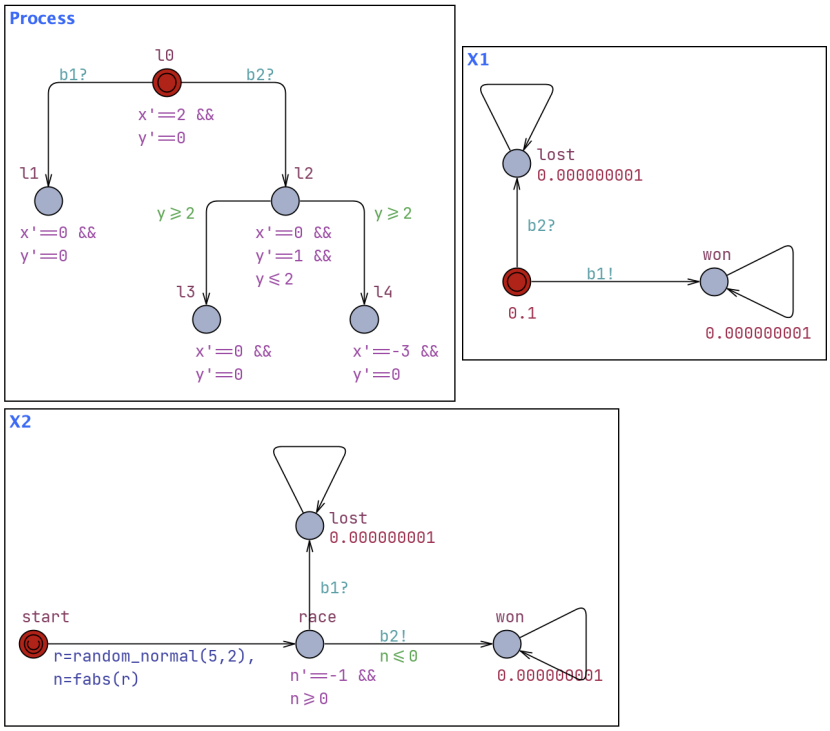

Figure 5.4: Normal variant of Case B. X_1 has an exponential delay with rate 0.1. X_2 samples one delay from the folded normal $|N(5,2)|$ and emits $b2!$ when a local clock reaches that value. After $b2?$ the system waits exactly 2 time units in ℓ_2. At $y = 2$ the two urgent edges are resolved uniformly at random.

5.4 Case C: stochastic flow noise

Case C adds Gaussian noise to the flow of y in ℓ_2, written $\dot{y} = 1+n_1$ with $n_1 \sim N(0,2)$. UPPAAL does not support stochastic noise in flows, so we approximate it by fixed ticks of size $\Delta t = 0.01$ and resample the slope on each tick. The ARCH'22 report likewise notes that no participating tool supports such noise and therefore lists no official results for Case C [ABD+22].

5.4.1 Exponential baseline

We keep $X_1 \sim \text{Exp}(0.1)$ and $X_2 \sim \text{Exp}(0.08)$. In ℓ_2 a local timer c produces ticks. At each tick we draw $d_c \sim N(0,2)$ and use $\dot{y} = 1 + d_c$ until the next tick. Once $y \geq 2$ the urgent split is taken and the tie is resolved uniformly at random.

We estimate $\Pr(\phi)$ and $\Pr(\phi')$ with fixed N Monte Carlo using
`E[<=10; 1000000] (max: (x <= -1))` and `E[<=12; 1000000]`

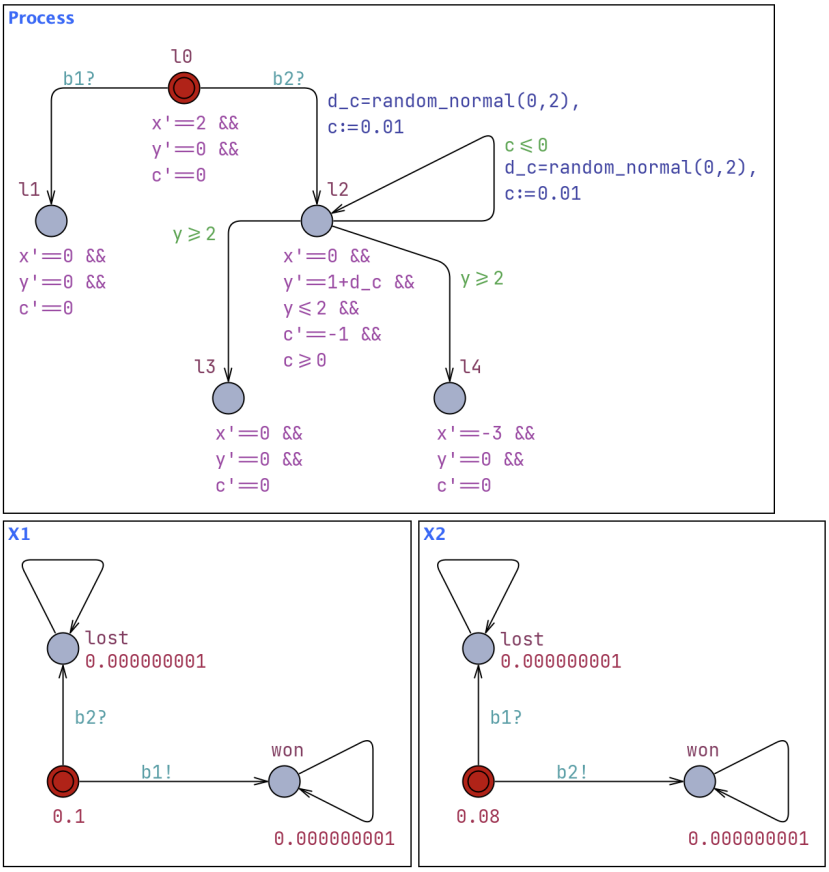

Figure 5.5: Case C with stepwise noise in ℓ_2. A timer c enforces a fixed step of $\Delta t = 0.01$. When $y \geq 2$ the urgent split fires with uniform resolution.

`(max: (x <= -1))`. For this exponential baseline we obtain $\hat{p}_\phi = 0.125755 \pm 0.000649$ and $\hat{p}_{\phi'} = 0.143548 \pm 0.000687$ at 95% confidence.

5.4.2 Normal delay for X_2

We keep $X_1 \sim \text{Exp}(0.1)$ and draw $r \sim N(5,2)$ for X_2. We enforce a non-negative delay by setting $n = |r|$ and realize it by a countdown with $n' == -1$ until $n \leq 0$, at which point $b2!$ fires. The rest of the system, including the ticked noise in ℓ_2, remain unchanged.

We estimate $\Pr(\phi)$ and $\Pr(\phi')$ with the same fixed N Monte Carlo as in Sec. 5.4.1 using `E[<=10; 1000000] (max: (x <= -1))` and `E[<=12; 1000000] (max: (x <= -1))`. This yields $\hat{p}_\phi = 0.154234 \pm 0.000708$ and $\hat{p}_{\phi'} = 0.224111 \pm 0.000817$ at 95% confidence.

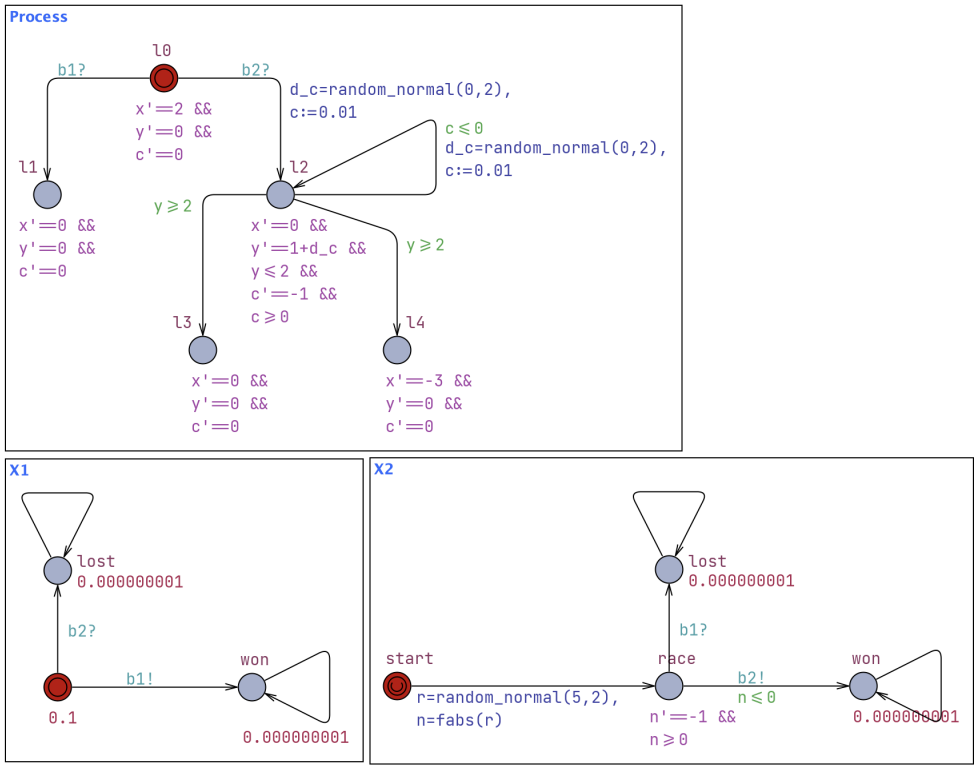

Figure 5.6: Case C with a normal race variable for X_2. We sample $r \sim N(5,2)$, set $n = |r|$, and trigger $b2!$ once $n \leq 0$.

5.5 Case D: timelock through an invariant

We target $\phi = F^{\leq 10}(x \leq -1)$. The system component starts in ℓ_0 with $\dot{x} = 2$ and invariant $x \leq 6$. On $b1?$ it moves to ℓ_1 with $\dot{x} = 0$. On $b2?$ it moves to ℓ_2 with $\dot{x} = -3$. UPPAAL stops analysis when a timelock is reachable which here would happen if no broadcast occurs before x reaches 6 in ℓ_0. We therefore add an edge from ℓ_0 to a sink location `TimeLock` guarded by $x \geq 6$. Together with the invariant $x \leq 6$ this forces a jump at the boundary so a timelock cannot occur. Sinks are non-blocking and use a tiny exponential rate so time can progress. Broadcast channels deliver the winning event to the system component and the losing timer moves to a sink.

5.5.1 Exponential baseline

We keep $X_1 \sim \text{Exp}(0.1)$ and $X_2 \sim \text{Exp}(0.08)$. The property holds iff $b2!$ occurs before $b1!$ and before $t = 3$ since at $t = 3$ the invariant forces the jump to `TimeLock`. With independent exponentials the first event time is exponential with rate $\lambda_1 + \lambda_2$ and the winner is X_2 with probability $\lambda_2/(\lambda_1 + \lambda_2)$. Hence (with $\lambda_1 = 0.1$ and $\lambda_2 = 0.08$):

$$\Pr(\phi) = \int_0^3 \lambda_2 e^{-(\lambda_1+\lambda_2)t}\,\mathrm{d}t = \frac{\lambda_2}{\lambda_1+\lambda_2}\left(1 - e^{-(\lambda_1+\lambda_2)\cdot 3}\right) \approx 0.185$$

We estimate $\Pr(\phi)$ with a run indicator using `E[<=10; 1000000] (max: (x <= -1))`. For $N = 1000000$ we obtain $\hat{p} = 0.185505 \pm 0.000762$ at 95% confidence.

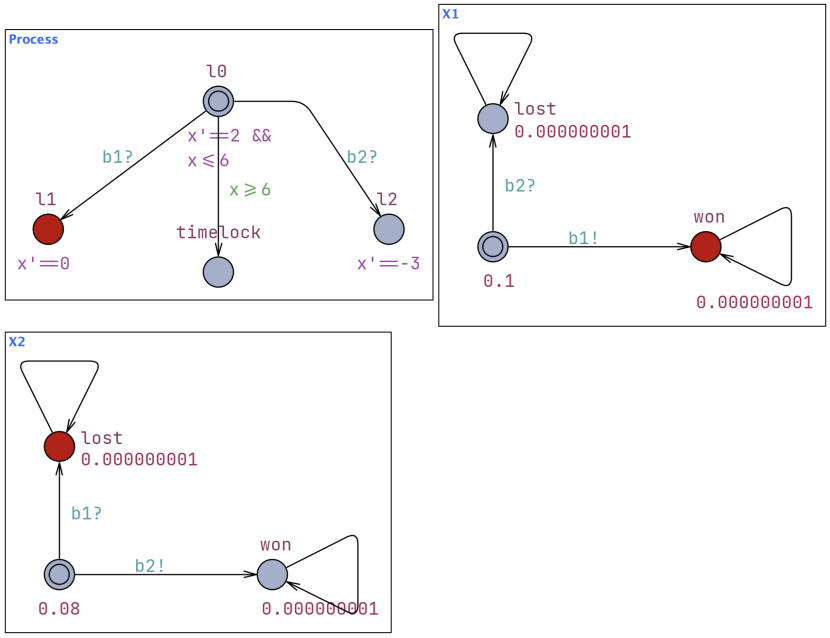

Figure 5.7: UPPAAL model for Case D. The system has $\dot{x} = 2$ with $x \leq 6$ in ℓ_0 and an urgent edge to `TimeLock` when $x \geq 6$. X_1 and X_2 emit $b1!$ and $b2!$ after exponential delays with rates 0.1 and 0.08. Broadcast channels deliver the winning event.

This aligns with the RealySt value 0.185280 reported in ARCH'22 [ABD+22], which is within our 95% band and differs by 2.25×10^{-4}.

5.5.2 Normal delay for X_2

We keep $X_1 \sim \text{Exp}(0.1)$ and set X_2 to a single sampled delay from the folded normal $|N(5,2)|$. On entry X_2 draws $r \sim N(5,2)$, sets $y_2 = |r|$, and emits $b2!$ when a local countdown reaches zero. The system component remains as in Section 5.5.1 with invariant $x \leq 6$ and an urgent escape to `TimeLock` when $x \geq 6$.

We evaluate $\phi = F^{\leq 10}(x \leq -1)$ with `E[<=10; 1000000] (max: (x <= -1))`. For $N = 1000000$ we obtain $\hat{p} = 0.130076 \pm 0.000659$ at 95% confidence. This matches the ARCH'22 normal table, e.g., RealySt 0.130280 and modes SMC 0.130161, both within our 95% band.

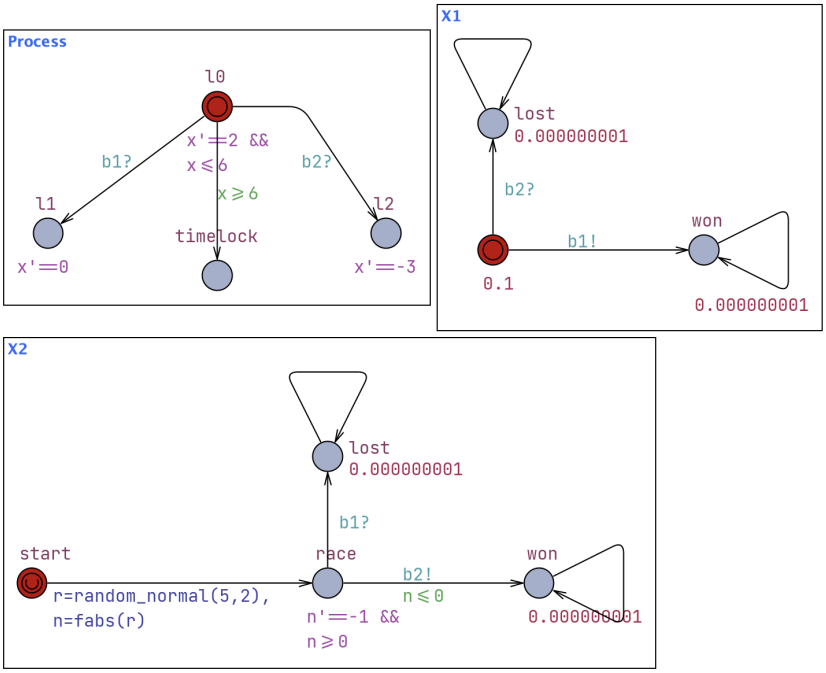

Figure 5.8: Normal variant of Case D. X_2 draws a single delay from $|N(5,2)|$. The escape at $x \geq 6$ avoids timelocks during simulation.

5.6 Results

Across all cases UPPAAL SMC reproduces the SMC rows in ARCH'22 where they exist and agrees with RealySt when the semantics matches. In Case A and D our estimate matches the RealySt value. In Case B our probabilities are about half of the RealySt max line because we resolve the discrete split uniformly rather than maximizing, and they align with the probabilistic SMC entries. In Case C there are no official ARCH'22 results and our values come from the ticked noise approximation.

Chapter 6

UPPAAL vs. RealySt: CAMELS Expressivity

In this chapter we want to define as to what extent and how CAMELS semantics are representable in UPPAAL (SMC and stratego) and in RealySt. We summarize the criteria based on the make-up of CAMELS automata:

- scheduling granularity (composed vs. decomposed)
- realizability handling (lazy vs. eager (non-)predictive)
- explicit enabledness timers (random clocks)
- treatment of non-determinism

6.1 UPPAAL

We analyse UPPAAL under its SMC semantics on a single automaton. Our focus is the stochastic timing behavior captured by CAMELS, not controller synthesis, so stratego is out of scope. In SMC there is no adversarial non-determinism, so time delays and branch choices are sampled from the given distributions or weights. Within this scope we discuss CL, CEP, CENP, DL, and DENP. Decomposed Eager Predictive is excluded as it is not well defined.
As a modeling convention we use countdown clocks for random clocks. When we sample d, we set the clock $c = d$ and let $c' == -1$; the jump fires when $c \leq 0$. This avoids equality tests against real-valued random samples and is robust under SMC's numerical integration. To rule out time slip at expiry we add the invariant $c \geq 0$ in locations with enabled random clocks and guard the firing (or resampling) edges by $c \leq 0$, forcing an immediate transition at expiry. In UPPAAL we encode boundaries with $\leq$, $\geq$ (and a committed splitter at $c \leq 0$ if needed) so each boundary belongs to exactly one edge.

In all UPPAAL experiments we evaluate the time-bounded reachability of the location "free" from the Sisyphus model `Process.l1` with one million simulations for two horizons. We use the following SMC queries:

```
1 E[<=30; 1000000] (max: (Process.l1 ? 1 : 0))
2 E[<=100; 1000000] (max: (Process.l1 ? 1 : 0))
```

We report the estimated value and the confidence interval returned by the verifier.

6.1.1 Decomposed eager non-predictive

We model the DENP automaton from Figure 2.3 and implement it in UPPAAL as shown in Figure 6.1. A single automaton encodes DENP by giving each stochastic label a_i its own stopwatch c_i and a sampled delay d_i. When the guard g_i becomes true for the first time in a cycle we draw d_i, set $c_i = d_i$, and let the clock evolve with $c_i' == -1$ while g_i holds and $c_i' == 0$ otherwise. The jump a_i fires when g_i is true and $c_i \leq 0$. After firing, the label is re-armed and the next sample is drawn at the next enabled event. If two labels are enabled at the same time they race and the one that reaches its sampled duration first fires. Ties have probability zero.

UPPAAL fixes each flow per location. You cannot write a conditional rate such as $c_i' == -1$ when $10 \leq x \leq 20$ or $20 \leq x \leq 30$ and $c_i' == 0$ otherwise inside one location. To realise DENP we therefore refine a location with random jumps into several windows. Each window is a separate location in which the enabledness of every random guard is constant, so for each label a_i the guard g_i is always true or always false while the automaton stays there.

For the automaton in Figure 2.3 the cut points are $x = \{10,20,24,26\}$. Thus we refine ℓ_0 into the windows [0,10], [10,20], [20,24], [24,26], and [26,30], as shown in Figure 6.1. In each window we implement the countdown style for every stochastic label a_i:

- when a_i becomes enabled at a window boundary, sample d_i, set $c_i = d_i$, and use $c_i' == -1$ while g_i holds in that window;
- when a_i is disabled in a window, use $c_i' == 0$ so the remaining time is preserved and resumes in the next window where a_i is enabled;
- a_i fires from any window where g_i holds with guard $c_i \leq 0$; after firing we re-arm a_i and will draw a fresh d_i at the next enable event.

If a guard is a union of intervals, the refinement creates the corresponding on/off windows (for c_1 this yields [10,20] and [20,24] on, [24,26] off, and [26,30] on). We avoid time slip at expiry by a committed pre-state or an invariant $c_i \geq 0$ in the counting windows.

In summary, DENP in UPPAAL requires precomputing the enable/disable cut points and generating one sub-location per window. This can be done manually by the modeller or by a translator that derives the windows and wires the stopwatches automatically.

This construction is decomposed because timers are per label. It is eager non-predictive because delays are interpreted as enabledness durations and we do not precompute enabling times. Enabledness timers are explicit random clocks c_i. Under SMC the adversarial notion of non-determinism is not present. It is replaced by probabilistic races governed by the sampled delays and the given weights.

UPPAAL SMC reported for $t \leq 30$ the estimate $0.222 \pm 8.14545 \times 10^{-4}$ with 95% confidence and for $t \leq 100$ the estimate $0.617393 \pm 9.5259 \times 10^{-4}$ with 95%

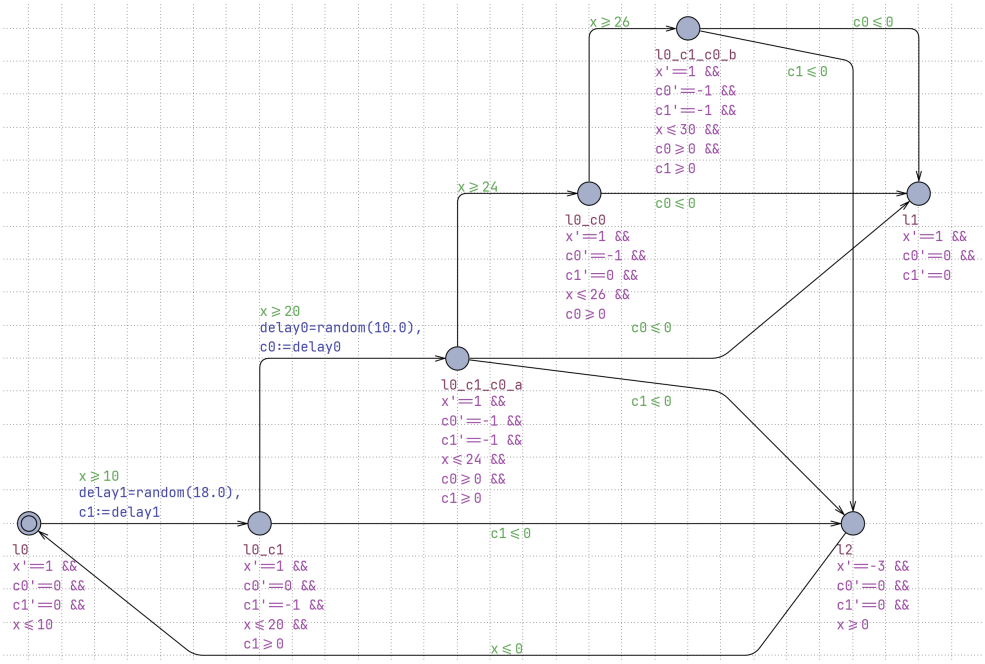

Figure 6.1: UPPAAL SMC model for the DENP Sisyphus automaton (Figure 2.3). Each label has its stopwatch c_i and sampled duration d_i. Clocks run only while enabled and fire when $c_i \leq 0$.

confidence. RealySt results for the same setup are reported in Section 6.2 and both tools are compared in Section 6.3.

6.1.2 Decomposed lazy

We take the Sisyphus automaton of Figure 2.3 and implement the DL version of scheduling in UPPAAL by modifying the DENP template. Each label a_i keeps a countdown clock c_i and sampled delay d_i, but now the clock runs with $c_i' == -1$ in the location that contains the random jump. There is no pausing outside enablement. We keep the invariant $c_i \geq 0$ and guard all expiry-triggered edges by $c_i \leq 0$, so that expiry is urgent. At expiry we branch between firing and resampling: a_i fires iff its guard g_i holds with $c_i \leq 0$; otherwise we take a resampling edge that redraws d_i. Because UPPAAL guards do not allow disjunctions, we materialize $\neg g_i$ as separate windows and add one resampling self-loop per maximal interval of the complement of the enablement set. Concretely in ℓ_0 (see Figure 6.2): $g_1 \equiv (10 \leq x \leq 24) \vee (26 \leq x \leq 30)$ and $g_0 \equiv (x \geq 20)$, hence $\neg g_1 : [0{,}10) \cup (24{,}26)$ and $\neg g_0 : [0{,}20)$. This is implemented by three self-loops in the model: the long loop labeled "resample before gap" with guard $x < 10 \wedge c_1 \leq 0$ that executes `delay1=random(18.0), c1:=delay1`, the small loop labeled "resample c_1 during gap" with guard $24 < x \wedge x < 26 \wedge c_1 \leq 0$ and the same update, and the short loop labeled "resample c_0" with guard $x < 20 \wedge c_0 \leq 0$ that executes `delay0=random(10.0), c0:=delay0`. The extra conjunct $x < 20$ on the c_0-resampler is crucial. Resampling must only be used when the sampled deadline expired while the action was disabled, which for a_0 can happen only before

$x < 20$ (since $g_0 \equiv x \geq 20$). With these split guards and explicit resampling loops, at each expiry exactly one of "fire" or "resample" is enabled per label, so the UPPAAL model in Figure 6.2 realizes DL semantics for the Sisyphus example.

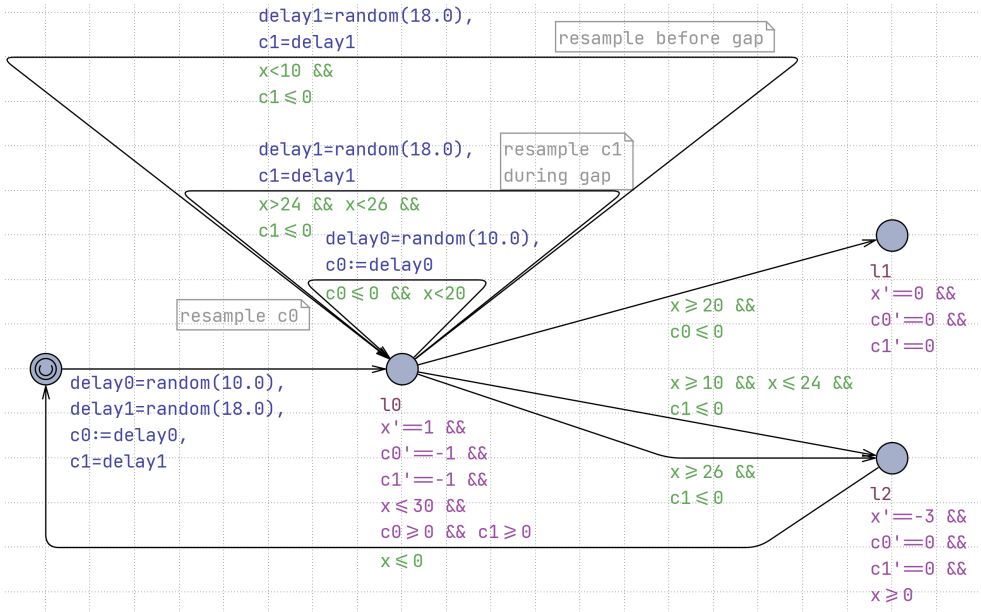

Figure 6.2: UPPAAL SMC model for the DL version of the Sisyphus automaton (Figure 2.3). For every gap in the complement of the enablement set we have to create its own resampling selfloop.

In summary, our DL encoding in UPPAAL keeps one automaton with one countdown clock c_i and sampled delay d_i per stochastic label a_i, but, unlike DENP, the clocks never pause. In the location with stochastic jumps we set $c_i' == -1$ unconditionally. Expiry is made urgent by the invariant $c_i \geq 0$ and by guarding all expiry-triggered edges with $c_i \leq 0$, which prevents time slip at $c_i = 0$. At each expiry we choose between firing a_i and resampling by compiling the complement of the enablement guard g_i into its "gaps" and adding one resampling self-loop per gap, guarded by $(\text{gap}) \wedge (c_i \leq 0)$. The firing edge is guarded by $g_i \wedge (c_i \leq 0)$. Because gap guards and g_i partition the state space at expiry, exactly one of "fire" or "resample" is enabled for each label when c_i expires. When g_i is a union of intervals, this yields one self-loop per gap of $\neg g_i$ (not per enabling window), so the model remains a single location with several self-loops rather than a window-refined template. This template is decomposed in scheduling granularity (timers are per label within one automaton), lazy in realizability handling (samples represent global delays and no enabling times are precomputed), and it uses explicit random clocks that are global countdowns rather than enabledness stopwatches (there is no pausing outside enablement). With the guard-splitting at expiry, the local branch is deterministic. Under SMC, the remaining concurrency across labels is resolved by probabilistic races between independently sampled delays, and simultaneous expiries have probability zero.

UPPAAL SMC reported for $t \leq 30$ the estimate $0.07293 \pm 5.09634 \times 10^{-4}$ with 95% confidence and for $t \leq 100$ the estimate $0.281954 \pm 8.81889 \times 10^{-4}$ with 95%

confidence.

6.1.3 Composed lazy

Starting from Figure 2.3 we switch to composed scheduling by replacing the per-label stopwatches c_0,c_1 with one global countdown c. On (re-)entry to ℓ_0 we draw $c = \texttt{random}(15.0)$ and two weights w_0,$w_1 = \texttt{random}(1.0)$ that parameterize the discrete choice at expiry. UPPAAL probability labels use nonnegative weights and select an enabled edge with probability proportional to its weight, drawing w_0,w_1 uniformly gives an unbiased choice when both edges are enabled and fixes that choice for the current countdown. These weights belong to the composed variants and do not correspond to the decomposed models where the winner is decided by the race of independent delays rather than a probability label. We keep $c' == -1$ in ℓ_0, enforce $c \geq 0$, and guard all expiry-triggered edges by $c \leq 0$ to make expiry urgent. We adjust the slip guard and keep both fire edges enabled on overlaps $g_{\text{top}} := x \geq 20$ and $g_{\text{slip}} := (15 \leq x \leq 24) \vee (x \geq 26)$. At $c = 0$ UPPAAL samples between the enabled fire edges using the weights w_0 for $l0$ and w_1 for $l1$. If c expires where no label is enabled (here $x < 15$) a self-loop guarded by $(x < 15) \wedge (c \leq 0)$ resamples c,w_0,w_1 and returns to ℓ_0. We redraw w_0,w_1 together with c only at initialization and on that resampling loop. The resulting template (shown in Figure 6.3) uses one location with three fire edges and one resampling loop. There is no window refinement.

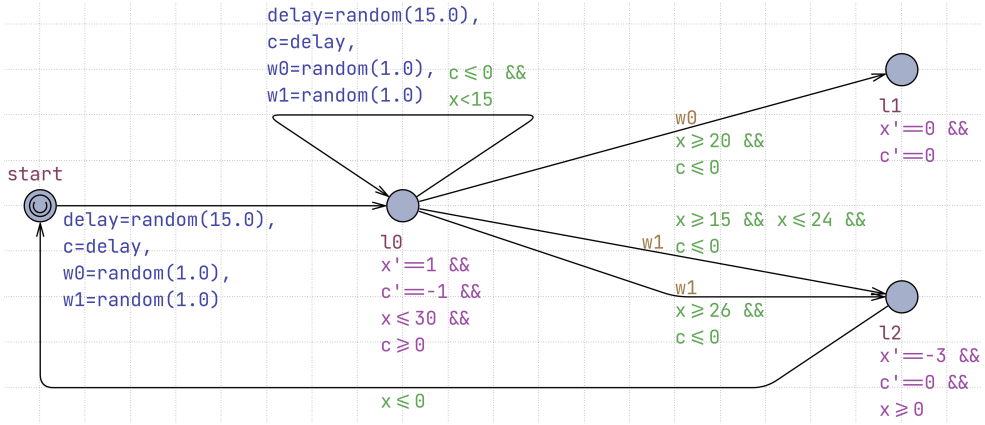

Figure 6.3: UPPAAL model for the CL version of Sisyphus. A single global countdown c schedules the next expiry for the random jumps. At expiry ($c \leq 0$) the enabled fire edges are chosen probabilistically via weights w_0 (top) and w_1 (slip). If $x < 15$ at expiry, the resampling loop redraws c,w_0,w_1 and returns to ℓ_0.

In summary, the UPPAAL encoding of CL is composed in scheduling granularity (a single global clock c), lazy and non-predictive in realizability handling. If the sampled expiry falls in a gap we resample, so no enabling times are precomputed and we use an explicit random clock c rather than enabledness stopwatches. Local non-determinism at overlaps is resolved stochastically by the probability labels w_0,w_1. When only one label is enabled the choice is deterministic. Timing and branching are governed by the sampled delay and the weights. Finally, whenever several random jumps leave the same location and their enabledness can overlap, we draw choice

variables for those labels (e.g. weights w_i) at the expiry or upon resampling and use them in probability labels on the overlapping fire edges. This yields a single stochastic branch on intersections, while the gap case is handled by the resampling loop.

UPPAAL SMC reported for $t \leq 30$ the estimate $0.296113 \pm 8.94806 \times 10^{-4}$ with 95% confidence and for $t \leq 100$ the estimate $0.709751 \pm 8.89584 \times 10^{-4}$ with 95% confidence.

6.1.4 Composed eager predictive

Starting from the CL template in Figure 6.3 we move to CEP. To avoid resampling, instead of drawing a global delay c that may fall into a gap, we sample c only from times at which at least one random jump will be enabled. The structure (one location with three fire edges and the same guards) stays, but the sampling update becomes predictive and the resampling loop disappears. UPPAAL does not precompute this predictive set. The modeller (or a translator) encodes the draw manually. Concretely we keep $c' == -1$ in ℓ_0, the invariant $c \geq 0$, and the fire edges guarded by $g_{\text{top}} := x \geq 20$ and $g_{\text{slip}} := (10 \leq x \leq 14) \vee (x \geq 16)$. On overlaps ($x \geq 20$) both fire edges remain enabled and we keep probability labels (e.g. w_0 for top, w_1 for slip). The change is in the update that draws c. In our Sisyphus we always re-enter ℓ_0 at $x = 0$, so valid expiries are exactly the union [10,14]∪[16,30]. We therefore sample uniformly on this union by a single update.

Listing 6.1: CEP: predictive draw for fixed entry x=0

```
/* Windows: [10,14] U [16,30] (L1=4, L2=14).
On entry to l0: sample c on that union */
delay = random(18.0),                    //4 + 14
c = (delay < 4.0)
      ? (10.0+delay)                     //uniform in [10,14]
      : (16.0+(delay-4.0))               //uniform in [16,30]
```

If entry could have happened at arbitrary x, one can compute the two windows on the fly and still draw in one update, for a horizon $x \leq H$:

Listing 6.2: CEP: predictive draw from arbitrary x with horizon x <= H.

```
/* Compute windows from current x, sample c on union */
double a1 = (10.0-x > 0 ? 10.0-x : 0.0), //max(0,10-x)
double b1 = (14.0-x > 0 ? 14.0-x : 0.0), //max(0,14-x)
double a2 = (16.0-x > 0 ? 16.0-x : 0.0), //max(0,16-x)
double b2 = H-x,                         //from invariant x <= H

double L1 = (b1-a1 > 0 ? b1-a1 : 0.0),
double L2 = (b2-a2 > 0 ? b2-a2 : 0.0),

double u  = random(L1+L2),   //pick interval by length
c = (u < L1) ? (a1+u) : (a2+(u-L1)) //place in window
```

Keep expiry urgent by $c \geq 0$ in ℓ_0 and $c \leq 0$ on firing edges.

In summary, the UPPAAL encoding of CEP is composed in scheduling granularity (a single global clock c), eager and predictive in realizability handling (we sample only times at which some label is enabled, so no resampling and no deadlocks at expiry),

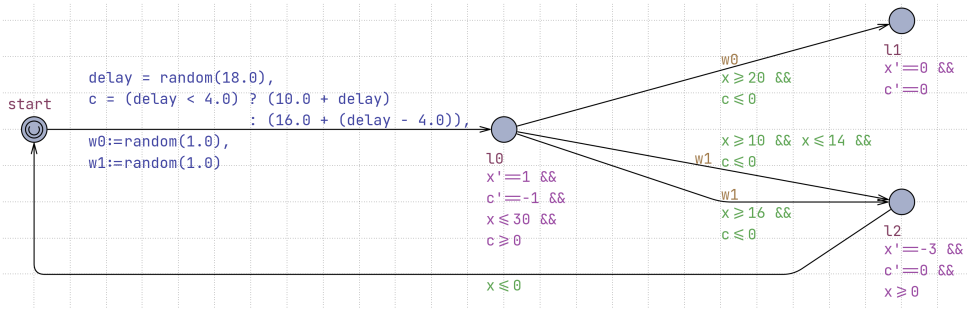

Figure 6.4: UPPAAL model for the CEP version of Sisyphus. A single global countdown c is drawn predictively so that $x + c$ lands in an enabled window (here, from $x = 0$: $[10,14] \cup [16,30]$), hence no resampling edge is needed. At expiry ($c \leq 0$) the enabled fire edges are taken. On overlaps ($x \geq 20$) the choice is resolved by probability labels w_0 (top) and w_1 (slip).

and it uses an explicit random clock c rather than enabledness stopwatches. Local non-determinism on overlaps is treated stochastically by the probability labels on the overlapping fire edges. When only one guard holds the choice is deterministic. Timing and branching are governed by the predictive draw for c and the chosen weights on overlaps.

UPPAAL SMC reported for $t \leq 30$ the estimate $0.276903 \pm 8.77023 \times 10^{-4}$ with 95% confidence and for $t \leq 100$ the estimate $0.677297 \pm 9.16305 \times 10^{-4}$ with 95% confidence.

6.1.5 Composed eager non-predictive

Starting from Figure 6.1 we switch to composed scheduling. All stochastic labels a_i share a single stopwatch c for the composed event a. UPPAAL still cannot make a clock rate depend on guards inside one location, so we keep the window refinement of ℓ_0 at the cut points $x = \{10,20,24,26\}$ where the enabledness of g_0 and g_1 is constant. CENP runs c exactly while the union $g := g_0 \vee g_1$ holds. In our Sisyphus this means $c' == -1$ in all windows with $x \geq 10$ and $c' == 0$ before. On the first enable of the union we sample once and arm the clock with `delay = random(18.0), c = delay, w0 = random(1.0), w1 = random(1.0)`. Expiry is urgent via the invariant $c \geq 0$ and guards $c \leq 0$ on all expiry edges.

A key difference to DENP is the treatment of the gap (24,26). In DENP a per label stopwatch pauses when its label is disabled. In CENP the single global clock does not pause in (24,26) because g_0 is already true from $x \geq 20$ and keeps the union g true. The (24,26) window exists only to model enabled jumps. It is a state where only the top label is enabled and the slip label is not yet enabled again. Because g_0 holds from $x \geq 24$, the union $g_0 \vee g_1$ remains true in (24,26). Consequently the single stopwatch c continues running through the gap with $c' == -1$. At $c \leq 0$ the composed event fires. If exactly one label is enabled in the current window the successor is deterministic. If both are enabled in the overlap windows [20,24) and [26,30] we resolve the choice with UPPAAL probability labels that carry weights w_0 and w_1. Figure 6.5 shows the resulting template.

In summary, the UPPAAL encoding of CENP is composed in scheduling granular-

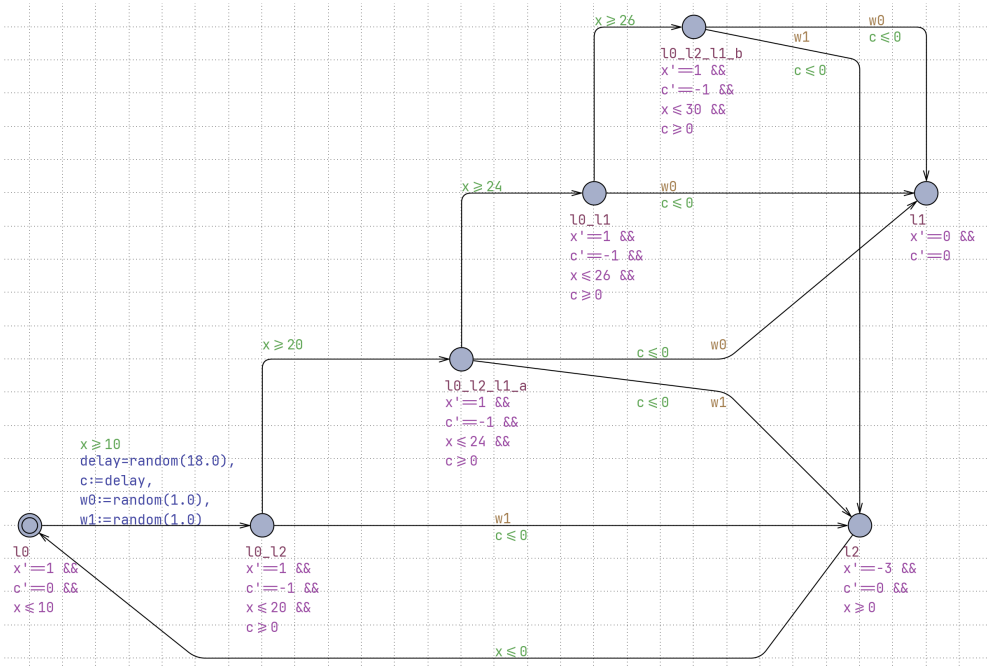

Figure 6.5: UPPAAL SMC model for the CENP Sisyphus automaton. One stopwatch c is sampled when the union $g_0 \vee g_1$ becomes true at $x \geq 10$ and then counts down through all union on windows including the gap (24,26) with $c' == -1$ and invariant $c \geq 0$. On expiry with $c \leq 0$ the enabled jump fires. On overlaps the successor is chosen stochastically via weights w_0 and w_1. The global stopwatch continues through (24,26) because g_0 keeps the union true.

ity with one clock c for the event. It is eager non-predictive in realizability handling since we sample on union enable and do not precompute enabling times. It uses an explicit enabledness timer given by the single countdown c with $c \geq 0$ and $c \leq 0$ at expiry and it continues to run in the (24,26) gap because the union remains true. non-determinism on overlaps is treated by probability labels w_i, and when only one guard holds the choice is deterministic.

UPPAAL SMC reported for $t \leq 30$ the estimate $0.277712 \pm 8.77812 \times 10^{-4}$ with 95% confidence and for $t \leq 100$ the estimate $0.712563 \pm 8.87017 \times 10^{-4}$ with 95% confidence.

6.2 RealySt

RealySt does not target the composed variants (lazy or eager), which assume a single global delay and a probabilistic label choice at expiry. RAC times transitions per label. DL is also not the execution semantics of RealySt. Its timers measure enabledness time, whereas lazy semantics sample absolute delays and cancel them when disabled. Another fundamental obstacle is label resolution. Composed variants require a probabilistic choice at expiry, whereas RealySt resolves such choices non-deterministically (maximizing). CAMELS shows that CL and CEP are expressively equivalent, while

DENP (RAC) and CL are not. Hence a general encoding of composed semantics in RealySt does not preserve probabilities. Two limited cases are useful in practice:

1. CENP can match in timing via a single event clock for all random jumps. However, at expiry RealySt resolves the label non-deterministically and maximizes instead of sampling if more than one random jump is enabled simultaneously.

2. CEP can match if we precompute the set of possible samples for the global random clock. Otherwise RealySt again maximizes instead of sampling.

Consequently both cases would only model faithfully if at expiry exactly one random jump is enabled (e.g. disjoint windows).

In the current implementation of RealySt a random clock or stochastic jump can fire at most once. This implies that loops or iterations cannot reuse a previously armed random clock, which means that only the first pass through a loop will schedule and fire the stochastic jump. As a consequence we cannot encode CENP and CEP natively, since both require a global random clock that is reused across multiple random jumps. In RealySt such a clock would expire once and would not schedule a second random jump on a later overlap.

In [DSSR24] a practical workaround is used. The automaton is unrolled for every intended repetition so that each copy carries its own locations, edges and fresh random clocks. We were able to reproduce this idea at the model level and observed the expected change in the reachability graph. In our experiments the reported probability did not change accordingly. We therefore proceed conservatively and report RealySt results under a fixed time-bound and jump bound, measuring the initial stochastic race in each unrolled copy rather than attempting to reuse a stochastic jump across loop iterations. This reflects the behavior of the implementation we used and does not rule out future extensions that support reusable stochastic timers. We used RealySt from an internal development branch named alpha. The repository is not publicly accessible and access requires approval from the maintainers.

In our evaluation we thus compare DENP in both tools.

For DENP on the Sisyphus model with the goal to reach L1_FREE and with time-bound 30 and jump bound 100 we used the following command.

```
./realyst -t 30 -d 100 -b SISYPHUS -m A
          --plotDimensions 0 1 -l trace
```

The tool reported a reachability probability of 0.2221555005133634 with statistical error $5.267938246430821 \times 10^{-5}$.

6.3 Results

We first summarize which CAMELS variants we could realize in each tool and under which constraints. All references to figures and templates follow the modeling described above.

6.3.1 Implementability UPPAAL

- DENP supported. This requires window refinement of locations at all cut points where any random guard toggles. Each label a_i uses a stopwatch c_i that pauses

outside enablement and runs with $c_i' == -1$ while enabled. Expiry is urgent via invariant $c_i \geq 0$ and guard $c_i \leq 0$. See Figure 6.1.

- DL supported. One countdown clock c_i per label that never pauses. At expiry we branch deterministically between fire with guard $g_i \wedge c_i \leq 0$ and resample via one self loop per gap of $\neg g_i$ guarded by $(\text{gap}) \wedge c_i \leq 0$. Resampling edges must be enabled only if the label was disabled when the deadline expired. For a_0 this means $x < 20$. See Figure 6.2.

- CL supported. A single global countdown c. At expiry UPPAAL chooses among enabled random jumps using probability weights w_i. A resampling loop handles expiry in gaps. No window refinement. See Figure 6.3.

- CEP supported via modeling only when the predictive sampling distribution over the union of future enabling windows can be computed for the entry condition under consideration. If this distribution cannot be derived or depends on history that cannot be encoded in UPPAAL expressions the template does not faithfully model CEP. See Figure 6.4.

- CENP supported via modeling. A single stopwatch on the union $g := \bigvee_i g_i$ that runs across gaps of individual labels because the union remains true. At expiry overlapping labels are resolved by probability weights w_i.

- For all variants we make expiry urgent. In locations that count down we add the invariant $c \geq 0$. On every expiry edge we guard with $c \leq 0$. Window boundaries are encoded with $\leq$ and $\geq$. A committed splitter at $c \leq 0$ is used when needed. Under SMC there is no adversarial non-determinism. Races and branch choices are sampled.

6.3.2 Implementability RealySt

- Targets RAC or DENP style semantics with per label enabledness timers and a prophetic maximizing scheduler. This matches the DENP timing and allows a sound comparison on timing races.

- Composed variants CL or CEP or CENP are not natively supported in general. Label resolution is non-deterministic and maximizing on overlaps rather than probabilistic. In the current implementation a random clock or stochastic jump fires at most once. A reusable global random clock across multiple overlaps or loops cannot be expressed directly.

- A practical workaround from prior work unrolls loops and duplicates the stochastic structure per intended repetition with fresh clocks. This changes the reachability graph. In our experiments we could reproduce the reachability graph but not the quantitative probabilities. For this reason we restrict the cross tool quantitative comparison to DENP.

6.3.3 Experimental setup

All UPPAAL SMC results use one million simulations and time-bounded reachability of location `Process.l1` with the following queries

```
E[<=30; 1000000] (max: (Process.l1 ? 1 : 0))
E[<=100; 1000000] (max: (Process.l1 ? 1 : 0))
```

RealySt was run on the same model under DENP with time-bound 30 and jump bound 100

```
./realyst -t 30 -d 100 -b SISYPHUS -m A \
          --plotDimensions 0 1 -l trace
```

6.3.4 Quantitative results

Table 6.1 reports the estimates with their uncertainty.

Table 6.1: Time-bounded reachability to $l1$ from Figure 2.3/Figure 4.1

Variant	**Tool**	**Bound**	**Estimate**	**Uncertainty**
DENP	UPPAAL SMC	$t \leq 30$	0.222000	$\pm 8.14545 \times 10^{-4}$
		$t \leq 100$	0.617393	$\pm 9.52590 \times 10^{-4}$
DL	UPPAAL SMC	$t \leq 30$	0.072930	$\pm 5.09634 \times 10^{-4}$
		$t \leq 100$	0.281954	$\pm 8.81889 \times 10^{-4}$
CL	UPPAAL SMC	$t \leq 30$	0.296113	$\pm 8.94806 \times 10^{-4}$
		$t \leq 100$	0.709751	$\pm 8.89584 \times 10^{-4}$
CEP	UPPAAL SMC	$t \leq 30$	0.276903	$\pm 8.77023 \times 10^{-4}$
		$t \leq 100$	0.677297	$\pm 9.16305 \times 10^{-4}$
CENP	UPPAAL SMC	$t \leq 30$	0.277712	$\pm 8.77812 \times 10^{-4}$
		$t \leq 100$	0.712563	$\pm 8.87017 \times 10^{-4}$
DENP	RealySt	$t \leq 30$	0.2221555	$\pm 5.26794 \times 10^{-5}$

For $t \leq 30$ UPPAAL SMC yields approximately 0.222 and RealySt reports 0.2221555 with very small statistical error. The values agree with each other and with the analytic ground truth derived next.

6.3.5 Analytic check for DENP case

Under DENP the enabledness window lengths are $L_1 = 18$ for slip and $L_0 = 10$ for top. Sample $u \sim U[0,10]$ for top and $v \sim U[0,18]$ for slip independently. Top fires first exactly in the following three cases:

1. $u \in [0,4]$ and $v - 10 > u$. Slip has already been active for ten units before $x = 20$. For top to win in the first overlap [20,24) its sampled duration u must be at most four and slip must need more than u additional units beyond the ten it already collected. The vertical range for v at each u is $18 - (10 + u) = 8 - u$. The area is $\int_0^4 (8 - u)\, du = 24$.

2. $u \in (4,6]$ and both survive the first overlap which means $v \in (14,18]$. In the gap (24,26) only top is active and it wins provided $u \leq 6$. This gives a rectangle of width 2 and height 4. The area is 8.

3. $u \in (6,10]$ and after the gap top needs less remaining time than slip which is $u-6 < v-14$. At the start of the second overlap both are still waiting. Top requires $u-6$ more units while slip requires $v-14$. The condition is $v > u+8$. The vertical range is $18-(u+8) = 10-u$. The area is $\int_6^{10}(10-u)\,du = 8$.

The sample rectangle has area $10 \cdot 18 = 180$. The favorable area is $24+8+8 = 40$. Hence

$$\Pr(\text{top before slip}) = \frac{40}{180} = \frac{2}{9} \approx 0.222$$

which matches both tools for DENP.

Chapter 7

Conclusion

7.1 Summary

This thesis compares UPPAAL and RealySt for stochastic hybrid systems through the CAMELS classification with a focus on time-bounded reachability. In Chapter 1 we state the motivation and research questions. In Chapter 2 we introduce CAMELS and the formal background used throughout the work. Together these chapters set the stage for a semantics first comparison that makes tool assumptions explicit.

In Chapter 3 we delimit the subset of the UPPAAL language and its stochastic semantics that are used as a formal reference in this work. We give a precise account of what our automata can express and how races on sampled delays are interpreted. The chapter fixes notation and interpretation and is included for completeness and clarity.

In Chapter 4 we give a translation from decomposed eager non-predictive specifications to rectangular automata with random clocks together with the assumptions under which time-bounded reachability is preserved. The construction produces RAC models that are accepted by RealySt and that stay faithful to the intended DENP timing view. This gives a common target for cross tool comparison.

In Chapter 5 we implement the ARCH-COMP 2022 minimal examples in UPPAAL and report time-bounded probability estimates with their confidence information. We note required approximations such as ticked noise in flows where continuous stochastic effects are not available in native form. The case studies provide a controlled environment to exercise the formal setting from Chapter 3 and to prepare matched RAC instances from Chapter 4.

In Chapter 6 we place UPPAAL and RealySt in the CAMELS landscape and compare expressivity and guarantees. We align cases where semantics coincide and we discuss divergences where support differs. Where semantics are aligned the reported probabilities agree within statistical error in UPPAAL and within numerical tolerance in RealySt. In our Sisyphus DENP toy model both tools match the analytic value 2/9. When adversarial resolution of discrete choices is relevant RealySt computes maximal probabilities while UPPAAL with SMC resolves choices uniformly unless modeled otherwise. These differences follow from the CAMELS perspective developed in Chapter 2 and made concrete across Chapter 3 and Chapter 4.

The thesis addresses the objectives from Chapter 1. It fixes a clear UPPAAL fragment and stochastic semantics in Chapter 3. It supplies a DENP to RAC translation

with preservation assumptions in Chapter 4. It evaluates ARCH-COMP minimal examples in UPPAAL with documented approximations in Chapter 5. It compares UPPAAL and RealySt within CAMELS and explains agreements and gaps in Chapter 6. Our scope is time-bounded reachability. We do not address unbounded properties or expected reward objectives. Continuous stochastic noise is handled only through the stated approximations when needed. UPPAAL offers flexible modeling with statistical estimation. RealySt offers maximal probabilities on RAC under prophetic schedulers with exact geometric reachability and integration. Once semantics are aligned the outcomes coincide.

7.2 Future work

We keep the focus and point to concrete extensions grounded in Chapter 3, Chapter 4, Chapter 5 and Chapter 6

A central step is native continuous stochastic noise in the tools. UPPAAL and RealySt would benefit from direct support for continuous disturbances. This would let us replace the ticked approximations used in Chapter 5 and then measure the effect on reported probabilities and on runtime in a controlled way.

Another direction is controller synthesis with stratego studied on its own. The goal is to formalize how strategies interact with stochastic timing in the SMC setting and to relate the observed performance to the RAC based view when such a relation is meaningful.

RealySt would profit from support for several goal locations. Unions of targets and goal multiplicity enable simpler modeling tricks such as probability unwrapping by duplicating goals.

RealySt is also moving toward native random clock reuse. Once reuse is available it becomes possible to model CAMELS variants beyond DENP including CL, CEP, CENP and DL. These models can then be validated against the UPPAAL realizations discussed in Chapter 6.

Finally we observed that UPPAAL SMC estimates can be less stable in some runs. It is useful to study the influence of seeds, confidence settings and race tie handling and to propose checks or defaults that yield more consistent estimates for the experiments of Chapter 5.

Bibliography

[ABD+22] Alessandro Abate, Henk Blom, Joanna Delicaris, Sofie Haesaert, Arnd Hartmanns, Birgit van Huijgevoort, Abolfazl Lavaei, Hao Ma, Mathis Niehage, Anne Remke, Oliver Schön, Stefan Schupp, Sadegh Soudjani, and Lisa Willemsen. ARCH-COMP22 category report: Stochastic models. In Goran Frehse, Matthias Althoff, Erwin Schoitsch, and Jeremie Guiochet, editors, *Proceedings of 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH22)*, volume 90 of *EPiC Series in Computing*, pages 113–141. EasyChair, 2022.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on UPPAAL*, pages 200–236. Springer Berlin Heidelberg, 2004.

[CK14] Erhard Cramer and Udo Kamps. *Grundlagen der Wahrscheinlichkeitsrechnung und Statistik*. Springer-Lehrbuch. Springer Spektrum, Berlin, Heidelberg, 3 edition, 2014.

[DJL+15] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. UPPAAL stratego. In Christel Baier and Cesare Tinelli, editors, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, pages 206–211, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[DLL+15] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. UPPAAL smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[DRÁ+25] Joanna Delicaris, Anne Remke, Erika Ábrahám, Stefan Schupp, and Jonas Stübbe. Maximizing reachability probabilities in rectangular automata with random events. *Science of Computer Programming*, 240:103213, 2025.

[DSSR24] Joanna Delicaris, Jonas Stübbe, Stefan Schupp, and Anne Remke. Realyst: A C++ tool for optimizing reachability probabilities in stochastic hybrid systems. In Evangelia Kalyvianaki and Marco Paolieri, editors, *Performance Evaluation Methodologies and Tools*, pages 170–182. Springer Nature Switzerland, 2024.

[HKPV98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.

[JMG+20] Kenneth Yrke Jørgensen, Marius Mikučionis, Martijn Goorden, Thorulf Neustrup, Brian Nielsen, Mario Merlo, Asger Horn Brorholt, Peter Gjøl Jensen, sdfg610, Damiloju, and PHIKN1GHT. UPPAAL documentation. `https://docs.uppaal.org/`, 2020. [Accessed 30-09-2025].

[LMT15] Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Safe and optimal adaptive cruise control. In Roland Meyer, André Platzer, and Heike Wehrheim, editors, *Proceedings of the Correct System Design: Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday*, pages 260–277. Springer International Publishing, 2015.

[WRÁ23] Lisa Willemsen, Anne Remke, and Erika Ábrahám. Comparing two approaches to include stochasticity in hybrid automata. In Nils Jansen and Mirco Tribastone, editors, *Proceedings of the Quantitative Evaluation of Systems*, pages 238–254. Springer Nature Switzerland, 2023.

[WRÁ25] Lisa Willemsen, Anne Remke, and Erika Ábrahám. *(de-)Composed And More: Eager and Lazy Specifications (CAMELS) for Stochastic Hybrid Systems*, pages 309–337. Springer Nature Switzerland, 2025.

Appendix A

Appendix: RealySt Sisyphus DENP/RAC example

Listing A.1: RealySt model snippet for `CASE_A` (goal: reach `L1_FREE`)

```
case modelCase::CASE_A : {

  auto time = poolPtr->createVariable("time", librealyst::variableType::CONTINUOUS);
  auto x    = poolPtr->createVariable("x",    librealyst::variableType::CONTINUOUS);
  auto c0   = poolPtr->createVariable("c0",   librealyst::variableType::STOCHASTIC);
  auto c1   = poolPtr->createVariable("c1",   librealyst::variableType::STOCHASTIC);
  auto iT = poolPtr->getIndexForVariable(time);
  auto iX = poolPtr->getIndexForVariable(x);
  auto iC0= poolPtr->getIndexForVariable(c0);
  auto iC1= poolPtr->getIndexForVariable(c1);

  // Locations
  auto* L0_0_10  = automaton.createLocation("l0_[0,10]");
  auto* L0_10_20 = automaton.createLocation("l0_[10,20]");
  auto* L0_20_24 = automaton.createLocation("l0_[20,24]");
  auto* L0_24_26 = automaton.createLocation("l0_(24,26)");
  auto* L0_26_30 = automaton.createLocation("l0_[26,30]");
  auto* L1_FREE  = automaton.createLocation("l1");
  auto* L2_ROLL  = automaton.createLocation("l2");

  DistributionMap dmap;
  if (distributions.size() < 2) {
    if (distributions.empty())
      distributions.push_back(std::make_shared<librealyst::UniformDistribution>(0,
     10)); // c0
    if (distributions.size() == 1)
      distributions.push_back(std::make_shared<librealyst::UniformDistribution>(0,
     18)); // c1
  }
  // assume order: [0] -> c0, [1] -> c1
  dmap[iC0] = distributions.at(0);
  dmap[iC1] = distributions.at(1);
  automaton.setDistributionMapping(std::move(dmap));

  // flows
  VariableIntervalMap F_0_10 {
    { iT,  carl::Interval<Number>{1} },
    { iX,  carl::Interval<Number>{1} },
    { iC0, carl::Interval<Number>{0} },
    { iC1, carl::Interval<Number>{0} },
  };
  VariableIntervalMap F_10_20{
    { iT,  carl::Interval<Number>{1} },
    { iX,  carl::Interval<Number>{1} },
```

```
43      { iC0, carl::Interval<Number>{0} },
44      { iC1, carl::Interval<Number>{1} },
45    };
46    VariableIntervalMap F_20_24{
47      { iT,  carl::Interval<Number>{1} },
48      { iX,  carl::Interval<Number>{1} },
49      { iC0, carl::Interval<Number>{1} },
50      { iC1, carl::Interval<Number>{1} },
51    };
52    VariableIntervalMap F_24_26{
53      { iT,  carl::Interval<Number>{1} },
54      { iX,  carl::Interval<Number>{1} },
55      { iC0, carl::Interval<Number>{1} },
56      { iC1, carl::Interval<Number>{0} },
57    };
58    VariableIntervalMap F_26_30{
59      { iT,  carl::Interval<Number>{1} },
60      { iX,  carl::Interval<Number>{1} },
61      { iC0, carl::Interval<Number>{1} },
62      { iC1, carl::Interval<Number>{1} },
63    };
64    VariableIntervalMap F_FREE{
65      { iT,  carl::Interval<Number>{1} },
66      { iX,  carl::Interval<Number>{1} },
67      { iC0, carl::Interval<Number>{0} },
68      { iC1, carl::Interval<Number>{0} },
69    };
70    VariableIntervalMap F_ROLL{
71      { iT,  carl::Interval<Number>{1} },
72      { iX,  carl::Interval<Number>{-3} },
73      { iC0, carl::Interval<Number>{0} },
74      { iC1, carl::Interval<Number>{0} },
75    };
76
77    automaton.setRectangularFlowInLocation(L0_0_10,  F_0_10);
78    automaton.setRectangularFlowInLocation(L0_10_20, F_10_20);
79    automaton.setRectangularFlowInLocation(L0_20_24, F_20_24);
80    automaton.setRectangularFlowInLocation(L0_24_26, F_24_26);
81    automaton.setRectangularFlowInLocation(L0_26_30, F_26_30);
82    automaton.setRectangularFlowInLocation(L1_FREE,  F_FREE);
83    automaton.setRectangularFlowInLocation(L2_ROLL,  F_ROLL);
84
85    // --- invariants ---
86    VariableIntervalMap timeInvariant{
87      { iT, carl::Interval<Number>{ Number(0), timebound } }
88    };
89
90    VariableIntervalMap inv_l0_0_10{
91      { iT, carl::Interval<Number>{ Number(0), timebound } },
92      { iX, carl::Interval<Number>{ 0, 10 } }
93    };
94
95    VariableIntervalMap inv_l0_10_20{
96      { iT, carl::Interval<Number>{ Number(0), timebound } },
97      { iX, carl::Interval<Number>{ 10, 20 } }
98    };
99
100   VariableIntervalMap inv_l0_20_24{
101     { iT, carl::Interval<Number>{ Number(0), timebound } },
102     { iX, carl::Interval<Number>{ 20, 24 } }
103   };
104
105   VariableIntervalMap inv_l0_24_26{
106     { iT, carl::Interval<Number>{ Number(0), timebound } },
107     { iX, carl::Interval<Number>{ 24, carl::BoundType::STRICT, 26, carl::BoundType::
        STRICT } } // (24,26)
108   };
109
110   VariableIntervalMap inv_l0_26_30{
111     { iT, carl::Interval<Number>{ Number(0), timebound } },
112     { iX, carl::Interval<Number>{ 26, 30 } }
113   };
114
```

```
115   // l1: only time invariant (x unbounded)
116   VariableIntervalMap inv_l1 = timeInvariant;
117
118   // l2: time invariant + x >= 0
119   VariableIntervalMap inv_l2{
120     { iT, carl::Interval<Number>{ Number(0), timebound } },
121     { iX, carl::Interval<Number>{ Number(0), timebound} }
122   };
123
124   automaton.setInvariantsInLocation(L0_0_10,  inv_l0_0_10);
125   automaton.setInvariantsInLocation(L0_10_20, inv_l0_10_20);
126   automaton.setInvariantsInLocation(L0_20_24, inv_l0_20_24);
127   automaton.setInvariantsInLocation(L0_24_26, inv_l0_24_26);
128   automaton.setInvariantsInLocation(L0_26_30, inv_l0_26_30);
129   automaton.setInvariantsInLocation(L1_FREE,  inv_l1);
130   automaton.setInvariantsInLocation(L2_ROLL,  inv_l2);
131
132   // --- transitions ---
133   // window switches (x == 10, 20, 24, 26)
134   auto* t01 = automaton.createTransition(L0_0_10,  L0_10_20);
135   auto* t12 = automaton.createTransition(L0_10_20, L0_20_24);
136   auto* t23 = automaton.createTransition(L0_20_24, L0_24_26);
137   auto* t34 = automaton.createTransition(L0_24_26, L0_26_30);
138
139   VariableIntervalMap g_eq10{ { iX, carl::Interval<Number>{ 10 } } };
140   VariableIntervalMap g_eq20{ { iX, carl::Interval<Number>{ 20 } } };
141   VariableIntervalMap g_eq24{ { iX, carl::Interval<Number>{ 24 } } };
142   VariableIntervalMap g_eq26{ { iX, carl::Interval<Number>{ 26 } } };
143
144   automaton.setGuardsOfTransition(t01, g_eq10);
145   automaton.setGuardsOfTransition(t12, g_eq20);
146   automaton.setGuardsOfTransition(t23, g_eq24);
147   automaton.setGuardsOfTransition(t34, g_eq26);
148
149   // stochastic expirations (unguarded)
150   auto* slip_10_20 = automaton.createTransition(L0_10_20, L2_ROLL, iC1, true);
151   auto* slip_20_24 = automaton.createTransition(L0_20_24, L2_ROLL, iC1, true);
152   auto* slip_26_30 = automaton.createTransition(L0_26_30, L2_ROLL, iC1, true);
153
154   auto* top_20_24  = automaton.createTransition(L0_20_24, L1_FREE, iC0, true);
155   auto* top_24_26  = automaton.createTransition(L0_24_26, L1_FREE, iC0, true);
156   auto* top_26_30  = automaton.createTransition(L0_26_30, L1_FREE, iC0, true);
157
158   // rolling down -> restart at x == 0
159   auto* t_roll0 = automaton.createTransition(L2_ROLL, L0_0_10);
160   VariableIntervalMap g_eq0{ { iX, carl::Interval<Number>{ 0 } } };
161   automaton.setGuardsOfTransition(t_roll0, g_eq0);
162
163   // --- initial set ---
164   VariableIntervalMap initialConstraints{
165     { iT,  carl::Interval<Number>{ 0 } },
166     { iX,  carl::Interval<Number>{ 0 } },
167     { iC0, carl::Interval<Number>{ 0 } },
168     { iC1, carl::Interval<Number>{ 0 } },
169   };
170   automaton.setInitialStates(L0_0_10, initialConstraints);
171
172   // Spec: reach l1
173   librealyst::Specification S{};
174   S.goalLocation = L1_FREE;
175   spec.addSpecification(S);
176   break;
177 }
```