Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

UPPAAL SMC Tutorial

Alexandre David[1], **Kim G. Larsen**[1], **Axel Legay**[2], **Marius Mikučionis**[1], **Danny Bøgsted Poulsen**[1]

[1] Department of Computer Science, Aalborg University, Denmark
[2] INRIA/IRISA Rennes, France

The date of receipt and acceptance will be inserted by the editor

Abstract. This tutorial paper surveys the main features of UPPAAL SMC, a new branch of the UPPAAL family that allows us to reason on networks of complex real-timed systems with a stochastic semantic. We expose the modeling features of the tool, and new verification algorithms that are applied to potentially complex case studies.

1 Introduction

Computer systems play a central role in modern societies and their errors can have dramatic consequences. Proving the correctness of computer systems is therefore a highly relevant activity, on which both industry and academics invest a considerable amount of effort. Among such techniques, one finds (1) *testing* [BJK$^+$05], the traditional approach that detects bugs by exercising the system with test cases, and (2) *formal methods*, e.g., model checking [CGP99], that are a more mathematical approach that can *guarantee* the absence of bugs. Both approaches have been largely deployed on complex case studies

Originally, formal verification was devoted to hardware and to software systems with discrete behaviors. However, over the past years, one has observed that real-time plays a central roles in systems, and that this feature should be taken into account in the verification process. Developping 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 [AD94], a natural model to capture real-time systems whose behaviors depends on clocks that can be reset. Among all the tools that have been developped 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 (network of) timed automata extended with integer variables, structured date taypes, and channel synchronization. The tool is jointly developed by Uppsala University and Aalborg University. It has been applied successfully in case studies ranging from communication protocols to multimedia applications (see [BDL04] and [BDL$^+$11] for concrete examples). The first version of UPPAAL was released in 1995 [LPY97]. Since then it has been in constant development. In the same spirit as any other professional model checker such as SPIN, UPPAAL proposes efficient data structures [LLPY97],a distributed version of UPPAAL [BHV00,Beh05], guided and minimal cost reachability [BFH$^+$01a,LBB$^+$01,BFH$^+$01b], work on UML Statecharts [DMY02], acceleration techniques [HL02], and new data structures and memory reductions [BLP$^+$99,BDLY03].

Unfortunately, timed automata is not a panacea. In fact, albeit powerful, the model is not expressive enough to capture behaviors of complex cyper physical systems. Indeed, the continuous time behaviors of those systems often relies on rich and complex dynamics as well as on stochastic behaviors. The model checking problem for such systems is in fact undecidable, and the best one could originally do in UPPAAL was to approximate those behaviors with timed automata.

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 features. Concretelly, in UPPAAL SMC, each component of the system is described with an automaton whose clocks can involve with different rates. Such rates being 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) [You05,SVA04], 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 statistic area (including sequential hypothesis testing or Monte Carlo simulation) in order 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, 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 [CFL+08, JCL+09, KNP04, DDL+12, JLS13], energy-centric systems [DDL+13], 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 for our results and tools) [BDL+12c, BCLS13, ?]. 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 planing and robotics.

This paper is a complete tutorial on Uppaal SMC. 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 dynamic 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 [AD94] used in the classical model-checking version of Uppaal [?]. For *individual TA components* the stochastic interpretation refines the non-deterministic choices between multiple enable transition 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 by uniform distributions in cases of time-bounded delays or by 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 (intially) the weight annotations on locations

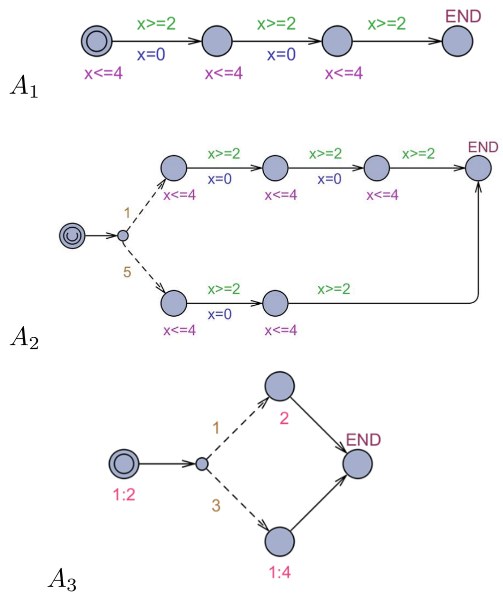
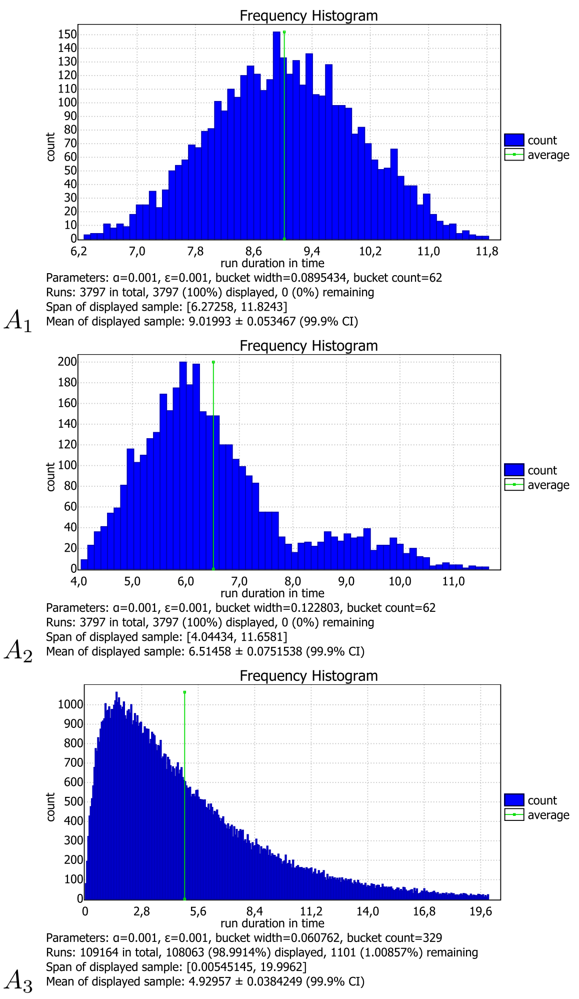


Fig. 1: Three stochastic timed automata

and edges, the END-locations in the three automata are easily seen to be reachable within the time-intervals $[6, 12]$, $[4, 12]$ and $[0, +\infty[$. Now the stochastic interpretation of the three TAs provides a refinement of these intervals in terms of distributions of the reachability time. For A_1, the delay of the three transitions will all be (automatically) resolved by independent, uniform distributions on $[2, 4]$. Thus the overall reachability time is given as the sum of three uniform distributions as illustrated in Fig. 2 (a). For A_2, the delay distributions determined by the upper and lower path to the END-location are similarly given by sums of uniform distributions. Subsequently the distribution of the overall delay is obtained by a weighted combination (1/6 to 5/6) of these as illustrated in Fig. 2 (b). Finally, in A_3 – in the absence of invariants – delays are choosen according to exponential distributions with user-supplied rates (here 1/2, 2 and 1/4). In addition, after the initial delay a discrete probabilistic choice (1/4 versus 3/4) is made. The resulting distribution of the overall reachability time is given in Fig. 2 (c).

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 that of DTMC and CTMC. 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



Parameters: α=0.001, ε=0.001, bucket width=0.0895434, bucket count=62
Runs: 3797 in total, 3797 (100%) displayed, 0 (0%) remaining
Span of displayed sample: [6.27258, 11.8243]
Mean of displayed sample: 9.01993 ± 0.053467 (99.9% CI)

Parameters: α=0.001, ε=0.001, bucket width=0.122803, bucket count=62
Runs: 3797 in total, 3797 (100%) displayed, 0 (0%) remaining
Span of displayed sample: [4.04434, 11.6581]
Mean of displayed sample: 6.51458 ± 0.0751538 (99.9% CI)

Parameters: α=0.001, ε=0.001, bucket width=0.060762, bucket count=329
Runs: 109164 in total, 108063 (98.9914%) displayed, 1101 (1.00857%) remaining
Span of displayed sample: [0.00545145, 19.9962]
Mean of displayed sample: 4.92957 ± 0.0384249 (99.9% CI)

Fig. 2: Distributions of reachability time

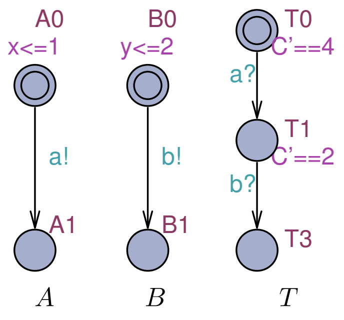


Fig. 3: An NPTA, $(A|B|T)$.

is assumed that these components are input-enabled, determistic (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 be broadcast to keep a clean semantic of non-blocked components that are racing against each other with the corresponding distributions.

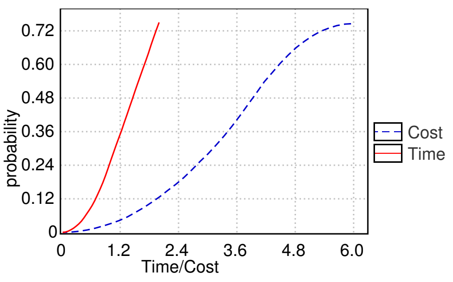


Fig. 4: Cumulative probabilities for `time` and `Cost`-bounded reachability of T_3.

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

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

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.

For components, as stated in the previous section, UPPAAL SMC applies uniform distributions for bounded delays and exponential distributions for the case where a component STA can remain indefinitely in a state. 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 NPTA of Fig. 3, A wins the initial race over B with probability 0.75.

As observed in [DLL$^+$11], 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 NSTAss 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.

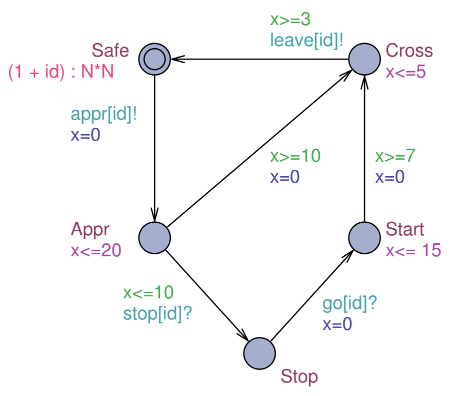
To illustrate the extended input language, we consider a train-gate example. This example is available in the distributed version of UPPAAL SMC. A number of trains are approaching a bridge on which there is only one track. To avoid collisions, a controller stops the trains. It restarts them when possible 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 5(a) 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 and synchronize with `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 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. This happens in `Appr`, `Cross`, and `Start`, e.g., it takes some time picked uniformly between 3 and 5 time units to cross the bridge. Figure 5(b) 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 is approaching while the bridge is occupied in `Occ`) or dequeue them when possible (when the bridge is free and some train is queued).

3 Query Language

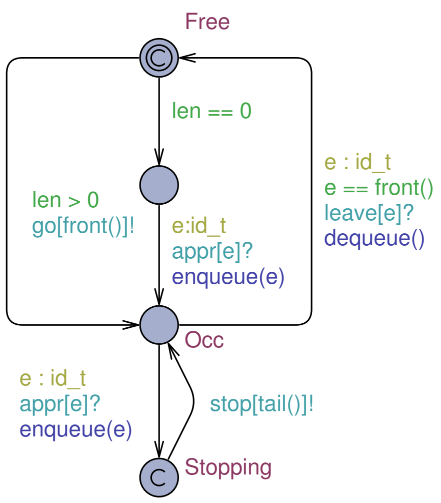
In addition to the standard model checking queries – i.e. reachability, invariance, inevitability and leads-to, which are still available – UPPAAL SMC provides a number of new queries related to the stochastic interpretation of timed automata. UPPAAL SMC allows the user to visualize the values of expressions (evaluating to integers or clocks) along 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



(a) Train.



(b) Gate controller.

Fig. 5: Templates for the train-gate example.

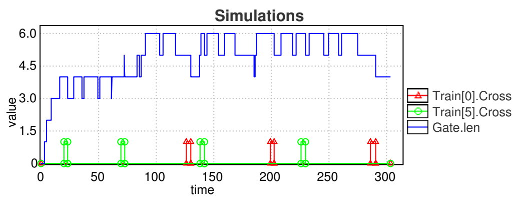


Fig. 6: Visualizing the gate length and when `Train(0)` and `Train(5)` cross on one random run.

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

This gives us the plot of Figure 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 of NSTAs, we use a weighted extension of temporal logic MITL [AFH96] expressing properties over runs [BDL+12a], defined by the grammar:

$$\varphi ::= ap \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{O}\varphi \mid \varphi_1 \mathsf{U}^{x}_{\leq d} \varphi_2$$

where ap is an atomic proposition, 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. For an NSTA M we define $\mathbb{P}_M(\psi)$ to be the probability that a random run of M satisfies ψ.

The problem of checking $\mathbb{P}_M(\psi) \geq p$ ($p \in [0,1]$) is undecidable in general [1]. For the sub-logic of cost-bounded reachability problems $\mathbb{P}_M(\Diamond_{x \leq C}\varphi) \geq p$, where φ is a state-predicate, x is a clock and C is bound, Uppaal SMC approximates the answer using simulation-based algorithms known under the name of statistical model checking algorithms (SMC). We briefly recap statistical algorithms permitting to answer the following three types of questions:

1. *Probability evaluation:*
   What is the probability $\mathbb{P}_M(\Diamond_{x \leq C}\varphi)$ for a given NSTA M?
2. *Hypothesis Testing:*
   Is the probability $\mathbb{P}_M(\Diamond_{x \leq C}\varphi)$ for a given NSTA M greater or equal to a certain threshold $p \in [0,1]$?
3. *Probability comparison:*
   Is the probability $\mathbb{P}_M(\Diamond_{x \leq C}\varphi_2)$ greater than the probability $\mathbb{P}_M(\Diamond_{y \leq D}\varphi_2)$?

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 qualitative questions (1 and 3), we shall use sequential hypothesis testing, while for the quantitative question (2) we will use an estimation algorithm that resemble the classical Monte Carlo simulation. The two solutions are detailed hereafter.

Probability Estimation This algorithm [HLMP04] computes the number of runs needed in order to produce an approximation interval $[p-\epsilon, p+\epsilon]$ for $p = Pr(\psi)$ with a confidence $1-\alpha$. The values of ϵ and α are chosen by the user and the number of runs relies on the Chernoff-Hoeffding bound. In Uppaal SMC we use the following query:

$$\texttt{Pr[}\mathit{bound}\texttt{](}\varphi\texttt{)}$$

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:

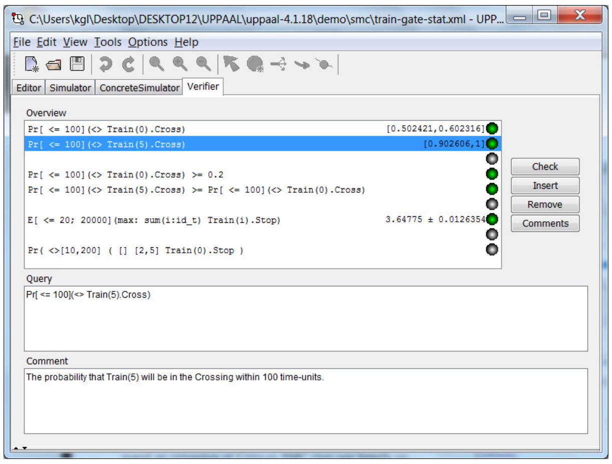
$$\begin{array}{l}\texttt{Pr[<= 100](<> Train(0).Cross)} \\ \texttt{Pr[<= 100](<> Train(5).Cross)}\end{array} \qquad (1)$$

Figure 7 shows how these (and other) queries are entered in the "Query" field of the Verifier tap 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 detialed information in terms of (cumulative, confidence interval, frequency histogram) probability distribution of the time-bounded reachability property, e.g. Fig. 8.

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

$$H : p = \mathbb{P}_M(\Diamond_{x \leq C}\varphi) \geq \theta$$

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


C:\Users\kgl\Desktop\DESKTOP12\UPPAAL\uppaal-4.1.18\demo\smc\train-gate-stat.xml - UPP...
File Edit View Tools Options Help
Editor Simulator ConcreteSimulator Verifier
Check
Insert
Remove
Comments

Overview

```
Pr[ <= 100](<> Train(0).Cross)                          [0.502421,0.602316]
Pr[ <= 100](<> Train(5).Cross)                                  [0.902606,1]

Pr[ <= 100](<> Train(0).Cross) >= 0.2
Pr[ <= 100](<> Train(5).Cross) >= Pr[ <= 100](<> Train(0).Cross)

E[ <= 20; 20000](max: sum(i:id_t) Train(i).Stop)         3.64775 ± 0.0126354

Pr( <>[10,200] ( [] [2,5] Train(0).Stop )
```

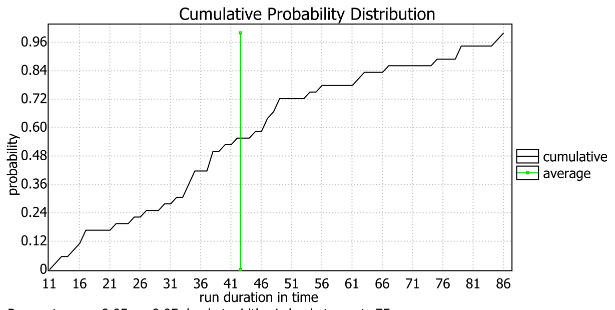
Query

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

Comment

The probability that Train(5) will be in the Crossing within 100 time-units.

Fig. 7: The Verifier of Uppaal SMC



Parameters: α=0.05, ε=0.05, bucket width=1, bucket count=75
Runs: 36 in total, 36 (100%) displayed, 0 (0%) remaining
Span of displayed sample: [11.0447, 85.2899]
Mean of displayed sample: 42.5735 ± 7.10607 (95% CI)

Fig. 8: The cumulative probability distribution of `Pr[<=`T`](<> Train(5).Cross)`.

against the alternative hypothesis:

$$K : p = \mathbb{P}_{\boldsymbol{M}}(\Diamond_{x\leq C}\varphi) < \theta$$

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* [Wal45]. 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:

$$\texttt{Pr[}\mathit{bound}\texttt{](}\varphi\texttt{)>=}\ \ p_0$$

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 φ is either `<>`q or `[]`q where q is a state predicate.

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. 7):

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

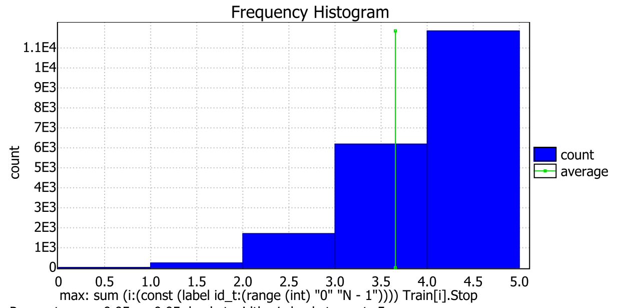
Within a number of runs significantly smaller that that of estimating the same probability (383 runs), this property may be confirmed. The number of runs needed by Walds sequential hypothesis testing method varies, e.g. posing the above query 5 times, the property was confirming within 66, 62, 65, 67, and 49 runs respectively with 5% level of significance.

Probability Comparison This algorithm, which is detailed in [DLL+11], exploits an extended Wald testing. In Uppaal SMC, we use the following query:

$$\texttt{Pr}[\mathit{bound}_1](\varphi_1) >= \texttt{Pr}[\mathit{bound}_2](\varphi_2).$$

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)`. Possing the query:

```
Pr[<=100](<>Train(5).Cross)
        >= Pr[<=100](<>Train(0).Cross)}
```



```
Parameters: α=0.05, ε=0.05, bucket width=1, bucket count=5
Runs: 20000 in total, 20000 (100%) displayed, 0 (0%) remaining
Span of displayed sample: [0, 5]
Mean of displayed sample: 3.65475 ± 0.0125406 (95% CI)
```

Fig. 9: Frequency histogram of maximum number of trains stopped within 20 time-untis.

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:

$$\texttt{E}[\mathit{bound}; N](\texttt{min}:\mathit{expr})$$

or

$$\texttt{E}[\mathit{bound}; N](\texttt{max}:\mathit{expr})$$

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.

Example 4. As an intersting 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. 9.

Full Weighted MITL Regarding implementation, we note that both of 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 [BDL+12b], we proposed an extension of Uppaal SMC that can handle arbitrary formulas of weighted MITL. Given a property φ, our implementation 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

[BDL+12a], the *exact* evaluation of whether the generated run satisfies a given weighted MITL formula is done on-line 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 `Pr(`ψ`)`, where

$$\begin{aligned} \psi ::= \;& \texttt{BExpr} \\ \mid\;& \psi\,\texttt{\&\&}\,\psi \mid \psi\,\texttt{||}\,\psi \\ \mid\;& \psi\,\texttt{U[a,b]}\,\psi \mid \psi\,\texttt{R[a,b]}\,\psi \\ \mid\;& \texttt{<>[a,b]}\,\psi \mid \texttt{[][a,b]}\,\psi \end{aligned}$$

$a, b \in \mathbb{N}$, $a \leq b$ 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 consequtive time-units somewhere in the time-interval `[10,10]`. 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 system, 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. Fig.10(a) shows the automaton for the heater. It turns itself on with a uniform distribution over time 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. 10(b-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

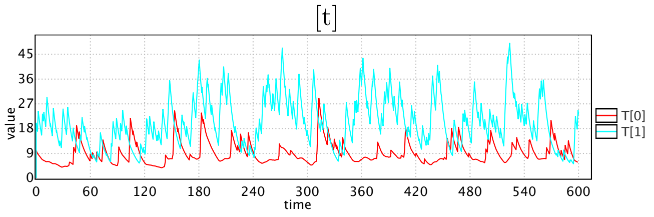


Fig. 11: Evolution of the temperatures of the two rooms.

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 arithmetics 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.

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* neither these clocks or the floating-point variables affect the control of the automata, i.e., such variables are inactive and used as costs.

4.2 Example

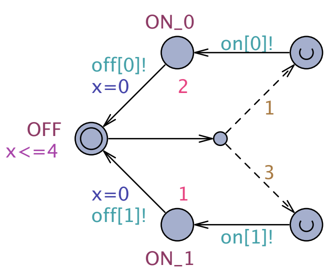
All the new queries of UPPAAL SMC described in Section 3 are available for stochastic hybrid systems. We illustrate this on by a number of queries related to the two-room example from the previous Section.

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

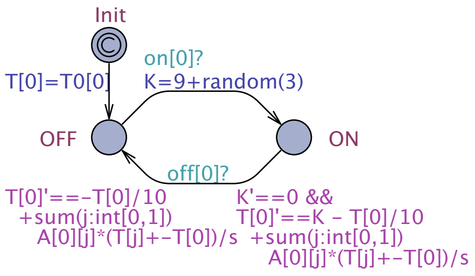
```
simulate 1 [<=600]{T[0],T[1]}
```

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. 11 shows that the heater is able to maintain the temperatures within (mostly) distinct intervals.

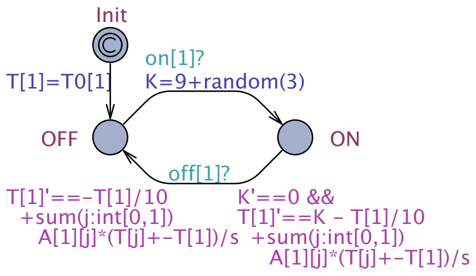
We can evaluate on a shorter time scale the probability for the temperature of `Room(0)` to stay below 30 and the temperature of `Room(1)` to stay above 5 with the queries



(a) stochastic heater.



(b) room 0.



(c) room 1.

Fig. 10: A simple two room example with an autonomous heater.

```
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 these so-called 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 with a confidence of 95%, we need 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.

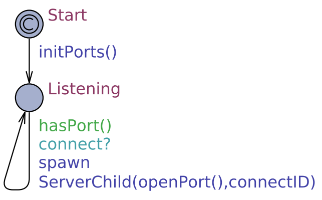
We can test the hypothesis that the heater is better at keeping the temperature of `Room(1)` above 8 than keeping the temperature of `Room(0)` below 20 by the following comparison query:

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

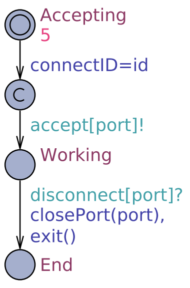
which is accepted in this case with 95% level of significance with just 258 runs.

5 Extension to Dynamic Creation of Processes

Computer systems are contrary to the assumption of networks of timed automata not statically encoded entities. Instead they 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 would be available for the resource manager to "start" whenever a spawn request was made in the model. A necessary assumption for modeling this resource manager



(a) Server



(b) ServerChild

Fig. 12: Modelling a server with dynamic spawning

is 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 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.

A prototypical example of a system exhibiting spawning is that of a server system. A server system usually consists of a main thread responsible of accepting connections from clients. The actual processing of the clients requests is handled by a thread spawned by the main thread. In Fig. 12 we show how this is modeled in Uppaal SMC.

In Fig. 12a is shown the automaton modeling the server thread listening for connections. The function initPorts is initialising an array used internally by the server. When a connection is attempted by a client (template not shown) the server immidiatly checks if it has communication channels available (using the aforementioned array). If succesful a ServerChild (Fig. 12b) is spawned using the Spawn ServerChild (..) construct. The port number this client should use to communicate with the client is passed by parameters to ServerChild (the `openPort`() function returns the port number as an integer). The second parameter to the ServerChild is the id of the client. The

```
dynamic ServerChild (int port, int id);
```

Fig. 13: Syntax for defining a spawnable template ServerChild with integer parameter port and id.

server child then accepts the connection (using the global variable connectID to identify the client) and awaits a disconnect request from the client i.e. in the model we abstract from the actual work requested by the client. When the client disconnects the ServerChild terminates with the exit() construct.

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.

In Fig. 13 the declaration for the ServerChild template is shown. The template takes two parameters port and id, where port is the port used for communcation with the client identified by 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 behaviour 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 ServerChild example this means that the parameters both in the dynamic declaration and the definition must be int port, ind id.

Spawnable templates may be spawned by any template during a transition using the spawn keyword. For instance, adding spawn ServerChild (5,2) to the update expression of an edge will spawn an instance of the template ServerChild with paramters 5 and 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 Section 3. For the dynamically created components of the system have three additional constructions are available:

$$\texttt{forall}(\texttt{t} : \texttt{T})(\texttt{q}),$$

$$\texttt{exists}(\texttt{t} : \texttt{T})(\texttt{q}) \text{ and}$$

$$\texttt{sum}(\texttt{t} : \texttt{T})(\texttt{a}),.$$

that may be used anywhere in a specification.

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

Example 6. Returning to the Server example from before, we may consider the probability that a ServerChild is not released for 10 time units i.e. that it is working in the Working location in 10 time units. In the extended specification formalism this can be checked using the query:

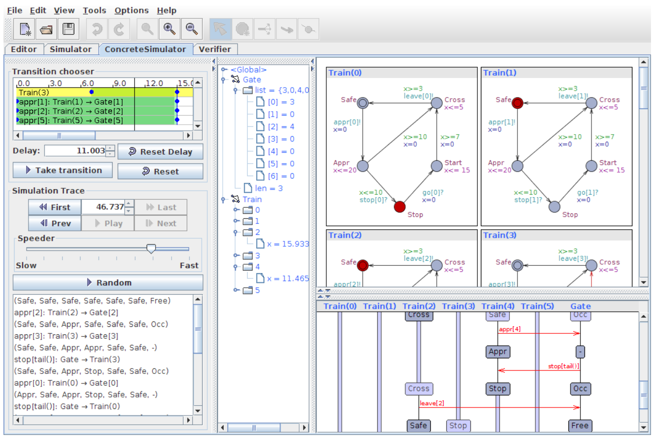
$$\texttt{Pr}(<> [0,10](\texttt{exists}(\texttt{s} : \texttt{ServerChild})(\\ ([][0,10]\texttt{s.Working}))))$$

sum(t : T)(a) is an expression that can be used in arithmetic expressions and simply evaluates a for all the instances of T. An interesting use of sum is to count the number of active components of a template: this is easily accomplished using sum(t : T)(1). An optimised version of this is also available as numOf(T).

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 refered to [?].

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. 14. 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



```
Train(3)
appr[1]: Train(1) → Gate[1]
appr[2]: Train(2) → Gate[2]
appr[5]: Train(5) → Gate[5]
```

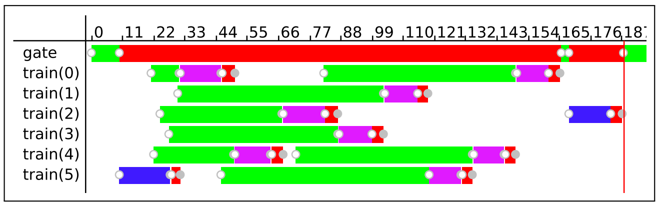
```
(Safe, Safe, Safe, Safe, Safe, Safe, Free)
appr[2]: Train(2) → Gate[2]
(Safe, Safe, Appr, Safe, Safe, Safe, Occ)
appr[3]: Train(3) → Gate[3]
(Safe, Safe, Appr, Appr, Safe, Safe, -)
stop[tail()]: Gate → Train(3)
(Safe, Safe, Appr, Stop, Safe, Safe, Occ)
appr[0]: Train(0) → Gate[0]
(Appr, Safe, Appr, Stop, Safe, Safe, -)
stop[tail()]: Gate → Train(0)
```

```
<Global>
Gate
  list = {3,0,4,0
    [0] = 3
    [1] = 0
    [2] = 4
    [3] = 0
    [4] = 0
    [5] = 0
    [6] = 0
  len = 3
Train
  0
  1
  2
    x = 15.933
  3
  4
    x = 11.465
  5
```

Fig. 14: The concrete simulator in UPPAAL.

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

(a) Definition in System declarations.



(b) Trace visualization in Concrete Simulator.

Fig. 15: Gantt chart.

user can show and hide variables in different scopes. In the example, only the clocks of Train(2) and Train(4) are shown.

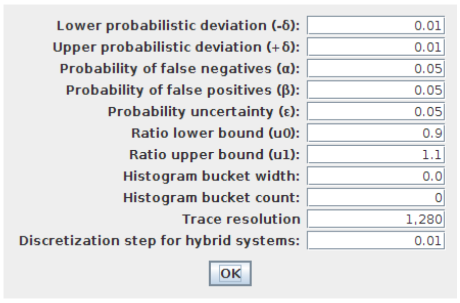
The concrete simulator also supports Gantt chart visualization of the interactive concrete trace. Figure 15 shows a sample use case of Gantt chart for the train-gate example. The chart is defined in system declarations (Fig. 15a), where each chart line is defined by a statement separated by a semicolon. 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; }.

SMC Options. Under the menu *Options* the user can choose *Statistical parameters*. This opens the window shown in Fig. 16.

– $-\delta$ 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 $[\mu - \epsilon, \mu + \epsilon]$.
– u_0 and u_1 are the lower and upper bounds used in probability comparison. Similarly to hypothesis testing, the algorithm tests two hypothesis: $H_0 : \frac{Pr(\varphi_1)}{Pr(\varphi_2)} \geq u_1$ 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.
– 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



Lower probabilistic deviation (-δ):	0.01
Upper probabilistic deviation (+δ):	0.01
Probability of false negatives (α):	0.05
Probability of false positives (β):	0.05
Probability uncertainty (ε):	0.05
Ratio lower bound (u0):	0.9
Ratio upper bound (u1):	1.1
Histogram bucket width:	0.0
Histogram bucket count:	0
Trace resolution	1,280
Discretization step for hybrid systems:	0.01

OK

Fig. 16: The statistical parameters from the options menu.

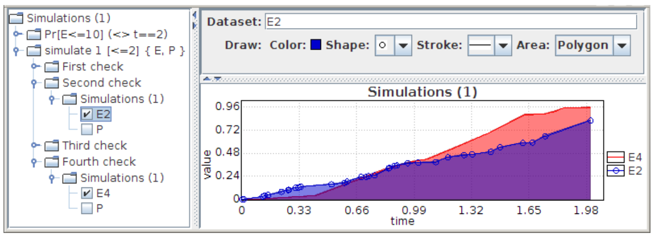


Fig. 17: Visual data comparison in the Plot Composer.

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 by 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.

The different data can also be contrasted and compared in one plot by using the Plot Composer from the Tools menu. Figure 17 shows a sample Plot Composer window with data from several verifications already loaded. The data is organized in the tree on the left. The root node controls the main attributes (the title and the labels of both axis) of the resulting plot which can be changed in the upper panel on the right. The bottom panel on the right shows the resulting plot. 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 check-box (`E2` and `E4` are ticked in Fig 17) and its drawing properties can be customized in the top-right panel when it is selected in the tree (`E2` is selected). It is also possible to compose several 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 Subsection 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. 18. 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 by 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 variable for some other peripherical 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. 19. 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

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

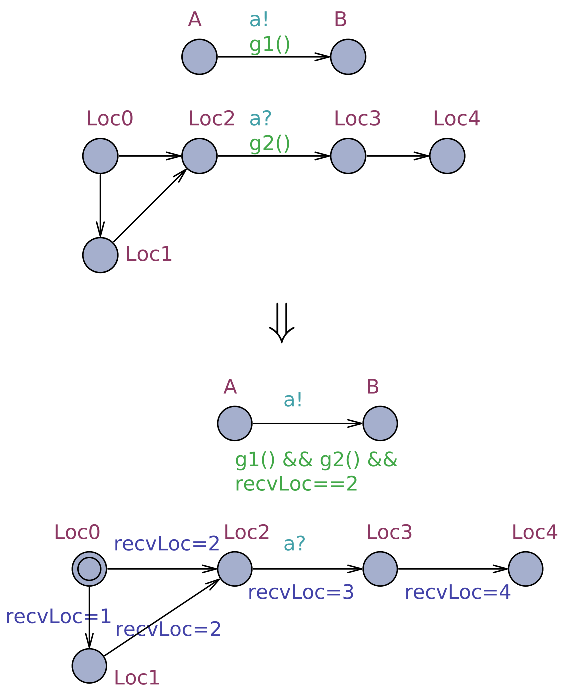


Fig. 18: Basic case of a one-to-one channel synchronization and its translation to a broadcast channel synchronization.

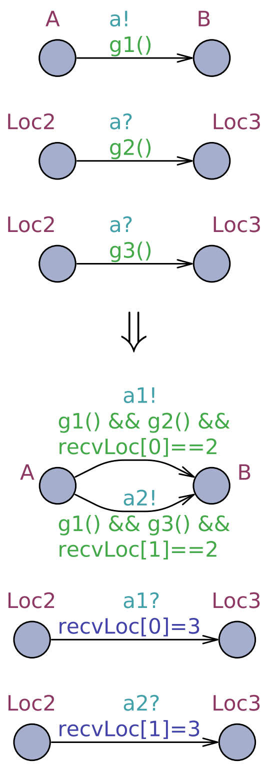


Fig. 19: Extended case of a one-to-any channel synchronization (only two here) and its translation to a broadcast channel synchronization.

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.20. Any translation of this model will violate the so called *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.

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. 22. 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

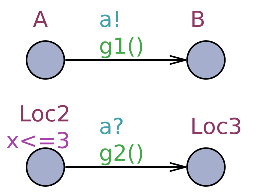
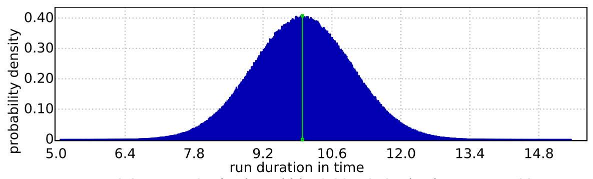


Fig. 20: Problematic case where the translation to broadcast channel is not possible.

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 [Har10].

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 uniform 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 define generate random numbers



```
Parameters: α=0.05, ε=5e-07, bucket width=0.00540197, bucket count=1921
Runs: 3688878 in total, 3688878 (100%) displayed, 0 (0%) remaining
Span of displayed sample: [5.08135, 15.4585]
Mean of displayed sample: 10.0003 ± 0.00102015 (95% CI)
```

Fig. 21: Result from modeling a Gaussian distribution.

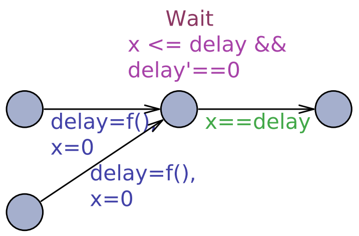


Fig. 22: Pattern for custom delay distributions.

according to a normal distribution using the Box-Müller method, we can use the following function:

```
const double PI = 3.14159265358979323846;
double f() {
  return 10+sqrt(-2.0*ln(1.0-random(1.0))) *
      cos(2.0*PI*random(1.0));
}
```

The distribution obtained is shown in Fig. 21 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 derivatives 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 double 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 evaluate 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 reactions and other interactions can be modeled by adding additional locations, edges and channel synchronizations.

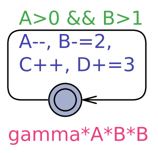
Stochastic model. Figure 23 shows a stochastic model of the reaction and its behavior. The discrete quantities (molecules) of the materials involved are counted by the corresponding integers A, B, C and D. The reaction rate is represented by the double precision floating point variable gamma. The automaton in Fig. 23b captures the interaction between chemicals A and B in the following way:

- The automaton takes a discrete transition when the reaction 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 consummes A and $2B$ and produces C and $3D$, hence the edge has the update A--, B-=2, C++, D+=3.
- In a well mixed (homogeneous) compound the probability 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 Poisson process and the delay until the next reaction follows an exponential distribution with the rate gamma*A*B*B.

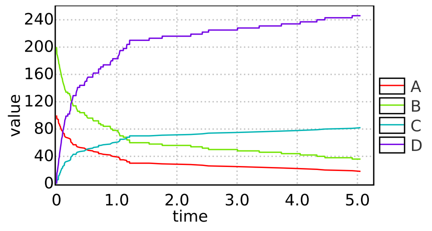
If there are more reactions, then they have to be modeled by another parallel process. The trajectory of the quantities can be inspected by the following query: `simulate 1 [<=5]{A,B,C,D}`. The resulting plot is shown in Fig. 23c: A and B are slowly decaying, replaced by C and D. We notice that the trajectory is jittery and can be slightly different with every new simulation due to probabilistic 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 continuous dynamics.

```
int A=100, B=200, C=0, D=0;
double gamma=0.0001;
```

(a) Declarations.



(b) Automaton.

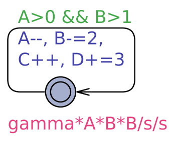


(c) Simulation.

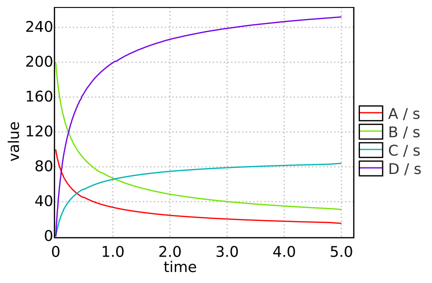
Fig. 23: Stochastic model and its behavior.

```
typedef int[-(1<<31),(1<<31)-1] int32_t;
const int s=1000; // scale by a thousand
int32_t A=100*s, B=200*s, C=0, D=0;
double gamma=0.0001;
```

(a) Scaled declarations.



(b) Scaled rate.



(c) Scaled trajectories.

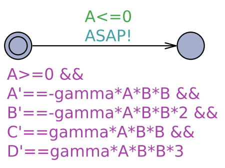
Fig. 24: Scaled stochastic model and its scaled behavior.

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 1000, 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 overflow the default range of int. Figure 24 shows the same model but with molecule quantities scaled by 1000. 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).

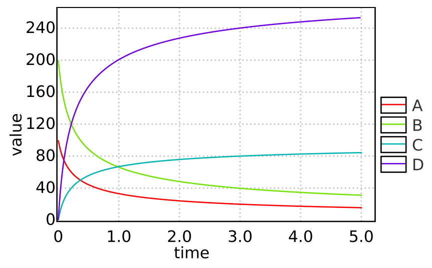
The default integer range is rather small ($\pm 2^{16}$), thus one may need to broaden it by defining a custom range. UPPAAL supports integer ranges up to 32 bits, hence the type declaration typedef int[-(1<<31),(1<<31)-1] int32_t; corresponds to a range of signed 32 bit integer. The range can be expanded 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 in-

```
clock A=100.0, B=200.0, C=0.0, D=0.0;
double gamma=0.0001;
urgent broadcast chan ASAP;
```

(a) Declarations.



(b) Automaton.



(c) Simulation.

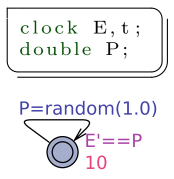
Fig. 25: Dynamic model and behavior.

crements (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:

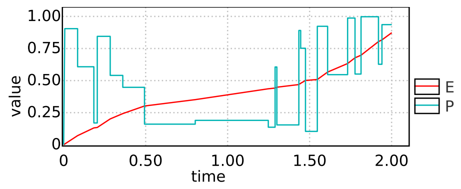
$$\begin{cases} \frac{d[A]}{dt} = -\gamma \cdot [A] \cdot [B]^2 \\ \frac{d[B]}{dt} = -\gamma \cdot [A] \cdot [B]^2 \cdot 2 \\ \frac{d[C]}{dt} = \gamma \cdot [A] \cdot [B]^2 \\ \frac{d[D]}{dt} = \gamma \cdot [A] \cdot [B]^2 \cdot 3 \end{cases}$$

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. Fig. 25 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.25b). 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.25c. 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

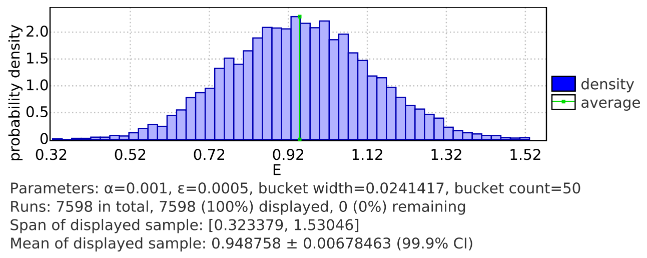


```
clock E,t;
double P;
```

(a) Model.



(b) Trajectories of energy and power.



```
Parameters: α=0.001, ε=0.0005, bucket width=0.0241417, bucket count=50
Runs: 7598 in total, 7598 (100%) displayed, 0 (0%) remaining
Span of displayed sample: [0.323379, 1.53046]
Mean of displayed sample: 0.948758 ± 0.00678463 (99.9% CI)
```

(c) Estimated energy probability distribution.

Fig. 26: Cost estimation in terms of energy.

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 more computationally 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 [DLL+12, DDL+12].

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 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 it can be in function of discrete steps if a bound of the form `#<100` is used.

Now suppose that we want to estimate a cost expressed as some energy consumption. To illustrate this, let us consider the example in Fig. 26a. In this model, a random power level is choosen 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 26b 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. 26c.

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 Section 6. Specifically, Fig. 26c 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.

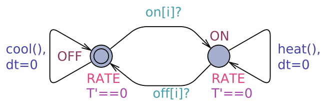
Modeling. The modeling trick consists of using a "high" exponential rate on the locations where the manual discretization is needed. The tool will then take small delay steps, albeit random according to an exponential distribution with high rate, which allows for custom discretization. Fig. 27 shows 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 27a. 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. 27. This manual encoding replaces, resp.,

[3] Uppaal SMC detects zeno runs and rejects models producing them.

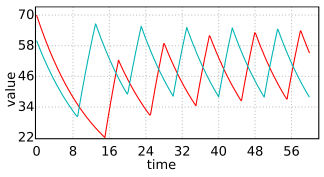
[4] The actual controller is not important for this example and is not given here.

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

(a) Variable and function declarations.



(b) The model.



(c) Temperature trajectories.

Fig. 27: The temperature of a heated room with a manual discretization using a high exponential rate RATE.

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}.

References

AD94. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

AFH96. Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. he benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, January 1996.

BCLS13. Benoît Boyer, Kevin Corre, Axel Legay, and Sean Sedwards. Plasma-lab: A flexible, distributable statistical model checking library. In *QEST*, pages 160–164, 2013.

BDL04. G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. *Lecture Notes in Computer Science*, pages 200–236, 2004.

BDL$^+$11. Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing uppaal over 15 years. *Softw., Pract. Exper.*, 41(2):133–142, 2011.

BDL$^+$12a. Peter Bulychev, Alexandre David, Kim G. Larsen, Axel Legay, Guangyan Li, and Danny Bøgsted Poulsen. Rewrite-based statistical model checking of wmtl. In *Runtime Verification*, volume 7687 of *LNCS*, pages 260–275, 2012.

BDL$^+$12b. Peter Bulychev, Alexandre David, Kim G. Larsen, Axel Legay, Guangyuan Li, Danny Bøgsted Poulsen, and Amelie Stainer. Monitor-based statistical model checking for weighted metric temporal logic. In Nikolaj Bjørner and Andrei Voronkov, editors, *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *LNCS*, pages 168–182. Springer, 2012.

BDL$^+$12c. Peter E. Bulychev, Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Checking and distributing statistical model checking. In *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 449–463. Springer, 2012.

BDLY03. Gerd Behrmann, Alexandre David, Kim G. Larsen, and Wang Yi. Unification & sharing in timed automata verification. In *SPIN Workshop 03*, volume 2648 of *LNCS*, pages 225–229, 2003.

Beh05. Gerd Behrmann. Distributed reachability analysis in timed automata. *STTT*, 7(1):19–30, 2005.

BFH$^+$01a. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient guiding towards cost-optimality in UPPAAL. In T. Margaria and W. Yi, editors, *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, pages 174–188. Springer–Verlag, 2001.

BFH$^+$01b. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits Vaandrager. Minimum-cost reachability for priced timed automata. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Proceedings of the 4th International Workshop on Hybris Systems: Computation and Control*, number 2034 in Lecture Notes in Computer Sciences, pages 147–161. Springer–Verlag, 2001.

BHV00. Gerd Behrmann, Thomas Hune, and Frits Vaandrager. Distributed timed model checking - How the search order matters. In *Proc. of 12th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Chicago, Juli 2000. Springer-Verlag.

BJK$^+$05. M. Broy, B. Jonsson, J-P. Katoen, M. Leucker, and A. Pretschner, editors. *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*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

BLP$^+$99. Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference d iagrams. In *Proceedings of the 12th Int. Conf. on Computer Aided Verificat ion*, volume 1633 of *Lecture Notes in Computer Science*. Springer–Verlag, 1999.

CFL$^+$08. Edmund M. Clarke, James R. Faeder, Christopher James Langmead, Leonard A. Harris, Sumit Kumar Jha, and Axel Legay. Statistical model checking in biolab: Applications to the automated analysis of t-cell receptor signaling pathway. In *CMSB*, LNCS, pages 231–250, 2008.

CGP99. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

DDL+12. Alexandre David, Dehui Du, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny Bøgsted Poulsen, and Sean Sedwards. Statistical model checking for stochastic hybrid systems. In Ezio Bartocci and Luca Bortolussi, editors, *HSB*, volume 92 of *EPTCS*, pages 122–136, 2012.

DDL+13. Alexandre David, Dehui Du, Kim Guldstrand Larsen, Axel Legay, and Marius Mikucionis. Optimizing control strategy using statistical model checking. In *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2013.

DLL+11. Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, Jonas Van Vliet, and Zheng Wang. Statistical model checking for networks of priced timed automata. In *FORMATS*, LNCS, pages 80–96. Springer, 2011.

DLL+12. Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, and Sean Sedwards. Runtime verification of biological systems. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (1)*, volume 7609 of *Lecture Notes in Computer Science*, pages 388–404. Springer, 2012.

DMY02. Alexandre David, M. Oliver Möller, and Wang Yi. Formal verification of UML statecharts with real-time extensions. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002*, volume 2306 of *LNCS*, pages 218–232. Springer–Verlag, 2002.

Har10. Arnd Hartmanns. Model-checking and simulation for stochastic timed systems. In *FMCO*, pages 372–391, 2010.

HL02. Martijn Hendriks and Kim G. Larsen. Exact acceleration of real-time model checking. In E. Asarin, O. Maler, and S. Yovine, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, April 2002.

HLMP04. Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 73–84. Springer Berlin Heidelberg, 2004.

JCL+09. Sumit Kumar Jha, Edmund M. Clarke, Christopher James Langmead, Axel Legay, André Platzer, and Paolo Zuliani. A bayesian approach to model checking biological systems. In *CMSB*, volume 5688 of *LNCS*, pages 218–234. Springer, 2009.

JLS13. Cyrille Jégourel, Axel Legay, and Sean Sedwards. Importance splitting for statistical model checking rare properties. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 576–591. Springer, 2013.

KNP04. M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism 2.0: A tool for probabilistic model checking. In *Proc. of 1th Int. Conference on the Quantitative Evaluation of Systems (QEST)*, pages 322–323. IEEE, 2004.

LBB+01. Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV 2001*, number 2102 in Lecture Notes in Computer Science, pages 493–505. Springer–Verlag, 2001.

LLPY97. Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: Compact data structures and state-space reduction. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

LPY97. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

SVA04. Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *CAV*, LNCS 3114, pages 202–215. Springer, 2004.

Wal45. A. Wald. Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics*, 16(2):117–186, 1945.

You05. H. L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events.* PhD thesis, Carnegie Mellon, 2005.