HyperProb: A Model Checker for Probabilistic Hyperproperties [⋆]

Oyendrila Dobe[1], Erika Ábrahám[2], Ezio Bartocci[3], and Borzoo Bonakdarpour[1]

[1] Michigan State University, East Lansing, MI, USA, {dobeoyen,borzoo}@msu.edu
[2] RWTH-Aachen, Aachen, Germany, abraham@informatik.rwth-aachen.de
[3] Technische Universität Wien, Vienna, Austria, ezio.bartocci@tuwien.ac.at

Abstract. We present HyperProb, a model checker to verify probabilistic hyperproperties on Markov Decision Processes (MDP). Our tool receives as input an MDP expressed as a PRISM model and a formula in Hyper Probabilistic Computational Tree Logic (HyperPCTL). By restricting the domain of scheduler quantification to memoryless non-probabilistic schedulers, our tool exploits an SMT-based encoding to model check probabilistic hyperproperties in HyperPCTL. Furthermore, when the property is satisfied, the tool can provide a witness that can be used for synthesizing a DTMC that conforms with the specification.

1 Introduction

Stochastic phenomena appear in many systems. In computing systems that interact with the physical environment, the modeling of physical processes is usually probabilistic due to environmental uncertainties like thermal fluctuations, random message loss, and processor failure. Probabilistic models can be also used as approximations to analyze very complex deterministic physical processes.

Certain requirements of computing systems prescribe the behavior of the system as a whole – they *simultaneously* argue about different system executions and compare observations on them. Such requirements are called *hyperproperties* [3]. Hyperproperties can describe the requirements of probabilistic systems as well. They generally express probabilistic relations between multiple experiments. For example, in information-flow security, adding probabilities is motivated by establishing a connection between information theory and information flow across multiple traces. A prominent example is probabilistic schedulers that open up an opportunity for an attacker to set up a probabilistic covert channel. Or, *probabilistic causation* compares the probability of occurrence of an effect between scenarios where the cause is or is not present. Also, the goal of *quantitative information flow* is to measure the amount of information leaked about a secret by observing different runs of a program.

As a more concrete example, consider the MDP in Fig. 1b, we can express the property where a computation tree starting from a state labeled *h0*, reaches

⋆ This work is sponsored in part by the United States NSF SaTC-1813388 grant.

a state labeled $l1$ with probability at least 0.5 as the classical PCTL formula: $h0 \wedge \mathbb{P}(\Diamond l1) > 0.5$. But we cannot use PCTL to express a property where all pairs of computation trees, starting at states labeled $h0$ and $h1$ respectively, reach a state labeled $\ell 1$ with equal probability. In a nondeterministic system, for all schedulers, the above property can be expressed using HyperPCTL [1,2] as follows:

$$\forall \hat{\sigma}.\ \forall \hat{s}(\hat{\sigma}).\ \forall \hat{s}'(\hat{\sigma}).\ \Big((h0_{\hat{s}} \wedge h1_{\hat{s}'}) \Rightarrow \big(\mathbb{P}\ (\Diamond\ \ell 1_{\hat{s}}) = \mathbb{P}\ (\Diamond\ \ell 1_{\hat{s}'}) \big) \Big)$$

In this paper, we introduce the tool HyperProb, a model checker for verifying probabilistic hyperproperties expressed in the temporal logic HyperPCTL [1,2] on Markov Decision Processes (MDP) given as a PRISM model [7]. HyperProb reduces the model checking problem to a satisfiability modulo theory (SMT) problem, implemented by the SMT-solver Z3 [4]. Furthermore, the tool may provide with the verdict of model checking accompanied by a witness or a counterexample represented as deterministic memoryless scheduler. A witness is provided when the probabilistic hyperproperty containing an existential quantifier over all the possible schedulers, holds, and it can be used to synthesize the induced discrete-time Markov chain (DTMC) satisfying the desired probabilistic hyperproperty. A counterexample is provided when the probabilistic hyperproperty containing a universal quantifier over all possible schedulers, does not hold, and it can be exploited to synthesize an adversarial attack that may violate the desired probabilistic hyperproperty.

This tool demonstration paper provides guide for potential users of HyperProb, focusing on the usage and implementation aspects of the tool. For details on theoretical and algorithmic aspects of HyperProb, we refer to [1,2]. Our implementation is available at: https://www.cse.msu.edu/tart/tools.

2 Input to the Tool

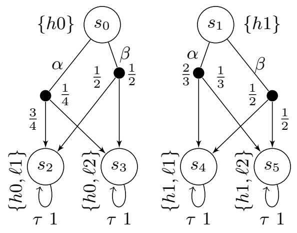
Input modeling language. The input model is provided as a high-level PRISM[4] program [7]. We illustrate the language here on a simple example shown in Fig. 1. This program takes a high-security input h and computes a low-security output ℓ. The execution steps are represented symbolically by probabilistic actions. For example, line 5 in Fig. 1a declares that action `alpha` can be executed if $\ell = 0$ and $h = 0$, and the action resets ℓ to the value 1 with probability $3/4$, and to the value 2 with probability $1/4$. Line 10 defines that $\ell = 0$ initially. Modeling each state explicitly yields the Markov decision process (MDP) depicted in Fig. 1b, where the state labeling with atomic propositions is defined in the lines 11-14 in Fig. 1a. For the given PRISM program, we use STORMpy to parse the model and to generate the underlying MDP.

Specification language: To protect the secret value h, there should be no probabilistic dependency between observations on the low variable ℓ and the value

[4] https://www.prismmodelchecker.org/

```
mdp
module basic_mdp
  h: [0..1];  // high input
  ℓ: [0..2]; // low output
  [alpha] (ℓ=0 & h=0) →3/4: (ℓ'=1) + 1/4: (ℓ'=2);
  [alpha] (ℓ=0 & h=1) →2/3: (ℓ'=1) + 1/3: (ℓ'=2);
  [beta]  (h=0) →  1/2: (ℓ'=1) + 1/2: (ℓ'=2);
  [beta]  (h=1) →  1/2: (ℓ'=1) + 1/2: (ℓ'=2);
  [tau]   (!ℓ=0) →1: true;
endmodule
init  (ℓ=0) endinit   //initial states
label "h0" = (h=0);   //h0 label on s0
label "h1" = (h=1);   //h1 label on s1
label "ℓ1"  = (ℓ=1); //ℓ1 label on s2,s4
label "ℓ2"  = (ℓ=2); //ℓ2 label on s3,s5
```

(a) PRISM model generating the MDP in Fig.1b



(b) MDP of the PRISM program in Fig 1a.

Fig. 1: An MDP and its corresponding PRISM program.

of h. For example, an attacker that chooses a scheduler that always takes action α from states s_0 and s_1 can learn whether or not $h = 0$ by observing the probability of obtaining $\ell = 1$ (or $\ell = 2$). Such properties, which compare observations on system execution when using different schedulers in different initial states are *probabilistic hyperproperties*. These properties, which should be model checked by HyperProb, can be specified in the temporal logic HyperPCTL [1]. We refer to [1] for a detailed description of HyperPCTL but show the tool's input grammar in Fig. 2. For example, the property that we expect from the MDP in Fig 1b is that for any scheduler (*AS sched.*) and any two initial states $s1$ and $s2$ ($A\ s1.\ A\ s2.$) with different high input values ($h0(s1) \wedge h1(s2)$), the probability to reach the low value 1 from $s1$ ($P\ F\ \ell 1(s1)$) is the same as from $s2$ ($P\ F\ \ell 1(s2)$), and analogously for the low value 2:

$$\begin{aligned}&\textit{AS sched. } A\ s1.\ A\ s2.\ (h0(s1) \wedge h1(s2)) \ {=}{>}\\ &\qquad (P\ F\ \ell 1(s1) = P\ F\ \ell 1(s2))\ \&\ (P\ F\ \ell 2(s1) = P\ F\ \ell 2(s2))\end{aligned}$$

Note that in the above formula we use expressions of the form $a(s)$ to state that the atomic proposition a holds in the computation tree starting from the (quantified) state s. For this formula, our tool will provide a violation as output with a deterministic memoryless scheduler that would leak information to an attacker.

3 Tool Structure and Usage

3.1 Implementation

HyperProb, is an optimized implementation of the algorithm presented in [1]. Given an MDP and a HyperPCTL property, it verifies if the given hyperproperty holds in the input MDP. Depending on the scheduler quantifier in the hyperproperty, if it holds, we get a witness to the property (in case of $\exists$) or if the

φ^{sched} ::= "AS" NAME "." φ^{state} | "ES" NAME "." φ^{state}
φ^{state} ::= ϕ | "A" NAME "." φ^{state} | "E" NAME "." φ^{state}
ϕ ::= "t" | "f" | NAME "(" NAME ")" | "∼" ϕ |
ϕ "&" ϕ | ϕ "|" ϕ | ϕ "=>" ϕ | ϕ "< − >" ϕ | c
c ::= p "<" p | p "<=" p | p "=" p | p ">=" p | p ">" p
p ::= "P" ψ | p "+" p | p "-" p | p "." p | NUM
ψ ::= "(X" ϕ ")" | "(" ϕ "U" ϕ ")" |
"(" ϕ "U[" NUM "," NUM "]" ϕ ")" | "(F" ϕ ")"

Fig. 2: Grammar defining HyperPCTL inputs to HyperProb, where the NUM token is a decimal number and NAME is a non-empty string.

hyperproperty does not hold, we get a counterexample in the model (in case of ∀). The witness or counterexample is defined by the actions chosen at each state to obtain the required DTMC.

The tool has been implemented using Python3 and depends on several python packages. Computer Arithmetic and Logic (CArL), is an open source C++ library for handling of complex mathematical computations and is needed by STORM. Carl-parser is an ANTLR based parser which is meant as an extension to CArL. pyCArL essentially provides python bindings for CArL and is a dependency for STORM. STORM is an academically developed probabilistic model checker. STORMpy is the python binding for STORM that we use to parse the input model. For the ease of usage, we have provided a docker image for the user. The image comes pre-installed with Ubuntu, all the required dependencies, and the tool. Docker makes it easier for the user to run the tool, as they do not have to install any of the dependencies mentioned above. The main advantage of docker is that running of the tool becomes independent of the operating system the user has. The overall view of the docker container can be seen in Fig. 3.

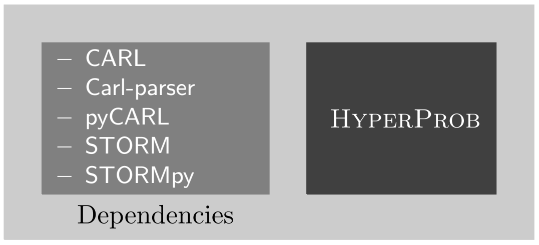


Fig. 3: Overview of the docker container with the tool and its dependencies

Inside the tool, as shown in Fig. 4 , we first parse the model using STORMpy [9] to store them in an optimized way and for easy retrieval of the details of the model like labels, transitions, states, and actions. We, then, parse the input hyperproperty into a syntax tree that allows us to recursively encode the property in the next stages. Using the parsed model, we first encode all possible actions in each state as SMT constraints. Using the parsed model and the parsed property, we encode, as SMT constraints, both, the quantifier combinations by generating constraints that should be satisfied at all states (for $\forall$) or by at least one state (for $\exists$), and the semantic interpretation of the operators in the formula.

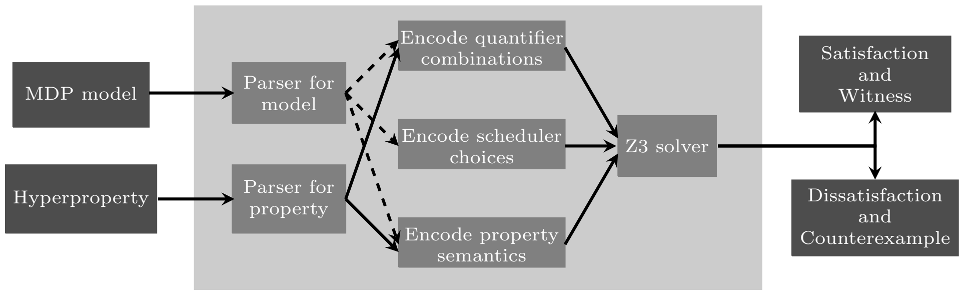


Fig. 4: Dataflow inside the tool

In the final step of the algorithm, we feed these constraints to the SMT solver Z3 [4]. If our scheduler quantifier is a $\exists$, and the constraints are satisfiable, the tool outputs a witness to the property, as a set of actions that should be chosen at each state of the model. If our scheduler quantifier is a $\forall$, and the constraints are unsatisfiable, the tool outputs a counterexample to the property as a set of actions that should be chosen at each state of the model. For all the other combinations of the scheduler quantifier and satisfiability, the tool returns a true/false but with no sequence of actions.

In our previous work [1], we present the HyperPCTL model checking algorithm for MDPs. HyperProb, incorporates additional optimizations whose impact is reflected in Table 1 (columns **N** referring to original implementation in [1], and **O** referring to the optimized implementation). In particular, instead of considering all possible state combinations when encoding a state formula for Z3, we consider only the relevant state combinations. For example, let us consider the scenario in Fig. 1 with six states, and its HyperPCTL specification as described in Section 2 with two state quantifiers. Our goal is to encode the atomic proposition $h0(s1)$. In the unoptimized version, this is encoded for 36 state combinations $((s_{0_1}, s_{0_2}), (s_{0_1}, s_{1_2}), \ldots, (s_{1_1}, s_{0_2}), \ldots, (s_{5_1}, s_{5_2}))$, since we have two copies of the MDP. However, in HyperProb, we consider the information that while encoding $h0(s1)$, only states of the second copy of the MDP are relevant. Hence, we encode it for just six state combinations, $((s_{0_1}, s_{0_2}), \ldots, (s_{0_1}, s_{5_2}))$. We keep the first state for the irrelevant copy as the first member of every state combination. This not only reduces the number of constraints generated but also relaxes the constraints Z3 needed work on.

3.2 Usage

In order to use the tool, we will first need to ensure that our system has docker [5] installed in it. Then, download the *docker image* and create a container from it. Inside the container, all dependency packages are installed in the */opt* folder under the root directory. The main tool package is located in */home/HyperProb* folder under the root directory. We can add our model file anywhere in this folder. The tool can be run by invoking the **source.py** script with appropriate inputs in the format,

python source.py **file_path_for_model hyperproperty**

Here *file_path_for_model* refers to the file path with respect to */home/HyperProb* directory as base and the hyperproperty is written according to the grammar in Section 2. For example, if your file is */home/HyperProb/models/mdp.nm*, your command would be,

python source.py **models/mdp.nm hyperproperty**

The commands to replicate all our case studies have been placed under */home/HyperProb/benchmark_files/Experiments.txt*.

4 Evaluation

4.1 Case Studies

Side-channel timing leaks open a channel for attackers to infer the value of a secret by observing the execution time of a function. For example, the heart of the RSA public-key encryption algorithm is the modular exponentiation algorithm that computes ($a^b \mod n$), where a is an integer representing the plaintext and b is the integer encryption key. A careless implementation can leak b through a probabilistic scheduling channel (see Fig. 5). This program is not secure since the two branches of the *if* have different timing behaviors. Under a fair execution scheduler for parallel threads, an attacker thread can infer the value of b by running in parallel to a modular exponentiation thread and iteratively incrementing a counter variable until the other thread terminates (lines 12-14). To model this program by an MDP, we can use two nondeterministic actions for the two branches of the *if* statement, such that the choice of different schedulers corresponds to the choice of different bit configurations `b(i)` for the key `b`. This

```
 1 void mexp(){
 2   c = 0; d = 1; i = k;
 3   while (i >= 0){
 4     i = i-1; c = c*2;
 5     d = (d*d) % n;
 6     if (b(i) = 1)
 7       c = c+1;
 8       d = (d*a) % n;
 9     }
10 }
11 ************
12 t = new Thread(mexp());
13 j = 0; m = 2 * k;
14 while (j < m & !t.stop) j++;
15 ************
```

Fig. 5: Modular exponentiation.

algorithm should satisfy the following property: the probability of observing a concrete value in the counter `j` should be independent of the bit configuration of the secret key `b`:

$$\forall\hat{\sigma}_1.\forall\hat{\sigma}_2.\forall\hat{s}(\hat{\sigma}_1).\forall\hat{s}'(\hat{\sigma}_2).\Big(init_{\hat{s}} \wedge init_{\hat{s}'}\Big) \;\Rightarrow\; \bigwedge_{\ell=0}^{m}\Big(\mathbb{P}(\Diamond(j=\ell)_{\hat{s}}) = \mathbb{P}(\Diamond(j=\ell)_{\hat{s}'})\Big)$$

Another example of timing attack that can be implemented through a probabilistic scheduling side channel is password verification. It is typically implemented by comparing an input string with another confidential string (see Fig 6). Also here, an attacker thread can measure the time necessary to break the loop, and use this information to infer the prefix of the input string matching the secret string.

```
int str_cmp(char * r){
  char * s = 'Bg\$4\0';
  i = 0;
  while (s[i] != '\0'){
    i++;
    if (s[i]!=r[i]) return 0;
    }
    return 1;
}
```

Fig. 6: String comparison.

Scheduler-specific observational determinism policy (SSODP) [8] is a confidentiality policy in multi-threaded programs that defends against an attacker choosing an appropriate scheduler to control the set of possible traces. In particular, given any scheduler and two initial states that are indistinguishable with respect to a secret input (i.e., low-equivalent), any two executions from these two states should terminate in low-equivalent states with equal probability. Formally, given a proposition h representing a secret:

$$\forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\big(h_{\hat{s}} \oplus h_{\hat{s}'}\big) \;\Rightarrow\; \bigwedge_{\ell\in L}\big(\mathbb{P}(\Diamond\,\ell_{\hat{s}}) = \mathbb{P}(\Diamond\,\ell_{\hat{s}'})\big)$$

where $\ell \in L$ are atomic propositions that classify low-equivalent states and $\oplus$ is the exclusive-or operator. A stronger variation of this policy is that the executions are stepwise low-equivalent:

$$\forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\big(h_{\hat{s}} \oplus h_{\hat{s}'}\big) \;\Rightarrow\; \mathbb{P}\Box\Big(\bigwedge_{\ell\in L}\big((\mathbb{P}\bigcirc\ell_{\hat{s}}) = (\mathbb{P}\bigcirc\ell_{\hat{s}'})\big)\Big) = 1.$$

Probabilistic conformance describes how well a model and an implementation conform with each other with respect to a specification. As an example, consider a six-sided die. The probability to obtain one possible side of the die is $1/6$. We would like to synthesize a protocol that simulates the six-sided die behavior only by repeatedly tossing a fair coin. We know that such an implementation exists [6], but our aim is to find such a solution automatically by modeling the die as a DTMC and by using an MDP to model all the possible coin-implementations with a given maximum number of states, including six absorbing final states to model the outcomes. In the MDP, we associate with the states, a set of possible nondeterministic actions, each of them choosing two states as successors with

Case Study		**Running time(s)**						**#SMT variables**		**#op**	**#st**	**#tr**
		SE		**SS**		**Total**						
		N	**O**	**N**	**O**	**N**	**O**	**N**	**O**			
TA	$m = 2$	5	2	< 1	< 1	5	2	8088	2520	14	24	46
	$m = 4$	114	18	20	1	134	19	50460	14940		60	136
	$m = 6$	1721	140	865	45	2586	185	175728	51184		112	274
	$m = 8$	12585	952	**TO**	426	**TO**	1378	388980	131220		180	460
PW	$m = 2$	5	2	< 1	< 1	6	3	8088	2520	14	24	46
	$m = 4$	207	26	40	1	247	27	68670	20230		70	146
	$m = 6$	3980	331	1099	41	5079	372	274540	79660		140	302
	$m = 8$	26885	2636	**TO**	364	**TO**	3000	657306	221130		234	514
TS	$h = (0, 1)$	< 1	< 1	< 1	< 1	1	1	1379	441	28	7	13
	$h = (0, 15)$	60	8	1607	< 1	1667	8	34335	8085		35	83
	$h = (4, 8)$	12	3	17	< 1	29	3	12369	3087		21	48
	$h = (8, 15)$	60	8	1606	< 1	1666	8	34335	8085		35	83
	$h = (10, 20)$	186	19	13707	1	13893	20	52695	13095		45	108
PC	s=(0)	277	10	1996	5	2273	15	21220	6780	44	20	188
	s=(0,1)	822	13	5808	5	6630	18	21220	6780		20	340
	s=(0..2)	1690	15	**TO**	5	**TO**	20	21220	6780		20	494
	s=(0..3)	4631	16	**TO**	7	**TO**	23	21220	6780		20	648
	s=(0..4)	7353	22	**TO**	21	**TO**	43	21220	6780		20	802
	s=(0..5)	10661	19	**TO**	61	**TO**	80	21220	6780		20	956
	s=(0..6)	13320	18	**TO**	41	**TO**	59	21220	6780		20	1110

Table 1: Experimental results and comparison. **TA:** Timing attack. **PW:** Password leakage. **TS:** Thread scheduling. **PC:** Probabilistic conformance. **TO:** Timeout. **N:** Prototype presented in [1]. **O:** HyperProb,. **SE:** SMT encoding. **SS:** SMT solving. #op: Formula size (number of operators). #st: Number of states. #tr: Number of transitions.

equal probability 1/2. Then, each scheduler corresponds to a particular implementation. Our goal is to check whether there exists a scheduler that induces a DTMC over the MDP, such that repeatedly tossing a coin simulates die-rolling with equal probabilities for all possible outcomes:

$$\exists\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\exists\hat{s}'(\hat{\sigma}).\Big(init_{\hat{s}} \wedge init_{\hat{s}'}\Big) \Rightarrow \bigwedge_{\ell=1}^{6}\Big(\mathbb{P}(\Diamond(die = \ell)_{\hat{s}}) = \mathbb{P}(\Diamond(die = \ell)_{\hat{s}'})\Big)$$

4.2 Results and Discussions

All of our experiments in section 4.1 were run on a MacBook Pro with a 2.3GHz quad-core i7 processor with 32GB of RAM. The results are presented in Table 1.

For our first case study **TA**, described in 4.1, models and analyzes information leakage in the modular exponentiation algorithm. We experimented with up to four bits for the encryption key (hence, $m \in \{2, 4, 6, 8\}$). The specification checks whether there is a timing channel for all possible schedulers, which is the case for the implementation in `modexp`.

Our second case study **PW**, described in 4.1 handles the verification of password leakage through the string comparison algorithm. Here, we experimented with $m \in \{2, 4, 6, 8\}$.

In our third case study **TS**, described in 4.1, we assume two concurrent processes. The first process decrements the value of a secret h by 1 as long as the value is still positive, and after this it sets a low variable ℓ to 1. A second process just sets the value of the same low variable ℓ to 2. The two threads run in parallel; as long as none of them terminates. A fair scheduler chooses for each CPU cycle the next executing thread. This opens a probabilistic thread scheduling channel and leaks the value of h. We compare observations for executions with different secret values h_1 and h_2 (denoted as $h = (h_1, h_2)$). There is an interesting relation between the data for **TS**. Both the encoding and running time for the experiment is proportional to the higher value in the tuple h.

Our last case study **PC**, described in 4.1, is on probabilistic conformance. The input is a DTMC modeling a fair 6-sided die as well as an MDP whose actions model single fair coin tosses with two successor states each. We are interested in finding a scheduler that induces a DTMC which simulates the die outcomes using a fair coin. Given a fixed state space, we experiment with different numbers of actions. In particular, we started from the implementation in [6] and for the state space of the die section of the protocol, we added all the possible nondeterministic transitions from the first state to all the other states (denoted $s = 0$), from the first and second states to all the others ($s = 0, 1$), and, similarly scaled it stepwise to include transitions from all states to all others ($s = 0 \dots 6$). Each time, we were not only able to satisfy the formula, but also obtain the witness corresponding to the scheduler satisfying the property.

In our previous prototypical implementation [1], due to encoding of all formula for all composed states, both the encoding as well as SMT solving time were significantly higher. Hence, we opted for a timeout for cases where the timing did not seem practically useful. For **TA**, **PW**, and **PC**, we used a timeout of 10000s for the SMT solving.

5 Conclusion

We introduced HyperProb, a fully automated tool for model checking probabilistic hyoerproperties expressed in the temporal logic HyperPCTL for DTMCs and as well as MDPs. HyperProb reduces the model checking problem to a satisfiability modulo theory (SMT) problem, implemented by the SMT-solver Z3 [4]. Furthermore, the tool may provide with the verdict of model checking, a witness or a counterexample represented as a deterministic memoryless scheduler. A witness is provided when the probabilistic hyperproperty contains an existential scheduler quantifier and it can be used to synthesize the induced discrete-time Markov chain (DTMC) satisfying the desired probabilistic hyperproperty. The counterexample is provided when the probabilistic hyperproperty contains a universal scheduler quantifier and it can be exploited to synthesize an adversarial attack that may violate the desired probabilistic hyperproperty. We also provided detailed experimental results that evaluate the effectiveness of our tool.

References

1. E. Ábrahám, E. Bartocci, B. Bonakdarpour, and O. Dobe. Probabilistic hyperproperties with nondeterminism. In *Proc. of ATVA'20*, volume 12302 of *LNCS*, pages 518–534, 2020.
2. E. Ábrahám and B. Bonakdarpour. HyperPCTL: A temporal logic for probabilistic hyperproperties. In *Proc. of QEST'18*, pages 20–35, 2018.
3. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
4. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS'08*, pages 337–340, 2008.
5. Docker. https://www.docker.com/get-started.
6. D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.
7. Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. CAV 2011*, volume 6806 of *LNCS*, pages 585–591, 2011.
8. T. M. Ngo, M. Stoelinga, and M. Huisman. Confidentiality for probabilistic multi-threaded programs and its verification. In *Proc. of ESSoS'13*, pages 107–122, 2013.
9. STORMpy. https://moves-rwth.github.io/stormpy/.