Electronic Communications of the EASST
Volume 66 (2013)

Proceedings of the
Automated Verification of Critical Systems
(AVoCS 2013)

Statistical Model Checking of Dynamic Networks of Stochastic Hybrid Automata

Alexandre David, Kim G. Larsen, Axel Legay, and Danny Bøgsted Poulsen

15 pages

Guest Editors: Steve Schneider, Helen Treharne
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Statistical Model Checking of Dynamic Networks of Stochastic Hybrid Automata

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

[1] {adavid,kgl,dannybp}@cs.aau.dk
Aalborg University, Denmark
[2] alegay@irisa.fr
Aalborg University, Denmark and
INRIA/IRISA Rennes, France

Abstract: In this paper we present a modelling formalism for dynamic networks of stochastic hybrid automata. In particular, our formalism is based on primitives for the dynamic creation and termination of hybrid automata components during the execution of a system. In this way we allow for natural modelling of concepts such as multiple threads found in various programming paradigms, as well as the dynamic evolution of biological systems.

We provide a natural stochastic semantics of the modelling formalism based on repeated output races between the dynamic evolving components of a system. As specification language we present a quantified extension of the logic Metric Temporal Logic (MTL). As a main contribution of this paper, the statistical model checking engine of UPPAAL has been extended to the setting of dynamic networks of hybrid systems and quantified MTL. We demonstrate the usefulness of the extended formalisms in an analysis of a dynamic version of the well-known Train Gate example, as well as in natural monitoring of a MTL formula, where observations may lead to dynamic creation of monitors for sub-formulas.

Keywords: Hybrid automata, statistical model-checking, process creation

1 Introduction

A computer program was originally seen as a single stream of instructions performed in a linear sequence. In contrast to this are multitasking systems where one program has multiple computational threads with their instructions interleaved in each other. To complicate matters, computational threads are even allowed to *spawn* other threads.

The study of such systems was pioneered by the introduction of process algebras, e.g. CSP [Hoa85] and CCS [Mil80]. Process algebras describe the behaviour of systems with a minimal set of primitives and allow us to reason about the equivalence of systems using bisimulation relations. Adding a recursion/replication operator permits expressing spawning of new threads.

Besides having multiple computational threads, modern software is getting more complex due to the distribution of labour: clients connect to servers, servers may delegate work to others and servers may need to contact some other service etc. In general, systems may establish connections to other systems and share communication links. The π-calculus [MPW92b, MPW92a], an extension of CCS, allows processes to pass communication links to each other. The minimalistic approach of process algebras, however, makes modelling actual systems tedious.

In model checking communities the formalism timed automata [AD94] is one of the most successful modelling formalisms for concurrent systems. Each computational thread can be modelled as a single automaton and all the automata coordinate their actions through synchronisations on channels. Because the timed automata formalism was developed for model checking, it defines a finite (symbolic) state space. The number of automata is as a result fixed during execution of the model thus a spawning primitive is not part of the formalism.

Statistical model checking is a software verification technique that relaxes the requirements to the modelling language. In particular, the state space does not need to be finite as an execution of the model is always terminated at a specific time point - provided the model is time diverging. We will construct a formalism that allows spawning new threads as in process algebras.

In this paper, we present a new modelling formalism founded on the basis of timed input-output (IO) transition systems. The formalism operates on a collection of timed IO-transition systems (templates) that can be instantiated during transitions of active templates. The templates could be generated by any model with semantics given as timed IO-transition system but in our implementation we rely on Hybrid Automata. We present a stochastic semantics for our formalism based on races between the instantiated components. We develop a specification logic based on MTL [Koy90]. The main difference from MTL is the addition of two operators, one to quantify on the (unknown) number of components of the network, and another to reason on arithmetic operations on this number. We have made an implementation of the modelling formalism and the monitoring technique inside UPPAAL SMC [DLL+11].

Related Work. Dynamic creation of processes is already part of extensions of process algebras. An example is the fork calculus [HL94] that extends CCS with a fork primitive. The extensions do not consider quantities and runtime verification of complex requirements expressed in MTL. As said above, the study of dynamical architecture is an intensive research topic. At the software engineering level, several works propose extensions of UML/MODAF/DODAF to handle dynamicity. Those extensions do not rely on a formal semantic which makes run-time (verification) almost impossible. Additionally, timed and stochastic information are rarely considered in those works. From a more formal perspective, the work of Chen [Ca09] deals with adaptive systems, but again assume that the state space is known in advance. Recently, Sharifloo proposed to avoid this assumption by combining verification and run-time of the deployed system within the Lover framework [SS12]. This work is in line with our objective, but ignores timed and stochastic aspects. Tools such as BIP have been extended to deal with dynamical architecture [BJMS12]. BIP focuses on interactions, while UPPAAL proposes a quantitative framework. Other approaches such as PRS also consider dynamical networks. However, they remain at a highly theoretical level, mostly studying what is decidable and what is not [ST09]. Those approaches do not consider effective and efficient algorithms. Finally, Henzinger et al., have also considered dynamical extension of reactive module with an application to systems biology. The theory presented in [FHN+11] remains very complex, there is no run-time monitoring procedure and the verification process is limited to conformance. There are also a wide range of dynamical architectures dedicated to a specific problem [FL10]. Our approach is more generic and hence incomparable to those approaches.

2 Dynamic Networks of Hybrid Automata

We introduce a general framework for dynamically evolving networks of real-time components. As the semantical basis of the individual components and the network itself, we use the notion of timed IO-transition systems.

Definition 1 (Timed IO-transition System) A *timed IO-transition system* $\mathscr{S}$ is a tuple $(S, s_0, \Sigma, \longrightarrow)$, where (i) S is a set of states, (ii) $s_0 \in S$ is the initial state, (iii) $\Sigma = \Sigma_0 \uplus \Sigma_i$ is a finite set of actions partitioned into inputs (Σ_i) and outputs (Σ_o), and $\longrightarrow \subseteq S \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times S$ is the transition relation.

As usual we write $s \xrightarrow{\alpha} s'$ whenever $(s, \alpha, s') \in \longrightarrow$, and $s \xrightarrow{\alpha}$ whenever $s \xrightarrow{\alpha} s'$ for some s'. We call $s \xrightarrow{\alpha} s'$ a discrete (input respectively output) transition whenever $\alpha \in \Sigma$ ($\alpha \in \Sigma_i$ respectively $\alpha \in \Sigma_o$), and a delay transition whenever $\alpha \in \mathbb{R}_{\geq 0}$.

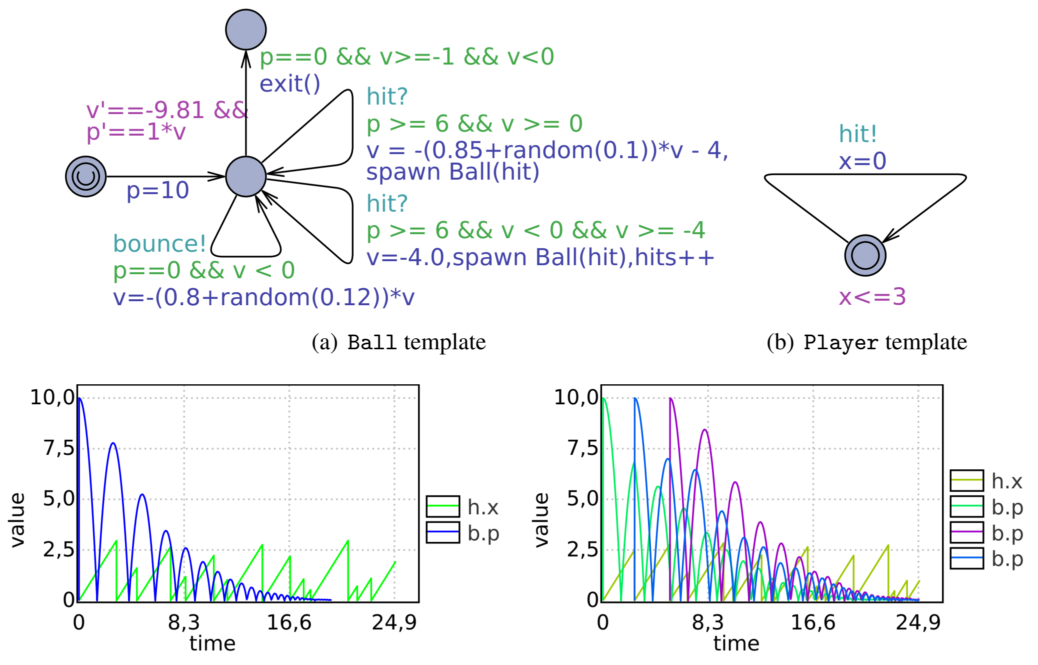
Following the compositional specification theory for timed systems in [DLL+10], we assume that $\mathscr{S}$ is *deterministic*, i.e. whenever $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ then $s' = s''$. We denote by s^α the unique state s' such that $s \xrightarrow{\alpha} s'$ (whenever it exists). Also we assume that $\mathscr{S}$ is *input enabled*, i.e. $s \xrightarrow{\alpha}$ for all $\alpha \in \Sigma_i$.

Well-known formalisms for expressing timed IO transition systems include timed automata [AD94], priced timed automata [BFH+01] and hybrid automata [HR98]. In these formalisms, states are of the type (ℓ, ν), with $\ell \in L$ being a location of the given automaton, and $\nu \in \mathscr{V}$ a valuation assigning values to the various continuous variables of the automaton (e.g. clocks, costs and hybrid variables). A *discrete transition*, $(\ell, \nu) \xrightarrow{\alpha} (\ell', \nu')$, corresponds to an edge between ℓ and ℓ' in the given automaton, whose guard is enabled by the source valuation ν and where the resulting valuation ν' is obtained from ν by performing the updates required by the edge. In *delay transitions*, $(\ell, \nu) \xrightarrow{d} (\ell, \nu')$, the values of the various continuous variables are changed according to a "flow" function $F_\ell : \mathbb{R}_{\geq 0} \times \mathscr{V} \to \mathscr{V}$ specified by the location ℓ, i.e. $\nu' = F_\ell(d, \nu)$. For timed automata $F_\ell(d, \_)$ simply corresponds to increasing the value of all clocks with d, whereas the "flow" function for hybrid automata are specified using differential equations.

Example 1 *Consider the variant of the bouncing ball in Fig. 1. Here a ball is repeatedly bouncing on the floor expressed by the hybrid automaton (template) in Fig. 1(a). In the model* $\texttt{p}$ *is the height of the and* $\texttt{v}$ *its velocity on the vertical axis. After being initialised to the state* $(\texttt{p} = 10, \texttt{v} = 0)$ *- by the first transition allowed in Fig. 1(a) - the following transition sequence may occur:*

$$(\texttt{p} = 10, \texttt{v} = 0) \xrightarrow{1.02} (\texttt{p} = 0, \texttt{v} = -10.00) \xrightarrow{\texttt{bounce!}} (\texttt{p} = 0, \texttt{v} = 9.01)$$

where in the $\texttt{bounce!}$*-transition the dampening factor has non-deterministically been chosen from the interval* $[0.80, 0.92]$ *as* 0.901*. Fig. 1(b) models an (inexperienced) player that attempts*



(a) `Ball` template (b) `Player` template

(c) Individual behaviours (d) Interacting behaviours

Figure 1: Bouncing `Ball`s with a `Player`.

to repeatedly `hit` *the ball after non-deterministic delays between* 0 *and* 3. *The individual behaviours of the ball and player are illustrated in Fig.* 1(c).

In our framework, a system consists of a dynamically changing number of interacting components, where each component is an instance of a template. The available templates is given by a template collection $\mathscr{T} = (\mathscr{T}_1, \dots, \mathscr{T}_n)$, describing *closed networks*: all templates have the same action set Σ, and their output action sets provide a partitioning of Σ, i.e. $\Sigma = \cup_j \Sigma_o^j$. For $a \in \Sigma$ we denote by $c(a)$ the unique j for which $a \in \Sigma_o^j$. For a set A, we shall denote all multisets over A by $\mathscr{M}(A)$. By $X \uplus Y$, we denote the multiset union of two multisets X and Y. Whenever $f : A \to B$, where A and B are sets, we shall extend f to the corresponding multisets in the obvious manner, i.e. for $X \in \mathscr{M}(A)$, $f(X) = \{\cdot\, f(a) : a \in X \cdot\} \in \mathscr{M}(B)$ where $\{\cdot \dots \cdot\}$ defines a multiset construction.

Definition 2 (Template Collection) Let Σ be a set of actions. A *template collection* over Σ is a tuple $\mathscr{T} = (\mathscr{T}_1, \dots, \mathscr{T}_n)$, where for $j = 1 \dots n$, $\mathscr{T}_j = (S^j, s_0^j, \longrightarrow^j, \Sigma^j, \Psi^j)$ with:

- $(S^j, s_0^j, \Sigma, \longrightarrow^j)$ is a timed IO transition systems over Σ with Σ^j as output action set;
- $\Sigma^1, \Sigma^1, \dots, \Sigma^n$ is a disjoint partitioning of Σ;
- $\Psi^j : \Sigma^j \times S^j \longrightarrow \mathscr{M}(\{\mathscr{T}_1, \dots, \mathscr{T}_n\})$ gives for each output-action-state-pair of $\mathscr{T}_j$ a multiset of templates that should be spawned while performing the output action. Whenever $s \xrightarrow{a}^j$

$$\text{DELAY} \quad (M_1,\dots,M_n) \xrightarrow{d} (M'_1,\dots,M'_n)$$
$$\text{if } d \in \mathbb{R}_{\geq 0} \text{ and for all } i \leq n, M_i \xrightarrow{d}_i M'_i;$$

$$\text{ACTION} \quad (M_1,\dots,M_j \uplus \{s\},\dots,M_n) \xrightarrow{a} (M''_1,\dots,M''_j,\dots,M''_n) \oplus P$$
$$\text{if } a \in \Sigma^j \text{ and } s \xrightarrow{a}{}^j_P s', M''_j = M_j \uplus \{s'\}, \text{ and } M_l \xrightarrow{a}_l M''_l \text{ for } l \neq j.$$

Table 1: Transition relation for $\mathscr{S}_{\mathscr{T}}$, where $\mathscr{T} = (\mathscr{T}_1,\dots,\mathscr{T}_n)$ is a template collection.

s' with $a \in \Sigma^j$ and $P = \Psi^j(a,s)$, we write $s \xrightarrow{a}{}^j_P s'$.

Formally, a template collection $\mathscr{T} = (\mathscr{T}_1,\dots,\mathscr{T}_n)$ describes a timed IO transition system of dynamic networks $\mathscr{S}_{\mathscr{T}} = (S_{\mathscr{T}}, s_0, \Sigma, \longrightarrow)$, where all actions are output actions. The set of states $S_{\mathscr{T}}$ are tuples $(M_1,\dots,M_n)$ with $M_j \in \mathscr{M}(S^j)$ describing the multiset of states comprising the currently active instances of template $\mathscr{T}_j$. The initial state s_0 is $(\{s_0^1\},\dots\{s_0^n\})$, i.e. initially one instance of each template $\mathscr{T}_i$ is instantiated. The transition relation $\longrightarrow$ of $\mathscr{S}_{\mathscr{T}}$ is given by the rules of Table 1. To delay from a state $\mathbf{s} = (M_1,\dots,M_n)$, all active instances of all templates must participate in the delay. An a-action transition is driven by an instance of the template $\mathscr{T}_j$ for which a is an output. All other instances of $\mathscr{T}_j$ ignore this output, whereas instances of other templates respond with a corresponding input transition on a. Importantly, instances of the templates in the multiset P are spawned and added to the new configuration. Formally, $(M_1,\dots,M_n) \oplus P$ is defined inductively in the size of P. As basis $(M_1,\dots,M_n) \oplus \emptyset = (M_1,\dots,M_n)$. If $P = P' \uplus \{\mathscr{T}_j\}$ and $(M_1,\dots,M_n) \oplus P' = (M'_1,\dots,M'_n)$, then $(M_1,\dots,M_n) \oplus P = (M'_1,\dots,M'_j \uplus \{s_0^j\},\dots,M'_n)$. An (infinite) *timed run* over $\mathscr{T}$ is a sequence $\omega = \mathbf{s}_0 d_0 \mathbf{s}_1 d_1 \dots \mathbf{s}_n d_n \mathbf{s}_{n+1} \dots$, where for all $i \geq 0$ $\mathbf{s}_i \in \mathbf{S}_{\mathscr{T}}$, $d_i \in \mathbb{R}_{\geq 0}$ and $\mathbf{s}_i \xrightarrow{d_i}\xrightarrow{a_i} \mathbf{s}_{i+1}$ for some $a_i \in \Sigma$. We denote by ω^i the suffix $\mathbf{s}_i d_i \mathbf{s}_{i+1} d_{i+1} \dots$.

Remark 1 *For readability, we only consider spawning of template instances on output actions. Extending the semantics to also allow spawning on input actions is, however, straightforward. Alternatively, a desired transition $s \xrightarrow{a}{}^j_P s'$, where a is an input of template $\mathscr{T}_j$, may be encoded as a sequence $s \xrightarrow{a}{}^j s^a \xrightarrow{o}{}^j_P$, with o being a new output action for $\mathscr{T}_j$ and s^a being a new intermediate state that can only output o while spawning P.*

Remark 2 *For simplicity our theoretical construction does not allow for parameterising templates. It is, however, allowed in our implementation in* UPPAAL *SMC.*

Example 2 *Reconsider the bouncing ball example from Fig. 1. Jointly the ball and the player constitutes a template collection $\mathscr{T}$ with two templates (*`Ball` *and* `Player`*), with initially one ball and one player. Figure 1(d) depicts the joint behaviour during the first 25 time-units, with* *the first transitions detailed as follows:*

$$
\begin{aligned}
\big(\{\langle p=10, v=0\rangle\}, \{\langle x=0\rangle\}\big) &\xrightarrow{1.02} \big(\{\langle p=0, v=-10.00\rangle\}, \{\langle x=1.02\rangle\}\big)\\
&\xrightarrow{\texttt{bounce!}} \big(\{\langle p=0, v=9.01\rangle\}, \{\langle x=1.02\rangle\}\big)\\
&\xrightarrow{0.8} \big(\{\langle p=6.1, v=1.16\rangle\}, \{\langle x=1.82\rangle\}\big)\\
&\xrightarrow{\texttt{hit!}} \big(\{\langle p=6.1, v=-5.04\rangle, \langle p=10, v=0\rangle\}, \{\langle x=1.82\rangle\}\big)
\end{aligned}
$$

In particular, we note that after the (initial) ball have bounced, the player successfully hits it resulting in a new (second) ball being spawned. We see that during the 25 *time-units the player is also successful in hitting that (second) ball. In the figure we can see a ball being spawned by the extra curves compared to Fig. 1(d).*

3 Stochastic Semantics for Dynamic Networks

Reconsidering our dynamic version of the bouncing ball from Section 2, we may consider that there is a constant race between the ball(s) `bounce!`ing on the floor and the player `hit!`ing the ball(s). Whereas the time of bouncing is deterministic – given by the ODE obtained from the (stochastic) effect of the previous `bounce!` or `hit!` – the time of the hitting by the player is stochastic according to a uniform distribution in the interval $[0,3]$. In the randomly generated trajectory of Fig. 1(d) it seems that the player was successful in hitting twice, thus generating two additional balls. In fact, a measure on sets of runs of the system is induced, according to which quantitative properties such as *"the probability that there are two or more balls with a height greater than 5 within 4 and 6 time-units"* become well-defined.

Our stochastic semantics is based on the principle of independence between components. Repeatedly each component decides on its own – based on a given delay density function and output probability function – how much to delay before outputting and what output to broadcast at that moment. Obviously, in such a race between components the outcome will be determined by the component that has chosen to output after the smallest delay: the output is broadcast and all other components may consequently change state.

Stochastic Template Collection Stochastic template collections refine the non-deterministic choices that may exist with respect to delay, output and next state in the specification of a template collection $\mathscr{T} = (\mathscr{T}_1, \ldots, \mathscr{T}_n)$. Let $\mathscr{T}_j$ be a template of the collection and let S^j denote the corresponding set of states. For each state $s \in S^j$, we assume that there exist probability distributions for delays, outputs as well as next-state:

- the *delay density function*, μ_s over delays in $\mathbb{R}_{\geq 0}$, provides stochastic information for when the component will perform an output, thus $\int \mu_s(t)dt = 1$;
- the *output probability function* γ_s assigns probabilities for resolving what output $o \in \Sigma_o^j$ to generate, i.e. $\sum_o \gamma_s(o) = 1$ [1].

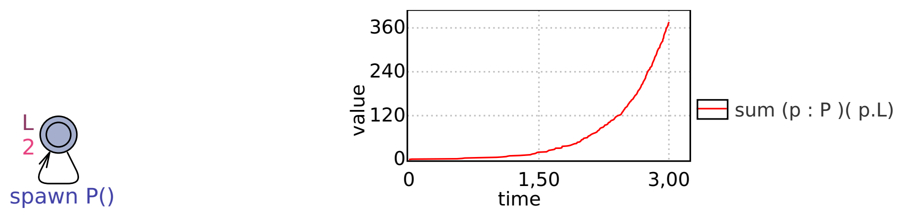
[1] For outputs happening deterministically at an exact time point d, μ_s becomes a Dirac delta function δ_d.

Remark 3 *In* UPPAAL *SMC uniform distributions are applied for states where delay is bounded, and exponential distributions (with location-specified rates) are applied for the cases, where a component can remain indefinitely in a location. Also,* UPPAAL *SMC provides syntax for assigning discrete probabilities to different outputs as well as specifying stochastic distributions on next-states (using the function* `random[b]` *denoting a uniform distribution on* $[0,b]$*).*

Stochastic Dynamic Networks A *stochastic* template collection $\mathscr{T} = (\mathscr{T}_1, \dots, \mathscr{T}_n)$ in turns induces a stochastic semantics of the dynamic network timed IO transition system $\mathscr{S}_{\mathscr{T}} = (S_{\mathscr{T}}, s_0, \longrightarrow, \Sigma)$. For $\mathbf{s} = (M_1, \dots, M_n) \in S_{\mathscr{T}}$, $\pi(\mathbf{s}, a_1 a_2 \dots a_k)$ denotes the set of all maximal runs from $\mathbf{s}$ with a prefix $t_1 a_1 t_2 a_2 \dots t_k a_k$ for some $t_1, \dots, t_n \in \mathbb{R}_{\geq 0}$ (a cylinder), that is runs where the i'th action a_i has been outputted by some instance of $\mathscr{T}_{c(a_i)}$. Providing the basic elements of a Sigma-algebra, we now inductively define the measure for such sets of runs:

$$\mathbb{P}_{\mathscr{T}}\big(\pi(\mathbf{s}, a_1 \dots a_n)\big) = \sum_{s \in M_c} \int_{t \geq 0} \mu_s \cdot \left(\prod_{j \neq c} \prod_{s' \in M_j} \left(\int_{\tau > t} \mu_{s'}(\tau) d\tau \right) \right) \cdot \left(\prod_{s'' \in M_c \setminus \{s\}} \left(\int_{\tau > t} \mu_{s''}(\tau) d\tau \right) \right) \cdot \gamma_{s^t}(a_1) \cdot \mathbb{P}_{\mathscr{T}}\big(\pi((\mathbf{s}^t)^{a_1}, a_2 \dots a_n)\big) dt$$

where $c = c(a_1)$. This definition requires an explanation: at the outermost level we sum over all states from M_c, i.e. active instances of the template $\mathscr{T}_c$ for which a_1 is an output. For a given delay t, the outputting component, s_c, will choose to make the broadcast at time t with the stated density. Independently, the other components (other components of the template $\mathscr{T}_c$ as well as components of other templates) will choose a delay amount, which – in order for c to be the winner – must be larger than t; hence the (two) products of the probabilities that they each make such a choice. Having decided for making the broadcast at time t, the probability of actually outputting a_1 is included. Finally, the probability of runs according to the remaining actions $a_2 \dots a_n$ is taken into account.



(a) Template P. (b) Number of instances.

Figure 2: An exploding template collection?

Example 3 *Given the dynamic spawning of new instances, the question arises whether the resulting network* explodes *in the sense that discrete actions may occur with shorter and shorter time between them as the number of instances grows, and hence the race between components* *become more and more intense over time. Stated differently, we worry that the dynamic network may exhibit Zeno behaviour with a non-zero probability. As a (potential) example consider the template* P *of Fig. 2, where each instance will spawn new instances according to an exponential distribution with rate* 2*. The resulting evolution of the number of instances during a random run is illustrated in Fig 2. Fortunately, it follows from Reuter's criteria for birth-and-death processes [Reu57], that for template collections where all delay densities are either exponential distributions (spanning a finite range of rates) or uniform (spanning a finite range of intervals), the system does not explode. This is important as termination of our method of statistical model checking relies on the assumption that random runs will eventually exceed any given time-bound with probability one.*

4 Dynamic Metric Interval Temporal Logic

In this section we present Dynamic Metric Temporal Logic (DMTL) for defining properties of runs of a dynamic network. The logic is based on Metric Temporal Logic (MTL), where atomic propositions have been extended with means for quantifying over the dynamic components of the systems.

Let $\mathscr{T} = (\mathscr{T}_1,\ldots,\mathscr{T}_n)$ be a template collection. For each template $\mathscr{T} \in \{\mathscr{T}_1,\ldots,\mathscr{T}_n\}$, we assume the existence of a syntactic category of arithmetic expressions, $\textsc{Expr}_{\mathscr{T}}$, interpreted over the states $S^{\mathscr{T}}$ of $\mathscr{T}$. Thus, whenever $\varepsilon \in \textsc{Expr}_{\mathscr{T}}$ and $s \in S^{\mathscr{T}}$ then $[\![\varepsilon]\!](s) \in \mathbb{N}$. Similarly we assume a syntactic category of Boolean expressions, $\textsc{Bool}_{\mathscr{T}}$, interpreted over the states $S^{\mathscr{T}}$ of $\mathscr{T}$. Thus, whenever $\beta \in \textsc{Bool}_{\mathscr{T}}$ and $s \in S^{\mathscr{T}}$ then $[\![\beta]\!](s) \in \mathbb{B}$.

Considering now global states $(M_1,\ldots,M_n)$ of the template collection $\mathscr{T}$, we introduce the sets of arithmetic expressions, $\textsc{Expr}$, and Boolean expressions, $\textsc{Bool}$ given by the following grammars:

$$e ::= c \mid e_1 \,\mathsf{op}\, e_2 \mid \mathsf{sum}(t:\mathscr{T}).\varepsilon_{\mathscr{T}}$$

$$b ::= \mathrm{tt} \mid \mathrm{ff} \mid \neg b \mid b_1 \wedge b_2 \mid e_1 \bowtie e_2 \mid \mathsf{forall}(t:\mathscr{T}).\beta_{\mathscr{T}}$$

where $c \in \mathbb{Z}$, op is a binary arithmetic operator, $\bowtie$ is a binary comparison operator, and $\mathscr{T}$ is a template from the collection $\mathscr{T}$. The semantics are:

$$[\![\mathsf{sum}(t:\mathscr{T}).\varepsilon_{\mathscr{T}}]\!](M_1,\ldots,M_n) = \sum_{s\in M_{\mathscr{T}}} [\![\varepsilon_{\mathscr{T}}]\!](s)$$

$$[\![\mathsf{forall}(t:\mathscr{T}).\beta_{\mathscr{T}}]\!](M_1,\ldots,M_n) = \bigwedge_{s\in M_{\mathscr{T}}} [\![\beta_{\mathscr{T}}]\!](s)$$

Definition 3 (Dynamic Metric Temporal Logic) Let $\mathscr{T} = (\mathscr{T}_1,\ldots,\mathscr{T}_n)$ be a collection of templates. A DMTL formula φ over $\mathscr{T}$ is defined by the grammar:

$$\varphi ::= b \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{X}\varphi \mid \varphi_1 \mathsf{U}_{[x,y]} \varphi_2$$

where $b \in \textsc{Bool}$, $x, y \in \mathbb{Q}$ with $x \leq y$.

We use ff as an abbreviation for $(b \wedge \neg b)$, tt for $\neg$ff, and exists$(t : \mathscr{T}).\beta_{\mathscr{T}}$ for $\neg$forall$(t : \mathscr{T}).\neg\beta_{\mathscr{T}}$. Other commonly used operators of MTL are derived in the usual manners, e.g.: $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \to \varphi_2 = (\neg\varphi_1 \vee \varphi_2)$, $\Diamond_{[x,y]}\varphi = \mathsf{tt}\,\mathsf{U}_{[x,y]}\varphi$, $\Box_{[a,b]}\varphi = \neg\Diamond_{[x,y]}\neg\varphi$, and $\varphi_1\mathsf{R}_{[x;y]}\varphi_2 = \neg(\neg\varphi_1\mathsf{U}_{[x,y]}\neg\varphi_2)$, where R is the "release" operator. For a given timed run $\omega = \mathbf{s}_0 d_0 \mathbf{s}_1 d_1 \ldots \mathbf{s}_n d_n \mathbf{s}_{n+1} \ldots$ over $\mathscr{S}_{\mathscr{T}}$ and a DMTL formula φ, we define satisfaction $\omega^i \models \varphi$ inductively as follows:

1. $\omega^i \models b$ iff $[\![b]\!]\mathbf{s}_i$
2. $\omega^i \models \neg\varphi$ iff $\omega^i \not\models \varphi$
3. $\omega^i \models \varphi_1 \wedge \varphi_2$ iff $\omega^i \models \varphi_1$ and $\omega^i \models \varphi_2$
4. $\omega^i \models \mathsf{X}\varphi$ iff $\omega^{i+1} \models \varphi$
5. $\omega^i \models \varphi_1\mathsf{U}_{[x,y]}\varphi_2$ iff there exists $j \geq i$ such that $\omega^j \models \varphi_2$ and $\sum_{k=i}^{j-1} d_k \in [x,y]$ and $\omega^k \models \varphi_1$ whenever $i \leq k < j$.

We say that a timed run ω satisfies φ if $\omega^0 \models \varphi$. We say that a template collection $\mathscr{T}$ satisfies φ, $\mathscr{T} \models \varphi$, iff all timed runs of $\mathscr{S}_{\mathscr{T}}$ starting in $\mathbf{s}_0$ satisfies φ. Given the stochastic semantics of $\mathscr{T}$, we define $\mathbb{P}_{\mathscr{T}}(\varphi)$ to be the probability that a random run of $\mathscr{T}$ satisfies φ. As we shall see later, this probability is well-defined as it may be characterized as a countable union and intersection of cylinders over $\mathscr{T}$ extended with a monitor $\mathscr{M}_\varphi$ for φ, and is thus measurable.

Example 4 *Reconsider the bouncing ball from Fig. 1. The property* "there are two or more balls with a height greater than 5 within 2 and 3 time-units" *may be expressed as the DTML formula* $\Diamond_{[2,3]}\big(\mathsf{sum}(\mathtt{b:Ball})(\mathtt{b.p} > 5)\big) \geq 2$. *Similarly, the property* "for any time-point within 1 and 3 time units, all balls have height less than or equal to 4" *corresponds to the formula* $\Box_{[1,3]}\mathsf{forall}(\mathtt{b:Ball})(\mathtt{b.p} \leq 4)$. *Using* UPPAAL *SMC, we find* $[0.164092, 0.264092]$, *resp.* $[0.82, 0.92]$, *to the interval of the probability that a random run will satisfy the first, resp. the second, property with* 95% *confidence.*

Theorem 1 *Let $\mathscr{T}$ be a template collection and φ be a DMTL formula . Then there exists a template collection $\mathscr{T}_\varphi = \{\mathscr{T}_\varphi, \mathscr{T}_{\varphi_1}, \ldots, \mathscr{T}_{\varphi_n}\}$ over Σ_ϕ with $\mathtt{tt}_\varphi!, \mathtt{ff}_\varphi! \in \Sigma_\phi$ associated with φ such that:*

$$\mathbb{P}_{\mathscr{T}}(\varphi) = \mathbb{P}_{\mathscr{T}\cup\mathscr{T}_\varphi}\left(\bigcup_{\omega\in\Sigma^*} \pi\left((\mathbf{s},\mathbf{s}_\varphi), \omega\mathtt{tt}_\varphi!\right)\right) \tag{1}$$

where $\mathbf{s}_\varphi = (M_\varphi, M_{\varphi_1}, \ldots M_{\varphi_n})$ with $M_\varphi = \{s_0^\varphi\}$ and for all $j \neq 0$, $M_{\varphi_j} = \emptyset$, and Σ is the combined alphabet of $\mathscr{T}$ and $\mathscr{T}_\varphi$ excluding $\mathtt{tt}_\varphi!$ and $\mathtt{ff}_\varphi!$.

5 Dynamic Networks of Hybrid Automata in UPPAAL

UPPAAL SMC has been extended to dynamic instantiation of templates. Templates are, as usual, defined as Stochastics Hybrid Automata (SHA). The extension includes extending the core language with two keywords (`spawn` and `exit`) to create and terminate processes, extending the notion of ranges to sets of processes using a similar syntax, and extending MTL to refer to the dynamic processes. We mention some interesting details that are important.

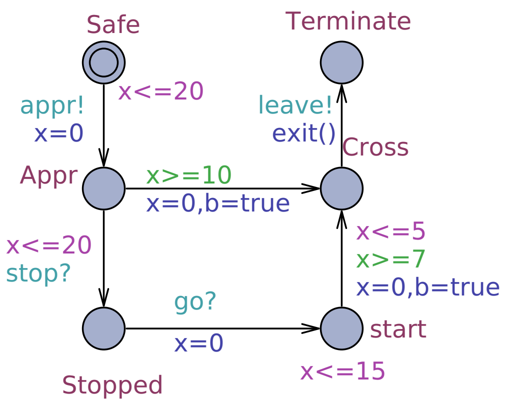
Core Language Extensions. New processes are created by calling `spawn T(args...)` where `T` is the name of a template. The argument list is similar to the ordinary process instances. However, the internal mechanism for dynamically creating processes is different. A dynamic process terminates by calling the special function `exit()`.

Normally, an instance corresponds to a template with its arguments substituted that is then compiled to obtain some byte-code specific for that instance. This is too expensive for dynamic instances not to mention the serious issue concerning memory management when terminating an instance (and the byte-code in the engine too). The mechanism we have creates local variables that correspond to the instance parameters of these templates. The templates are instantiated when needed but are recycled and kept in the engine when they terminate, in particular between each run. Instantiating a dynamic instance (`spawn`) is done by taking a process in a pool, writing to its local variables (instead of substituting and generating new byte-code), and adding it to the state. Termination of a process (`exit`) corresponds to recycling.

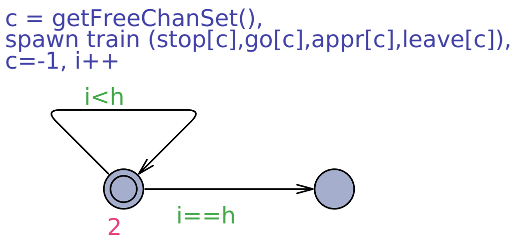
Sets of Processes. The language extends the notion of ranges to sets of processes. The processes are here instances of stochastic hybrid automata templates. Normally, the user can use `for` loops, and the statements `sum`, `forall`, and `exists` over ranges, e.g., `for(i:id_t)` for iteration purposes. Now, this is extended to sets of processes for a given template type. For a template of type `T`, it is now possible to use the syntax `t:T` to iterate over this set with, e.g., `sum(t:T) t.count`.

Statistical Model Checking (SMC). We use SMC [SVA04, YS02, LDB10] to estimate and test on the probability that a random run of a network of stochastic hybrid automata satisfies a given property. Given a model $\mathscr{H}$ and a trace property ϕ (expressed in LTL or MTL), SMC refers to a series of simulation-based techniques that can be used to answer two main questions: (1) *Qualitative:* is the probability that a random run of $\mathscr{H}$ satisfies ϕ greater or equal to a certain threshold θ? and (2) *Quantitative:* what is the probability that a random run of $\mathscr{H}$ satisfies ϕ? In both cases, the answer will be correct up to a user-specified level of confidence, providing a upper bound on the probability that the conclusion made by the algorithm is wrong. For the quantitative approach, the method computes a confidence interval that is an interval of probabilities that contains the true probability to satisfy the property. Here the confidence level provides the probability that the computed confidence interval indeed contains the unknown probability.

Implementation of DMTL Monitoring. When checking DMTL properties, the engine rewrites the DMTL formula ϕ on-the-fly in such a way that the new formula ϕ_1 is satisfied from the next state if and only if ϕ was satisfied from current state. The technique is similar to the one presented in [BDL+12], but the engine has been extended to check the construct $\mathsf{forall}(t:\mathscr{T}).\beta_{\mathscr{T}}$. methods.



(a) train (chan& stop, chan& go, chan& appr,chan& leave)



(b) trainSpawner ()

Figure 3: Automaton of a train (a) and an automaton for spawning trains (b)

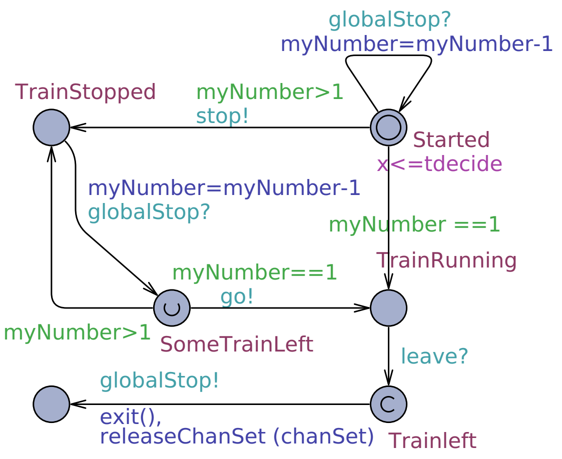
Visualization of Simulated Runs. As part of the statistical model-checking analysis output, plots are very useful tools. When generating processes dynamically, we need a new way to ask for plots on such dynamic instances. To do this, the special statement `foreach(t:T) expr` is used to tell the model-checker to generate plots for each of the expressions `expr` that depend on `t`. An example of such a plot is shown in Fig. 1(d).

6 Dynamic Train Gate

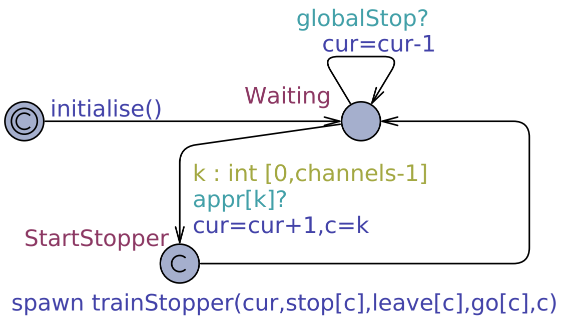
We consider a dynamic version of the train gate example distributed with UPPAAL: a gate controls the access to the one rail over a bridge. When a train approaches the bridge it signals the gate that it is approaching. Depending on the speed of the train, the train will proceed onto the crossing after 10 and before 20 time units. The actual crossing takes 5 time units. From the approach signal has been sent, the gate can stop the train to avoid a collision. When stopped, the train can be restarted which takes between 7 and 15 time units. Afterwards the train enters the bridge and cannot be stopped. To ensure only one train crosses at a time, the gate allows the train arriving first to proceed and queue the latter. When a train leaves the crossing, the gate will start the first train in the queue.

In our model each train is modelled as a single SHA, depicted in Fig. 3(a). The train-SHA has four parameters, namely the channels leave,stop,go and appr. The trains are spawned according to an exponential distribution with rate 2 by the SHA in Fig. 3(b). This uses a function to acquire channels, from a pre-allocated set of channels, to be used by the spawned train.

The gate, see Fig. 4(b), listens on all channels that can be used as appr channels and when a train signals its approach a train stopper, depicted in Fig. 4(a), is spawned . The train stoppers are given a number (myNumber) that they decrease when a train has left the crossing (signalled by a synchronisation on globalStop?) . When myNumber is equal to one, the train stopper broadcasts go! to its associated train. Whenever a train broadcast leave! the associated train stopper will broadcast globalStop!. This results in starting a stopped train. In this manner the train stoppers



(a) trainStopper (int myNumber, chan &stop, chan &leave, chan &go)



(b) gate()

Figure 4: Model of the queue (a) and the gate controller (b)

are encoding a queue. In the original model the queue was encoded explicitly in the C-like language of UPPAAL but as this does not allow dynamic allocation of memory we resorted to using templates. Initially the only instances in the system are the gate and train spawner..

Experiments Fig. 5 shows the number of trains in each of their possible location. This plot was obtained by running the UPPAAL SMC query:

```
simulate 1 [<=300] {sum (t:train)(t.Stopped),sum
                  (t:train)(t.Safe)...}
```

We see in the run that only one train is on the bridge at a time (i.e. in the location Cross). We also see that the trains are spawned at the beginning of the run, resulting in a high number of trains being stopped. At some point,approximately about 40 time units into the run, all the trains have been spawned and the number of stopped trains decreases, as the trains are moved one by one to the crossing.

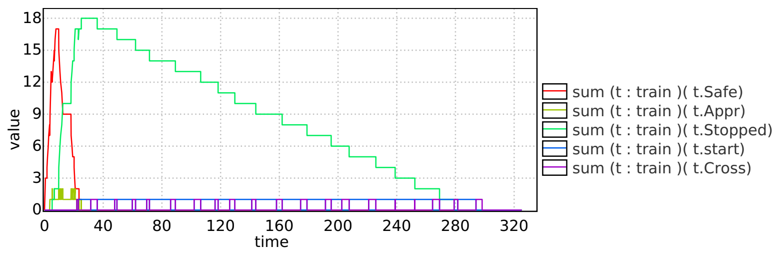


Figure 5: The number of trains in each location. The number of trains in this run is 20 and the trains stoppers time to decide (tdecide) is 0

In the model h trains are spawned by the the template in Fig. 3(b) with some delay in between. The gate is supposed to ensure only one train crosses the bridge at a time, and in order to do so it relies on the train stopper templates. The train stoppers need some time, tdecide, to decide if a train should be stopped or not. Using UPPAAL SMC we can find the probability that two trains are on the bridge simultaneously within 1100 time units through the query:

```
              Pr[<=X]
(<> exists (t : train) (t.Cross &&
exists (p : train)(p.Cross && p!=t)))
```

h	tdecide	Probability
20	10	[0.000; 0.097]
20	12	[0.091; 0.191]
20	15	[0.393; 0.494]
40	10	[0.000; 0.0974]
40	12	[0.156; 0.255]
40	15	[0.701; 0.801]

Table 2: Probability of collision within 110 time units with h trains and a decision time of tdecide

The delayed reaction from the train stoppers may, obviously, result in the gate not being safe. The results in Table 2 suggests, however, that if the time the train stoppers need is less than the minimum time for the train to enter the bridge then the probability is low.

7 Experiments with the Monitoring of DMTL

For experimenting with the MTL monitoring stated in Theorem 1 we have created an automaton that generate random runs over the propositions p,q and r. The automaton "flips" the truth assignment of p,q or r with a rate of 2 i.e. the delays between state changes is extracted from an exponential distribution with rate parameter 2.

The advantage of "implementing" the rewrite technique as observers is that we can use the plotting feature of UPPAAL SMC to obtain plots of the number of active observers during a simulation. This can be done with a query in the style of

```
Pr[<=10]{sum (b : T1) (1) +sum (b : T2) (1) ...}.
```

We show such a plot in Fig. 6. Obtaining this plot with the optimised version in UPPAAL SMC is more difficult because you would have to explicitly gather information during the rewriting. In addition to obtain such plots, we can experiment with variations of the rewrite technique. It is well-known, that for run time verification it is not feasible to observe a system every time it changes state. Instead the system is observed at distinct time points. This can result in different verification results thus we are interested in knowing how the verification result differs when using fewer observations. To exemplify this we have created three collections of observers for the formula $\Diamond_{[0;1]}(p \wedge \Box[0;1](\neg r) \wedge \Diamond_{[0;1]}(q))$ and parallel composed them with the random

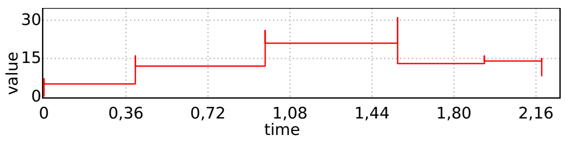


Figure 6: The number active observers during a verification of the formula $\Diamond_{[0;1]}(p \wedge \Box[0;1](\neg r) \wedge \Diamond_{[0;1]}(q))$

automata previously described: The first one, $\mathscr{T}^1_\phi$, observes every state change of the system. The second one $\mathscr{T}^2_\phi$ observes the system at time points randomly selected - with the restriction that there should be one observation each time unit. The third, $\mathscr{T}^3_\phi$, is similar to the second, but the maximal time between observations is 2.

Verifying the property with the different observer setups gives probability bounds $[0.02, 0.12]$ using $\mathscr{T}^1_\phi$ and $[0.03, 0.13]$ for $\mathscr{T}^2_\phi$. For $\mathscr{T}^3_\phi$ we obtain a probability bound $[0.00, 0.10]$. These results are not surprising, but they exemplify the possibility of using UPPAAL SMC to make an analysis of a run validation technique.

8 Conclusion & Future Work

We have presented a formalism that allows for dynamical instantiation of templates of hybrid automata. The formalism is given a natural stochastic semantics based on repeated output races. Also, we have extended MTL with the possibility of quantifying over components at the propositional level. In particular, the engine of UPPAAL SMC has been extended to allow for statistical model checking of DMTL properties for dynamic networks of stochastic hybrid systems. Also we added additional visualisation option to UPPAAL SMC.

Future extension involve extension to more advanced specification formalism in which we allow for nesting the quantifications over components.

References

[AD94] R. Alur, D. L. Dill. A Theory of Timed Automata. *TCS* 126(2):183–235, 1994.

[BDL+12] P. E. Bulychev, A. David, K. G. Larsen, A. Legay, G. Li, D. B. Poulsen. Rewrite-Based Statistical Model Checking of WMTL. In *RV*. LNCS 7687, pp. 260–275. 2012.

[BFH+01] G. Berhmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In Benedetto and Sangiovanni-Vincentelli (eds.), *4th HSCC*. Pp. 147–161. March 2001. http://www.mrtc.mdh.se/index.php?choice=publications&id=2849

[BJMS12] M. Bozga, M. Jaber, N. Maris, J. Sifakis. Modeling Dynamic Architectures Using Dy-BIP. In *SC*. LNCS 7306, pp. 1–16. 2012.

[Ca09] Cheng, all. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *SESAS*. LNCS 5525. 2009.

[DLL+10] A. David, K. G. Larsen, A. Legay, U. Nyman, A. Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *HSCC*. Pp. 91–100. ACM, 2010.

[DLL+11] A. David, K. G. Larsen, A. Legay, M. Mikucionis, Z. Wang. Time for Statistical Model Checking of Real-Time Systems. In *CAV*. LNCS 7687, pp. 349–355. 2011.

[FHN$^+$11] J. Fisher, T. A. Henzinger, D. Nickovic, N. Piterman, A. V. Singh, M. Y. Vardi. Dynamic Reactive Modules. In *CONCUR*. LNCS 6901, pp. 404–418. Springer, 2011.

[FL10] J. L. Fiadeiro, A. Lopes. A Model for Dynamic Reconfiguration in Service-Oriented Architectures. In *ECSA*. LNCS 6285, pp. 70–85. Springer, 2010.

[HL94] K. Havelund, K. G. Larsen. The Fork Calculus. *Nord. J. Comput.* 1(3):346–363, 1994.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HR98] T. A. Henzinger, V. Rusu. Reachability verification for hybrid automata. In *HSCC*. LNCS 1386, pp. 190–204. 1998.

[Koy90] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems* 2(4):255–299, 1990.

[LDB10] A. Legay, B. Delahaye, S. Bensalem. Statistical Model Checking: An Overview. In *RV*. LNCS 6418, pp. 122–135. Springer, 2010.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer, 1980.

[MPW92a] R. Milner, J. Parrow, D. Walker. A Calculus of Mobile Processes, I. *Inf. Comput.* 100(1):1–40, 1992.

[MPW92b] R. Milner, J. Parrow, D. Walker. A Calculus of Mobile Processes, II. *Inf. Comput.* 100(1):41–77, 1992.

[Reu57] G. E. H. Reuter. Denumerable Markov Processes and the associated contracting semigroups onL. *Acta Mathematica* 97(1-4):1–46, 1957.

[SS12] A. M. Sharifloo, P. Spoletini. LOVER: Light-Weight fOrmal Verification of adaptivE Systems at Run Time. In *FACS*. LNCS 7684, pp. 170–187. 2012.

[ST09] M. Sighireanu, T. Touili. Bounded Communication Reachability Analysis of Process Rewrite Systems with Ordered Parallelism. *ENTCS* 239:43–56, 2009.

[SVA04] K. Sen, M. Viswanathan, G. Agha. Statistical Model Checking of Black-Box Probabilistic Systems. In *CAV*. LNCS 3114, pp. 202–215. 2004.

[YS02] H. L. S. Younes, R. G. Simmons. Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling. In *CAV*. LNCS 2404, pp. 223–235. Springer, 2002.
doi:10.1007/3-540-45657-0_17