GPU Accelerating Statistical Model Checking for Extended Timed Automata

Oliver S. Bak[1], Mathias W. B. Christiansen[1], Oliver V. Eriksen[1], Sergio Feo-Arenis[2], Peter G. Jensen[1], Marcus D. Jensen[1], Simas Juozapaitis[1], Kim G. Larsen[1], Marius Mikučionis[1], Marco Muñiz[1], and Andreas Windfeld[1]

[1] Department of Computer Science, Aalborg University, Denmark
[2] NVIDIA Corp., München, Germany

Abstract. The core component of Statistical Model Checking (SMC) is the repeated sampling of a given system as to estimate statistical measures. To obtain probabilistic estimates with high confidence a significant number of simulations is required, in particular in the presence of *rare events*. In this paper we explore the use of Graphical Processing Unit (GPU) for accelerating SMC for Networks of Stochastic Extended Timed Automata (SXTA). We discuss the many challenges and solutions required to achieve significant speedups on a GPU architecture. In collaboration with NVIDIA we develop a prototype tool for parallel SMC using both GPU and multi-core CPU. Experimental results demonstrate trade-offs in the computation time when utilizing either CPU or GPU. In one case we observed the GPU using 20% of the power of the CPU equivalent while delivering a 2.73 time speedup.

1 Introduction

Statistical model checking (SMC) is a technique where the (statistical) correctness of a given system model is inferred via repeated statistical sampling. SMC is applied for both qualitative and quantitative studies of systems where classical model checking is unpractical due to e.g. the state-space explosion, or impossible e.g. due to intractability of a symbolic analysis [14]. Furthermore, repeated statistical simulation is required for both training and evaluating AI based controllers in a safe environment (see e.g. [7,21]). One formalism for expressing complex stochastic systems is Networks of Stochastic Extended Timed Automata (NSXTA), for which the model checking tool Uppaal provides an SMC engine. While the SMC technique generally enjoys good scalability, recent applications call for run-time reductions, e.g. training controllers for residential heating systems [21] or simulating large-scale agent-based COVID19-models [5,25]. Additionally, the recent energy cost surges, global warming and increased needs for large-scale AI system training call for more studies of cost of computation [32].

Numerical simulation problems have classically enjoyed good speedups from GPU acceleration, e.g. Computational Fluid Dynamics [2]. We note here that such classical problems generally enjoy matrix-like descriptions with little or no branching and little or no stochastics, a strong contrast to NSXTA.

In this paper, we study the use of parallel computing via GPUs and CPUs to increase the performance of SMC for NSXTA. Our main contribution is a parallel implementation of SMC for NSXTA on GPU. In addition, our experiments show substantial time and energy savings while using a GPU compared to an equivalent CPU cluster implementation. Another contribution is that estimation of probabilities for large real world scenarios e.g. COVID19-models [5,25] which required a CPU cluster, can now be carried out on workstation or laptop with a suitable GPU.

Relationship to Joost-Pieter Katoen The SMC engine of Uppaal dates back to the European FP7 project Quasimodo (2008-2011), where Joost-Pieter Katoen was a main contributor representing RWTH Aachen. Aalborg University was a project coordinator and the other partners of the Quasimodo project were ESI, University of Twente, Radboud University, ENS Cachan, Saarland University, University Libre de Bruxelles, Terma A/S, Chess, Inchron, and Hydac.

The objective of the Quasimodo project was to develop theory, techniques and tools for handling quantitative constraints in model-driven development of real-time embedded systems. This involved explicit handling of real-time, hybrid and stochastic constraints to capture use of resources, assumptions about an environment, as well as requirements on the services that the system has to provide. During the project, a large number of tools for probabilistic and real-time analysis were developed, refined, and combined, including MoDEST and Uppaal SMC.

The first presentation of Uppaal SMC was made at a Quasimodo project meeting in February 2011 in Aachen, with Joost-Pieter Katoen being the host and Kim G. Larsen (coauthor of this paper) giving the presentation. At that time Joost-Pieter Katoen was focusing on (exact) model checking of CTMC and stochastic hybrid systems against various specifications (including time automata), the probabilistic model checker MRMC, as well as compositional verification of probabilistic systems. The Quasimodo meetings were characterized by true scientific curiosity and frank discussions, and it is fair to say that the discussion after (and during) this first presentation of Uppaal SMC was *very* frank, with several of the other partners questioning the entire statistical model checking approach. Fortunately, we continued the work with Uppaal SMC becoming part of the standard Uppaal distribution in 2015 [16]. Throughout the years – in line with the core contribution of this paper – there has been a focus on constantly improving the performance of Uppaal SMC with contributions including importance sampling and splitting [24,26] for efficient estimation of rare events as well as distributed statistical model checking [9]. Most importantly, Uppaal SMC has since its introduction been successfully applied to the analysis of a number of highly critical systems ranging from maintenance of railway systems [31], performance analysis of routing protocols [35] to analysis of impact of close-down measures for controlling the spreading of Covid-19 [25,5]. Also, Joost-Pieter Katoen himself has made good use of Uppaal SMC with respect to the modeling and analysis of a microgrid with wind, microturbines, and the main grid as generation resources [10]. We hope all this work inspires

Joost-Pieter Katoen and his group to introduce statistical model checking into the STORM model checker [22].

Finally, in 2017 Joost-Pieter Katoen became Honoris doctor causa at Aalborg University in "recognition of his numerous contributions to the research community in model checking including probabilistic model checking and programming in particular".

Related Work GPUs have been explored in the context of model-checkers, e.g. for accelerating state-space generation [18], probabilistic model-checking [6], improving convergence of value-iteration [11] and speeding up SMT-solvers [28]. Additional examples of model checking using GPUs are GPUExplore [36] and Grapple [17], implemented using CUDA to perform state-space exploration in parallel. ParaFROST [30] uses GPUs to parallelize variable-clause elimination during bounded model checking with CBMC [12]. Closer to our approach is the work of Copik et al. [13] describing the implementation of an extension of *Prism* for statistic probabilistic model checking of Markov decision processes using GPUs, implemented using OpenCL. Similarly, the work of Gainer et al. [20] describes the use of CUDA to speed up model checking of parametric Markov chains. In contrast to previous work we extend the modeling language to Networks of Stochastic Extended Timed Automata and provide two ways of handling discrete data expressions: JIT-compiled C code and interpreted through Polish notation. To the best of our knowledge, this work is the first GPU-based statistical model checking engine for systems with mixed linear and non-linear behavior and complex interleaving stochastics.

The remainder of the paper is structured as follows; Section 2 builds up our modeling formalism from variables and automata to its semantics, introduces statistical model checking algorithms and NVIDIA CUDA programming framework, Section 3 presents our parallel version of the algorithm followed by a discussion in Section 4 on the required technical optimizations. This is followed by experimental findings in Section 5, shortly followed by conclusions in Section 6.

2 Preliminaries

We use *Timed Automata* [1] extended with discrete variables and given a stochastic semantics just like in statistical model checking (SMC) implemented in UPPAAL [3]. For parallel implementation of SMC algorithm we use NVIDIA CUDA framework.

2.1 Networks of Extended Stochastic Timed Automata (NSXTA)

Clocks and Discrete Variables. Let X be a finite set of *clocks.* A *clock valuation* is a function $\mu : X \to \mathbb{R}_{\geq 0}$. We use $\mathcal{V}(X)$ to denote the sets of all valuations for clocks in X. Let V be a set of *discrete variables.* The function D assigns to each variable $v \in V$ a finite domain $D(v)$. A *variable valuation* is a function

$\nu : V \to \bigcup_{v \in V} D(v)$ that maps variables to values such that $\nu(v) \in D(v)$. We use $\mathcal{V}(V)$ to denote the set of all variable valuations. We let μ_0 resp. ν_0 denote the valuation that maps every clock resp. variable to the value 0.

Expressions. We use expr to denote an expression over V. We assume that expressions are well typed. For an expression expr we use $D(\mathsf{expr})$ to denote its domain. Given a variable valuation ν and an expression expr, we use $\mathsf{expr}^{\nu} \in D(\mathsf{expr})$ to denote the value of expr under ν. We use $V(\mathsf{expr}) \in 2^V$ to denote the set of variables in expr such that if $\nu(v) = \nu'(v)$ for all $v \in V(\mathsf{expr})$ and for all $\nu, \nu' \in \mathcal{V}(V)$ then $\mathsf{expr}^{\nu} = \mathsf{expr}^{\nu'}$.

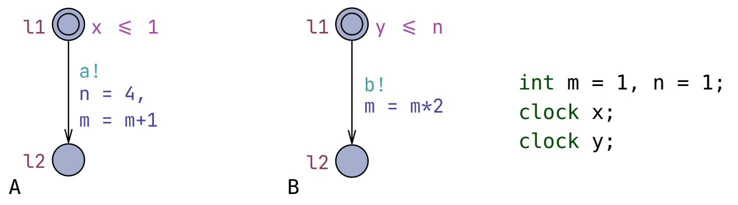
Constraints. The set $B(X)$ is the set of *clock constraints* generated by the grammar $\phi ::= x \bowtie \mathsf{expr} \mid \phi_1 \wedge \phi_2$, where $x \in X$, $D(\mathsf{expr})$ is the domain of all natural numbers $\mathbb{N}$ and $\bowtie \in \{<, \leq, \geq, >\}$. The set $B(V)$ is a set of *Boolean variable constraints* over V. The set $B(X, V)$ of constraints comprises $B(X)$, $B(V)$, and conjunctions over clock and variable constraints. Given a constraint $\phi \in B(X, V)$, we use $X(\phi)$ to denote the set of clocks in ϕ, and $V(\phi)$ to denote the set of variables in ϕ. A constraint $\phi \in B(X, V)$ evaluation under ν is denoted as ϕ^{ν}.

Updates. A *clock update* is of the form $x := \mathsf{expr}$ where $x \in X$, and $D(\mathsf{expr}) = \mathbb{N}$. A *variable update* is of the form $v := \mathsf{expr}$ where $v \in V$ and $D(v) = D(\mathsf{expr})$. The set $U(X, V)$ of *updates* contains all finite, possibly empty sequences of clock and variable updates. Given clock valuation $\mu \in \mathcal{V}(X)$, variable valuation $\nu \in \mathcal{V}(V)$, and update $r \in U(X, V)$, we use r^{ν} to denote the resulting update after evaluating all expressions in r under ν, we use $X(r)$ to denote the set of clocks in r, and $V(r)$ to denote the set of variables in r. We let $[\![r^{\nu}]\!] : \mathcal{V}(X) \cup \mathcal{V}(V) \to \mathcal{V}(X) \cup \mathcal{V}(V)$ be a map from valuations to valuations. We use $\mu[r^{\nu}]$ to denote the updated clock valuation $[\![r^{\nu}]\!](\mu)$. Analogously, for variable valuation ν', we use $\nu'[r^{\nu}]$ to denote the updated variable valuation $[\![r^{\nu}]\!](\nu')$.

Definition 1 (Extended Timed Automata XTA). *An* extended timed automaton *$\mathcal{A}$ is a tuple $(L, \ell_0, X, V, \Sigma, E, I)$ where: L is a set of locations, $\ell_0 \in L$ is the initial location, X is the finite set of clocks, V is the finite set of variables, Σ is a set of actions, $E \subseteq L \times \Sigma \times B(X) \times B(V) \times U(X, V) \times L$ is a set of edges between locations with an action, a clock guard, a variable guard, and an update set, $I : L \to B(X, V)$ assigns clock and variable invariants to locations.*

Semantics of an XTA. The semantics of an XTA is given by a timed transition system $(S, s_0, \to)$ where $S \subseteq L \times \mathcal{V}(X) \times \mathcal{V}(V)$ is the set of states comprising a location, a clock valuation, and a variable valuation, $s_0 = \langle \ell_0, \mu_0, \nu_0 \rangle$ is the initial state, and $\to \subseteq S \times (\mathbb{R}_{\geq 0} \cup \Sigma) \times S$ is the transition relation defined by: *Delay transitions* $\langle \ell, \mu, \nu \rangle \xrightarrow{d} \langle \ell', \mu', \nu \rangle$ with $d \in \mathbb{R}_{\geq 0}$, $\mu' = \mu + d$ and $\mu' \models I(\ell')^{\nu}$. *Discrete transitions* $\langle \ell, \mu, \nu \rangle \xrightarrow{a} \langle \ell, \mu', \nu' \rangle$ if exists $e = (\ell_i, a, \phi, \psi, r, \ell_i') \in E$ s.t. $\mu \models \phi^{\nu}$, $\nu \models \psi^{\nu}$, $\mu' = \mu[r^{\nu}]$, $\nu' = \nu[r^{\nu}]$, $\mu' \models I(\ell')^{\nu'}$, and $\nu' \models I(\ell)^{\nu'}$.

Given a transition system $(S, s_0, \to)$ we use $s \xrightarrow{d,a} s'$ as a shorthand for $s \xrightarrow{d} s'' \xrightarrow{a} s'$. A run is a finite (infinite) sequence of transitions $s_0 \xrightarrow{d_1,a_1} s_1 \xrightarrow{d_2,a_2} s_2 \ldots$. The set $\mathsf{En}(s) = \{a \in \Sigma \mid \exists d, s'.\ s \xrightarrow{d,a} s'\}$ is the set of enabled actions at



```
int m = 1, n = 1;
clock x;
clock y;
```

Fig. 1: NSXTA $\mathcal{N}_1$ with components `A` and `B` sharing two discrete variables `n` and `m` initialized with `1`.

s. Given state s and action a we define $T(s,a) = \{d \in \mathbb{R}_{\geq 0} \mid \exists s'.s \xrightarrow{d,a} s'\}$ and $T(s) = \bigcup_a T(s,a)$.

Definition 2 (Stochastic Extended Timed Automata (SXTA)). *A* Stochastic Timed Automaton *is a tuple* $(\mathcal{A}, w, r, \mathfrak{u})$ *where* $\mathcal{A}$ *is a timed automaton with local* [3] *semantics* $(S, s_0, \rightarrow)$*,* $w : \Sigma \rightarrow \mathbb{N}$ *assigns weights to actions,* $r : L \rightarrow \mathbb{N}$ *assigns rates to locations,* $\mathfrak{u}$ *is a family of probability measures,* $(\mathfrak{u}_s)_{s\in S}$*, such that* $\mathfrak{u}_s(T(s)) = 1$ *and if* $\sup\{T(s)\} < \infty$*, then* $\mathfrak{u}_s$ *is a uniform distribution on* $T(s)$*, otherwise* $\mathfrak{u}_s$ *is an exponential distribution with rate* $r(\ell)$*. For state* $s \in S$*. We assume a probability distribution* γ_s *over actions, such that for every action* $a \in \Sigma$*,* $\gamma_s(a) = 0$ *iff* $a \notin \mathsf{En}(s)$*, and otherwise* $\gamma_s(a) = w(a)/\sum_{a'\in\mathsf{En}(s)} w(a')$*.*

Network of Stochastic XTA (NSXTA). A formal definition for NSXTA requires additional notation and long definitions. We refer the reader to [15,4,24] for the precise semantics. In contrast we introduce NSXTA algorithmically and via an example. In a nutshell a network of SXTA $\mathcal{N} = (\mathcal{A}_i, w_i, r_i, \mathfrak{u}_i)$ refines the semantics of a network of XTA [27], non-deterministic time delays are refined by races and stochastic choices induced by $\mathfrak{u}_i$ and discrete non-deterministic choices are refined by probabilistic choices. The semantics of NSXTA assign a probability measure to sets of runs.

Example. As an example consider the network $\mathcal{N}_1$ of SXTA from Figure 1 with discrete variables n and m with initial value 1. The run $\langle \ell_2, \ell_1, x = y = 0.5, m = 2, n = 4\rangle \xrightarrow{3,b} \langle \ell_2, \ell_2, x = y = 3.5, m = 4, n = 4\rangle$ shows that variable m with value 4 can be reached in 3.5 timed units. The run $s_0 = \langle \ell_1, \ell_1, x = y = 0, m = n = 1\rangle \xrightarrow{0.6,b} \langle \ell_1, \ell_2, x = y = 0.6, m = 2, n = 1\rangle \xrightarrow{0.1,a} \langle \ell_2, \ell_2, x = y = 0.7, m = 3, n = 4\rangle$ shows that variable m with value 3 can be reached in 0.7 timed units. In particular we have that the probability of m to have value 4 within 2 time units, i.e. $\mathbb{P}(\mathcal{N}_1 \models \Diamond\, m = 4 \wedge x \leq 2) \approx 0.23$. Similarly, $\mathbb{P}(\mathcal{N}_1 \models \Diamond\, m = 3 \wedge x \leq 2) \approx 0.49$. In fact, the final value of m is decided by the outcome of the initial race between the components A and B for outputting a or b. Following the semantics of [15] the component with the smaller delay given by $\mathfrak{u}^A_{s_0}$ respectively $\mathfrak{u}^B_{s_0}$ wins the race. In the present case it is a fifty-fifty race given two uniform distributions on $[0,1]$. Note also, that component A writes on the shared variable n, which is used in the invariant (a uniform distribution) of the initial location of component B.

[3] as opposed to (global) semantics of network of timed automata.

This modification causes bigger delays which makes it less likely to reach $m = 4$ within two time units.

2.2 Statistical Model Checking

The core idea of Statistical Model Checking (SMC) [33] is to generate a number of independent runs for an NSXTA, while monitoring them with respect to a given temporal property. Standard statistics is then used to estimate the probability of system runs satisfying the property with some desired degree of confidence.

To generate a random run according to the stochastic semantics of NSXTA, we assume that the given network has a special clock $\hat{x}$ which is never updated. The clock is used as a time stamp. Algorithm 1 (as in [15,14]) describes the computation of a random run bounded by time horizon c. Line 3 computes the smallest delay (of the component winning the race). A delay can be infinite if no edge is enabled, e.g. a component is only expecting inputs. If the current time plus the smallest delay exceed the given time horizon c the algorithm returns. Otherwise at line 7 time elapses and at Line 8 an enabled action of the winning component is taken. The GPU implementation of Algorithm 1 is complex and a key contribution of this paper.

In this work we focus on the *Quantitative question* for a given NSXTA $\mathcal{N}$ and (time) bounded reachability property φ. Algorithm 2 [23] uses Chernoff-Hoeffding bound to estimate probability $\mathbb{P}(\mathcal{N} \vDash \varphi)$ by using N runs and providing an interval $p \pm \varepsilon$ with confidence level $1 - \alpha$ (where α is a probability that $\mathbb{P}(\mathcal{N} \vDash \varphi) \notin [p-\varepsilon, p+\varepsilon]$). Note that each run is sampled independently, therefore random variables derived from runs are independent and identically distributed, and thus Alg. 2 is suitable for a parallel implementation.

2.3 The GP-GPU Framework CUDA

CUDA is a programming interface developed by NVIDIA that allows issuing and managing data-parallel computations on specifically NVIDIA *Graphical*

Algorithm 1 Random run for $\mathcal{N} = \|_{i=1}^{n} (\mathcal{A}_i, w_i, r_i, \mathfrak{u}_i)$, state $\langle \vec{\ell}, \mu, \nu \rangle$, bound c

```
 1: $s := \langle \vec{\ell}_0, \mu_0, \nu_0 \rangle$, $run := (s)$          ▷ initialize with an initial state
 2: while $\mu(\hat{x}) < \mathsf{c}$ do
 3:     $d := min_{i=1}^{n}(delay(\mathfrak{u}_i^s))$          ▷ pick smallest delay of component $i$ from $s$
 4:     if $\mu(\hat{x}) + d \geq \mathsf{c}$ then          ▷ beyond time bound
 5:         $d := \mathsf{c} - \mu(\hat{x})$; return $run \oplus (\xrightarrow{d} \langle \vec{\ell}, \mu + d, \nu \rangle)$   ▷ concatenate remaining delay
 6:     else
 7:         let $\mu_d := \mu + d$ and $s' = \langle \vec{\ell}, \mu_d, \nu \rangle$          ▷ compute delay
 8:         pick $a \in \mathsf{En}(s')$ according to $\gamma_{s'}$          ▷ choose action
 9:         $run := run \oplus (\xrightarrow{d} s') \oplus (\xrightarrow{a} s'')$          ▷ concatenate delay and action
10:         $s := s''$
11: return $run$
```

*Processing Unit*s (GPU). It is a language extension of C and C++, however with certain limitations, such as no dynamic recursion or function pointers [29].

The CUDA interface discriminates between CPU and GPU computations, by viewing them as co-processing units named *host* and *device*, respectively. The *device* is viewed as a set of multiprocessors, each of which uses a *Single Instruction, Multiple Data* (SIMD) architecture. In this paradigm, each single sub-processor of a multiprocessor executes the same instruction at the same clock-cycle – although on different data. The *host* issues instructions to the *device* in the form of *kernels* – arbitrarily complex code units expressed as C++ functions. The kernels are executed by a *grid* of parallel threads. The grid is divided into equal size *blocks* with efficient memory sharing among threads within each block. The blocks are divided into SIMD groups called *warps* of synchronous execution where precisely 32 threads within each warp execute the same instruction. If some threads diverge within warp (e.g. due to conditional branching), then their instruction is executed on a different cycle from the rest, thus slowing down the overall progress.

The exact number of blocks and number of threads within a block is configurable upon each kernel launch. An example of a configuration could be 64 blocks with 256 threads per block, which would result in a grid of 16384 threads.

During the execution of a kernel, each *block* is mapped to a multiprocessor. Per clock cycle, the instruction unit of the multiprocessor will issue an instruction to a given *warp* from the set of *blocks* residing on the multiprocessor. This follows the SIMD architecture and allows the GPU a large degree of parallelism for computations where all threads in a given *warp* execute the same instruction. If the threads within a *warp* diverge in their execution path, the performance may be impacted. For example, for a given if-else statement **if (A) then B else C**, if A holds for at least one thread of a given warp, then the multiprocessor will execute the instruction for B, while the remaining threads of the warp will be *stalled*. Afterwards, when C is executed, the threads where A holds idle. By having this thread divergence, the warp may use additional clock cycles to execute instructions, significantly limiting parallelism. This implies that a *device* executes between $\#multiprocessors \cdot 1$ and $\#multiprocessors \cdot 32$ computations in parallel depending on the severity of thread divergence.

Algorithm 2 Prob. estimation of $\mathcal{N} \models \varphi$ with confidence $(1-\alpha)$ and error ε

```
1: $N := \ln(2/\alpha)/(2\varepsilon^2)$, $a := 0$
2: for $i = 1$ to $N$ do
3:     Use Alg. 1 to generate a random $run$ and observe random var $x := (run \models \varphi)$
4:     $a := a + x$                                   ▷ count satisfying runs
5: return $a/N$;          ▷ $\mathbb{P}(\mathcal{N} \models \varphi) \in [a/N - \varepsilon, a/N + \varepsilon]$ with confidence $(1-\alpha)$
```

3 Statistical Model Checking on GPU

Our tool SMAcc (Statistical Model-checking ACCelerated) [19] accepts NSXTA as a subset of the Uppaal specification language. Initially the input model is parsed in the *host* (CPU) and converted into an internal representation. After this, the model undergoes various optimization steps attempting to minimize thread divergence. These are discussed in detail in Section 4. In addition to facilitating computations on the GPU, CUDA facilitates running an almost equivalent code-base directly on the *host* CPU in a parallel fashion. We have utilized this feature to also allow for SMAcc to be executed on multi-core CPU systems.

Internally SMAcc represents an NSXTA as Abstract Syntax Trees and Expression Trees where each semantic component is decomposed into a tree with its leafs being atomic. This representation is designed for fast model traversal in order to perform analysis and optimizations efficiently. This structure further eases the migration of the model from the *host* to the *device* memory.

We utilize expression trees to represent user-supplied values and equations, such as update values, node exponential rates, edge weights, etc. We take particular care in implementing these tree structures such that they can be evaluated through a generic non-recursive post-order tree traversal algorithm using two stacks (one for operations and another for operands). This is particular to work around the limitations of CUDA where dynamic call stacks are not supported. This limitation implies that classical constructs using inheritance and recursion can not be utilized. In practice we determine the size of the required stack-size statically as $\mathcal{O}(depth(\mathsf{expr}))$ given that the expressions are evaluated bottom up. As we shall discuss later, these expression trees make the efficient implementation of a GPU accelerated SMC algorithm challenging, and remain a large source of thread divergence, as illustrated in Figures 2(a) and 2(b). We shall discuss different strategies of mitigation later in Section 4.

3.1 SMC Algorithm for the GPU

In a conventional use-case of SMC, e.g. for simulating disease propagation [25,5], the main challenge is to obtain a large number of simulations in limited time. For this work we thus focus on the parallelization of drawing a large set of

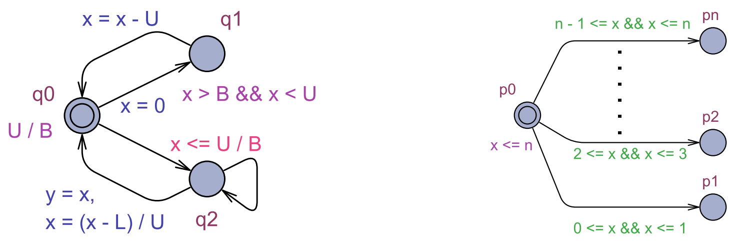


Fig. 2: Models triggering thread divergence due to: (a) expression complexity, (b) delay re-sampling.

Algorithm 3 CUDA probability estimation for network $\mathcal{N}$, property φ, confidence $(1-\alpha)$, error ε, and number of threads T

```
1: $N := \lceil (\ln \frac{2}{\delta})/(2\varepsilon^2) \rceil$; $N_t := \lceil \frac{N}{T} \rceil$; $A := \texttt{int}[T]\{0,\dots,0\}$
2: **Cuda parallel** $t \in \{0,\dots,T-1\}$ **do**                ▷ dispatch threads in warps
3:     **for** $i \in \{1,\dots,N_t\}$ **do**                      ▷ batch of simulations per thread
4:         observe $x := (\mathcal{N} \models \varphi)$ using GPU implementation of Alg. 1
5:         $A[t] := A[t] + x$                                      ▷ count satisfying runs per thread
6: **return** $\frac{1}{N_t \cdot T} \sum_{i=1}^{T} A[i]$          ▷ sum satisfying runs and divide by total
```

samples from the system. While the challenge of parallelizing a single simulation is interesting we leave that challenge for further work.

The GPU specific version of Alg. 2 is given in Alg. 3. The number of simulations each thread executes is predetermined and evenly distributed between all the threads in the execution (line 2), as to not introduce bias in the form of over-representing shorter traces through race-conditions [37]. This has the effect that the total number of simulations must be divisible by the number of threads used. As an example, with a 64-block 256-thread configuration, the total number of simulation runs must be divisible by the minimum number of simulations, which in this case is 16384. These threads are launched concurrently and run independently within their *warp* groups, as can be seen in line 2. A potential side-effect is the creation of *stragglers*: threads generating significantly longer simulations and thus taking longer computation times, which may become a bottleneck for simulation batches, especially when an entire *warp* group can be delayed by one straggler.

The simulation of a single trajectory concludes when one of a few conditions is met; the run can be bounded by time progression or a step counter, the trajectory has met a deadlock, or the monitored random variable has reached its terminal value (i.e. the property holds for the trajectory). This aligns with the semantics of properties as used in Uppaal [15]. The step-by-step progression of a single simulation trajectory follows the definition of NSXTA closely; for each individual XTA, sample the delay according to the delay distribution then resolve the race between the individual components in the network and conclude with sampling a winning out-edge in the winning XTA. This sampling procedure is also sketched in Alg. 1. Following this, the results from the individual simulations are aggregated (line 5) using dedicated counter $A[t]$ per thread t and thus avoid race conditions. Finally the result are tallied, normalized and returned (line 6).

3.2 Thread Divergence in SMC for NSXTA

For applications such as Computational Fluid Dynamics [2] and Value Iteration [11] both enjoy little to no thread-divergence, in part due to their roots in linear algebra. In the case of SMC for NSXTA, thread divergence is unavoidable for all but trivial stochastic systems, as traces of different executions naturally diverge, partly due to varied complexity of the expressions used in the system and partly due to the delay computation. In Fig. 2(a) we observe (1) that the complexity of the expressions used in invariants and guards differs and (2) that the out-degree of each location differs. For Alg. 1 this implies that the number (and sequence) of GPU instructions used for sampling of the *delay* on line 3 will vary depending on the location of the automata, leading to thread divergence. In the model seen in Fig. 2(b) we observe divergence related to the *delay* operation. Consider n threads sampling a delay for location p_0. At the first sample 50% of the threads (on average) will have picked a delay for which an edge can be taken. However, these 50% must await the remainder of the threads to re-sample and find a suitable delay. This re-sampling causes the successful branches to idle until all threads have completed the delay sampling. In the remainder of this work we address the former type of thread divergence and leave the latter to further work.

4 Optimizations

To fully utilize the computational capabilities of the GPU, we have implemented several optimization strategies as to maximize the parallelism and throughput of the simulations when executed on the GPU. The main cause of performance problems in CUDA programs is thread divergence. Our optimization strategies therefore largely focus on reducing thread divergence, either in size or frequency.

Expression Trees As discussed in Section 2.3, expression trees are a source of thread divergence when interpreted naively using the post-order-tree-traversal. Because expressions allow for ease of modeling complex arithmetic operations, these constructions are frequently used in practice. We attempt to combat this issue via two strategies: Just In Time (JIT) compilation of model expressions through CUDA `C` code, and interpreting the expression trees through Polish notation.

JIT compilation is accomplished through NVIDIA's runtime compilation library NVRTC. This translation is done by a straight-forward conversion of the internal representation of each model expression into a `C` equivalent formulation. Each expression is given a numerical identifier, and is added to a lookup table, implemented via a `switch` statement. This table is then injected to the source code of the engine, which is compiled at runtime. Using JIT compilation does not directly reduce thread divergence. However, it produces smaller and more uniform branches. In addition, the JIT compiler is at liberty to optimize and re-order the byte-code to be optimized towards the *device* architecture. However, JIT compilation of expressions introduces a significant overhead in preprocessing when running simulations – an effect observed in the experiments in Section 5.

Model expression interpretation through Polish notation is an alternative to JIT compilation, where the sequences of expression trees are translated to Polish notation and interpreted during simulation. This avoids having to traverse the expression trees at runtime, which vastly simplifies the evaluation. Furthermore, Polish notation has the added benefit of entirely eliminating branching caused by varying tree balance, while also limiting the branching caused by varying tree size.

Weakest Preconditions and Expression Simplification Existing tools such as UPPAAL implement the successor computation by first sampling delays, then applying guards, updates and finally invariants – with a potential reversal of the operation if the invariant is invalidated. This approach thereby has several potential points for thread divergence. This form of divergence can be limited to only a single branching point by "moving" the destination location invariant over to guards over edges that lead to such location: compute the weakest preconditions for the destination invariant after an update execution and add the resulting constraint to the guards. Additionally, we apply a set of identities to further simplify expressions and to remove trivial constraints added by computing weakest preconditions.

Shared Memory Shared memory is a part of the GPU's L1 cache memory, which can be accessed by all threads in a block and can be controlled by the programmer. The L1 memory bank of the device enjoys faster reading time compared to the global memory of the device. Typically, shared L1 memory is used for intra-block communication, increasing performance of parallel computing through synchronization of thread operations. We utilize this memory for storing the model that each thread will simulate. The L1 cache is though at a limited size, and is further restricted as the number of threads in a block dictates the amount of shared memory available, with up to 32 bytes per thread. Therefore, utilizing shared memory to store the model can be used if $threads \cdot 32B > modelsize$. In the future, the model representation can be further compressed by taking advantage of UPPAAL templates where many processes share the same automata structure.

5 Experiments

We experimentally show the applicability of GPUs for statistical model checking on stochastic systems through its use on seven different model families, whereof four are scalable problem instances. Specifically, our experiments are conducted with the CSMA and Aloha wireless communication protocol models [8], the Agent-Based and CTMC SEIHR Covid models [25], and the Bluetooth, Firewire

Table 1: Configurations of hardware used for the experiments. Experiments pitting different hardware platforms can be found in Appendix A, B and C.

	Power usage	SM/Cores	Clock speed	Release year
NVIDIA A100	250W	108	765 MHz	2020
2 × AMD EPYC 7642	2 × 225W	2 × 48	2.4 GHz	2019

and Fischer Uppaal SMC case study models [34]. The Fischer protocol model has modified timings s.t. multiple components can reside within the critical section concurrently, thus breaking the mutual exclusion of the protocol. We draw comparison between the performance of running the implementation on the CPU and GPU, using the hardware listed in Table 1, and CUDA version 11.8.

Furthermore, we investigate the impact of using the implemented optimization strategies from Section 4. As a final note, all experiments have been conducted with a kernel configuration of 64 blocks and 256 threads. The kernel configuration has significant impact on the GPU performance, and there is no way of deducing the optimal configuration for non-trivial kernel executions a priori. In addition, the optimal kernel configuration is dependent on both the specific model instance and the targeted GPU platform. This specific configuration has been chosen by sweeping through a grid of configurations and picking best performing largest numbers for CTMC Covid model.

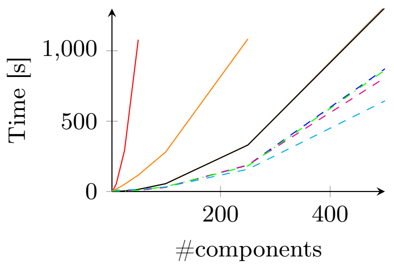
The parallel experiments executed on the CPU are conducted by oversubscribing the CPUs with threads; in preliminary experiments we found that a ratio of 10 threads per physical core yields good performance. We hypothesize that this is due to an uneven computational effort between within simulation batches, where the overall simulation time is straggled by a single thread having longer runs. This is similar to the effect observed across threads in the GPU simulation, and thus an artifact of the pipeline of SMAcc for the CPU execution also.

For each problem instance we test 8 different configurations:

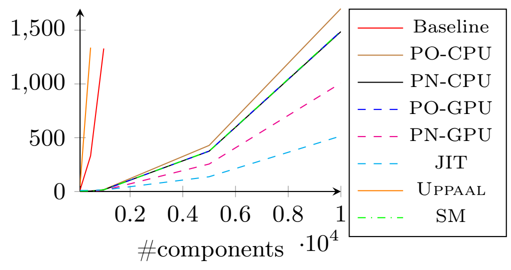
- Uppaal 4.1.26-2 statistical model checker, executed on a single core,
- a **Baseline**, executing SMAcc on a single core,
- a CPU (from Table 1) configuration using post order tree traversal **PO-CPU**, utilizing all the 96 cores of the CPU,
- a GPU (from Table 1) configuration using post order tree traversal **PO-GPU**, configured for 64 blocks of 256 threads,
- versions utilizing the Polish notation optimization for CPU (**PN-CPU**), executed on all 96 CPU cores,
- the GPU version with Polish notation optimization (**PN-GPU**),
- a configuration utilizing the **JIT** optimization – a feature limited to the GPU, vis-à-vis the restrictions of NVRTC framework, and
- a configuration utilizing the shared memory of the GPU (**SM**) when the problem instance fits into memory. Post order tree traversal is used for expression evaluation.

In all experiments we compute exactly 16384 simulations, corresponding to one simulation per GPU thread ($64 \cdot 256$). All experiments are limited to 1 hour of computation, and have been run 10 times giving an average runtime. The results have been compared to Uppaal to guarantee correctness.

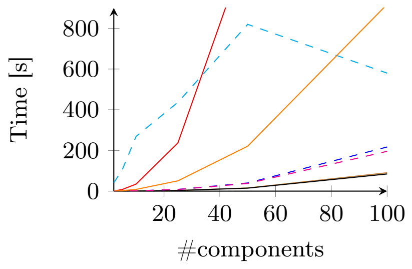
A note on the scalability of the models The scaling method of the scalable systems varies between models with and without local variables. Specifically, Aloha and Agent-Based SEIHR Covid are scaled by the number of components on the



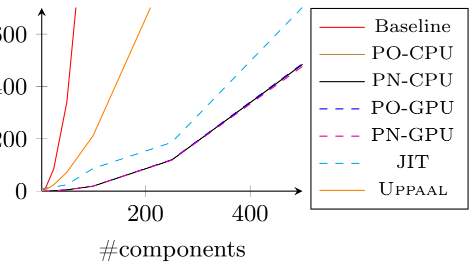
(a) ALOHA.



(b) Agent-Based SEIHR Covid.



(c) CSMA.



(d) Fischer Mutual Exclusion protocol.

Fig. 3: Simulation computation times (omitted points exceed 1h timeout).

same model, while CSMA and Fischer are scaled by duplicating the model and slightly changing the variables used for each component. Consequently, CSMA and Fischer suffer from increased model size for each added component thus requiring more memory to represent, making the Shared Memory optimization inapplicable. In addition, the Bluetooth and Firewire models are too large on their own to fit into shared memory. In these cases the **SM** configuration is omitted from the experiment. The only model families that fit within the Shared Memory are Agent-Based Covid, Aloha and CTMC Covid.

Discussion Let us initially compare the performance of Uppaal to the **Baseline** configuration. In general we observe that these two configurations are incomparable; Uppaal enjoys significantly lower computation times than the SMAcc baseline implementation in the ALOHA (Fig. 3a), CSMA (Fig. 3c), Fischer (Fig. 3d) and Firewire (Table 2) experiments. On the other hand, the **Baseline** configuration outperforms Uppaal on the remainder. We hypothesize that architectural differences, optimizations and fine-tuning are the cause of this discrepancy and leave it to further work to bring optimizations from Uppaal into SMAcc.

By comparing the **PO-GPU** and the **SM** configuration in Fig. 3a, Fig. 3b and Table 2 we observe marginal differences. We attribute this effect to the efficient memory-caches of the GPUs; for sufficiently small models, cache misses are rare, thus limiting its benefit of allocation directly in the shared memory. The largest improvements over the **Baseline** is enjoyed by the optimization techniques targeting thread divergence. Generally we observe that the Polish notation optimization has an insignificant effect for the CPU (**PN-CPU**) and only rarely a dramatic effect for the GPU (**PN-GPU**). Specifically the **PN-GPU** configuration enjoys good improvements for the Agent-Based SEIHR models (Fig. 3b) as the Bluetooth and Firewire models (Table 2). Importantly, the Polish notation never appears detrimental to the performance.

Contrary to **PN-GPU**, the **JIT** optimization has a varied impact on the performance. Specifically, **JIT** performs well on models where components have identical expressions, as is the case in the Aloha and Agent-Based Covid models.

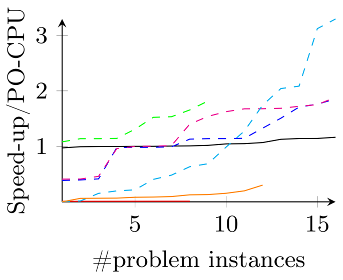
In the CSMA (Fig. 3c) and Fischer (Fig. 3d) models the **JIT** compilation instead causes a slowdown compared to the post order tree traversal **PO-GPU** implementation. We hypothesize that the effect of **JIT** compilation is especially subject to the number of unique expressions evaluated at a given time. Dissimilar expressions are guaranteed to cause thread divergence for **JIT**, whereas this is not necessarily the case for neither **PO-GPU** nor **PN-GPU**. Furthermore, for the CSMA case study we observe a surprising decrease in run time with increasing problem size. A functional error in the resulting GPU kernel is unlikely, as the obtained SMC results coincide. It is however plausible that larger problem sizes lead to better occupancy[1]. It may be the case that a larger problem results in more concurrent blocks being available, allowing the GPU to schedule interleaved work such that memory access can be coalesced, and that work can be performed on the same multiprocessor while some threads are waiting for data from device memory.

In Fig. 4a and Fig. 4b we see the cactus plots of the speed-up and power-ratio over all the experiments conducted. In these plots, the speedup (power-ratio respectively) is compared to the **PO-CPU** configuration using all 96 cores, sorted individually and then plotted. This implies that a given point in the x-axis may stem from different experiments for a given configuration. We observe in Fig. 4a that the **PN-GPU** configuration delivers the most stable performance compared to the reference with the **JIT** having a more varied effect on the performance. We can also observe that the Polish notation optimization has a generally positive impact on the performance for both GPU and CPU versions, albeit with a point-wise smaller degradation for the CPU-version. Studying instead the power consumption, we see in Fig. 4b that UPPAAL appears to be the most power effi-

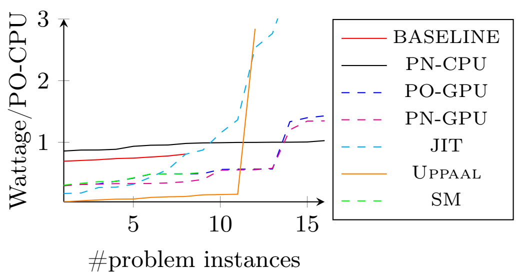
[1] `https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm`

Table 2: Performance measurements for various models in seconds.

Model	Baseline	PO-CPU	PN-CPU	PO-GPU	PN-GPU	JIT	UPPAAL	SM
CTMC Covid	263.633	3.777	3.405	2.080	**1.689**	11.019	565.199	2.295
Bluetooth	4.976	0.191	**0.165**	0.736	0.736	11.284	5.539	-
Firewire	9.315	**0.252**	0.277	0.811	1.689	10.914	1.197	-



(a) Speed-up relative to **PO-CPU** (higher is better).



(b) Power ratio relative to **PO-CPU** (lower is better).

Fig. 4: Cactus plots for speed-up and power ratio over all experiments. Problems solved by any configurations in $< 2sec$ are excluded.

cient configuration, however with notable exceptions and an inability to complete the computation within the provided 1 hour time limit. Across the majority of examples the **PN-GPU** and **PO-GPU** provide solid reductions in the power-usage, and peaking at a doubling of the power consumption compared to the reference. Studying Table 3, we observe that the degrading efficiency observed in Fig. 4b can be attributed to the CSMA model family. We hypothesize that these models exhibit the re-sampling branch divergence as illustrated by Fig. 2. Such divergence can be reduced by re-sampling only the processes affected by a recent transition, but it requires tracking the dependencies between transitions.

Table 3: Estimated power ratio of the GPU compared to the CPU: $(T_{GPU} \cdot P_{GPU} / T_{CPU} \cdot P_{CPU})$ where T and P denote the time and power consumption of the CPU and GPU by assuming that devices use max TDP for the entire computation time. The parentheses state the number of components in the scalable models. The best CPU and GPU configuration is chosen for each row.

Model	CPU(s)	GPU(s)	Speed-up	Power ratio
AB Covid (5K)	375.33	137.48	2.73	0.20
Aloha (500)	1306.40	644.42	2.03	0.27
CTMC Covid	2.31	1.69	1.37	0.41
Fischer (500)	484.23	476.94	1.02	0.55
CSMA (100)	84.49	195.65	0.43	1.29
Firewire	0.21	0.81	0.26	2.14
Bluetooth	0.13	0.74	0.18	3.16

6 Conclusion

In this work we show the applicability and advantages of GPU technologies in the context of Statistical Model Checking for Networks of Stochastic Extended Timed Automata. We presented SMAcc a prototype for performing SMC in NSXTA on the GPU. We have identified and experimented with several optimization and program transformation techniques to better accommodate the *Single Instruction Multiple Data* architecture employed by NVIDIA GPUs. Our experiments show reduced time and energy consumption which demonstrates the potential use of GPU technology for SMC algorithms for larger models. We observe that for one case GPU accelerated SMC uses as little as 20% of the energy of the CPU for completing an equivalent task while conducting the computation 2.73 times faster. Additionally we identify the bottlenecks of the current implementation, exemplified by a worse computation time and power consumption on single model instances.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, Apr. 1994.
2. T. Amada, M. Imura, Y. Yasumuro, Y. Manabe, and K. Chihara. Particle-based fluid simulation on GPU. In *ACM workshop on general-purpose computing on graphics processors*, volume 41, page 42. Citeseer, 2004.
3. G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer, 2004.
4. N. Bertrand, P. Bouyer, T. Brihaye, Q. Menet, C. Baier, M. Größer, and M. Jurdzinski. Stochastic timed automata. *Log. Methods Comput. Sci.*, 10, 2014.
5. A. Bilgram, P. G. Jensen, K. Y. Jørgensen, K. G. Larsen, M. Mikučionis, M. Muñiz, D. B. Poulsen, and P. Taankvist. An investigation of safe and near-optimal strategies for prevention of Covid-19 exposure using stochastic hybrid models and machine learning. *Decision Analytics Journal*, 5:100141, 2022.
6. D. Bosnacki, S. Edelkamp, D. Sulewski, and A. Wijs. Parallel probabilistic model checking on general purpose graphics processors. *International Journal on Software Tools Technology Transfer*, 13(1):21–35, 2011.
7. G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.
8. P. E. Bulychev, A. David, K. G. Larsen, A. Legay, and M. Mikučionis. Computing Nash equilibrium in wireless ad hoc networks: A simulation-based approach. *Electronic Proceedings in Theoretical Computer Science*, 78:1–14, feb 2012.
9. P. E. Bulychev, A. David, K. G. Larsen, M. Mikučionis, and A. Legay. Distributed parametric and statistical model checking. In J. Barnat and K. Heljanko, editors, *Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011*, volume 72 of *EPTCS*, pages 30–42, 2011.
10. S. Chakraborty, J. Katoen, F. Sher, and M. Strelec. Modelling and statistical model checking of a microgrid. *Int. J. Softw. Tools Technol. Transf.*, 17(4):537–554, 2015.

11. P. Chen and L. Lu. Markov decision process parallel value iteration algorithm on gpu. In *Proceedings of 2013 International Conference on Information Science and Computer Applications*, pages 299–304. Atlantis Press, 2013/10.
12. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
13. M. Copik, A. Rataj, and B. Wozna-Szczesniak. A GPGPU-based simulator for Prism: Statistical verification of results of PMC (extended abstract). In B. Schlingloff, editor, *Proceedings of the 25th International Workshop on Concurrency, Specification and Programming, Rostock, Germany, September 28-30, 2016*, volume 1698 of *CEUR Workshop Proceedings*, pages 199–208. CEUR-WS.org, 2016.
14. A. David, D. Du, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, and S. Sedwards. Statistical model checking for stochastic hybrid systems. *Electronic Proceedings in Theoretical Computer Science*, 92:122–136, aug 2012.
15. A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, J. van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In U. Fahrenberg and S. Tripakis, editors, *Formal Modeling and Analysis of Timed Systems*, pages 80–96, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
16. A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. Uppaal SMC tutorial. *Int. J. Softw. Tools Technol. Transf.*, 17(4):397–415, 2015.
17. R. DeFrancisco, S. Cho, M. Ferdman, and S. A. Smolka. Swarm model checking on the GPU. *Int. J. Softw. Tools Technol. Transf.*, 22(5):583–599, 2020.
18. S. Edelkamp and D. Sulewski. Efficient explicit-state model checking on general purpose graphics processors. In *Model Checking Software: 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings 17*, pages 106–123. Springer, 2010.
19. O. V. Eriksen, O. S. Bak, M. W. B. Christiansen, A. Windfeld, M. D. Jensen, S. Juozapaitis, P. G. Jensen, K. Guldstrand Larsen, M. Muniz, and S. Feo-Arenis. GPU accelerating statistical model checking for extended timed automata - artifact, Apr. 2024.
20. P. Gainer, E. M. Hahn, and S. Schewe. Accelerated model checking of parametric Markov chains. In S. K. Lahiri and C. Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 300–316. Springer, 2018.
21. I. R. Hasrat, P. G. Jensen, K. G. Larsen, and J. Srba. End-to-end heat-pump control using continuous time stochastic modelling and Uppaal Stratego. In Y. Aït-Ameur and F. Crăciun, editors, *Theoretical Aspects of Software Engineering*, pages 363–380, Cham, 2022. Springer International Publishing.
22. C. Hensel, S. Junges, J. Katoen, T. Quatmann, and M. Volk. The probabilistic model checker Storm. *CoRR*, abs/2002.07080, 2020.
23. T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 73–84, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
24. C. Jégourel, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, and S. Sedwards. Importance sampling for stochastic timed automata. In M. Fränzle, D. Kapur, and N. Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications*, pages 163–178, Cham, 2016. Springer International Publishing.

25. P. G. Jensen, K. Y. Jørgensen, K. G. Larsen, M. Mikučionis, M. Muñiz, and D. B. Poulsen. Fluid model-checking in Uppaal for Covid-19. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, pages 385–403, Cham, 2020. Springer International Publishing.
26. K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. Importance splitting in Uppaal. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part III*, volume 13703 of *Lecture Notes in Computer Science*, pages 433–447. Springer, 2022.
27. K. G. Larsen, M. Mikučionis, M. Muñiz, and J. Srba. Urgent partial order reduction for extended timed automata. In D. V. Hung and O. Sokolsky, editors, *Automated Technology for Verification and Analysis*, pages 179–195, Cham, 2020. Springer International Publishing.
28. G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 187–196, New York, NY, USA, 2010. Association for Computing Machinery.
29. NVIDIA. CUDA C++ programming guide, 2023. Last accessed 10 April 2024. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.
30. M. Osama and A. Wijs. GPU acceleration of bounded model checking with ParaFROST. In A. Silva and K. R. M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 447–460. Springer, 2021.
31. E. Ruijters, D. Guck, M. van Noort, and M. Stoelinga. Reliability-centered maintenance of the electrically insulated railway joint via fault tree analysis: A practical experience report. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 662–669. IEEE Computer Society, 2016.
32. A. Sasha Luccioni, S. Viguier, and A.-L. Ligozat. Estimating the carbon footprint of BLOOM, a 176b parameter language model. *arXiv e-prints*, page arXiv:2211.02001, Nov. 2022.
33. K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In R. Alur and D. A. Peled, editors, *Computer Aided Verification*, pages 202–215, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
34. Uppaal. Smc case studies, 2007. Last accessed 10 April 2024. `https://uppaal.org/casestudies/smc/`.
35. R. J. van Glabbeek, P. Höfner, M. Portmann, and W. L. Tan. Modelling and verifying the AODV routing protocol. *Distributed Comput.*, 29(4):279–315, 2016.
36. A. Wijs, T. Neele, and D. Bosnacki. Gpuexplore 2.0: Unleashing GPU explicit-state model checking. In J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 694–701, 2016.
37. H. L. S. Younes. Ymer: A statistical model checker. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification*, pages 429–433, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

A NVIDIA Tesla T4 vs dual AMD EPYC 7551

Tested using a configuration of 40 blocks and 256 threads, giving us 10240 sims. This has been tested using CUDA version 11.0, which lacks a few optimisations available in newer CUDA versions. These results are from a single run of the experiments.

Table 4: Different configurations of hardware used for the experiments.

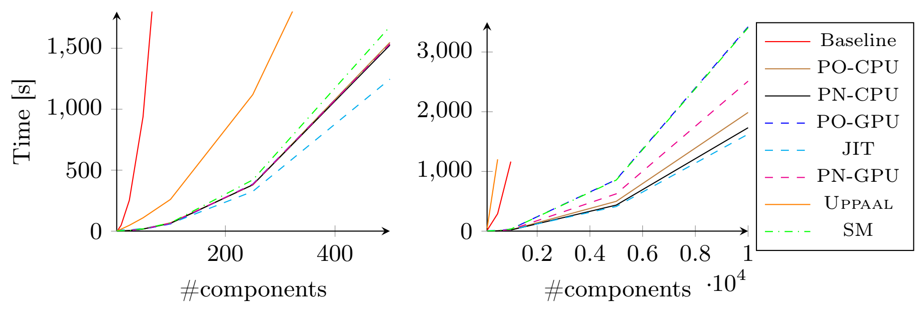
	Power usage	SM/Cores	Clock speed	Release year
NVIDIA Tesla T4	70W	40	585 MHz	2018
2 × AMD EPYC 7551	2 × 180W	2 × 32	2.0 GHz	2017

Table 5: The model families and their parameters. F indicates whether model fits in GPU shared memory.

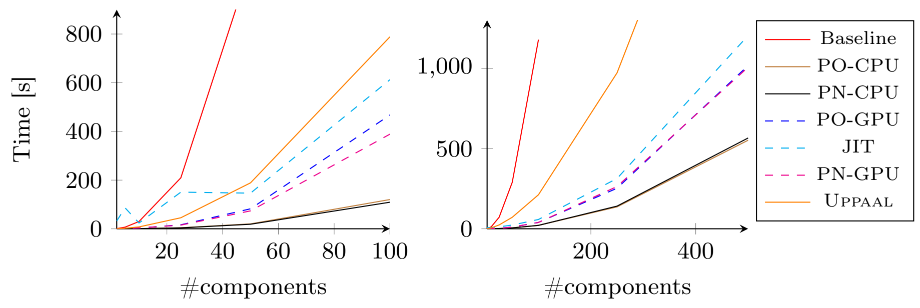
Model family	Property	#Components	F
AB Covid	`E[<=100;10240] (max: inf)`	100,500,1k,5k,10k	✓
Aloha	`E[<=100;10240] (max: nt)`	2,5,10,25,50,100, 250,500	✓
CSMA	`Pr[<=2000;10240](<> Proc(0).SUCCESS)`	2,5,10,25,50,100	×
Fischer	`E[<=300;10240] (max:in_critical)`	2,5,10,25,50,100, 250,500	×
CTMC Covid	`Pr[<=100;10240](<> I > 1000)`	5	✓
Bluetooth	`Pr[<=5000;10240](<> receiver1.Reply)`	4	×
Firewire	`Pr[<=1000;10240](<> node1.s5)`	4	×

Table 6: Tesla T4 results in seconds.

Model	Baseline	PO-CPU	PN-CPU	PO-GPU	PN-GPU	JIT	UPPAAL	SM
Bluetooth	2.718	**0.165**	0.265	0.868	0.969	8.490	5.858	-
CTMC Covid	158.445	3.428	2.818	3.574	**2.772**	10.844	513.376	3.523
Firewire	4.870	0.415	**0.265**	0.818	0.768	11.345	1.112	-

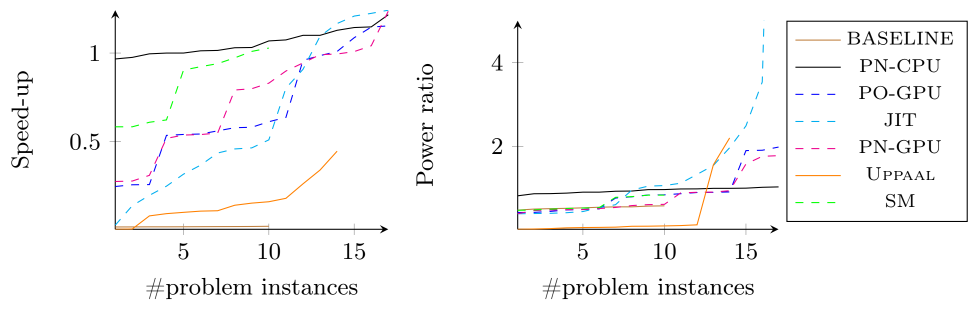


(a) Aloha. (b) Agent-Based SEIHR Covid.



(c) CSMA. (d) Fischer protocol.

Fig. 5: Tesla T4 results for scalable models (omitted points exceed 1h timeout).



(a) Speed-up cactus plot. (b) Power ratio cactus plot.

Fig. 6: Speed-up and power ratio comparison of Tesla T4 over PO-CPU.

Table 7: Comparison of CPU vs. GPU, estimating the power ratio of the GPU compared to the CPU: $(T_{GPU} \cdot P_{GPU}/T_{CPU} \cdot P_{CPU})$ where T and P denote the time and power-consumption of the CPU and GPU. The parentheses state the number of components in the scalable systems. The best CPU and GPU configuration is chosen for each row.

Model	CPU(s)	GPU(s)	Speed-up	Power ratio
Aloha (500)	1525.76	1246.60	1.22	0.16
AB Covid (5K)	436.30	412.94	1.06	0.18
CTMC Covid	2.82	2.77	1.02	0.19
Fischer (500)	551.89	1006.87	0.55	0.35
Firewire	0.27	0.77	0.29	0.68
CSMA (100)	109.28	388.67	0.28	0.69
Bluetooth	0.17	0.87	0.20	1.00

B NVIDIA RTX 3070 vs dual AMD EPYC 7642

Tested using a configuration of 46 blocks and 256 threads, giving us 11776 sims. This has been tested using CUDA version 11.8. These results are from a single run of the experiments.

Table 8: Different configurations of hardware used for the experiments.

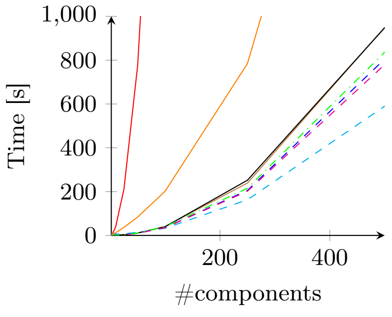
	Power usage	SM/Cores	Clock speed	Release year
NVIDIA RTX 3070	220W	46	1500 MHz	2020
2 × AMD EPYC 7642	2 × 225W	2 × 48	2.4 GHz	2019

Table 9: Model families and their parameters. F column indicates whether model fits in GPU shared memory.

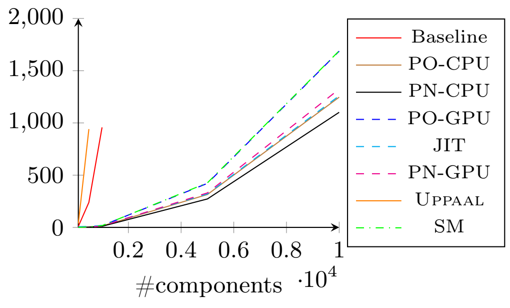
Model family	Property	#Components	F
AB Covid	`E[<=100;11776] (max: inf)`	100,500,1k,5k,10k	✓
Aloha	`E[<=100;11776] (max: nt)`	2,5,10,25,50,100, 250,500	✓
CSMA	`Pr[<=2000;11776](<> Proc(0).SUCCESS)`	2,5,10,25,50,100	×
Fischer	`E[<=300;11776] (max:in_critical)`	2,5,10,25,50,100, 250,500	×
CTMC Covid	`Pr[<=100;11776](<> I > 1000)`	5	✓
Bluetooth	`Pr[<=5000;11776](<> receiver1.Reply)`	4	×
Firewire	`Pr[<=1000;11776](<> node1.s5)`	4	×

Table 10: RTX 3070 results in seconds.

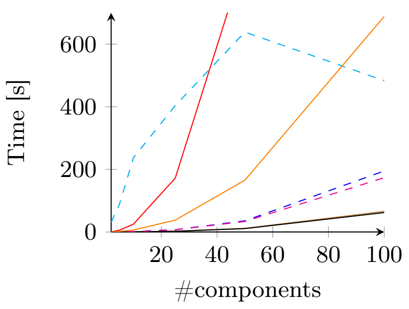
Model	Baseline	PO-CPU	PN-CPU	PO-GPU	PN-GPU	JIT	UPPAAL	SM
CTMC Covid	112.608	1.968	1.781	1.731	**1.378**	9.351	422.666	1.731
Bluetooth	2.220	0.216	**0.166**	0.368	0.368	7.988	4.041	-
Firewire	4.024	0.316	**0.216**	0.418	0.374	8.746	0.891	-



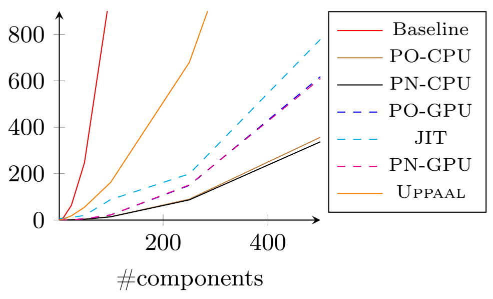
(a) Aloha.



(b) Agent-Based SEIHR Covid.

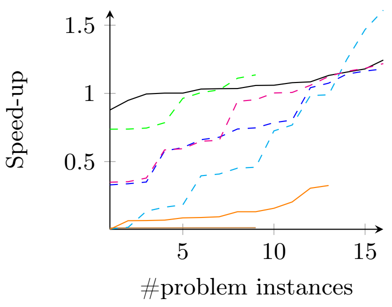


(c) CSMA.

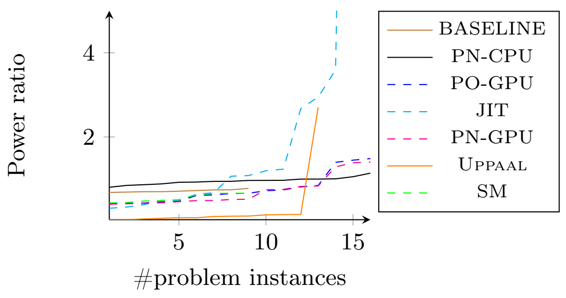


(d) Fischer protocol.

Fig. 7: RTX 3070 results for scalable models (omitted points exceed 1h timeout).



(a) Speed-up cactus plot.



(b) Power ratio cactus plot.

Fig. 8: Speed-up and power ratio comparison of RTX 3070 over PO-CPU.

Table 11: Comparison of CPU vs. GPU, estimating the power ratio of the GPU compared to the CPU: $(T_{GPU} \cdot P_{GPU} / T_{CPU} \cdot P_{CPU})$ where T and P denote the time and power-consumption of the CPU and GPU. The parentheses state the number of components in the scalable systems. The best CPU and GPU configuration is chosen for each row.

Model	CPU(s)	GPU(s)	Speed-up	Power ratio
Aloha (500)	947.96	590.05	1.61	0.30
CTMC Covid	1.78	1.38	1.29	0.38
AB Covid (5K)	272.65	320.45	0.85	0.57
Firewire	0.22	0.37	0.59	0.82
Fischer (500)	337.29	609.41	0.55	0.88
Bluetooth	0.17	0.37	0.46	1.06
CSMA (100)	62.09	173.38	0.36	1.37

C NVIDIA A100 vs dual AMD EPYC 7642

Tested using a configuration of 64 blocks and 256 threads, giving us 16384 sims. This has been tested using CUDA version 11.8. These results are from a single run of the experiments.

Table 12: Configurations of hardware used for the experiments. Experiments pitting different hardware platforms can be found in Appendix A, B and C.

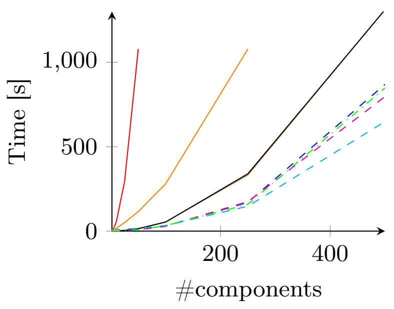
	Power usage	SM/Cores	Clock speed	Release year
NVIDIA A100	250W	108	765 MHz	2020
2 × AMD EPYC 7642	2 × 225W	2 × 48	2.4 GHz	2019

Table 13: Models and parameters. F indicates if model fits in GPU shared memory.

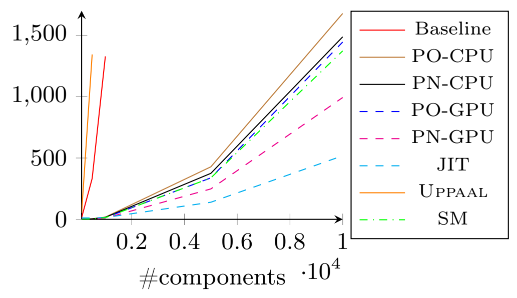
Model family	Property	#Components	F
AB Covid	`E[<=100;16384] (max: inf)`	100,500,1k,5k,10k	✓
Aloha	`E[<=100;16384] (max: nt)`	2,5,10,25,50,100, 250,500	✓
CSMA	`Pr[<=2000;16384](<> Proc(0).SUCCESS)`	2,5,10,25,50,100	×
Fischer	`E[<=300;16384] (max:in_critical)`	2,5,10,25,50,100, 250,500	×
CTMC Covid	`Pr[<=100;16384](<> I > 1000)`	5	✓
Bluetooth	`Pr[<=5000;16384](<> receiver1.Reply)`	4	×
Firewire	`Pr[<=1000;16384](<> node1.s5)`	4	×

Table 14: A100 results in seconds.

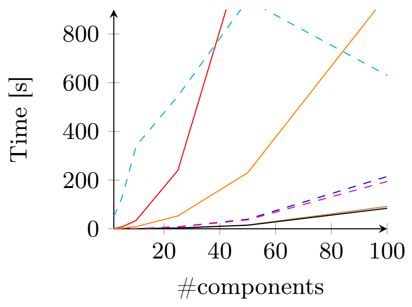
	Baseline	PO-CPU	PN-CPU	PO-GPU	PN-GPU	JIT	UPPAAL	SM
CTMC Covid	156.609	2.574	2.418	1.269	**1.119**	12.864	565.199	1.420
Bluetooth	2.972	0.166	**0.165**	0.266	0.265	13.464	5.539	-
Firewire	5.527	0.265	**0.215**	0.266	0.266	12.914	1.197	-



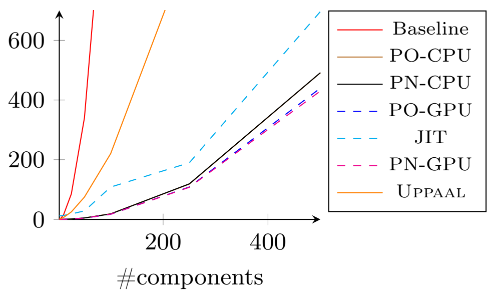
(a) ALOHA.



(b) Agent-Based SEIHR Covid.

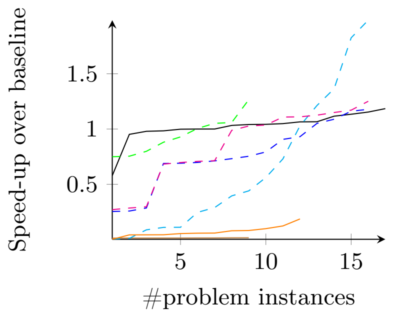


(c) CSMA.

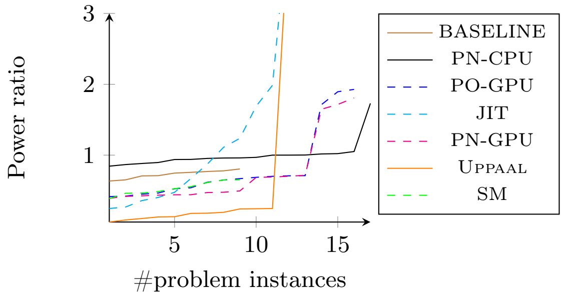


(d) Fischer protocol.

Fig. 9: A100 results for scalable models (omitted points exceeds 1 hour timeout).



(a) Speed-up cactus plot.



(b) Power ratio cactus plot.

Fig. 10: Speed-up and power ratio for A100 over CPU.

Table 15: Comparison of CPU vs. GPU, estimating the power ratio of the GPU compared to the CPU: $(T_{GPU} \cdot P_{GPU}/T_{CPU} \cdot P_{CPU})$ where T and P denote the time and power-consumption of the CPU and GPU. The parentheses state the number of components in the scalable systems. The best CPU and GPU configuration is chosen for each row.

Model	CPU(s)	GPU(s)	Speed-up	Power ratio
AB Covid (5K)	375.09	139.65	2.69	0.21
CTMC Covid	2.42	1.12	2.16	0.26
Aloha (500)	1311.32	649.20	2.02	0.28
Fischer (500)	491.23	430.13	1.14	0.49
Firewire	0.22	0.27	0.81	0.68
Bluetooth	0.17	0.27	0.63	0.88
CSMA (100)	84.13	194.38	0.43	1.28

D Kernel configuration

In order to determine well performing numbers of blocks and threads, we sweep through configurations. Fig. 11 shows that 40 blocks of 256 threads is the fastest configuration for CTMC Covid model.

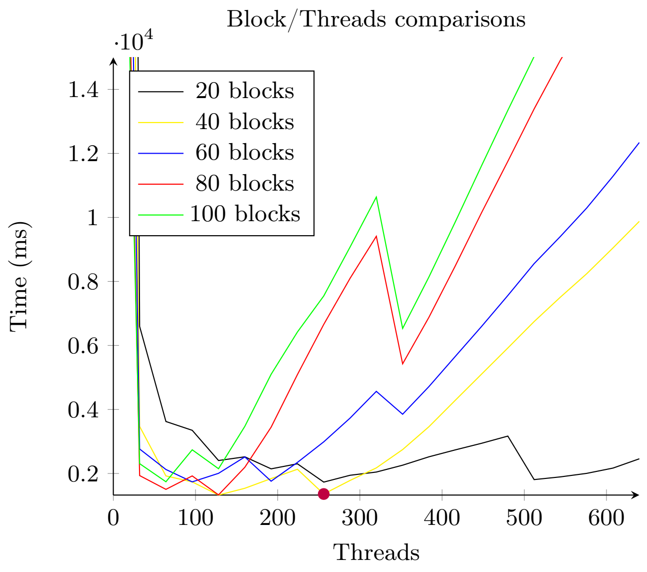


Fig. 11: Performance of various block/thread configurations, using the **PO-GPU** expression evaluation method on the CTMC Covid model. All experiments were run with 10k simulations on the NVIDIA Tesla T4 GPU. The red dot signifies the optimal configuration.

E Parallel Simulation on GPU

Listing 1.1: NVIDIA CUDA GPU C++ encoding of Alg. 3.

```
CPU GPU void simulate_automata(
    const unsigned idx,
    const network* model,
    const result_store* output,
    const sim_config* config)
{
    void* cache = static_cast<void*>(&static_cast<char*>(
        config->cache)[(idx*thread_heap_size(config)) / sizeof(char)]);
    curandState* r_state = &config->random_state_arr[idx];
    curand_init(config->seed, idx, idx, r_state);
    state sim_state = state::init(cache, r_state, model,
        config->max_expression_depth, config->max_backtrace_depth,
        config->max_edge_fanout);
    for (unsigned i = 0; i < config->sim_pr_thread; ++i) {
        const unsigned int sim_id = i + config->sim_pr_thread *
            static_cast<unsigned int>(idx);
        sim_state.reset(sim_id, model, config->initial_urgent,
            config->initial_committed);
        while (true) {
            // model->query->check_query()
            const int process = progress_sim(&sim_state, config);
            if (IS_NO_PROCESS(process))
                break;
            if (sim_state.models.store[process]->type == node::goal)
                break;
            do {
                const node* current = sim_state.models.store[process];
                const edge* e = pick_next_edge_stack(current->edges,
                    &sim_state);
                if (e == nullptr)
                    break;
                sim_state.traverse_edge(process, e->dest);
                e->apply_updates(&sim_state);
                sim_state.broadcast_channel(e->channel, process);
            } while (sim_state.models.store[process]->type == node::branch);
        }
        output->write_output(idx, &sim_state);
    }
}
```