CrossMark

UPPAAL **SMC tutorial**

**Alexandre David · Kim G. Larsen · Axel Legay ·
Marius Mikučionis · Danny Bøgsted Poulsen**

Abstract   This tutorial paper surveys the main features of
UPPAAL SMC, a model checking approach in UPPAAL family
that allows us to reason on networks of complex real-timed
systems with a stochastic semantic. We demonstrate the mod-
eling features of the tool, new verification algorithms and
ways of applying them to potentially complex case studies.

Keywords   Uppaal · Timed automata · Model-checking ·
Statistical model-checking · Stochastic · Hybrid ·
Dynamical · Probabilistic

1 Introduction

Computer systems play a central role in modern societies
and their errors can have dramatic consequences. Prov-
ing the correctness of computer systems is therefore a
highly relevant activity, on which both industry and aca-
demics invest a considerable amount of effort. Among
such techniques, one finds (1) testing [14], the traditional
approach that detects bugs by exercising the real system
with test cases, and (2) formal methods, e.g., model check-
ing [17], that is a more mathematical approach that can
guarantee the absence of bugs in the system design. Both
approaches have been largely deployed on complex case
studies.

Originally, formal verification was devoted to software
and hardware systems by considering their discrete behav-
iors. However, the past years shown that real-time aspects
play central roles in systems, and that this feature should be
taken into account in the verification process. Developing
formal techniques for such systems has thus been the subject
of intensive studies. One of the prominent results on the topic
was the introduction of model checking techniques for timed
automata [1], a natural model to capture real-time systems
whose behaviors depends on clocks that can be reset. Among
all the tools that have been developed to implement the timed
automata theory, one finds UPPAAL, which has now become
the leader in the area.

UPPAAL is a toolbox for verification of real-time systems
represented by (a network of) timed automata extended with
integer variables, structured data types, and channel synchro-
nization. The tool is jointly developed by Uppsala Univer-
sity and Aalborg University. It has been applied success-
fully in case studies ranging from communication protocols
to multimedia applications (see [4] and [5] for concrete exam-
ples). The first version of UPPAAL was released in 1995 [37].
Since then it has been in constant development. In the same
spirit as any other professional model checker such as SPIN,
UPPAAL proposes efficient data structures [36], a distributed
version of UPPAAL [10,13], guided and minimal cost reach-
ability [11,12,35], work on UML Statecharts [25], acceler-
ation techniques [29], and new data structures and memory
reductions [9,15].

Unfortunately, timed automata is not a panacea. In fact,
albeit powerful, the model is not expressive enough to cap-
ture behaviors of complex cyber-physical systems. Indeed,
the continuous time behaviors of those systems often rely
on rich and complex dynamics as well as on stochastic
behaviors. The model checking problem for such systems is
undecidable, and approximating those behaviors with timed
automata [28] was originally the best one could originally do
in UPPAAL.

A. David · K. G. Larsen · M. Mikučionis · D. B. Poulsen
Department of Computer Science, Aalborg University, Aalborg,
Denmark

A. Legay (✉)
INRIA/IRISA Rennes, Rennes, France
e-mail: axel.legay@inria.fr

In this paper, we introduce UPPAAL SMC that proposes an alternative to the above-mentioned problem. This new branch of UPPAAL proposes to represent systems via networks of automata whose behaviors may depend on both stochastic and non-linear dynamical features. Concretely, in UPPAAL SMC, each component of the system is described with an automaton whose clocks can evolve with various rates. Such rates can be specified with, e.g., ordinary differential equations.

To allow for the efficient analysis of probabilistic performance properties, UPPAAL SMC proposes to work with statistical model checking (SMC) [38,41], an approach that has been proposed as an alternative to avoid an exhaustive exploration of the state-space of the model. The core idea of SMC is to monitor some simulations of the system, and then use results from the statistics area (including sequential hypothesis testing or Monte Carlo simulation) to decide whether the system satisfies the property with some degree of confidence. By nature, SMC is a compromise between testing and classical model checking techniques. Simulation-based methods are known to be far less memory and time intensive than exhaustive ones, more expressive and are oftentimes the only option. SMC has been implemented in a series of tools that have been applied to a wide range of case studies. Unlike more "academic" exhaustive techniques, SMC gets widely accepted in various research areas such as systems biology [16,20,32–34], energy-centric systems [21], automotive/avionics, or software engineering, in particular for industrial applications. There are several reasons for this success. First, it is very simple to implement, understand and use (especially by industry, software engineers, and generally all people that are not pure researchers but customers of our results and tools) [3,4,8]. Second, it does not require extra modeling or specification effort, but simply an operational model of the system that can be simulated and checked against state-based properties. Third, it allows to model check properties that cannot be expressed in classical temporal logics. Aside from this, the flexibility of SMC allows it to be used in other areas than verification, including planning and robotics.

In this paper, SMC is presented as a technique for fully stochastic models thus it validates performance properties of a given deterministic (or stochastic) controller in a given stochastic environment. However, we note that SMC is applicable to systems exhibiting non-determinism (transitions with undefined probability distributions): for instance the SMC tool COSMOS has been used to find optimal schedulers for Markov decision processes [26] and in a more recent work [22] an experimental version of UPPAAL SMC was used for synthesizing controllers for priced timed Markov decision processes.

This paper is a complete tutorial on UPPAAL SMC for hybrid and fully stochastic systems. We illustrate most of

the modeling features of the tool, the usage of the graphical interface and of the simulation framework. We discuss the SMC algorithms that are available, and introduce some techniques to deal with dynamical systems. Finally, we present some modeling features and tricks.

2 Modeling formalism

The modeling formalism of UPPAAL SMC is based on a stochastic interpretation and extension of the timed automata (TA) formalism [1] used in the classical model checking version of UPPAAL [4]. For individual TA components the stochastic interpretation replaces the non-deterministic choices between multiple enabled transitions by probabilistic choices (that may or may not be user-defined). Similarly, the non-deterministic choices of time delays are refined by probability distributions, which at the component level are given either uniform distributions in cases with time-bounded delays or exponential distributions (with user-defined rates) in cases of unbounded delays.

Consider the three TAs A_1, A_2 and A_3 from Fig. 1. Ignoring (initially) the weight annotations on locations and edges, the END locations in the three automata are easily seen to be reachable within the time intervals [6, 12], [4, 12] and $[0, +\infty)$. The stochastic interpretation of the three TAs provides probability distributions over the reachability time. For A_1, the delay of the three transitions will all be (automatically) resolved by independent, uniform distributions over [2, 4]. Thus the overall reachability time is given as the sum of three uniform distributions as illustrated in Fig. 2a. For A_2, the delay distributions determined by the upper and lower
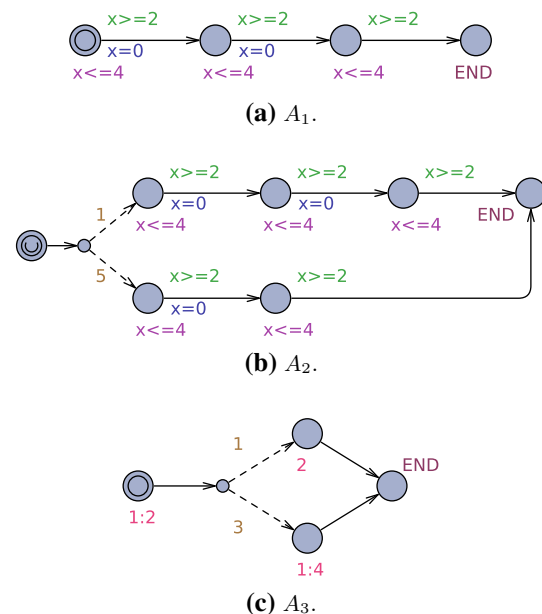

(a) A_1.

(b) A_2.

(c) A_3.

Fig. 1 Three stochastic timed automata

(a)

(b)

(c)

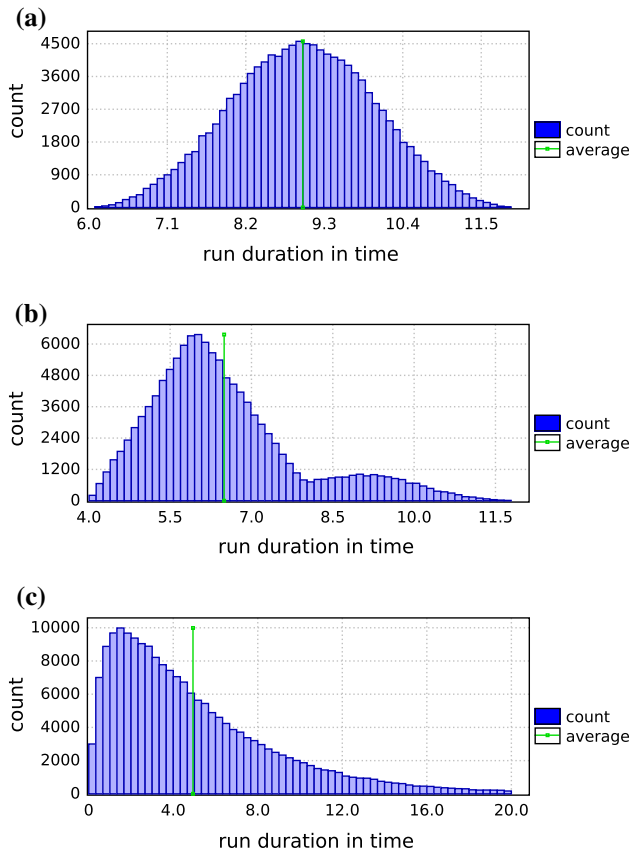**Fig. 2** Distributions of reachability time, **a** A_1 arrival to END, **b** A_2 arrival to END, **c** A_3 arrival to END

Fig. 3 An NSTA, $(A|B|T)$

Fig. 4 Cumulative probabilities for time and Cost-bounded reachability of T_3

path to the END location are similarly given by sums of uniform distributions. Subsequently, the combination ($\frac{1}{6}$ to $\frac{5}{6}$) of these as illustrated in distribution of the overall delay is obtained by a weighted Fig. 2b. Finally, in A_3—in the absence of invariants—delays are chosen according to exponential distributions with user-supplied rates (here $\frac{1}{2}$, 2 and $\frac{1}{4}$). In addition, after the initial delay a discrete probabilistic choice ($\frac{1}{4}$ versus $\frac{3}{4}$) is made. The resulting distribution of the overall reachability time is given in Fig. 2c.

Importantly, the distributions provided by the stochastic semantics are in agreement with the delay intervals determined by the standard semantics of the underlying timed automata. Thus, the distributions for A_1 and A_2 have finite support by the intervals [6, 12] and [4, 12], respectively. Moreover, as indicated by A_3, the notion of stochastic timed automata encompasses both discrete and continuous time Markov chains. In particular, the class of reachability time distributions obtained from the stochastic timed automata (STA) of UPPAAL SMC includes that of phase-type distributions.

Networks As in UPPAAL, a model in UPPAAL SMC consists of a network of interacting component STAs. Here, it
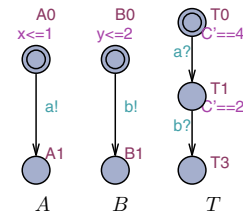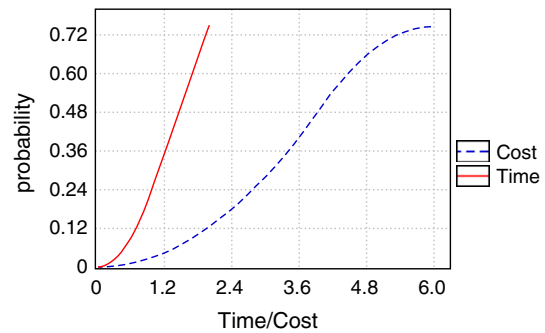
is assumed that these components are input-enabled, deterministic (with a probability measure defined on the sets of successors), and non-Zeno. The component STAs communicate via broadcast channels and shared variables to generate Networks of Stochastic Timed Automata (NSTA). The communication is restricted to broadcast synchronizations to keep a clean semantics of only non-blocked components which are racing against each other with their corresponding local distributions.

Figure 3 shows an NSTA with three parallel components A, B, and T as specified using the UPPAAL GUI. One can easily see that the composite system $(A|B|T)$ has a transition sequence:

$$((A_0, B_0, T_0), [x = 0, y = 0, C = 0]) \xrightarrow{1} \xrightarrow{a!}$$
$$((A_1, B_0, T_1), [x = 1, y = 1, C = 4]) \xrightarrow{1} \xrightarrow{b!}$$
$$((A_1, B_1, T_2), [x = 2, y = 2, C = 6]),$$

demonstrating that the final location T_3 of T is reachable. In fact, location T_3 is reachable within cost 0 to 6 and within total time 0 and 2 in $(A|B|T)$ depending on when (and in which order) A and B choose to perform the output actions $a!$ and $b!$. Given that the choice of these time delays is governed by probability distributions, a measure on sets of runs of NSTAs is induced, according to which quantitative properties such as *"the probability of T_3 being reached within a total cost-bound of 4.3"* become well defined (Fig. 4).

For components, as stated in the previous section, UPPAAL SMC applies uniform distributions for bounded delays and exponential distributions where a component STA

can remain indefinitely in the same location. In a network of STAs, the components repeatedly race against each other, i.e., they independently and stochastically decide on their own how much to delay before outputting, with the "winner" being the component that chooses the minimum delay. For instance, in the NSTA of Fig. 3, A wins the initial race over B with probability 0.75.

As observed in [23], though the stochastic semantic of each individual STA in UPPAAL SMC is rather simple (but quite realistic), arbitrarily complex stochastic behavior can be obtained by their composition when mixing individual distributions through message passing. The beauty of our model is that these distributions are naturally and automatically defined by the network of STAs.

Train crossing example UPPAAL SMC takes as input NSTAs as described above. Additionally, there is support for all other features of the UPPAAL model checker's input language such as integer variables, data structures and user-defined functions, which greatly ease modeling. UPPAAL SMC allows the user to specify an arbitrary (integer) rate for the clocks on any location. In addition, the automata support branching edges where weights can be added to give a distribution on discrete transitions. It is important to note that rates and weights may be general expressions that depend on the states and not just simple constants.

To illustrate the extended input language, we consider a train-gate example adapted from [42]. The example model is distributed together with UPPAAL SMC tool. A number of trains are approaching a bridge on which there is only one track. To avoid collisions, a controller stops the trains. It restarts them when possible to make sure that trains will eventually cross the bridge. There are timing constraints for stopping the trains modeling the fact that it is not possible to stop trains instantly. The interesting point w.r.t. SMC is to define the arrival rates of these trains. Figure 5a shows the template for a train. The location Safe has no invariant and defines the rate of the exponential distribution for delays. Trains delay according to this distribution and then approach by synchronizing on appr[i] with the gate controller. Here, we define the rational $\frac{1+id}{N^2}$ where id is the identifier of the train and N is the number of trains. Rates are given by expressions that can depend on the current states. Trains with higher id arrive faster. Taking transitions from locations with invariants is given by a uniform distribution over the time interval defined by the invariant. This happens in locations Appr, Cross, and Start, e.g., it takes some time picked uniformly between 3 and 5 time units to cross the bridge. Figure 5b shows the gate controller that keeps track of the trains with an internal queue data structure (not shown here). It uses functions to queue trains (when a train approaches and the bridge is occupied in Occ) or dequeue them (when some train leaves and the bridge is free).
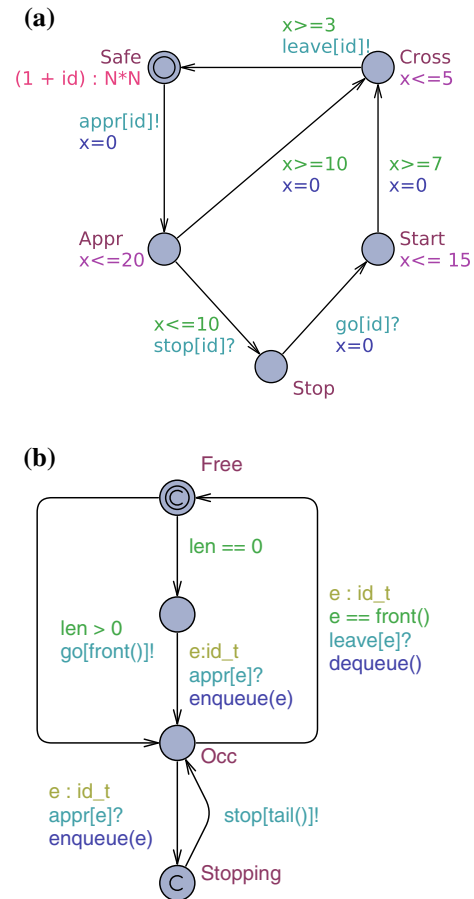
Fig. 5 Templates for the train-gate example, **a** train, **b** gate controller

3 Query language

In addition to the standard model checking queries—i.e., reachability, invariance, inevitability and leads-to, which are still available—UPPAAL SMC provides a number of new queries related to the stochastic interpretation of timed automata. UPPAAL SMC allows the user to visualize the values of expressions (evaluating to integers or clocks) along simulated runs. This gives insight to the user on the behavior of the system so that more interesting properties can be asked to the model checker. The concrete syntax applied in UPPAAL SMC is as follows:

simulate N [< = bound] {E1, .., Ek}

where N is a natural number indicating the number of simulations to be performed, bound is the time bound on the simulations, and E1, .., Ek are the k (state-based) expressions that are to be monitored and visualized. To demonstrate this on our previous train-gate example, we can monitor when Train(0) and Train(5) are crossing as well as the length of the queue. The query is
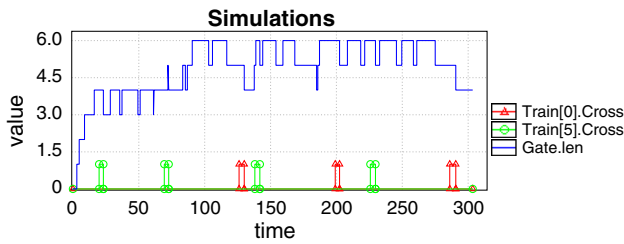
Fig. 6 Visualizing the gate length and when Train(0) and Train(5) cross on one random run

Fig. 7 True probability **P** and confidence intervals

```
simulate 1 [<=300]
  {Train(0).Cross, Train(5).Cross, Gate.len}
```

This gives us the plot of Fig. 6. Interestingly, Train(5) crosses more often (since it has a higher arrival rate). Secondly, it seems unlikely that the gate length drops below 3 after some time (say 20), which is not an obvious property from the model. We can confirm this by asking Pr[<= 300](<> Gate.len < 3 and t > 20) and adding a clock t. The probability is in [0.102, 0.123].

For specifying properties over NSTAs, we use a weighted extension of the temporal logic MITL [2] expressing properties over runs [6], defined by the grammar:

$$\varphi ::= \mathsf{ap} \,|\, \neg\varphi \,|\, \varphi_1 \wedge \varphi_2 \,|\, \mathsf{O}\varphi \,|\, \varphi_1 \mathsf{U}^x_{\leq d} \varphi_2$$

where ap is a conjunction of predicates over the state of a NSTA, d is a natural number and x is a clock. Here, the logical operators are interpreted as usual, and O is a next state operator. A weighted MITL formula $\varphi_1 \mathsf{U}^x_{\leq d} \varphi_2$ is satisfied by a run if φ_1 is satisfied on the run until φ_2 is satisfied, and this will happen before the value of the clock x exceeds d. As usual $\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2$ and we use standard MITL abbreviations $\mathsf{tt} = \varphi \vee \neg\varphi$, $\Diamond_{x \leq d}\varphi = \mathsf{tt}\mathsf{U}^x_{\leq d}\varphi$ and $\Box_{x \leq d}\varphi = \neg\Diamond_{x \leq d}\neg\varphi$.

For an NSTA M, we define $\mathbb{P}_M(\varphi)$ to be the probability that a random run of M satisfies φ. The problem of checking $\mathbb{P}_M(\varphi) \geq p$ ($p \in [0, 1]$) is undecidable in general.[1] For the sub-logic of cost-bounded reachability problems $\mathbb{P}_M(\Diamond_{x \leq C}\mathsf{ap}) \geq p$, where x is a clock and C is a bound, UPPAAL SMC approximates the answer using simulation-based algorithms known under the name of statistical model checking [41] algorithms (SMC). We briefly recap statistical algorithms permitting to answer the following three types of questions:

1. *Probability estimation*
   What is the probability $\mathbb{P}_M(\Diamond_{x \leq C}\mathsf{ap})$ for a given NSTA M?

2. *Hypothesis testing*
   Is the probability $\mathbb{P}_M(\Diamond_{x \leq C}\mathsf{ap})$ for a given NSTA M greater or equal to a certain threshold $p \in [0, 1]$?

3. *Probability comparison*
   Is the probability $\mathbb{P}_M(\Diamond_{x \leq C}\mathsf{ap}_1)$ greater than the probability $\mathbb{P}_M(\Diamond_{y \leq D}\mathsf{ap}_2)$?

From a conceptual point of view solving the above questions using SMC is simple. First, each run of the system is encoded as a Bernoulli random variable that is true if the run satisfies the property and false otherwise. Then a statistical algorithm groups the observations to answer the three questions. For the quantitative question (1), we will use an estimation algorithm that resemble the classical Monte Carlo simulation, while for the qualitative questions (2 and 3) we shall use sequential hypothesis testing. The two solutions are detailed hereafter.

Probability estimation The probability estimation algorithm [30] computes the number of runs needed to produce an approximation interval $[p - \varepsilon, p + \varepsilon]$ for $p = Pr(\psi)$ with a confidence $1 - \alpha$. A frequentist interpretation of this result tells us that if we repeat the interval estimation N times, then the estimated confidence interval $p \pm \varepsilon$ contains the true probability at least $(1 - \alpha)N$ times in the long run ($N \to \infty$). Figure 7 shows the relation between the estimated probability confidence intervals and the true (unknown) probability **P**.

The original algorithm for interval estimation decides the number of runs apriori based on the values of ε and α using Chernoff–Hoeffding inequality [18,31], however for practical purposes this inequality is too conservative. Moreover, the result can be even more improved when the probability is further from $\frac{1}{2}$. UPPAAL SMC implements a sequential method where a probability confidence interval (for given α) is derived with each new simulation measurement and the simulation generation is stopped when the confidence interval width is less than 2ε. The confidence interval is derived using Clopper–Pearson "exact" method [19] using the fact that the measurements are always binary (the property is either satisfied or not) and thus the result follows binomial distribution. The confidence level is also adjusted for one-
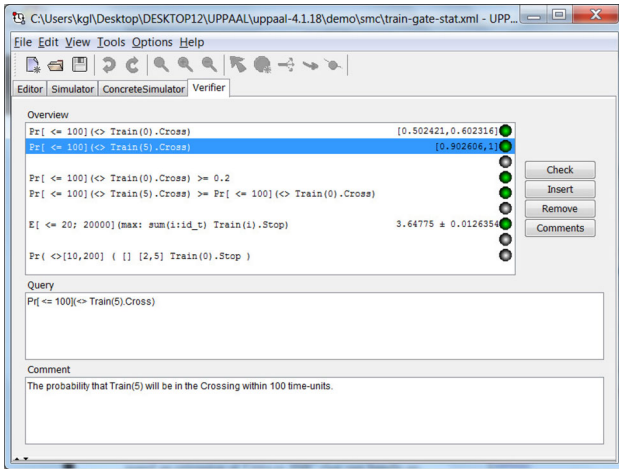
---

[1] Exceptions being stochastic TAs with 0 or 1 clocks and with p being 0 or 1.

Fig. 8 The Verifier of UPPAAL SMC

Fig. 9 The cumulative probability distribution of Pr[$<=T$]($<>$ Train(5).Cross)

sided intervals, where the measured property is always true or always false.

In UPPAAL SMC, the probability confidence interval can be estimated by the following query:

Pr[bound] (ap)

Example 1 Recall the Train Crossing example of the previous section. The following queries estimates the probabilities that Train(0) and Train(5) will be in the crossing before 100 time units:

Pr[$<= 100$]($<>$ Train(0).Cross)

Pr[$<= 100$]($<>$ Train(5).Cross)   (1)

Figure 8 shows how these (and other) queries are entered in the "Query" field of the Verifier tab of UPPAAL SMC. In the "Overview" field the answers are provided: [0.502421, 0.602316] and [0.902606, 1] are the two 95 % confidence intervals obtained from 383 and 36 runs, respectively. This shows—as we would expect—that the more eager Train(5) has a higher probability of reaching the crossing than Train(0) within the given time limit. Right clicking on the answers provide easy access to more detailed information in terms of (cumulative, confidence interval, frequency histogram) probability distribution of the time-bounded reachability property, e.g., Fig. 9.

Hypothesis testing This approach reduces the qualitative question to e test the null-hypothesis:

$H: p = \mathbb{P}_M(\psi) \geq \theta$

against the alternative hypothesis:

$K: p = \mathbb{P}_M(\psi) < \theta$

To bound the probability of making errors, we use strength parameters α and β and we test the hypothesis $H_0: p \geq p_0$
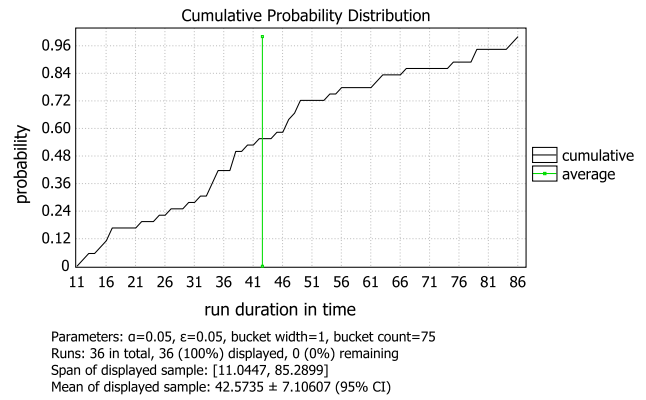
and $H_1: p \leq p_1$ with $p_0 = \theta + \delta_0$ and $p_1 = \theta - \delta_1$. The interval $p_0 - p_1$ defines an indifference region, and p_0 and p_1 are used as thresholds in the algorithm. The parameter α is the probability of accepting H_0 when H_1 holds (false positives) and the parameter β is the probability of accepting H_1 when H_0 holds (false negatives). The above test can be solved by using Wald's sequential hypothesis testing [40]. This test computes a proportion r among those runs that satisfy the property. With probability 1, the value of the proportion will eventually cross $\log(\beta/(1 - \alpha))$ or $\log((1 - \beta)/\alpha)$ and one of the two hypothesis will be selected. In UPPAAL SMC, we use the following query:

Pr[bound] (ψ) $>=$ p_0

where *bound* defines how to bound the runs. The three ways to bound them are (1) implicitly by time by specifying $<=M$ (where M is a positive integer), (2) explicitly by cost with $x<=M$ where x is a specific clock, or (3) by number of discrete steps with # $<=M$. In the case of hypothesis testing p_0 is the probability to test for. The formula ψ is either $<>q$ or $[]q$ where q is a state predicate.

Remark 1 Bounding runs for a number of discrete steps guarantees termination of the simulation. Bounding over time may however result in non-termination if the model is not time diverging. Similarly, bounding over a non-diverging clock can result in non-termination.

UPPAAL SMC cannot detect if a clock (or time) is diverging in a model thus the modeler needs to ensure this.

Example 2 Returning to the Train Crossing example, we may not be directly interested in the actual probability of Train(0) crossing within 100 time units, but merely whether this unknown probability is above 0.2, as reflected by the following query (see also Fig. 8):

Pr[$<= 100$]($<>$ Train(0).Cross) $>= 0.2$

Within a number of runs significantly smaller than that of estimating the same probability (383 runs), this property may be

confirmed. The number of runs needed by Wald's sequential hypothesis testing method varies, e.g., posing the above query 5 times, the property was confirming within 66, 62, 65, 67, and 49 runs, respectively, with 5 % level of significance.

Probability comparison This algorithm, which is detailed in [23], exploits an extended Wald testing. In UPPAAL SMC, we use the following query:

$Pr[bound_1](\varphi_1) >= Pr[bound_2](\varphi_2)$.

Example 3 In the train-gate example, it might be sufficient to confirm that the probability that Train(5) reaches the crossing within 100 time units is larger than that of Train(0). Posing the query:

Pr[<= 100](<> Train(5).Cross)

  >= Pr[<= 100](<> Train(0).Cross)

confirms this belief within 120 (132, 144, 108, 174) runs with 5 % level of significance.

In addition to those three classical tests, UPPAAL SMC also supports the evaluation of expected values of min or max of an expression that evaluates to a clock or an integer value. The syntax is as follows:

E[*bound*; *N*](min:*expr*)

or

E[*bound*; *N*](max:*expr*)

where *bound* is as explained in this section, *N* gives the number of runs explicitly, and *expr* is the expression to evaluate. Also for these properties a confidence interval is given using the fact that measurements follow Student's *t*-distribution (approaching Normal distribution when $N \to \infty$).

Example 4 As an interesting property of the Train Crossing example, we want to know the average of the maximum number of trains that are stopped within the first 20 time units:

E[<=20; 20000]
  (max: sum(i : id_t) Train(i) .Stop)

Using the explicitly required 20.000 runs, this average is estimated to be in the confidence interval 3.64775 ± 0.0126354. Right clicking gives easy access to more detailed views, e.g., the frequency histogram in Fig. 10.

Full weighted MITL Regarding the implementation, we note that both the above statistical algorithms are trivially implementable. To support the full logic of weighted MITL is slightly more complex as our simulation engine needs to rely on monitors for such logic. In [7], we proposed an extension of UPPAAL SMC that can handle arbitrary formulas of weighted MITL. Given a property φ, our implementation
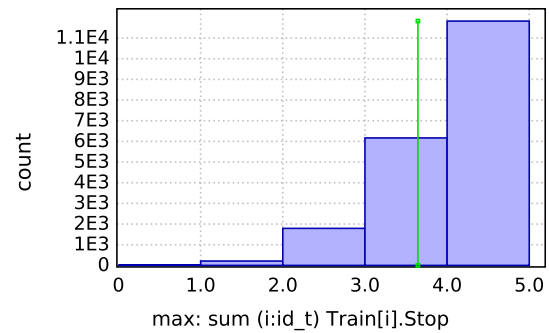

Fig. 10 Frequency histogram of maximum number of trains stopped within 20 time units

first constructs deterministic under- and over-approximation monitoring PTAs for φ. Then, it puts these monitors in parallel with a given model M, and applies SMC-based algorithms to bound the probability that φ is satisfied on M. More recently [6], the *exact* evaluation of whether the generated run satisfies a given weighted MITL formula is done online by constantly rewriting the formula during generation of the run.

The probability of satisfying an MITL property ψ is estimated by UPPAAL SMC using the query Prψ, where

$\psi ::= BExpr$

  $| (\psi \&\& \psi) | \quad (\psi || \psi)$

  $| (\psi U[a, b] \psi) | \quad (\psi R[a, b] \psi)$

  $| (<>[a, b] \psi) | \quad ([][a, b] \psi)$

$a, b \in \mathbb{N}$, $a \leq b$ and BExpr is a Boolean expression over clocks, variables and locations.

Example 5 The following query:

Pr(<>[10,100] ([ ][0,5] Train(0) .Stop))

asks for the probability that Train(0) will stopped for at least 5 consecutive time units somewhere in the time interval [10, 100]. Within 738 runs [0.880894, 0.980894] is returned as a 95 % confidence interval indicating that this happens with a very high probability.

4 Extension to hybrid systems

UPPAAL SMC allows for statistical model checking of stochastic hybrid systems, i.e., extensions of (stochastic) timed automata, where the rate of clocks may be given by general expressions involving clocks, thus effectively using ODEs.

To illustrate the various aspects of the (extended) modeling formalism supported by UPPAAL SMC, we consider the case of two independent rooms that can be heated by a single heater shared by the two rooms, i.e., at most one room can be heated at a time. Figure 11a shows the automaton for the heater. It turns itself on with a uniform distribution over time
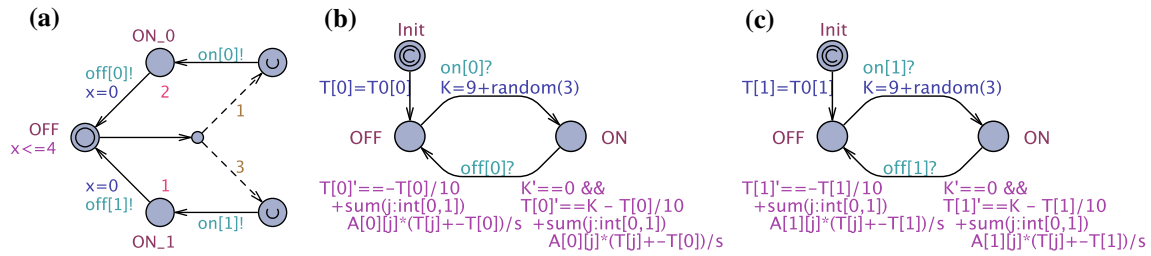
Fig. 11 A simple two-room example with an autonomous heater, **a** stochastic heater, **b** room 0, **c** room 1
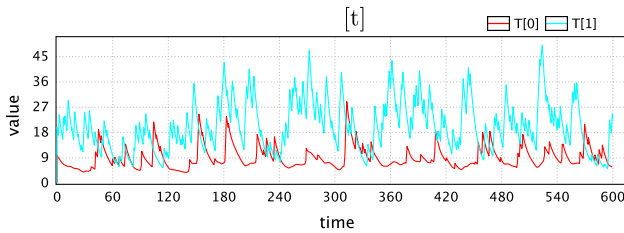

Fig. 12 Evolution of the temperatures of the two rooms (color figure online)

in-between [0, 4] time units. With probability 1/4 room 0 is chosen and with probability 3/4 room 1. The heater stays on for some time given by an exponential distribution (rate 2 for room 0, rate 1 for the room 1). In summary, one may say that the controller is more eager to initiate the heating of room 1 than room 0, as well as less eager to stop heating room 1. The rooms are similar and are modeled by the same template instantiated twice as shown in Fig. 11b, c. The room is initialized to its initial temperature and then depending on whether the heater is turned on or not, the evolution of the temperature is given by $T_i' = -T_i/10 + \sum_{j=0,1} A_{i,j}(T_j - T_i)$ or $T_i' = K - T_i/10 + \sum_{j=0,1} A_{i,j}(T_j - T_i)$ where $i, j = 0, 1$ are room identifiers. The sum expression corresponds to an energy flow between rooms and matrix A encodes the energy transfer coefficient between adjacent rooms. Furthermore, when the heater is turned on, its heating is not exact and is picked with a uniform distribution of $K \in [9, 12]$, realized by the update K=9+random(3).

This example illustrates the support for stochastic hybrid systems in UPPAAL SMC with extended arithmetic on clocks and generalized clock rates.

UPPAAL SMC takes as input networks of stochastic hybrid automata as described above. In addition, the automata support branching edges where weights can be added to give a distribution on discrete transitions. It is important to note that rates and weights may be general expressions that depend on the states and not just simple constants.

Remark 2 The ODE solver implemented within UPPAAL SMC is fixed time step Euler's integration method thus the results may be sensitive to the discretization step size.

Euler's method is known to be unstable for stiffs systems thus care must taken when deciding on the discretization step size controlled in the settings of statistical parameters.

4.1 Floating-point support

The syntax has been extended to support a double precision floating-point type (double). This type can be used mixed with clocks for computing or storing arithmetic expressions. Its rate cannot be changed. When using floating-point types or operations in a model, the model is marked as being hybrid. For such models, model checking is disabled, unless the clocks are declared to be hybrid clock and these clocks nor the floating-point variables affect the control of the automata, i.e., such variables are inactive and used as costs.

4.2 Example

All the new queries of UPPAAL SMC described in Sect. 3 are available for stochastic hybrid systems. We illustrate this by a number of queries related to the two-room example from the previous section.

We can simulate and plot the temperatures of the two rooms with the query

simulate 1 [<= 600] {T[0], T[1]}

The query request the checker to provide one simulate run over 600 time units and plot the temperatures of Room(0) and Room(1). The heater in this example is purely stochastic and is not intended to enforce any particular property. Yet, the simulation obtained from this query in Fig. 12 shows that the heater is able to maintain the temperatures within (mostly) distinct intervals.

We can evaluate on a shorter time scale the probability for the temperature of Room(0) to stay below 30 and the temperature of Room(1) to stay above 5 with the queries

Pr[<=100] ([ ] Room(0).Init || T[0] <= 20)
Pr[<=100] ([ ] Room(1).Init || T[1] >= 7)

The results are, respectively, in [0.45, 0.55] and [0.65, 0.75]. The precision and confidence of confidence intervals are user-defined (see later) and influence the number of runs

needed to compute the probability. In this example, for having the precision to be ±0.05 and a confidence of 95 %, we needed 738 runs. In fact, if we are only interested in knowing if the second probability is above a threshold it may be more efficient to test the hypothesis

Pr[<=100] ([ ] Room(1).Init || T[1] >= 7)
  >= 0.69

which is accepted in our case with 902 runs for a level of significance of 95 %. To obtain an answer at comparable level of precision with probability evaluation, we would need to use a precision of ±0.005, which would require 73778 runs instead.

We can test the hypothesis that the heater is better at keeping the temperature of Room(1) above 8 than keeping the temperature of Room(0) below 20 by the following comparison query:

Pr[<=100] ([ ] Room(1).Init || T[1] >= 7) >=
  Pr[<=100] ([ ] Room(0).Init || T[0] <= 20)

which is accepted in this case with 95 % level of significance with just 258 runs.

Remark 3 As it can be observed, the MITL specifications allowed in UPPAAL SMC are bounded properties. That is specifications only depend on a run up to a given time-bound, step-bound or bound on some other quantity defined in the model. Thus, specifications only express properties of transient behavior of systems, and may or may not be indicative of safety of a deployed system in steady operational state, depending on how long the system takes to settle. However, given knowledge of the size of the model, estimation of probability of unbounded properties may be obtained from the observation of finite runs as shown in [39].

5 Extension to dynamic creation of processes

An underlying assumption of networks of timed automata is that computer systems are statically encoded. This is however not reality. Instead, systems are composed of a number of threads/processes that interact and capable of spawning other processes/threads. Modeling such dynamic systems in standard UPPAAL requires the modeler to model an underlying resource manager. In addition, the model would consist of a large number of components in an inactive state available for the resource manager to "start" whenever a spawn request was made in the model. A necessary assumption for modeling this resource manager is thus that the maximum number of spawned threads during any execution is known in advance (or can be safely over-approximated). This does not only make modeling tedious but also affects analysis time. UPPAAL SMC supports instantiating dynamic processes out
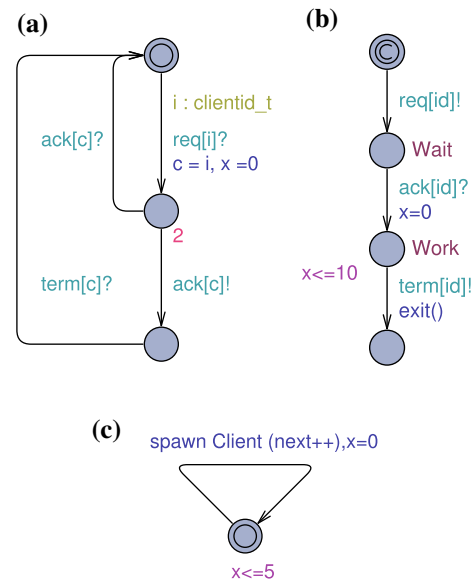

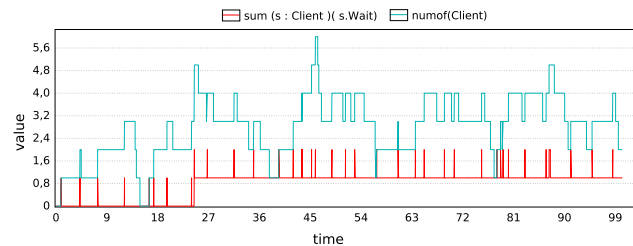**Fig. 13** Modeling a server with dynamic spawning, **a** server, **b** client, **c** clientSpawner

Fig. 14 Plot of the number of waiting clients and total number of clients. The plot was obtained with the query simulate 1[<= 100]{numOf(Client), sum (c : Client)(c.Wait)} (color figure online)

of the box. Any automata in the system can spawn instances of templates of the model that has been declared to be spawnable. Dynamically created instances act within the system as the static instances with the exception that they at any time may terminate, and thus remove themselves from the system.

In Fig. 13 is a high-level model of a client–server architecture. The model consists of a number of servers (10), shown in Fig. 13a, that listens on all possible input channels req. When a request arrives all the servers will "race" to acknowledge the connection over the channel ack[c]. The winner will proceed to communicate with the client (we abstract from this part), while the others return to their listening state. When a client has finished communication with the server it will terminate the connection by synchronizing on term[c]. Afterwards, the server returns to its listening state.

In Fig. 13b, we show the client side of the model. A client is given an id when spawned which tells it what channel to connect on (req[id]). A client is first attempting to get a connection, then it awaits an acknowledgment from a server and then do some work taking less than ten time units. Finally,
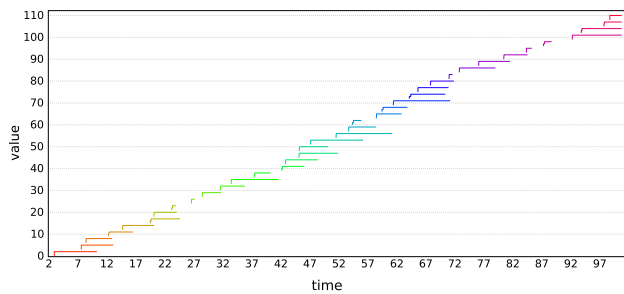
Fig. 15 Plot showing the life span of each client in the server example (color figure online)

it disconnects from the server by synchronizing on term[id] and at the same time tears itself down using the *exit*() construction.

Clients are spawned by the template in Fig. 13c using the **spawn** Client(next++) update. This instantiates a client and passes the value of next as a parameter to the client which binds that value to its own local variable id.

Remark 4 We realize that the template in Fig. 13c may create an unbounded number of clients, whereas the number of communication channels are bounded. For our particular use this is not a problem as we know the number of spawned clients will not exceed the number of communication channels within the time limit we work with in our queries.

5.1 Syntax in UPPAAL SMC

A template that will be dynamically spawned must be declared as a *dynamic* template. This is done in the global declaration of the UPPAAL model using the dynamic keyword. The declaration for the Client template would for instance be dynamic Client (int id). The template takes one parameter id. Parameters to spawnable templates are restricted to be pass-by-value parameters or a reference to a broadcast channel. The reasoning behind this restriction is that templates may cease to exist—invalidating any references to its local variables that it could have passed on to spawned templates.

The actual behavior of a spawnable template is defined as usual in the editor. Note, however, that there must be a correspondence between the parameters defined in the dynamic declaration and the definition. In the Client example this means that the parameters both in the dynamic declaration and the definition must be int id.

Spawnable templates may be spawned by any template during a transition using the **spawn** keyword. For instance, adding **spawn** Client (2) to the update expression of an edge will spawn an instance of the template Client with parameter 2. Obviously, there must be parameter compatibility between the actual and the formal parameters.

A spawnable template can tear itself down during a transition. This is expressed by adding the *exit*() expression to the update of an edge.

5.2 Extensions for queries

Having extended the modeling language of UPPAAL SMC to allow dynamically spawning templates, we also need an extended specification formalism.

For the statically defined components specifications are made as described in Sect. 3. For the dynamically created components of the system three additional constructions are available:

forall (i : T) (q),

exists (i : T) (q) and

sum (i : T) (a),

that may be used anywhere in a specification.

The predicate forall (i : T) (q) asserts that q is true for all the dynamically created instances of T. The name i may be used anywhere in q to refer to the variables of the instances of T, i.e., the name i is temporally bound to the instances of T while evaluating q. The construct exists (i : T) (q) is the dual of forall.

Example 6 Returning to the server example from before, we may consider the probability that a client is not served for 5 time units, i.e., that it is working in the Wait location for 5 time units. In the extended specification formalism this can be checked using the query:

Pr(<>[0,20] (exists (c : Client)
  (([ ][0,5]c.Wait))))

The expression sum (i : T) (a) can be used in arithmetic expressions and simply evaluates a for all the instances of T. In the Server example, we can for instance count the number of clients that are waiting for a connection with the expression sum (c : Client) (c.Wait). The sum construction can also be used to count the number of active clients sum (c : Client) (1). An optimized version of this is available as numOf(Client). In Fig. 14, we show one simulation, 100 time units long, where we observe these two expressions.

The sum operator is useful for computing aggregate data about all components of a given type but cannot give the exact value of each component. For instance, sum cannot be used to plot the location of each client. If this is wanted UPPAAL SMC supports the query:

simulate 1 [<=100] {foreach (c : Client)
  (3*C.id+c.Wait+2*c.Work)}

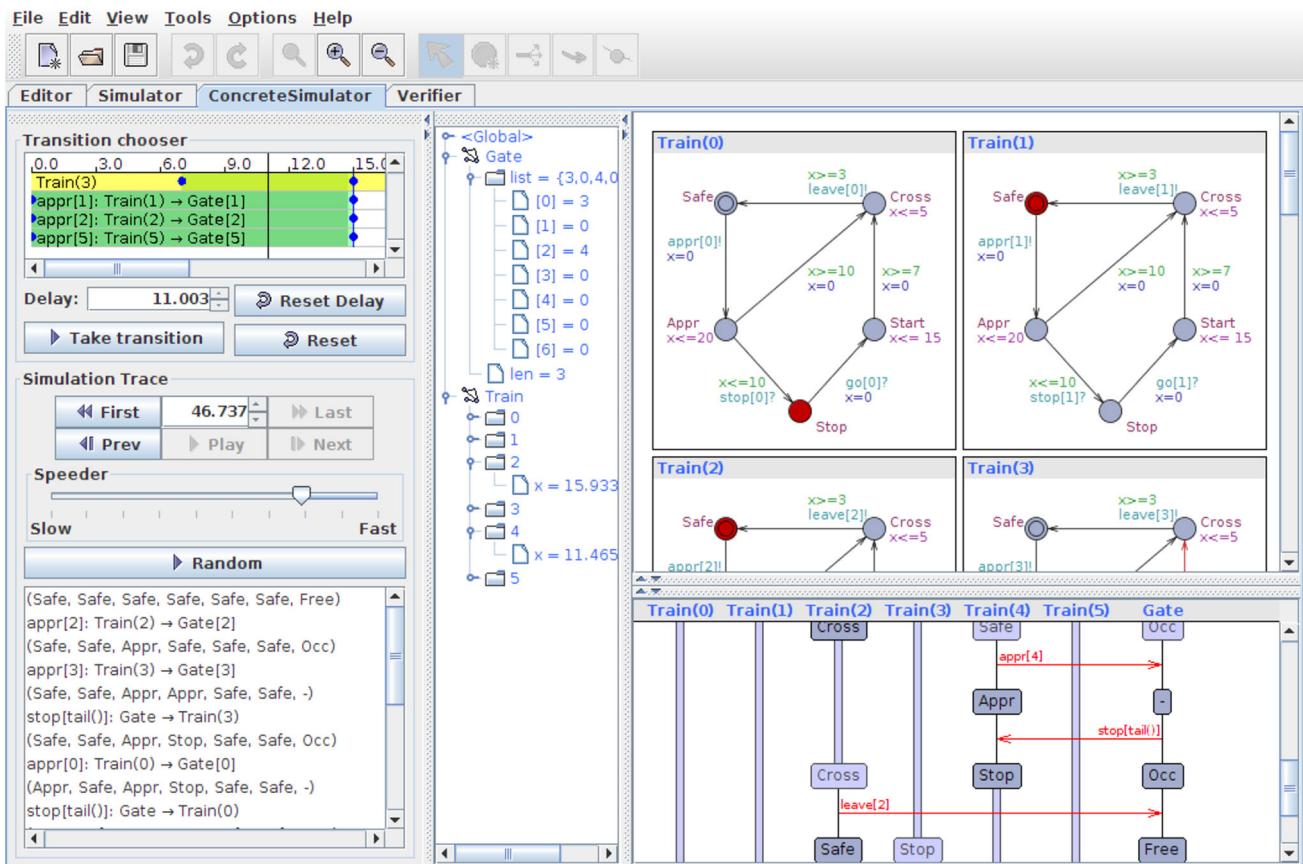The foreach statement is here used to tell the plotting facility of UPPAAL SMC to plot the expression

Fig. 16 The concrete simulator in UPPAAL

(3*c.id+c.Wait+2*c.Work) for each of the dynamically instances of Client. The actual expression is just a smart way to obtain a "Gantt-like" chart of each client. The result of the query is shown in Fig. 15 where each colored line correspond to a client.

6 Graphical interface

We focus in this section on the main features of the interface related to SMC. For a more complete overview of the interface the reader is referred to [4].

Overview The graphical interface of UPPAAL is divided into an editor, two simulators, and a verifier. The editor serves the purpose to define the automata and declaration of variables and functions. The verifier is used to specify and check different queries, and to get the results. Then there are two simulators, one is the well-known symbolic simulator that has been available in UPPAAL since the birth of this interface. The second simulator is a concrete simulator that was originally used in UPPAAL- TIGA. This simulator allows the user to simulate a system with concrete values of clocks, which is more intuitive than with the symbolic simulator. This simulator is shown in Fig. 16. The choice of transition is situated

in the upper-left corner. The user chooses with one click a transition (vertical choice) and a delay (horizontal choice). The simulator shows the automata and a message sequence chart on the right. On the lower left corner is the trace corresponding to the current simulation. The central view shows the variables and the user can show and hide variables in different scopes. In the example, only the clocks of Train(2) and Train(4) are shown.

The concrete simulator also supports Gantt chart visualization of the interactive concrete trace. Figure 17 shows a sample use case of Gantt chart for the train-gate example. The chart is defined in system declarations (Fig. 17a), where each chart line is defined by a statement separated by a semicolon. Each statement consists of a line label (e.g., gate and train) and a comma-separated list of predicates implying color numbers. For example, a line gate is painted in color #0 (red) whenever Gate.Occ is true and in color #1 (green) whenever Gate.Free. The colors are mixed when the corresponding predicates are true at the same time. It is also possible to define a chart line for a whole range of discrete values at once, like the parameterized definition of train(i:id_t), where the temporary variable i has a range of type id_t. For example, the first 32 colors can be rendered by the following definition: **gantt** { C(i:int[0,31]): true - > i; }.
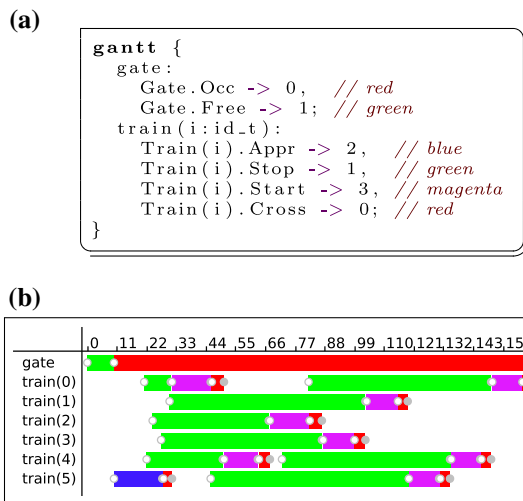
(a)

```
gantt {
  gate:
    Gate.Occ  -> 0,  // red
    Gate.Free -> 1;  // green
  train(i:id_t):
    Train(i).Appr  -> 2,  // blue
    Train(i).Stop  -> 1,  // green
    Train(i).Start -> 3,  // magenta
    Train(i).Cross -> 0;  // red
}
```

(b)

Fig. 17 Gantt chart, **a** definition in system declarations, **b** trace visualization in concrete simulator

SMC options Under the menu *Options* the user can choose *Statistical parameters*. This opens the window shown in Fig. 18.

- $-\delta$ and $+\delta$: When testing for hypothesis of the form $Pr(\varphi) \geq \theta$, the algorithm behind tests for two hypothesis. They are 1) $H_0 : Pr(\varphi) \geq \theta + \delta_+$ and 2) $H_1 : Pr(\varphi) \leq \theta - \delta_-$. These parameters define the *region of indifference*.
- α and β: α and β are used for hypothesis testing. The probability of accepting H_1 instead of H_0 is α and conversely for β. In the case of probability evaluation, α is also used and it is then the probability to be outside the result interval of probability. For probability comparison, the use of α and β is the same as for hypothesis testing.
- ε is the uncertainty for probability evaluations. The tool evaluate some probability μ and outputs the result $[\mu - \varepsilon, \mu + \varepsilon]$.
- u_0 and u_1 are the lower and upper bounds used in probability comparison. Similarly to hypothesis testing, the algorithm tests two hypotheses: $H_0 : \frac{Pr(\varphi_1)}{Pr(\varphi_2)} \geq u_1$ and $H_1 : \frac{Pr(\varphi_1)}{Pr(\varphi_2)} \leq u_0$. These parameters define the region of indifference for comparing probabilities.
- Histogram parameters: If the bucket width is set to a positive value, its value determines the width of the bars in the histogram and the number of bars depends on the range of the obtained results. Otherwise if the bucket count is positive then the number of bars is set to this value and the width of the bars depends on the range of the obtained result. Otherwise if both parameters are set to zero (default), the number of bars in the histogram is set to the square root of the number of runs used to obtain the graph.

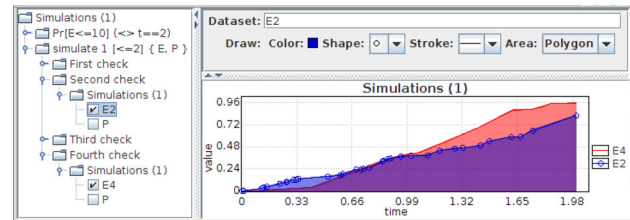**Fig. 18** The statistical parameters from the options menu

Fig. 19 Visual data comparison in the plot composer

- Trace resolution: When computing a simulation using the simulate query, the tool filters out the data on-the-fly and retains points that are distinguishable w.r.t. a certain resolution when plotted on a screen. This parameter controls the maximum width of the plot in pixels.
- Discretization step: This is used for integration when ODEs are used in the model. We note that defining rates as constants does not qualify as ODE, but having x'==y does.

Plotting and composing Most of SMC queries also provide quick result visualization in a form of data plots accessible in the Verifier by right clicking on a selected property and choosing one of the available plots from a pop-up menu. Simulation queries display all the requested trajectories in one plot with different colors assigned to various expressions. Statistical queries result in a number of different histograms showing the data scattered along time, cost or discrete transitions horizontal axis. The displayed plot elements (like title, legend, transparency, comments and logarithmic scale) can be customized by right clicking on the plot and choosing appropriate items from a pop-up menu. The plotted data can be exported as either a picture or a text file using the same plot pop-up menu. The size of the exported plot can be customized by resizing the plot window. Note that larger window will result in smaller fonts when rescaled for inclusion into a document, so smaller window will result in fewer details but clearer picture with larger fonts. The dark-colored areas are printer-friendlier when the plot is brightened by choosing Areas/Bright in the plot pop-up menu.

Different data can also be contrasted and compared in one plot using the Plot Composer from the Tools menu. Figure 19 shows a sample Plot Composer window with data from several verifications already loaded. The bottom panel on the right shows the resulting plot and the data is organized in the tree on the left. Each verification data is appended to the tree to its corresponding query. For example, simulate query has been checked four times and each result contains one plot with two datasets. The data can be added to the composite plot by ticking its checkbox and its drawing properties can be customized in the top-right panel when it is selected in the tree. For example E2 and E4 are ticked in Fig. 19 and E2 is selected and drawing properties can be changed. The main plot attributes like the title and the labels of both axis can be changed by selecting the root node and changing its properties in the upper panel on the right. It is also possible to edit several composite plots at the same time by invoking Plot Composer several times from the Tools menu.

7 Modeling tricks

7.1 How to convert channel synchronizations into broadcast synchronizations

Problem It is common that a user wants to analyze performance of a given model previously model checked with UPPAAL. This model may contain ordinary channel synchronizations that work by hand shake. The problem is that the SMC extension does not support them as explained in Sect. 2. Here, we present a translation to convert these models so that they can be analyzed by UPPAAL SMC.
Translation We distinguish three cases: the basic simple one-to-one synchronization, the one-to-any synchronization, and a problematic case.

The common simple case is of one process synchronizing with exactly one other process on a channel as shown in Fig. 20. The sender in state A may have an invariant or not. The receiver in state Loc2 does *not* have an invariant. The synchronization may be guarded by, resp. g1() and g2(), for resp. the sender and the receiver. To convert this model, the user should redeclare the channel a as broadcast, move the guard of the receiver to the sender,[2] and make the actual location visible from the sender using a simple encoding with the extra integer variable recvLoc. Other encodings may be used, e.g., with booleans, but the integer presents the advantage to keep the translation of several synchronizations simple. The integer allows the user to map each location to a unique value that is used by the sender to allow the synchronization only in the right state. The example illustrates the update of this

---

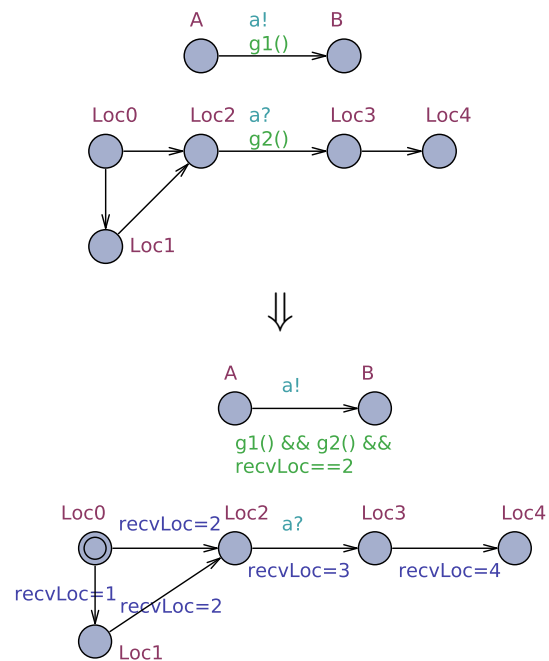[2] This may require moving local variables to the global scope to make the state visible.

Fig. 20 Basic case of a one-to-one channel synchronization and its translation to a broadcast channel synchronization

variable for some other peripheral locations Loc0, Loc1, and Loc4.

The second more general case is of one process synchronizing with one process out of several ones. There is a choice of one-to-any synchronization shown in Fig. 21. Here as well, the receiver is in a location without invariants. In this case, the same principle as the simple case is used with in addition a renaming of the channel. The initial transition in the sender has a copy with a unique channel name for each possible synchronization that is possible in the original model. Each copy uses the right associated guard and looks up the state of the right process. In the example, we illustrate with the use of an array a generic encoding where there would be several instances of the same template for the receiver. If the guards g2() and g3() are generic or depend on some id used to instantiate the receivers, the *select* construct can be used, in which case the original transition is not copied and the channel a is renamed as a[id] with an array.

The last case is the problematic one where a receiver has an invariant as shown in Fig. 22. Any translation of this model will violate the independent progress condition because here a receiver would force another sender process to synchronize. Not synchronizing would result in a deadlock. We note that if there is an output from that location, i.e., some b! synchronization, then there is no problem.

The last technical detail to take care of is to add exponential rates to the locations without invariants and that have output synchronizations (or *tau* transitions). This is the rate of the exponential distribution used for picking delays.
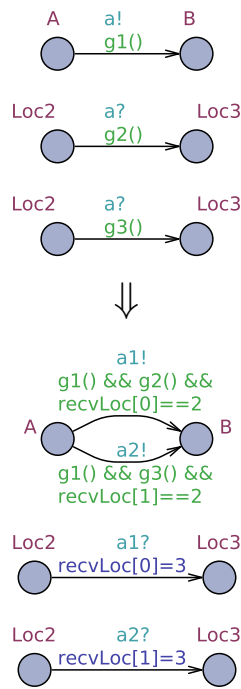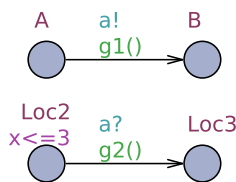
Fig. 21 Extended case of a one-to-any channel synchronization (only two here) and its translation to a broadcast channel synchronization

Fig. 22 Problematic case where the translation to broadcast channel is not possible

7.2 How to encode custom distributions

Problem Sometimes, the default uniform or exponential distributions available in UPPAAL SMC are not enough. The user needs a simple way to encode any distribution into the model to generalize the ones illustrated in Fig. 2.

Encoding The pattern for encoding general distributions is given in Fig. 23. The principle is that upon entry of a given location Wait where the actual custom delay is to take place, the actual delay is computed and stored into a clock delay. The function f() that computes this delay returns a floating-point value of type double. The automaton will then delay for this amount and take the transition. The location Wait has its invariant set to x $<=$ delay *and* delay'==0. The clock delay is used here only for storage. This technique is similar to the one used for computing stochastic simulations in Modest [27].

Implementation of f() The function that computes the delay may use the *random*(n) function with n being a floating-point value. The function returns a number in $[0, n[$ with a uni-
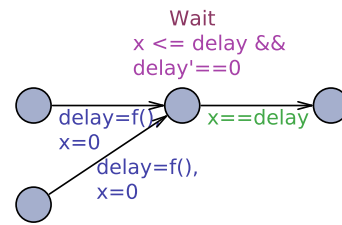

Fig. 23 Result from modeling a Gaussian distribution
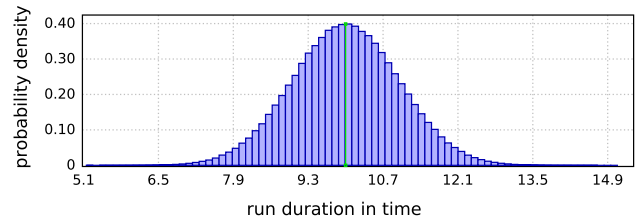

Fig. 24 Pattern for custom delay distributions

form distribution. This can then be transformed to return a delay with another distribution. We note that the function may keep a state as well, by storing what it wants into global variables (also of type double), which allows the encoding of virtually any distribution. For example, to generate random numbers according to a normal distribution using the Box-Müller method, we can use the following function:

```
const double PI = 3.14159265358979323846;
double stdNormal() { // N(0, 1)
  return sqrt(-2*ln(1-random(1))) *
    cos(2*PI*random(1));
}
double Normal(double mean, double stdDev) {
  return mean + stdDev * stdNormal();
}
double f() { // N(10.0, 1.0)
  return Normal(10.0, 1.0);
}
```

Listing 1: Normal distribution generating functions

The distribution obtained is shown in Fig. 24 together with the parameters used.

Remark The reader may wonder why the pattern proposes to use a clock for the variable delay instead of a variable of type double. In fact, it is possible to use double, which saves the trouble of setting its rate to 0. However, the performance of the model checker may drop. In its current implementation, UPPAAL SMC uses a fined-grained discretization if guards or invariants contain a "general" floating-point expression. The syntax analyzer will not recognize that the discretization is not needed in this case. Using clocks alleviates the problem.

7.3 How to model physics

Problem The formalism of UPPAAL SMC is stochastic hybrid automata so modeling physics is a simple matter of writing

the ODEs in the model. However, only first-degree deriva-
tives are allowed.

Modeling To model an n-degree derivative, the user should
use a clock variable for every intermediate derivative. This
is standard renaming technique used in other tools, e.g.,
Matlab. For example, instead of modeling y''==-9.81 for a
falling object, the user should declare y'==v and v'==-9.81.
Using different clocks or arithmetic expressions mixing dou-
ble typed variables is also supported.

7.4 How to model biochemistry

Problem Cyber-physical systems may involve chemical and
even biological processes and hence there is a need to eval-
uate the performance of control systems in such a context.
Suppose the reaction involves a mixed solution of materials
A and *B* and produce *C* and *D* with reaction speed of γ:

$$A + 2B \xrightarrow{\gamma} C + 3D$$
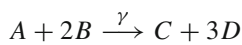
Here, we show how this reaction can be modeled as either
probabilistic or dynamical system. The containment of reac-
tions and other interactions can be modeled by adding addi-
tional locations, edges and channel synchronizations.

Stochastic model Figure 25 shows a stochastic model of the
reaction and its behavior. The discrete quantities (molecules)
of the materials involved are counted by the correspond-
ing integers A, B, C and D. The reaction rate is represented
by the double precision floating-point variable gamma. The
automaton in Fig. 25b captures the interaction between chem-
icals *A* and *B* in the following way:

- The automaton takes a discrete transition when the reac-
  tion happens.
- The reaction requires at least one molecule of *A* and at
  least two molecules of *B*, hence the edge is guarded by
  an expression A > 0&&B > 1.
- Each reaction consumes *A* and 2*B* and produces *C* and
  3*D*, hence the edge has the update A--, B-=2, C++, D+=3.
- In a well-mixed (homogeneous) compound the probabil-
  ity of a reaction is proportional to its speed γ and the
  probability of meeting the required three molecules (*A*,
  B and another *B*) in one place. The probability of reaction
  remains the same as long as the conditions (quantities and
  temperature) do not change, hence the reaction is a Pois-
  son process and the delay until the next reaction follows
  an exponential distribution with the rate gamma*A*B*B.

If there are more reactions, then they have to be modeled
by another parallel process. The trajectory of the quantities
can be inspected by the following query:
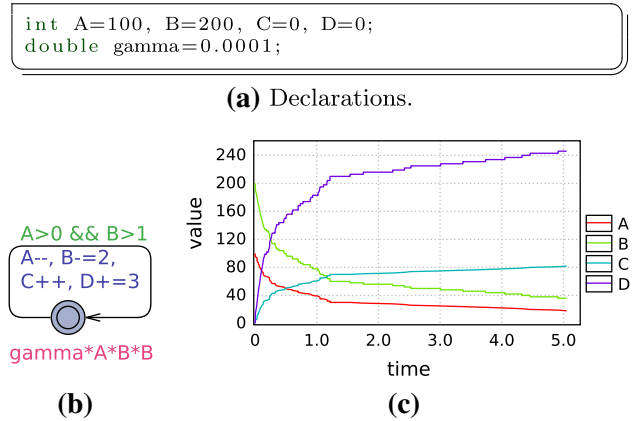simulate 1[<= 5]{A, B, C, D}. The resulting plot is shown

(a) Declarations.

A>0 && B>1

A--, B-=2,
C++, D+=3

gamma*A*B*B

(b)

(c)

Fig. 25 Stochastic model and its behavior, **a** declarations, **b** automa-
ton, **c** simulation (color figure online)

(a)

(b)

A>0 && B>1
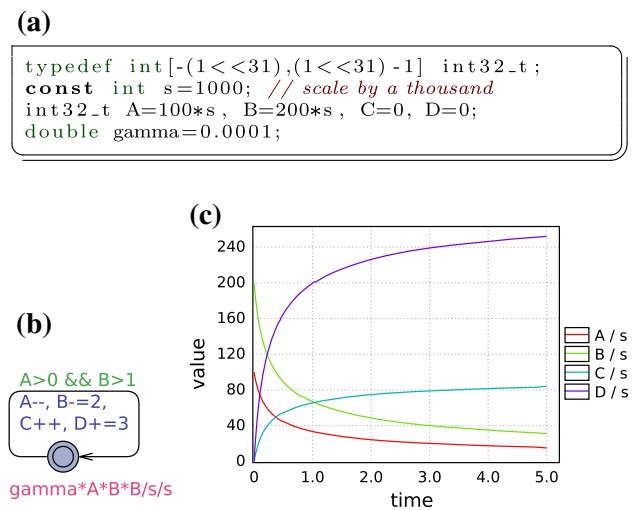
A--, B-=2,
C++, D+=3

gamma*A*B*B/s/s

(c)

Fig. 26 Scaled stochastic model and its scaled behavior, **a** scaled dec-
larations, **b** scaled rate, **c** scaled trajectories (color figure online)

in Fig. 25c: *A* and *B* are slowly decaying, replaced by *C*
and *D*. We notice that the trajectory is jittery and can be
slightly different with every new simulation due to proba-
bilistic nature of the stochastic process and relatively small
amounts of molecules. The trajectories are smoother when
quantities are much larger and approach the limit of the con-
tinuous dynamics.

Scaling Usually, chemical reactions involve huge numbers
of molecules with different orders of magnitude and thus
some scaling of dimensions may be desired. Note that if
the quantities are scaled by 1,000, then the exponential rate
gamma*A*B*B has to be scaled by 10^6 (while the dynamical
coefficients are scaled by 10^9) and thus it is very easy to over-
flow the default range of int. Figure 26 shows the same model
but with molecule quantities scaled by 1,000. The simulated
trajectories are divided by s back down to a comparable scale
as in previous and next example. The simulated behavior is
smoother and closer to the dynamical model (shown next).

The default integer range is rather small ($\pm 2^{16}$), thus one may need to broaden it by defining a custom range. UPPAAL supports integer ranges up to 32 bits, hence the type declaration typedef int[$-(1 << 31)$, $(1 << 31) - 1$] int32_t; corresponds to a range of signed 32 bit integer. The range can be expanded further to a double precision floating point, but note that its precision is limited to 52 bits ($\approx 4.5 \times 10^{12}$) and hence beyond that point minor increments (like $+1$) will not affect the variable value anymore.

Dynamical model The same reaction can be rewritten using a set of differential equations describing the rate of change of the quantities:

$$\begin{cases} \dfrac{d[A]}{dt} = -\gamma \cdot [A] \cdot [B]^2 \\[2mm] \dfrac{d[B]}{dt} = -\gamma \cdot [A] \cdot [B]^2 \cdot 2 \\[2mm] \dfrac{d[C]}{dt} = \gamma \cdot [A] \cdot [B]^2 \\[2mm] \dfrac{d[D]}{dt} = \gamma \cdot [A] \cdot [B]^2 \cdot 3 \end{cases}$$

The idea here is that the rate of change in quantities is proportional to the speed of reaction and concentration of materials. The contribution to various materials is then scaled by coefficients from the original reaction. We have one equation per each material mentioned. If there are more reactions then their contributions can be added up to the same system of differential equations either as separate extra terms or a separate equation for each new chemical. Figure 27 shows the dynamical model and its behavior. The quantities are captured by dynamical clock variables A, B, C and D and the same reaction coefficient gamma. The differential equations are then typeset as a single invariant of derivative expressions in Lagrange's prime notation (Fig. 27b). We also added an escape transition if/when the quantity of *A* reaches zero, i.e., the reaction stops. The trajectories can be inspected by the same simulation query as previously and the result is shown in Fig. 27c. Notice that the trajectory is smoother, very close to the scaled-up stochastic simulation, and is the same every time (deterministic), because ordinary differential equations have one fixed solution for the same initial conditions. Some ODE systems might require tuning the discrete integration step in the Statistical parameters from the Options menu: the smaller the step the more precise simulation is, but it is also computationally more expensive. Stiff systems may require smaller integration steps. A more complicated biochemical model can be found in a study of a circadian rhythm genetic oscillator [20,24].

7.5 How to obtain distributions over costs

When the user checks queries to evaluate probabilities, e.g., Pr[<= 100](<> Proc.Goal), UPPAAL SMC keeps track of
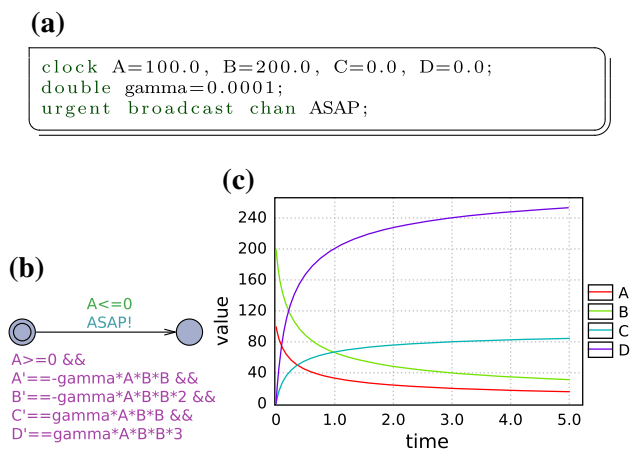

Fig. 27 Dynamical model and its behavior, **a** declarations, **b** automaton, **c** simulation (color figure online)
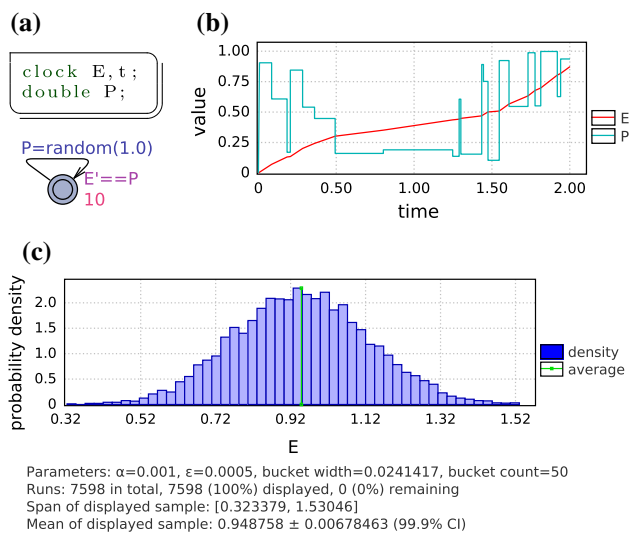

Fig. 28 Cost estimation in terms of energy, **a** model, **b** trajectories of energy and power, **c** estimated energy probability distribution (color figure online)

when the runs satisfy the specified goal state and uses this information to build a frequency histogram. Specifically, what is counted is the number of runs that were satisfied at a given "time" as defined by the bound of the run. When no explicit variable is used, e.g., <= 100, the plot is the count of satisfied runs as a function of time, discretized in the histogram bars (so in fact in function of time intervals). When a clock variable is used, the plot is in function of this variable. Alternatively, the runs can be bound by number of discrete steps of the form # <= 100.

Now suppose that we want to estimate a cost expressed as some energy consumption. To illustrate this, let us consider the example in Fig. 28a. In this model, a random power level is chosen stochastically and the corresponding energy consumption is integrated by UPPAAL SMC. The evolution of the energy is naturally expressed by the equation E'==P.

Figure 28b shows one stochastic simulation bounded by two time units obtained with the query simulate 1[<= 2]{E, P}. Every run will have its own energy consumption. The question is to know the mean of the energy consumption and its distribution over runs bounded by two time units. To obtain this we check the query Pr[E <= 10](<> t == 2). The trick is that first we bound the actual energy by a *high enough* bound that covers the reachable range for all runs. It could be E <= 1000 if the user is unsure. Second, the goal state is the time bound that will be reached since time progresses.[3] The result probability is one but this is not the point. The point is the distribution generated by this query. UPPAAL SMC will record "when" (in function of the bound) the runs reach the goal, here t==2. We obtain now a distribution of energy consumption on runs bounded by two time units as shown in Fig. 28c.

Remarks If the suggested query is checked with the default settings the obtained histogram will have poor precision because UPPAAL SMC does not need many runs to conclude that the result probability is one. The user should increase the precision by changing the SMC options as described in Sect. 6. Specifically, Fig. 28c was obtained from 7598 runs using $\alpha = 0.001$ and $\varepsilon = 0.0005$.

It is also possible to estimate discrete costs even though the tool does not support integers as bounds. Users can use clocks for this purpose by maintaining their rates to zero and updating them manually. For example, if c is a counter, then it is declared as a clock. Then the user adds one process with one location and no transition with the invariant c'==0. Finally, the increment c = c + 1 is used wherever necessary and the bound c <= 100 can now be used.

7.6 How to model custom discretizations

Problem Sometimes users want to use a custom integration method or want to change the integration granularity at the level of locations. UPPAAL SMC uses a global time step when it detects that some integration is needed. It may be better for performance or precision to change this step depending on the locations and the type of equation to integrate.

Modeling The modeling trick consists of using a "high" exponential rate on the locations where the manual discretization is needed. The tool will then take small delay steps, albeit random according to an exponential distribution with high rate, which allows for custom discretization. Figure 29 shows an example of the temperature of a room that can have a heater turned on or off.[4] The value of RATE controls the precision. The functions for cooling and heating are depicted in Listing

---

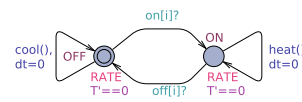[3] UPPAAL SMC detects Zeno runs and rejects models producing them.

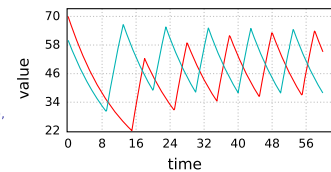[4] The actual controller is not important for this example and is not given here.

```
clock T = T0[i], dt;
void cool()
{
  T = T - (T*dt)/KCOOL;
}
void heat()
{
  cool();
  T = T + KHEAT*dt;
}
```

(a)

(b)   **(c)**

Fig. 29 The temperature of a heated room with a manual discretization using a high exponential rate RATE, **a** variable and function declarations, **b** the model, **c** temperature trajectories

29a. The value of the clock dt is the time elapsed and is used for the integration. KHEAT and KCOOL are constants used in the model. The result of a simulation is shown in Fig. 29. This manual encoding replaces, resp., T'==-T/KCOOL and T'==KHEAT-T/KCOOL for, resp., cooling and heating. The example also illustrates a recent new feature of the language, namely initializers for clocks with the declaration T = T0[i], where T0 is declared as const double T0={70.0,60.0}.

8 Conclusion

This paper presented UPPAAL SMC as an efficient tool for evaluating performance properties of stochastic hybrid systems. The modeling language has been extended to handle dynamical behaviors, discrete probabilities, a stochastic interpretation for timed delays and even dynamic process creation—far beyond analytically tools reach. Most importantly the old UPPAAL models require only small changes to benefit also from UPPAAL SMC features, thus it is straightforward to gain also performance measures in addition to firm results. The paper also includes tricks for handling more problematic corner cases to satisfy UPPAAL SMC assumptions and in particular how to transform handshake synchronization to broadcast synchronization. The query language has been expanded to request simulation trajectories, compute probabilistic aspects and evaluate weighted MITL formulas.

In the future, we intend to include better ODE solvers to improve dynamical simulations and improve the interactive concrete run simulator including a Gantt chart visualization of a run.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)

2. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM **43**(1), 116–146 (1996)

3. Boyer, B., Corre, K., Legay, A., Sedwards, S.: Plasma-lab: a flexible, distributable statistical model checking library. In: QEST, pp. 160–164 (2013)

4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. Lecture Notes in Computer Science. pp. 200–236 (2004)

5. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Developing uppaal over 15 years. Softw. Pract. Exp. **41**(2), 133–142 (2011)

6. Bulychev, P., David, A., Larsen, K.G., Legay, A., Li, G., Poulsen, D.B.: Rewrite-based statistical model checking of wmtl. In: Runtime Verification, vol. 7687 of LNCS, pp. 260–275 (2012)

7. Bulychev, P., David, A., Larsen, K.G., Legay, A., Li, G., Poulsen, D.B., Stainer, A.: Monitor-based statistical model checking for weighted metric temporal logic. In: Nikolaj, B., Voronkov, A. (eds.) 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, vol. 7180 of LNCS, pp. 168–182. Springer (2012)

8. Bulychev, P.E., David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B.: Checking and distributing statistical model checking. In: NASA Formal Methods, vol. 7226 of Lecture Notes in Computer Science, pp. 449–463. Springer (2012)

9. Behrmann, G., David, A., Larsen, K.G., Yi, W.: Unification & sharing in timed automata verification. In: SPIN Workshop 03, vol. 2648 of LNCS, pp. 225–229 (2003)

10. Behrmann, G.: Distributed reachability analysis in timed automata. STTT **7**(1), 19–30 (2005)

11. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J.: Efficient guiding towards cost-optimality in uppaal. In: Margaria, T., Yi, W. (eds.) Proceedings of the 7th International Conference on Tools and Algorithms for the construction and analysis of systems, number 2031 in Lecture Notes in Computer Science, pp. 174–188. Springer (2001)

12. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.: Minimum-cost reachability for priced timed automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A. (eds.) Proceedings of the 4th International Workshop on Hybris Systems: Computation and Control, number 2034 in Lecture Notes in Computer Sciences, pp. 147–161 Springer-Verlag (2001)

13. Behrmann, G., Hune, T., Vaandrager, F.: Distributed timed model checking: How the search order matters. In: Proceedings of 12th International Conference on Computer Aided Verification, Lecture Notes in Computer Science, Chicago, Springer, Jul (2000)

14. Broy, M., Jonsson, B., Katoen, J-P., Leucker, M., Pretschner, A. (eds.): Model-based testing of reactive systems, advanced lectures the volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004, vol. 3472 of Lecture Notes in Computer Science. Springer (2005)

15. Behrmann, G., Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Efficient timed reachability analysis using clock difference d iagrams. In: Proceedings of the 12th Int. Conf. on Computer Aided Verificat ion, vol. 1633 of Lecture Notes in Computer Science. Springer (1999)

16. Clarke, E.M., Faeder, J.R., Langmead, C.J., Harris, L.A., Jha, S.K., Legay, A.: Statistical model checking in biolab: applications to the automated analysis of t-cell receptor signaling pathway. In: CMSB, LNCS, pp. 231–250 (2008)

17. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)

18. Chernoff, H.: A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. Ann. Math. Stat. **23**(4), 493–507 (1952)

19. Clopper, C.J., Pearson, E.S.: The use of confidence or fiducial limits illustrated in the case of the binomial. Biometrika **26**(4), 404–413 (1934)

20. David, A., Du, D., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Sedwards, S.: Statistical model checking for stochastic hybrid systems. In: Bartocci, E., Bortolussi, L. (eds.) HSB, vol. 92 of EPTCS, pp. 122–136 (2012)

21. David, A., Du, D., Larsen, K.G., Legay, A., Mikučionis, M.: Optimizing control strategy using statistical model checking. In: NASA formal methods, vol. 7871 of Lecture Notes in Computer Science, pp. 352–367. Springer (2013)

22. David, A., Jensen, P.G., Larsen, K.G., Legay, A., Lime, D., Søresensen, M.G., Taankvist, J.H.: On time with miniam expected cost

23. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Vliet, J.V., Wang, Z.: Statistical model checking for networks of priced timed automata. In: FORMATS, LNCS, pp. 80–96. Springer (2011)

24. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Sedwards, S.: Runtime verification of biological systems. In: Margaria, T., Steffen, B. (eds.), ISoLA (1), vol. 7609 of Lecture Notes in Computer Science, pp. 388–404. Springer (2012)

25. David, A., Möller, M.O., Yi, W.: Formal verification of UML statecharts with real-time extensions. In: Kutsche, R.-D., Weber, H. (eds.) Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, vol. 2306 of LNCS, pp. 218–232. Springer (2002)

26. Henriques, D., Martins, J.G., Zuliani, P., Platzer, A., Clarke, E.M.: Statistical model checking for markov decision processes. In: Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, Sept 17–20, 2012, pp. 84–93. IEEE Computer Society (2012)

27. Hartmanns, A.: Model-checking and simulation for stochastic timed systems. In: Bernhard, K.A., De Boer, F.S., Marcello M.B. (eds.) FMCO, vol. 6957 of Lecture Notes in Computer Science, pp. 372–391. Springer (2010)

28. Henzinger, T.A., Ho, P.-H.: Algorithmic analysis of nonlinear hybrid systems. In: Wolper, P. (ed.) Computer Aided Verification, 7th International Conference, Liège, Belgium, July, 3–5, 1995, Proceedings, vol. 939 of Lecture Notes in Computer Science, pp. 225–238. Springer (1995)

29. Hendriks, M., Larsen, K.G.: Exact acceleration of real-time model checking. In: Asarin, E., Maler, O., Yovine, S. (eds.) Electronic Notes in Theoretical Computer Science, vol. 65. Elsevier Science Publishers (2002)

30. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) Verification, Model Checking, and Abstract Interpretation, vol. 2937 of Lecture Notes in Computer Science, pp. 73–84. Springer, Berlin, Heidelberg (2004)

31. Hoeffding, W.: Probability inequalities for sums of bounded random variables. J. Am. Stat. Assoc. **58**(301), 13–30 (1963)

32. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A bayesian approach to model checking biological systems. In: CMSB, vol. 5688 of LNCS, pp. 218–234. Springer (2009)

33. Jégourel, C., Legay, A., Sedwards, S.: Importance splitting for statistical model checking rare properties. In: CAV, vol. 8044 of Lecture Notes in Computer Science, pp. 576–591. Springer (2013)

34. Kwiatkowska, M.Z., Norman, G., Parker, D.: Prism 2.0: A tool for probabilistic model checking. In: Proc. of 1th Int. Conference on the Quantitative Evaluation of Systems (QEST), pp. 322–323. IEEE (2004)

35. Larsen, K.G., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., Romijn, J.: As cheap as possible: efficient cost-optimal reachability for priced timed automata. In: Berry, G., Comon, H., Finkel, A. (eds.) Proceedings of CAV 2001, number 2102 in Lecture Notes in Computer Science, pp. 493–505. Springer (2001)

36. Larsson, F., Larsen, K.G., Pettersson, P., Yi, W.: Efficient verification of real-time systems: Compact data structures and state-space reduction. In: Proc. of the 18th IEEE Real-Time Systems Symposium, pp. 14–24. IEEE Computer Society Press (1997)

37. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a Nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 134–152 (1997)

38. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: CAV, LNCS 3114, pp. 202–215. Springer (2004)

39. Theelen, B.D.: Performance modelling for system-level design. Ph.D. thesis, Eindhoven University of Technology, (2004) ISBN 90-386-1633-3

40. Wald, A.: Sequential tests of statistical hypotheses. Ann. Math. Stat. **16**(2), 117–186 (1945)

41. Younes, H.L.S.: Verification and planning for stochastic processes with asynchronous events. Ph.D. thesis, Carnegie Mellon (2005)

42. Yi, W., Pettersson, P., Daniels, M.: Automatic verification of real-time communicating systems by constraint-solving. In: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII, pp. 243–258, London, UK, UK, Chapman & Hall Ltd (1995)