STAMINA in C++: Modernizing an Infinite-State Probabilistic Model Checker

Joshua Jeppson[1], Matthias Volk[2], Bryant Israelsen[1], Riley Roberts[1], Andrew Williams[1], Lukas Buecherl[3], Chris J. Myers[3], Hao Zheng[4], Chris Winstead[1], and Zhen Zhang[1]

[1] Utah State University, Logan, UT, USA
{joshua.jeppson, bryant.israelsen, chris.winstead, zhen.zhang}@usu.edu, rileylroberts.97@gmail.com, andreww900@gmail.com
[2] University of Twente, Enschede, The Netherlands
m.volk@utwente.nl
[3] University of Colorado Boulder, Boulder, CO, USA
{lukas.buecherl, chris.myers}@colorado.edu
[4] University of South Florida, Tampa, FL, USA
haozheng@usf.edu

Abstract. Improving the scalability of probabilistic model checking (PMC) tools is crucial to the verification of real-world system designs. The STAMINA infinite-state PMC tool achieves scalability by iteratively constructing a partial state space for an unbounded continuous-time Markov chain model, where a majority of the probability mass resides. It then performs time-bounded transient PMC. It can efficiently produce an accurate probability bound to the property under verification. We present a new software architecture design and the C++ implementation of the STAMINA 2.0 algorithm, integrated with the STORM model checker. This open-source STAMINA implementation offers a high degree of modularity and provides significant optimizations to the STAMINA 2.0 algorithm. Performance improvements are demonstrated on multiple challenging benchmark examples, including hazard analysis of infinite-state combinational genetic circuits, over the previous STAMINA implementation. Additionally, its design allows for future customizations and optimizations to the STAMINA algorithm.

Keywords: Probabilistic Model Checking, Infinite-state Systems, Markov Chains

1 Introduction

Continuous-time Markov Chain (CTMC) can represent real-time probabilistic systems (e.g., genetic circuits [15], *Dynamic Fault Trees* (DFTs) [20]). Unfortunately, *probabilistic model checking* (PMC) may not always be feasible due to their infinite or finite but large state spaces. For instance, one might prefer to have unbounded species molecule count in a genetic circuit model due to insufficient information at design time,

which results in an infinite state space. Furthermore, the required explicit-state representation challenges scalable CTMC numerical analysis for verifying time-bounded transient properties. Existing PMC tools such as PRISM [13] and STORM [9] can efficiently analyze reasonably-sized *finite-state* CTMCs. INFAMY [8] approximates state-spaces with breadth-first search to some depth k and truncates any state beyond it. STAR [14] and SeQuaiA [5] approximate the most probable behavior for population Markov models of biochemical reaction networks. The STORM-DFT library [20] implements an approximation algorithm for DFTs based on partial state space generation of CTMCs.

STAMINA [16, 17, 19] performs *on-the-fly* state truncation using estimated state reachability probability to enable efficient PMC of CTMCs with an extremely large or infinite state space. STAMINA differentiates from the aforementioned techniques as follows: It can analyze both bounded and unbounded CTMCs and is not restricted to specific input models such as DFTs or biochemical population models. Specifically, DFTs typically incur an acyclic state space, whereas cycles commonly exist in genetic circuit models. Also, STAMINA does not truncate the state space with a fixed depth.

Contributions. This paper presents a new software architecture of STAMINA and a first re-implementation in C++ that interfaces the STORM model checker. Modularity plays a central role in STAMINA at both the core model builder class and other specialized classes. It allows optimized heuristics for state-space truncation and clean compartmentalization of functionality. Additionally, it includes memory optimizations tailored to the STAMINA 2.0 algorithm [19]. STAMINA demonstrated marked performance improvements on multiple challenging benchmarks, including hazard analysis of infinite-state genetic circuits, over the previous STAMINA implementation, due to both the speed of STORM's internals, and increased opportunities for optimization within STAMINA during its integration with STORM. STAMINA and STORM are both licensed under the GPLv3 license.

2 Overview of STAMINA

STAMINA takes an *unbounded* CTMC model in the PRISM modeling language and generates and truncates its state space where the probability mass resides. It calculates a probability bound $[\mathsf{P}_{\mathsf{min}}, \mathsf{P}_{\mathsf{max}}]$ based on a partial state space. STAMINA 1.0 [16, 17] truncates the state space by preventing expansion of states whose reachability fall below a threshold κ; and 2.0 [19] supports state re-exploration to obtain more accurate probability estimates. Both are written in Java and interface with the PRISM model checker. STAMINA has now been rewritten in C++ to integrate with STORM. It is available on GitHub and https://staminachecker.org with extensive documentation. It is now available via REST API for web-based usage.

The state truncation algorithm in STAMINA 1.0 [16, 17] performs breadth-first state expansion in multiple iterations. In every iteration, it terminates a state-transition path exploration *on-the-fly* when the estimated state reachability probability $\hat{\pi}(s)$ of state s, also called *terminal state*, falls below a user-specified bound κ. It walks through the explored state space at the end of each iteration to find all terminal states to be re-explored in the next iteration. It repeats this process until the change in state space size between iterations becomes sufficiently small. This method has the following main drawbacks:

(1) Inefficiency in repeatedly re-exploring the state space to find terminal states, and (2) inaccurate state space truncation due to its inability to update state reachabilities of previously visited non-terminal states in each iteration. STAMINA then interfaces the PRISM CTMC transient analysis engine to compute a probability bound $[\mathsf{P}_{\mathsf{min}}, \mathsf{P}_{\mathsf{max}}]$ that encloses the actual probability of the property under verification. STAMINA accepts non-nested time-bounded until CSL formulas. It computes $\mathsf{P}_{\mathsf{max}}$ by assuming the unexplored state space, abstracted as one artificial absorbing state, *satisfies* the property and $\mathsf{P}_{\mathsf{min}}$ by assuming it *violates* the property. If $\mathsf{P}_{\mathsf{max}} - \mathsf{P}_{\mathsf{min}} \not\leqslant w$, STAMINA reduces κ in order to further expand the state space until $\mathsf{P}_{\mathsf{max}} - \mathsf{P}_{\mathsf{min}} \leqslant w$. The user-specified probability window w allows the user to control the tightness of the result.

The STAMINA 2.0 algorithm [19] balances re-exploring states to distribute their estimated state reachability probabilities $\hat{\pi}$ and the number of states to re-explore. Specifically, it addresses both drawbacks in STAMINA 1.0 by repeatedly pushing the updated $\hat{\pi}(s)$ of state s to its successor states sufficiently frequently so that it prevents $\hat{\pi}$ from being ignored for states in a cycle in the state space. To optimize the performance, it limits the number of states to re-explore by setting a smaller amount to reduce κ in each iteration. It also reduces the expensive CTMC model checking procedure from two to one. Additionally, STAMINA 2.0 provides reasonable defaults for both κ and w, relieving the end-user's burden in guessing their appropriate values. The STAMINA tool presented in this work interfaces the STORM model checker, described next.

The STORM Model Checker. STORM [9] is a probabilistic model checker that supports the analysis of discrete- and continuous-time variants of both Markov chains and Markov decision processes. STORM is written in C++, open-source, and publicly available at stormchecker.org. It is one of the state-of-the-art tools for probabilistic model checking as witnessed in the Quantitative Verification Competition (QComp) [4]. The performance of STORM stems from its modular design where solvers, model representations, and model checking algorithms can easily be exchanged and combined. Lastly, STORM provides a rich C++ API which allows fine-granular access to its model checking algorithms and underlying datastructures. This allows other tools such as STAMINA to tightly integrate with STORM and benefit from its model checking performance.

Models such as CTMCs can be provided to STORM in many modelling languages including the PRISM and JANI modelling languages, DFTs, or generalized stochastic Petri nets. The Markov models can be built both in an explicit matrix-based or a symbolic decision-diagram-based representation. STAMINA uses the explicit representation based on an efficient sparse matrix data structure. State space generation within STORM is performed via the `NextStateGenerator` interface which iteratively generates successors states for a given state. Analysis of CTMC models can be performed with respect to CSL using efficient algorithms [3]. Time-bounded properties are solved via transient analysis of the CTMC using uniformisation and the approach by Fox and Glynn [7] for approximating Poisson probabilities. Time-unbounded properties are solved on the underlying discrete-time model using algorithms such as value iteration or via linear equation system solvers.

3 Software Architecture

STAMINA has been modularized as shown in Figure 1. It has a CLI entry point, but STAMINA/S's modularity allows alternate entry points into the program, which in the future will facilitate rapid development of a GUI. This entry point instantiates an instance of the `Stamina` class, which contains all of the information needed to run STAMINA. The static `Options` class allows the CLI and GUI to specify input parameters, such as w and κ. From there, the `Stamina` class instantiates one `ModelModify` and `StaminaModelChecker` objects, whose functions are explained below.

Model Building. The core of STAMINA is the `StaminaModelBuilder` base class. Although only one type is presented in this paper, this polymorphism allows multiple types of model builders, which reuse code for shared functionality. Since STORM requires that the CTMC transition rate matrix entries to be inserted in order, this base class orders transitions, connects terminal states to the artificial absorbing state, and keeps track of all other data needed for model checking with STORM. It employs a `ProbabilityState` data type to store the estimated state reachability probability and the state status, including whether the state is new, terminal, or deadlock.

To support different types of model builders, `StaminaModelBuilder` easily allows sub-classes to explore (and re-explore) states without enforcing an exploration order. For example, `StaminaReExploringModelBuilder` implements the algorithm in STAMINA 2.0 discussed in this paper. While it's exploration order is similar to breadth-first, other sub-classes, such as `StaminaIterativeModelBuilder` and its cousin `StaminaPriorityModelBuilder` explore states in a different order by nature of their heuristics. Additionally, generic design of the `StaminaModelBuilder` can eventually allow multithreading, which is under active development.

Specialized Classes. A number of specialized classes are delegated certain actions in order to compartmentalize functionality and increase modularity. The `ModelModify` class takes the non-nested time-bounded CSL property on time interval I, $\mathsf{P}_{=?}(\varphi U^I \phi)$ under verification, extracts the path formula ϕ, and automatically generates $\phi \wedge \neg\hat{s}$ and $\phi \vee \hat{s}$, respectively. $\hat{s}$ is the artificial absorbing state created by STAMINA. These are passed to the `StaminaModelChecker` class responsible for calling STORM to compute $\mathsf{P}_{\mathsf{min}} = \mathsf{P}_{=?}(\varphi\, U^I\, \phi \wedge \neg\hat{s})$ and $\mathsf{P}_{\mathsf{max}} = \mathsf{P}_{=?}(\varphi\, U^I\, \phi \vee \hat{s})$, respectively. Finally, there are static `StateSpaceInformation` and `StaminaMessages` classes which read information from the generated state-space and write log messages respectively.

Memory Optimization. Throughout its execution, STAMINA creates many small state-probability objects in memory. While in Java this is fairly fast (due to the built-in memory management), in C++ this is delegated to the OS, making it slower. One solution to this is to use a memory pool. Rather than using a memory-pool library data structure, STAMINA is cognisant of object lifetime. Since each state exists for the lifetime of the entire state space, the memory pool does not have a method to deallocate a single state, reducing bookkeeping and CPU usage.

The custom memory pool (`StateMemoryPool`) is optimized for the STAMINA algorithm. It makes one allocation request to the OS per "page" of size $B_{mp} = 2^{b_{mp}} \times \texttt{sizeof}(s)$ — where b_{mp} is an integer chosen at compile time based on system resources — and allocating the next B_{mp} instances from that block. These requests only occur when a particular "page" is full. B_{mp} is chosen to be reasonably large so that such

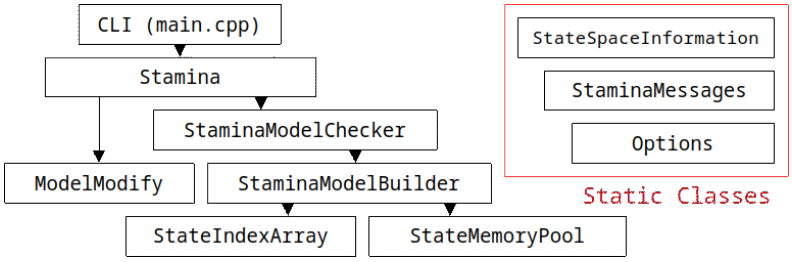

Fig. 1: Architecture of STAMINA: Arrows indicate object instantiations and data ownership. Classes in the red box are static and accessible everywhere.

requests do not happen frequently during state-space construction when state instances are allocated. All states are deallocated the end of `StateMemoryPool`'s lifetime. This, in tandem with the `StateIndexArray` class, allows STAMINA to access states by index in $O(N/B_{mp})$. Due to the high value of B_{mp}, while linear, this averages close to constant time on most models.

While some optimizations, such as keeping active tally of terminal states for w-estimate, are more general, many are possible due to differences in STORM and PRISM. PRISM provides an easy-to-use Java API which does most of the work under the hood, and STORM exposes more of its internals, allowing STAMINA to optimize more extensively. We acknowledge the importance of both tools during STAMINA's development; PRISM's coherent Java API allowed the first two versions of STAMINA to be developed quickly, and STORM's modularity was the inspiration for STAMINA's.

4 Results

All results were obtained on an AMD Ryzen Threadripper machine with a 16-core 3.5 GHz processor with 64 GB of RAM, running Debian 11 (Linux 5.10). Default parameters were used for all user-specifiable variables except where indicated. Both versions of STAMINA limit the probability window ($w = \mathsf{P}_{\mathsf{max}} - \mathsf{P}_{\mathsf{min}}$) to be $w \leqslant 10^{-3}$ for all models, as was used in [19]. The STAMINA 2.0 algorithm was tested in both STAMINA/PRISM and STAMINA/STORM for comparison. Note that STAMINA 1.0 was not implemented in STAMINA/STORM, because of the marked advantage of STAMINA 2.0 already demonstrated in [19]. The source code for STAMINA is available at [1].

Hazard Analysis in Genetic Circuits. The emerging *genetic design automation* (GDA) tools enables the design of genetic circuits. However, due to the inherent noisy behavior of biological systems, the predictability of genetic circuits remains largely unaddressed. The state space of models of genetic circuit designs are infinite, and therefore great case studies to test STAMINA. For this work, circuit 0x8E was selected to verify STAMINA's functionality. The circuit is part of 60 combinational genetic circuits designed and built as part of the development of the genetic design automation tool *cello* [18]. It has three inputs and one output, indicated by yellow fluorescence proteins. The circuit was specifically chosen since it exhibits an unwanted output behavior. While the output is supposed to be high throughout an entire input transition, it turns

Table 1: Probability comparisons. Digits in bold font show differences in STAMINA/PRISM and STAMINA/STORM.

Model	STAMINA/PRISM 2.0			STAMINA/STORM 2.0		
	$\mathsf{P}_{\mathsf{min}}$	$\mathsf{P}_{\mathsf{max}}$	w	$\mathsf{P}_{\mathsf{min}}$	$\mathsf{P}_{\mathsf{max}}$	w
010to100	0.3950**48522**	0.3951**15209**	6.66874E-05	0.39505**093**	0.3951**00628**	4.96987E-05
010to111	0.016**594466**	0.01**678343**	0.000188964	0.016**474768**	0.02**0078199**	0.003603431
100to111	0.016**627243**	0.01**6822966**	0.000195723	0.016**504701**	0.02**0246672**	0.003741971
111to010	0.694**56523**	0.694**808764**	0.000152241	0.694**61415**	0.694**745396**	8.39815E-05
100to010	0.4550**29708**	0.455**120325**	9.06176E-05	0.4550**34656**	0.455**085702**	5.10458E-05
111to100	0.735722**567**	0.735**846036**	0.00012347	0.735725**864**	0.735**790289**	6.44251E-05
000to011	0.82**6019153**	0.82**6159295**	0.000140142	0.82**5468467**	0.82**7041855**	0.001573389
011to101	0.989**516179**	0.9**89742961**	0.000226783	0.989**23085**	0.9**90289532**	0.001058682
000to101	0.990**235688**	0.990**472905**	0.000237217	0.990**019346**	0.990**893279**	0.000873933
101to000	0.864410**883**	0.8644**64495**	5.36124E-05	0.8644**17637**	0.8644**56068**	3.84307E-05
011to000	0.857436475	0.857504873	6.83981E-05	0.857436475	0.857504873	6.83981E-05
101to011	0.989**490726**	0.9**89802643**	0.000311917	0.989**025011**	0.9**90641893**	0.001616882
TQN2047	0.498962403	0.498968523	6.11917E-06	0.498962404	0.498968523	6.11917E-06
TQN4095	0.499263342	0.499269661	6.3189E-06	0.499263342	0.499269661	6.3189E-06
Polling20	1	1	0	1	1	0
JQN 4/5	0.865**393696**	0.865**546456**	0.00015276	0.865**436981**	0.865**452685**	1.57041E-05
JQN 5/5	0.819**441133**	0.82**0159054**	0.000717921	0.819**599345**	0.81**9810381**	0.000211036

off briefly before returning to a high output. Analysis by Fontanarrosa et al. showed that the behavior is explained due to a *function hazard* (i.e., a property of the function being implemented) [6]. To validate STAMINA, twelve models representing the 12 input transitions, shown as the first 12 rows in Tables 1 and 2, were analyzed to calculate the likelihood of that unwanted switching behavior.

Other Benchmarks. STAMINA was also evaluated on the same subset of examples from the PRISM benchmark suite [12] and the INFAMY case studies [2] as in [19], shown in the last few rows in Tables 1 and 2. One exception is the grid-world robot examples, since STAMINA does not yet support nested properties. The *Tandem Queueing Network* (TQN) models [10] consist of two interconnected queues with capacity 2047 and 4095, respectively, whose property queries the probability that the first queue is full before 0.23 time units. The *Jackson Queuing Network* (JQN) [11] models both have 5 as the arrival rate and contain 4 and 5 interconnected queuing stations, respectively. The property queries the probability that at least 4 and 6 jobs are in the first and the second queue, within 10 time units. Note that the property is satisfied in the initial state in the polling model, so the probability is always 1.0 and the runtime is negligible.

5 Discussion

In all but four tests, state count and probabilities, namely $\mathsf{P}_{\mathsf{min}}$ and $\mathsf{P}_{\mathsf{max}}$, were different between both tools. However, all cases are comparable and the probability win-

Table 2: State count and runtimes (in seconds) comparison for STAMINA 2.0.

Model	State Count		Runtime		
	STAMINA/P	STAMINA/S	STAMINA/P	STAMINA/S	Improve.(%)
010to100	85160	87902	47.79	28.67	40.00
010to111	3527020	2479199	3337.96	2843.89	14.8
100to111	3568525	2510758	3280.53	2873.05	12.42
111to010	467635	490145	283.53	236.78	16.49
100to010	165043	175194	100.78	73.61	26.95
111to100	406424	425443	216.10	210.31	2.68
000to011	2543695	1839612	2393.88	2109.79	11.87
011to101	2813395	2234057	2612.57	2262.65	13.39
000to101	2829690	2331091	2417.28	2557.20	-5.79
101to000	327687	337394	193.00	136.80	29.12
011to000	381372	381372	230.41	156.23	32.19
101to011	3006113	2302933	2896.84	2452.61	15.33
TQN2047	21293	21293	15.59	3.44	77.90
TQN4095	42469	42469	52.80	6.80	87.12
JQN4/5	187443	257265	33.38	59.88	-79.39
JQN5/5	1480045	1896415	419.47	577.40	-37.65

dows w always overlap as shown in Table 1. That is, $\mathsf{P_{min}}^P \leqslant \mathsf{P_{max}}^S$ where $\mathsf{P_{min}}^P$ is the lower bound provided by STAMINA/PRISM and $\mathsf{P_{max}}^S$ the upper bound from STAMINA/STORM. This is also true for the condition $\mathsf{P_{min}}^S \leqslant \mathsf{P_{max}}^P$, the lower bound from STAMINA/S and the upper bound from STAMINA/P, respectively. In order for the actual probability to exist within these bounds, both conditions must—and did—hold. Although the core algorithm is the same, the different but overlapping bounds may be due to differences such as property-based truncation and CTMC numerical analysis that exist in PRISM and STORM. There were several cases which STAMINA/STORM did not meet the requirement, i.e., $w \leqslant 10^{-3}$, due to a limit of 10 iterations in the outer while-loop in [19]. Since PMC occurs in each iteration of that loop, this limit exists in [19] and both versions of STAMINA to prevent excessive PMC with diminishing returns. However, after lifting this iteration limit, STAMINA/STORM was able to meet the window requirement. Increasing this limit resulted in the windows of the two reactions which transition to 111 to be 8.4×10^{-4} and 8.7×10^{-4}, in 16 iterations (4669.08 and 4754.01 seconds), respectively. While this increased runtimes, the worst case being the 100 to 111 circuit transition, modifying other parameters, such as a smaller κ, would reduce the number of iterations. All of these results are available online.

Table 2 shows that, STAMINA/STORM outperformed STAMINA/PRISM in most cases, with substantial reductions in runtime, except for the two JQNs, and one circuit model. However, for the JQN models, the runtime improved by over two times as the model scaled from $4/5$ to $5/5$. State counts are generally comparable. More optimization opportunities presented when interfacing STAMINA with STORM, although more considerations such as the order of state insertion into the transition matrix had to be made.

6 Conclusion

This paper presents the first C++ implementation of the infinite-state CTMC model checker STAMINA with increased modularity in software architecture and memory optimizations. By interacting with the CTMC analysis engine in STORM, STAMINA further improved model checking efficiency over the previous implementation, while maintaining comparable scalability. Future improvements to STAMINA may involve additionaly algorithmic efficiency, further optimizations with respect to STORM integration, tuning the algorithm for rare events, and more challenging benchmarks or requirements, such as bounding $w \leqslant 10^{-7}$. We also are working on several user-experience improvements to STAMINA, such as its REST API and a GUI, the former currently available for public use and the latter under active development.

Data availability. An artifact with both versions of STAMINA, as well as STORM and PRISM, and the testing suite used in this paper is provided in the open virtual appliance (OVA) format. It has been submitted to the QEST 2023 artifact evaluation, and is publicly available on STAMINA's website.

Acknowledgment We thank Tim Quatmann at RWTH Aachen University for his help with interfacing the STORM model checker. This work was supported by the National Science Foundation under Grant No. 1856733, 1856740, and 1900542. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

1. https://github.com/fluentverification/stamina-storm
2. https://depend.cs.uni-saarland.de/tools/infamy/casestudies/
3. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Software Eng. **29**(6), 524–541 (2003). https://doi.org/10.1109/TSE.2003.1205180
4. Budde, C.E., Hartmanns, A., Klauck, M., Kretínský, J., Parker, D., Quatmann, T., Turrini, A., Zhang, Z.: On correctness, precision, and performance in quantitative verification - QComp 2020 competition report. In: ISoLA (4). Lecture Notes in Computer Science, vol. 12479, pp. 216–241. Springer (2020). https://doi.org/10.1007/978-3-030-83723-5_15
5. Češka, M., Křetínský, J.: Semi-quantitative abstraction and analysis of chemical reaction networks. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 475–496. Springer International Publishing, Cham (2019)
6. Fontanarrosa, P., Doosthosseini, H., Borujeni, A.E., Dorfan, Y., Voigt, C.A., Myers, C.: Genetic Circuit Dynamics: Hazard and Glitch Analysis. ACS Synthetic Biology p. 15 (2020)
7. Fox, B.L., Glynn, P.W.: Computing poisson probabilities. Commun. ACM **31**(4), 440–445 (1988). https://doi.org/10.1145/42404.42409
8. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: Infamy: An infinite-state markov model checker. In: Proceedings of the 21st International Conference on Computer Aided Verification. pp. 641–647. CAV '09, Springer-Verlag, Berlin, Heidelberg (2009)
9. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. International Journal on Software Tools for Technology Transfer **24**(4), 589–610 (Aug 2022). https://doi.org/10.1007/s10009-021-00633-z
10. Hermanns, H., Meyer-Kayser, J., Siegle, M.: Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In: Plateau, B., Stewart, W., Silva, M. (eds.) NSMC. pp. 188–207 (1999)
11. Jackson, J.: Networks of Waiting Lines. Operations Research **5**, 518–521 (1957)
12. Kwiatkowsa, M., Norman, G., Parker, D.: The PRISM benchmark suite. In: Quantitative Evaluation of Systems, International Conference on(QEST). vol. 00, pp. 203–204 (09 2012). https://doi.org/10.1109/QEST.2012.14, doi.ieeecomputersociety.org/10.1109/QEST.2012.14
13. Kwiatkowska, M., Norman, G., Parker, D.: Prism 4.0: Verification of probabilistic real-time systems. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 585–591. CAV'11, Springer-Verlag, Berlin, Heidelberg (2011)
14. Lapin, M., Mikeev, L., Wolf, V.: Shave: Stochastic hybrid analysis of markov population models. In: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control. pp. 311–312. HSCC '11, ACM, New York, NY, USA (2011)
15. Madsen, C., Zhang, Z., Roehner, N., Winstead, C., Myers, C.: Stochastic model checking of genetic circuits. J. Emerg. Technol. Comput. Syst. **11**(3), 23:1–23:21 (Dec 2014). https://doi.org/10.1145/2644817, http://doi.acm.org/10.1145/2644817
16. Neupane, T., Myers, C.J., Madsen, C., Zheng, H., Zhang, Z.: STAMINA: Stochastic approximate model-checker for infinite-state analysis. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 540–549. Springer International Publishing, Cham (2019)
17. Neupane, T., Zhang, Z., Madsen, C., Zheng, H., Myers, C.J.: Approximation techniques for stochastic analysis of biological systems. In: Liò, P., Zuliani, P. (eds.) Automated Reasoning for Systems Biology and Medicine, vol. 30, chap. 12, pp. 327–348. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17297-8_12, https://doi.org/10.1007/978-3-030-17297-8_12

18. Nielsen, A.A.K., Der, B.S., Shin, J., Vaidyanathan, P., Paralanov, V., Strychalski, E.A., Ross, D., Densmore, D., Voigt, C.A.: Genetic circuit design automation. Science **352**(6281), aac7341 (2016). https://doi.org/10.1126/science.aac7341, https://www.science.org/doi/abs/10.1126/science.aac7341
19. Roberts, R., Neupane, T., Buecherl, L., Myers, C.J., Zhang, Z.: STAMINA 2.0: Improving scalability of infinite-state stochastic model checking. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 319–331. Springer International Publishing, Cham (2022)
20. Volk, M., Junges, S., Katoen, J.P.: Fast dynamic fault tree analysis by model checking techniques. IEEE Trans. Ind. Informatics **14**(1), 370–379 (2018). https://doi.org/10.1109/TII.2017.2710316