International Journal on Software Tools for Technology Transfer (2022) 24:589–610
https://doi.org/10.1007/s10009-021-00633-z

COMPETITIONS AND CHALLENGES

Regular

The probabilistic model checker STORM

Christian Hensel[1] · Sebastian Junges[2] · Joost-Pieter Katoen[1] · Tim Quatmann[1] · Matthias Volk[1]

Accepted: 23 June 2021 / Published online: 6 July 2021

Abstract

We present the probabilistic model checker STORM. STORM supports the analysis of discrete- and continuous-time variants of both Markov chains and Markov decision processes. STORM has three major distinguishing features. It supports multiple input languages for Markov models, including the JANI and PRISM modeling languages, dynamic fault trees, generalized stochastic Petri nets, and the probabilistic guarded command language. It has a modular setup in which solvers and symbolic engines can easily be exchanged. Its Python API allows for rapid prototyping by encapsulating STORM's fast and scalable algorithms. This paper reports on the main features of STORM and explains how to effectively use them. A description is provided of the main distinguishing functionalities of STORM. Finally, an empirical evaluation of different configurations of STORM on the QComp 2019 benchmark set is presented.

Keywords Verification · Model checking · Probilistic systems · Markov chain · Markov decision process

1 Introduction

The verification of systems involving stochastic uncertainty is a prominent research challenge. Among the many techniques is probabilistic model checking, a mature technique that grew out of model checking.

A model checker takes the formal system model and the formal property as inputs and, somewhat simplifying, returns one of three results, see Fig. 2. It reports that the property holds or is violated, and these reports are—given a correct implementation—guaranteed to be correct. The third outcome is that the model checker ran out of computational resources. Model checking has written numerous success stories [16,79], and major contributors Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis were awarded the Turing Award in 2007. *Probabilistic model checking* extends traditional model checking with tools and techniques for the analysis of systems involving random phenomena or other forms of behavior that can be approximated by randomization. Alur, Henzinger and Vardi [3] state: "A promising new direction in formal methods is probabilistic model checking, with associated tools for quantitative evaluation of system performance along with correctness." Distributed algorithms and communication protocols are natural examples, as they often use randomization to efficiently break symmetry. Another example are cyber-physical systems that tightly integrate software and hardware such as sensors, actors and microcontrollers. In particular, sensor readings may be noisy, actors may not always have the same effects, and physical components may fail. Other domains that give rise to models involving probabilistic aspects include, e.g., security protocols and systems biology. All these systems are naturally mapped to Markov models, and probabilistic model checking takes exactly such models as input.

Probabilistic model checking is not new. Initial theoretical results and algorithms for Markov chains [65,66] and Markov decision processes [37,116] were provided about thirty years ago. First tool support using explicit [53] and symbolic data structures [10,73] followed. Tool realizations for continuous-

✉ Joost-Pieter Katoen
katoen@cs.rwth-aachen.de

Christian Hensel
dehnert@cs.rwth-aachen.de

Sebastian Junges
sjunges@berkeley.edu

Tim Quatmann
tim.quatmann@cs.rwth-aachen.de

Matthias Volk
matthias.volk@cs.rwth-aachen.de

[1] RWTH Aachen University, Aachen, Germany

[2] University of California, Berkeley, CA, USA

time Markov chains appeared shortly thereafter [78]. Prism evolved as one of the main probabilistic model checkers[1] covering all these models in a symbolic way [91]. In more recent years, tool support extended to cover probabilistic real-time and hybrid systems, as well as multi-player games.

Meanwhile, research in probabilistic model checking continued, changed directions, and progressed in new application areas. The diversity of this field motivated the development of a modular and adaptive model checker, called Storm. Storm's main aim is to be a performant, easily extendible platform supplying various probabilistic model checking algorithms. After five years of development, Storm was released as open-source project in 2017 [41]. Despite its relative young age, Storm has established the following in pursuit of its original goals:

- In the first edition of QComp [60], Storm compared favorably with other model checkers. Consider the *quantile plot* in Fig. 1. The quantile plot expresses how many benchmark instances (on the x-axis) *each* were solved in at most the time given on the y-axis. In other words, the point $\langle x, y \rangle$ is contained in the quantile plot for tool c if the *maximal* runtime when using c on the x fastest solved instances (for c) is y seconds. Storm solved more instances and was generally faster in solving these instances. We elaborate these results in Sect. 7.4.
- Storm's modularity paid off in various occasions: The tool has been adapted to include various novel variants to the typical value iteration algorithm and has been extended with parameter synthesis for probabilistic systems and multi-objective model checking. In many of these areas, Storm has helped to push the state of the art considerably. We elaborate these results in Sect. 4.

In this paper, we report on Storm's main features and how to use them. We start with a very quick overview introducing Storm before elaborating the supported models and properties. We survey Storm's most prominent building blocks and unique features in greater detail and discuss the possibilities to interface with these features in Storm. Finally, we report on its internal tool architecture and provide some empirical evaluation of the main configurations of Storm on the QComp 2019 benchmark set.

A video tutorial covering Storm and some of its core features is available at

http://stormchecker.org/video-tutorial.

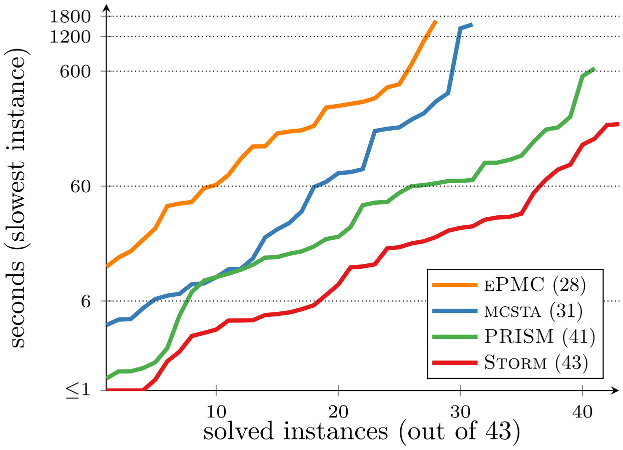


Fig. 1 Runtime comparison of general-purpose probabilistic model checkers taken from the QComp 2019 report [60] licensed under Creative Commons Attribution 4.0 International License: http://creativecommons.org/licenses/by/4.0/

2 Storm in a nutshell

Research to advance concepts and methods for probabilistic model checking often combines key routines and a variety of essential model checking algorithms. Storm delivers these. Some main characteristic features of Storm that help to push the state of the art in probabilistic model checking are that Storm

- contains efficient implementations of well-known and mature model checking algorithms for discrete-time and continuous-time Markov chains and Markov decision processes, but also for the more general *Markov automata* [49], a model containing probabilistic branching, nondeterminism, and exponentially distributed delays[2];
- supports *explicit state* and *symbolic* (BDD-based) model checking as well as a *mixture* of these modes to handle a wider range of models;
- has a *modular* setup, enabling the easy exchange of different solvers and distinct decision diagram packages; its current release supports about 15 solvers and two BDD packages.
- extends probabilistic model checking with the possibility of generating (high-level) counterexample [39], synthesizing permissive schedulers [46], symbolic bisimulation minimization [119,121] as well as game-based abstraction of infinite-state MDPs [120].

[1] Resulting in the Haifa Verification Conference 2016 Award.

[2] Markov automata can be used to provide a compositional semantics to modeling formalisms such as arbitrary generalized stochastic Petri nets [48], dynamic fault trees [20], and AADL extended with the error annex [22].

- offers the possibility to improve the reliability of model checking by supporting exact rational arithmetic using recent techniques [18] and techniques to avoid premature termination of value iteration [110].
- supports advanced properties such as multi-objective model checking [51,52,109], efficient algorithms for conditional probabilities and rewards [13], and long-run averages on MDPs [6,44] and MAs [28]. STORM also contains (the essential building blocks) for handling parametric models such as [38,108,114];

STORM can also be used to investigate the application of model checking in novel domains: In particular,

- STORM supports *various native input formats*: the PRISM and JANI languages, generalized stochastic Petri nets, dynamic fault trees, and conditioned probabilistic programs. This support makes it easier to apply probabilistic model checking, and amounts not to just providing another parser; state-space reduction and generation techniques as well as analysis algorithms are partly tailored to these modeling formalisms;
- besides a command line interface with many optional arguments, STORM provides a *Python API* facilitating easy and rapid prototyping of other tools using the engines and algorithms of STORM;
- it provides advanced approaches to model checking (see above) and good performance in terms of verification speed and memory footprint, cf. Fig. 1, under one roof.

How does STORM *relate to other probabilistic model checkers?* STORM has not reinvented the wheel, but has rather been inspired and learned from the successes of in particular PRISM [93] and the explicit model checker MRMC [88]. Like its main competitors PRISM, MCSTA [67], and EPMC [62], STORM relies on numerical and symbolic computations. Although many functionalities are covered by STORM, there are some significant areas that STORM has not been extended to. It does not support discrete-event simulation against temporal logic formulas, known as statistical model checking [2,97]. STORM does not support LTL model checking (as supported by EPMC and PRISM), does not support probabilistic timed automata (as supported by MCSTA and PRISM), has no equivalent of PRISM's hybrid engine (a crossover between full MTBDD and STORM's hybrid engine), and does not support the analysis of stochastic games. A longer survey of both features and performance of the various model checkers can be found in [26,60]. A detailed comparison between STORM, EPMC, MCSTA, and PRISM is given in [76].

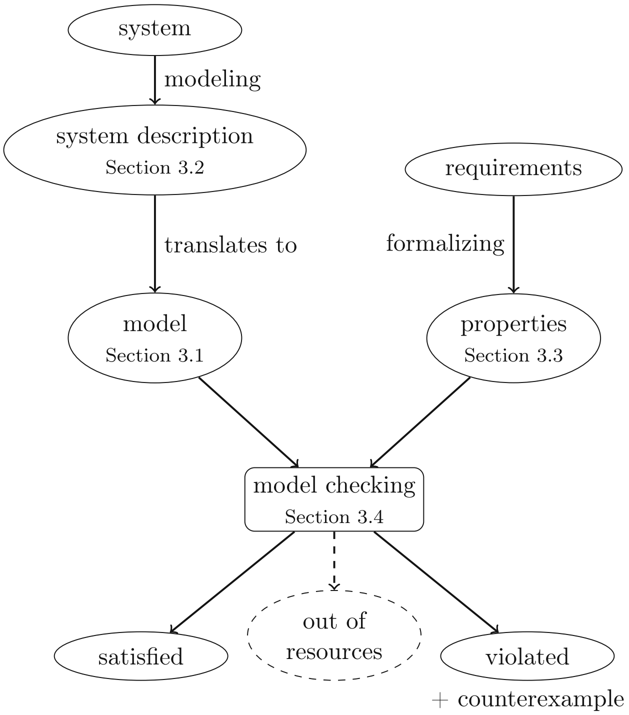


Fig. 2 Overview of the model checking approach [12]

Table 1 Overview of model types

	Deterministic	Nondeterministic
Discrete time	DTMCs	MDPs and PAs
Continuous time	CTMCs	MAs

3 Probabilistic model checking with STORM

We give a gentle introduction to probabilistic model checking[3] with STORM, clarifying the different parts as outlined in Fig. 2. For surveys and more formal introductions to probabilistic model checking, we refer to [9,12,86].

3.1 Model types

STORM supports the analysis of several different formalisms. They differ regarding (i) their notion of time and (ii) whether or not nondeterministic choices are allowed. Table 1 shows a categorization of the models supported by STORM along the two dimensions. In a third dimension, STORM supports partially observable models, in which the way nondeterminism is resolved is restricted.

Discrete-time models abstract from timing behavior by viewing the progression of time in terms of discrete steps. In contrast, continuous-time models use real numbers to model

[3] Readers familiar with probabilistic model checking may safely skip this section.

the flow of time and therefore have a dense notion of time. Deterministic models (also referred to as Markov chains from now on) behave purely probabilistically. Dually, in MDPs and MAs, nondeterministic choices can be used to model, for instance, the interaction with an adversarial environment or underspecification of the model with the goal to synthesize the optimal concrete system. In general, all model types can be enriched with cost structures. Together with the probabilities in the model this allows for reasoning over, for instance, expected costs until a certain goal is reached. Rather than providing formal definitions, we will illustrate a typical use case for each model type.

3.1.1 Discrete-time Markov chains (DTMCs)

We start with the simplest model. In DTMCs [103], every state is equipped with a single probability distribution over successor states. The evolution of the system therefore is fully probabilistic in the sense that it is governed only by repeated randomized trials. A famous example that can be captured in terms of a DTMC is the Herman protocol [94]. The general setting is this: a ring consisting of identical processes that each start either with a token or without one. If more than one process holds a token, the protocol is in an *unstable* state. The goal is to reach a configuration in which *exactly one* token remains, a situation called a stable configuration. This problem cannot be solved by deterministic algorithms and randomization is crucial. Herman's protocol uses synchronous, unidirectional communication and can be shown to eventually reach a stable configuration with probability 1.

3.1.2 Continuous-time Markov chains (CTMCs)

CTMCs [103] extend DTMCs with a continuous notion of time. Here, the sojourn time of the system in a state is also determined by a random experiment. More specifically, the time is sampled according to a negative exponential distribution. The transitions between states happen just like for DTMCs, i.e., governed by the associated probability distributions. Examples for CTMCs can be found in, for instance, systems biology [29]. In this work, they are used to analyze the effect of concentrations of proteins and reaction rates on signal transduction pathways. In other words, the model combines discrete aspects (the molecule concentration) and continuous aspects (time). Here, not only the probabilistic but also the timing effects are important: Since both the underlying chemical reactions and the spatial distribution of molecules take time, fundamental questions like "what is the probability that the concentration of X is high after 10 seconds?" require a proper modeling of time.

3.1.3 Markov decision processes (MDPs)

MDPs [107] extend DTMCs with nondeterminic actions. That is, instead of a single distribution governing the successor states, the system can nondeterministically select between several actions, each identifying a different distribution. After a selection has been made, the successor states are resolved probabilistically, and in the successor state, a new selection process is initiated. As already mentioned, nondeterminism can be used to model the possible interaction with an adversarial environment. An important example for this are distributed protocols. Such protocols are often randomized to efficiently break symmetry. However, because of their distributed nature, the progress of the processes is not synchronized and they may be scheduled differently. A well-known example is the randomized consensus algorithm by Aspnes and Herlihy [95]. In this protocol, the participating processes repeatedly modify a shared global counter based on the outcome of a coin flip until the whole system agrees on one of two outcomes, i.e., consensus has been reached. To faithfully model the protocol, nondeterminism can be used to account for the missing information about the scheduling of the competing accesses to the counter. Probabilistic automata (PAs) [112] extend MDPs with a more flexible action labeling.

3.1.4 Markov automata (MAs)

Finally, MAs [49] extend PAs using the notion of continuous time that CTMCs use. In probabilistic states no time passes, and the system nondeterministically selects one of the available probability distributions. In Markovian states, an amount of time passes that is distributed in a negative exponential manner, as in CTMCs. A well-known example is the stochastic job scheduling problem [109]. Here, the task is to schedule n jobs with (different) exponential service times onto k processors. The processors are assumed to run a preemptive scheduling strategy: Upon completion of any job, all k processors can take over any of the remaining jobs. The corresponding MA uses nondeterministic choices to model the assignment of jobs to processors whenever such a choice can be made. Thus, the nondeterminism is used to *underspecify* the concrete behavior. Determining the job assignment that maximizes the probability for completion within a given time limit can thus be seen as synthesizing a scheduling policy that one would like to impose in the actual system.

3.1.5 Partially observable MDPs (POMDPs)

Partially observable MDPs [7,85] are a popular extension that cater for a common issue with the analysis of MDPs. That analysis typically assumes that the nondeterminism can be resolved arbitrarily. The policy resolving the nondeterminism might, for example, depend on the internal state of a remotely running process. Consequently, the policies that are synthesized by such an analysis are *unrealistic*, and the verification results are too pessimistic. Consider a game like mastermind, where the adversary has a trivial strategy if it knows the secret they have to guess. Intuitively, to analyze an adversary that has to find a secret, we must assume it cannot observe this secret. For a range of privacy, security, and robotic domains, we may instead assume that the adversary must decide based on system observations. In widespread examples [85], the position of a robot is unknown and can only be determined by landmarks (such as doors), or the position of other agents in the same environment can only be observed if these agents are sufficiently close.

Formally, POMDPs extend MDPs by a set of observations and label every state with a one of these observations. Extensions in which actions are labeled or where states are labeled with distributions over observations can be reduced to this simpler case.

3.2 Modeling languages

Markov models for practical purposes are often too large to denote explicitly, but may be described by various more powerful and concise modeling languages. Depending on the domain, different modeling languages are more or less suitable. Furthermore, the structure of the model is often more apparent from a symbolic description than on the state level. STORM therefore tries to support a variety of different input languages. In order to be compatible with the widespread usage of PRISM, the PRISM language is supported. For testing small models, explicit enumeration of states and transitions is supported in two different formats. Furthermore, STORM accepts models given in JANI [25], a modeling language that was devised in a joint effort across multiple tools (involving EPMC, MODEST, FIG) in an attempt to unify the cluttered language landscape. STORM supports three other modeling languages. First, the user can input generalized stochastic Petri nets (GSPNs) [100] specified in an extension of the Petri net Markup Language PNML, which is then translated to JANI automatically. GSPNs are an important modeling formalism in dependability and performance evaluation. Secondly, dynamic fault trees (DFTs) are a means to specify the fault behavior of systems and is a reliability engineering formalism that is widely used in industry [111]. DFTs can be specified in the GALILEO format [115]. Finally, a recent trend in the analysis of probabilistic systems is probabilistic programming [55]. The latter refers to programs written in a probabilistic extension of regular programs. An extension to imperative while programs is PGCL [74], and can additionally be extended with statements expressing conditional reasoning [104], an ingredient that is essential to describe inference as in Bayesian networks. STORM can parse and translate programs written in PGCL to JANI, which makes such programs amenable to existing probabilistic model checking techniques.

3.3 Properties

STORM offers support for a multitude of properties. The most fundamental properties are reachability properties. Intuitively, they ask for the probability with which a system reaches a certain state. One may, e.g., ask

- "is the probability to reach an unsafe state of the system less than 0.1?"
- "is the probability to reach a target within 20 steps at least 0.9?"

For models involving nondeterministic choices, such an analysis will reason about all possible resolutions of nondeterminism and assert that the desired property holds *in all cases*. Alternatively, an easy extension is to ask for *some* resolution of the nondeterminism such that the property holds. Besides asking for whether the probability meets some threshold, one may also ask "what is the probability to reach an unsafe state of the system?."

As models can be equipped with cost structures, properties allow for retrieving, e.g.,

- "what is the expected number of coin flips until consensus has been reached?"
- "what is the expected energy consumption after t time units?"
- "what is the expected molecule concentration at time point t?"

Further properties include temporal logic formulas based on PCTL [66] and CSL [8,11], conditional probability and cost queries [13,14], long-run average values [6,28,44] (also known as steady-state or mean payoff values), cost-bounded properties [69] (see Sect. 4.2), and support for multi-objective queries [51,109] (see Sect. 4.5).

3.4 Model checking methods

In probabilistic model checking and arguably in verification in general, (sadly) there is no known "one-size-fits-all" solution. Instead, the best tools and techniques depend heavily on the input model and the properties. STORM—as well as other model checkers—implements a variety of approaches that allow a knowledgeable user to pick the appropriate method as part of the input, and allows developers to extend and combine their favorite methods. In particular, we provide approaches based on solving (explicit) linear (in)equation systems, value iteration variants on explicit or symbolic representations of (parts of the) model, policy iteration methods, methods using abstraction techniques and bisimulation minimization. We refer to Sect. 4 for some of STORM's distinguishing features for model checking, and Sect. 6 for specifics on the technical realization.

4 STORM's features

In this section, we detail some of the outstanding features of STORM that go beyond conventional probabilistic model checking methods. We give an overview in Table 2.

In particular, we have chosen four aspects that improve probabilistic model checking of standard properties such as reachability or expected rewards. These are reflected by the first four rows. *Sound/exact* model checking reflects a collection of approaches that, compared to the classical numerical algorithms, provide stronger guarantees on the accuracy of the obtained results. *Cost-bounded* model checking, *symbolic bisimulation minimization*, and *game-based abstraction* reduce the size of the analyzed model in various ways to make probabilistic model checking more scalable.

Furthermore, we have selected three extensions that go beyond the classical variants of probabilistic model checking: We discuss how to extract *counterexamples* using STORM, how to handle finding strategies that satisfy multiple properties simultaneously using *multi-objective* model checking, and we discuss *parametric* models in which probabilities are not fixed constants but rather unknown symbols.

Finally, we discuss tailored model checking methods for *POMDPs* and *dynamic fault trees*. We stress that the modular structure of STORM enables these approaches to easily reuse the regular model checking methods and the other methods outlined in this section.

4.1 Exact and sound model checking

Several works [18,58,121,123] observed that the numerical methods applied by probabilistic model checkers are prone to numerical errors. This has mostly two reasons. First, the floating point data types used by the tools are inherently imprecise. For example, representing the probability $\frac{1}{10}$ using IEEE 754 compliant double precision introduces an error of $5{\cdot}10^{-18}$. In the presence of numerical algorithms, these errors accumulate and may lead to incorrect results. An alternative to the above is to employ *rational arithmetic*. That is, by representing probabilities (and costs) in the model and also the results as rational numbers, models may be analyzed without introducing any numerical errors. STORM implements these ideas and allows for the exact solution of many properties. However, efficient approaches for floating point arithmetic such as value iteration become inefficient when using rational numbers, as the representation of the latter grow very large. STORM offers two tailored techniques to solve systems of (in)equations using rational arithmetic. The first is based on policy iteration and Gaussian elimination and the second on a recent technique called *rational search* [18]. The idea of the latter is to use an (imprecise) approximation of the exact solution and then sharpen this to a precise rational solution using the Kwek–Mehlhorn algorithm [90]. If a straightforward check then returns that the sharpened values constitute an actual solution, the technique can return it. Otherwise, the precision of the imprecise underlying solver is increased and the loop is restarted.

Secondly, the numerical algorithms sometimes themselves are strictly speaking unsound. For example, standard value iteration for computing reachability probabilities approximates the solution in the limit, but the termination criterion implemented by most tools does not guarantee that the obtained result is differing by at most the given precision ϵ from the actual solution. One way to combat these problems is to approach the solution from both directions, a technique referred to as *interval iteration* [15,23,58]. STORM implements the latter and additionally the more recent *sound value iteration* [110] and *optimistic value iteration* [71]. Numerical errors aside[4], these methods ensure a correct result within a user-defined accuracy and come with a small time penalty as shown in Sect. 7.

4.2 Cost-bounded reachability

A typical application for Markov models is to analyze the probability to, e.g., reach a goal state before some resource like time or energy is depleted. Another typical application is to analyze the expected time before a number of tasks have been fulfilled. Both instances can be generalized to cost-bounded reachability. In cost-bounded reachability, one is interested in the behavior of the system that does not violate the bounds on the resources. The classical approach to analyze cost-bounded reachability is to model this behavior in the model description by keeping track of the resources explicitly and then rely on standard reachability queries [5]. That is, the states of the model keep track of the consumed resources, and the reachability query asks, e.g., what the probability is that one of the target states is reached in which the resource bounds are not violated. The downside is that the model grows with these bounds.

STORM alternatively allows modeling the (nonnegative) costs of actions or states in the modeling language. These costs are attached in the model, and then, one may analyze cost-bounded reachability with the adequate query. The

[4] The implementation of these methods still uses finite precision floating point arithmetic.

Table 2 Overview of distinguishing features of STORM and their applicability based on the model types

Feature	Reference	DTMC	CTMC	MDP	MA
Sound/exact model checking	Section 4.1	✓	✓*	✓	✓*
Cost-bounded model checking	Section 4.2	✓	×	✓	×
Symbolic bisimulation minimization	Section 4.3	✓	✓	✓	✓
Game-based abstraction refinement	Section 4.4	✓	×	✓	×
Multi-objective model checking	Section 4.5	(✓)	(✓)	✓	✓
High-level counterexamples	Section 4.6	✓	×	✓	×
Parametric model checking	Section 4.7	✓	✓*	✓	✓*
Partial observations	Section 4.8	(·)	(·)	✓	×
Dynamic fault trees	Section 4.9	(·)	✓	(·)	✓
Permissive scheduler synthesis	[82]	(✓)	(×)	✓	×
Quantiles	[70]	✓	×	✓	×

✓* = except for time-bounded reachability properties
(·) = not meaningful

clear advantage of this approach is that the resources are not encoded in the state space which keeps the model much smaller. Rather, STORM does a series of model checking calls on the much smaller model [69,70], generalizing ideas from [59,89] to multiple cost dimensions. The reduced memory footprint allows to handle much larger models, and often the reduced memory consumption also yields faster verification times.

Cost-bounded reachability is closely related to quantile properties [70,89], where one fixes a desired reachability probability and asks how many resources have to be invested in order to achieve this probability.

4.3 Symbolic bisimulation minimization

A typical approach to alleviate the state-space explosion is to represent the state space *symbolically*. In the probabilistic setting, employing variants of *decision diagrams* (DDs) such as multi-terminal binary DDs (MTBDDs) or multi-valued DDs (MDDs) is the most widely used approach to deal with large state spaces [10]. They are a graph-based data structure that can exploit structure and symmetry in the underlying model to represent gigantic models very compactly.

A different angle to approach the problem is abstraction. Here, the idea is to remove details from the model that are unnecessary for the desired analysis. A well-studied technique is *bisimulation minimization*. Its core idea is that states with equivalent behavior (in some suitable sense) can be merged to obtain a quotient model that preserves the properties of the original input. Then, the (potentially much smaller) quotient can be analyzed instead. Bisimulation minimization was shown to yield substantial reductions in the case that models are represented explicitly (for instance, in terms of a probability matrix) [87].

STORM allows to combine a symbolic representation with bisimulation minimization, thereby extending previous work [119,121]. We extended the approach to deal with nondeterministic models, which makes it available on all four model types supported by STORM (see Sect. 3.1). This combination leads to significant reductions in memory and time consumption for a variety of models, and enables the analysis of models that are otherwise out of reach [76]. The resulting quotient model is often small enough to be represented explicitly which enables a wide range of efficient analysis methods.

4.4 Game-based abstraction–refinement

Even though bisimulation minimization effectively helps reducing the model, it has two major drawbacks. First, it is not guided by the concrete analysis that is to be performed. The quotient model may be much too fine for the analysis of a given property as it preserves a whole *class* of properties. Secondly, with few exceptions [42], the algorithms to compute the bisimulation quotient require the entire state space and transitions to be available. If the model is very large or even infinite, the algorithms fail to produce a quotient even if the quotient is very small.

Game-based abstraction [92] addresses these two challenges. It is based on two fundamental ideas. The first is that states are merged much more aggressively than in bisimulation minimization. That is, they may be collapsed even if they have distinguishable behavior. The behavior of the original model is over-approximated by the abstraction, and the latter can therefore be used to obtain sound bounds for the measures on the former. Note that the abstraction contains two sources of nondeterminism: the one present in the original model and the nondeterminism that is introduced by the abstraction process. Merging these sources of nondeterminism results in very loose and unsatisfactory bounds on the target values. The second idea therefore is to keep the two kinds of nondeterminism apart. This gives rise to a stochas-

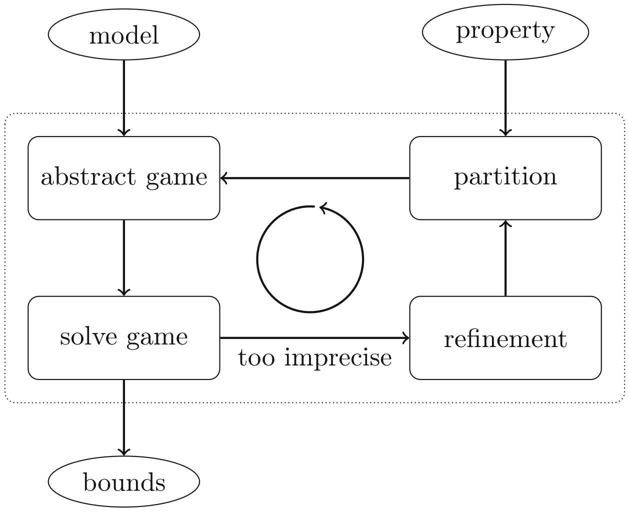


Fig. 3 Overview of abstraction–refinement using games

tic game [35] whose solution gives lower and upper bounds on both minimal and maximal probabilities in the original model.

Storm implements a game-based abstraction–refinement loop based on the ideas in [120]. The loop is illustrated in Fig. 3. As a first step, the abstract game is derived from the model and the current partitioning of the states, which is initially induced by the given property. If the bounds obtained by the analysis of the game are precise enough, they can be returned. Otherwise, the abstraction is refined by splitting the partition in a suitable way and the process is repeated.

To enable the analysis of gigantic or even infinite models, the abstraction is extracted directly from the high-level model description (given in terms of a Prism or Jani model). This extraction is achieved by the formulation as a (series of) satisfiability problem(s), which are dispatched to an off-the-shelf solver. While this has the aforementioned advantages, it is often the computationally most expensive part of the overall procedure. To combat this, Storm implements several optimizations outlined in [76, Ch. 6].

4.5 Multi-objective model checking

Initially, the focus in many probabilistic model checkers was mostly on computing the probability that a certain event happens. However, probabilistic model checking can provide meaningful data beyond the probability to reach some state, such as the optimal strategies for MDPs, i.e., functions that describe how to resolve the nondeterminism in an MDP such that the induced behavior satisfies a given property.

However, if a strategy should satisfy multiple properties, standard model checking techniques do not suffice. Consider two properties limiting time and energy usage. Standard techniques would independently compute two strategies, one optimizing time, the other optimizing energy consumption. Both strategies might be wasting the other resource, thus violating the limits described in the matching combined property. Multi-objective model checking [50,51] helps in finding strategies that satisfy multiple properties at once, and can be used to clarify the trade-offs between various properties.

Essentially, state-of-the-art multi-objective model checking boils down to a series of preprocessing steps on the model, and then either solving a linear program [51] or iteratively applying standard model checking techniques [52]. Storm supports multi-objective model checking on MDPs, and in addition on MAs [109] under general as well as more restricted strategies [43]. Furthermore, it allows for a more flexible combination of various properties, including properties with (multiple) cost bounds [69,70], and incorporates some particularly efficient preprocessing steps.

4.6 Synthesis of high-level counterexamples

Besides the computation of a single strategy, the synthesis of counterexamples and/or of sets of strategies that all satisfy or violate a given property has gained some attraction. Here, we discuss counterexamples, but similar ideas have been used for so-called permissive strategies [46] as implemented in Storm using [82].

Suppose that a system reaches a bad state with a probability above some threshold. To locate the reason for this behavior, it is helpful to obtain the part of the system that leads to this behavior, by means of a counterexample. Counterexamples try capturing the essence of the failed verification attempt and help the user of the model checker—being a human or another algorithm—to revise the system or its model accordingly. In the nonprobabilistic setting, a counterexample may be represented as *one* offending run of the system. However, such a representation is not necessarily possible in the probabilistic setting as there may be infinitely many paths that contribute to the overall probability mass reaching the bad state [63]. A single run ending in a bad state is therefore typically insufficient as a counterexample. While it is possible to consider sets of paths for probabilistic safety properties, the resulting counterexamples are large and hard to comprehend. Alternatively, counterexamples can be computed as sub-Markov models [1,31].

Rather than considering counterexamples at the state-space level, Storm computes counterexamples in terms of the high-level model specification using the ideas of [122]. More concretely, given a Jani (or Prism) model that violates a safety property, Storm computes the smallest portion of the Jani code that already witnesses the violation based on the method proposed in [39]. It does so by a guided exploration of all candidate sub-models. Ultimately, the smallest sub-model highlights the core of the problem. It does so at the abstraction level of the user. High-level counterexamples are thus a valuable as diagnostic feedback to tool users (by humans). Recent work has illustrated that these examples can be effectively used in a counterexample-guided inductive synthesis approach of finite Markov chains [30].

4.7 Parametric model checking

Naturally, the model checking result of Markov models crucially depends on the transition probabilities. Often, these probabilities are approximations based on data or reflect configurable parts of a modeled system. To represent the uncertainty about the probabilities, parametric Markov models have been first considered in [38,96]. In parametric Markov models, the probabilities are symbolic expressions rather than concrete values. For any valuation of the parameters, replacing the parameters in a parametric Markov model yields an *instantiated* parameter-free Markov model.

There are many interesting questions that one can ask revolving around parametric systems. The simplest is *feasibility*, i.e., whether there exist a valuation such that the instantiated Markov model satisfies a property. More advanced is *parameter space partitioning* where the goal is to decompose the parameter space into regions in which a predefined property is either satisfied or violated. Such a decomposition indicates for *most* parameter valuations whether they lead to a system that satisfies the given property. An alternative question is to find the *solution function*, i.e., a function in closed form that gives the model checking result of the instantiated Markov model in terms of the parameter values. Already the feasibility problem is ETR-complete, that is, it is asymptotically as hard as finding the root of a multivariate polynomial [124].

Storm supports the construction and analysis of parametric Markov models. Besides handling models and supporting efficient instantiation of parametric models, Storm provides three methods to perform parameter synthesis. The first is based on computing the aforementioned solution function through state elimination [38,61] that can also be seen as Gaussian elimination. This basic algorithm is improved by heuristics that order the operations, and a representation of the rational functions that allows for faster operations [40]. The second method, referred to as *parameter lifting*, avoids computing a potentially large rational function and determines validity of a formula over a region of parameter valuations through a sound abstraction into a nonparametric system [108]. The third method [114] aims to analyze whether the solution function is monotonic in some parameter without actually computing the solution function, as the latter can be exponential in the number of parameters. These and further methods are all used by the parameter synthesis tool PROPhESY [81] which provides a playground for parameter synthesis approaches using Storm as a back end.

4.8 Partially observable Markov decision processes

Storm supports three methods for POMDP analysis:

First, Storm supports the *verification* of (quantitative) reachability in POMDPs, e.g., to check whether for each policy resolving the nondeterminism based on the available observations, the probability to reach a bad state is less than 0.1. In general, this problem is undecidable [99]. We consider an equivalent reformulation of the POMDP as an (infinite) belief MDP: Here, each state is a distribution over POMDP states. Such a belief MDP has additional properties that have been exploited to allow verification [80,98,102]. Storm uses a combination of abstraction-and-refinement techniques to iteratively generate a finite abstract belief MDP that soundly approximates the extremal reachability probabilities in the POMDP [19].

Often, POMDPs are analyzed in settings where nondeterminism is controllable: The main interests is than in the dual of the verification problem: Find a policy such that the induced probability to reach a bad state is less than 0.1. The problem remains undecidable. A popular approach to overcome the hardness of the problem is to limit the policies, i.e., by putting a (small) a priori bound on the memory of the policy [4,24,64,101,105,125]. Such limits are especially reasonable when the nondeterminism is controllable, i.e., if a policy is to be synthesized. There are various cases in which small memory policies deliver adequate performance. Additionally, these policies are small (and arguably simple) by construction. Storm translates POMDPs under observation-based policies with a fixed amount of memory to parametric DTMCs [84]. Consider memoryless, observation-based policies: These policies map the current observation to a distribution over the available actions. We can encode all possible actions with the help of parameters. Finding values for these parameters then corresponds to finding an observation-based policy, and arguing over all parameters corresponds to arguing over all observation-based policies adhering to the memory limit.

Third, in POMDPs, even a qualitative variant of reachability is hard: In particular, to decide whether there exists a policy—resolving the nondeterminism based on the available observations—such that the probability to reach a bad state is 1 is EXPTIME-complete [33]. Storm can compute small memory policies via SAT encodings [32], and finds more general policies by an incremental procedure [83].

4.9 Model checking dynamic fault trees

Fault trees [111] are widely used in reliability engineering and model how component failures lead to failures of the complete system. Dynamic fault trees (DFTs) [47] extend (static) fault trees by dynamic gates. DFTs more faithfully model systems by allowing order-dependent failures, functional dependencies and spare management.

Dynamic fault trees may be translated into corresponding Markov models [21,47] whose analysis yields common measures on dynamic fault trees, such as reliability and mean time to failure. The analysis of the corresponding Markov models also allows more complex measures, e.g., dealing with degraded modes [54]. The essential step here is that STORM supports all these queries out of the box. Due to the modular architecture of STORM features such as parametric DFTs are supported off the shelf without dedicated implementation.

To drastically improve the analysis of DFTs, STORM contains a dedicated translation of such models into Markov models [117]. To make the state-space generation as fast as possible, STORM utilizes the structure of the DFT, and constructs a Markov model that contains only the relevant behavior of the DFT. Symmetries in the fault trees are exploited to further collapse the model with is then subject to regular model checking with STORM. As the state-space explosion might still be present during translation, STORM also supports a partial state-space generation for DFTs [117]. This partial state space yields a sound abstraction, which may be model checked to obtain safe lower and upper bounds. The state space can be iteratively extended to obtain the desired precision of the analysis result.

5 Using STORM

STORM is available as free and open software. Below, we give an overview how to use STORM. A detailed and up-to-date guide may be found on STORM's website:

http://stormchecker.org

Before you start. STORM has to be configured and compiled on the target machine. This procedure automatically looks up various dependencies and (optionally) adds them if they are not found on the system. While this configuration and compilation procedure offers some advantages, see Sect. 6.5, it is often cumbersome. Therefore, we recommend users which only want to experiment to rely on the *docker containers*[5] containing STORM with all the key dependencies, and all interfaces and extensions. One may start right away, at the cost of slightly reduced performance.
Model descriptions. STORM can be used with a variety of input languages including JANI and PRISM. A complete up-to-date list and further resources can be found at STORM's website[6]. For the sake of conciseness, we do not discuss the details of these languages here.

Below, we consider a PRISM description of the Bounded Retransmission Protocol (brp) [75]. This and many other examples can be found in the *Quantitative Verification Benchmark Set (QVBS)*[7] [72].

5.1 Command line interface

The key way to interact with STORM is through its command line interface. The command line interface allows to specify the input model and properties, and after analysis reports on the requested results. The command

```
storm --prism brp.pm --prop brp.props
```

invokes STORM with a PRISM description in `brp.pm`, and the properties listed in a file `brp.props`. STORM will build the model and perform model checking on each property. For advanced users, the methods used for model checking can be flexibly yet simply set, e.g.,

```
storm ... --engine hybrid --eqsolver elimination
```

sets the engine to hybrid (see Sect. 6.3) and sets the linear equation solver to state elimination, see Sect. 6.4. Experts may exploit the possibility to configure even details of the various procedures, e.g., the order in which state elimination is applied.

5.2 C++ extensions

To be able to flexibly use the internal data structures of STORM, one may build an own tool using STORM as a library. This approach is also taken by the STORM command line interface, as well as other extensions shipped and tightly bound to STORM, such as the analysis of DFTs outlined in Sect. 4.9. This approach is the most flexible and powerful way of using STORM, but also requires most effort. We illustrate model checking DTMCs with the sparse engine in Fig. 4. The code parses a string and a property, builds a DTMC corresponding to the model, and applies model checking on the property to compute the corresponding probability for all states. The output is then created based on the model checking result of (some) initial state. We provide a minimal working example to build your own C++ tool based on STORM as a template repository[8].

5.3 Python interface

A much quicker way to flexibly interact with (a selection of) STORM's internal data structures is the Python API called

[5] Docker containers are a lightweight alternative to virtual machines. See http://stormchecker.org/getting-started for more details.

[6] http://stormchecker.org/documentation/languages.

[7] http://qcomp.org/benchmarks.

[8] http://stormchecker.org/api/starter-project.

```
#include ...

typedef storm::models::sparse::Dtmc<double> Dtmc;
typedef storm::modelchecker::SparseDtmcPrctlModelChecker<Dtmc> DtmcModelChecker;

bool check(std::string const& path_to_model, std::string const& property_string) {
    auto program = storm::parser::PrismParser::parse(path_to_model);
    // Code snippet assumes a Dtmc
    assert(program.getModelType() == storm::prism::Program::ModelType::DTMC);
    auto properties = storm::api::parsePropertiesForPrismProgram(property_string, program);
    auto formulae = storm::api::extractFormulasFromProperties(properties);
    auto model = storm::api::buildSparseModel<double>(program, formulae)->template as<Dtmc>();
    auto checker = std::make_shared<DtmcModelChecker>(*model);

    auto result = checker->check(storm::modelchecker::CheckTask<>(*(formulae[0]), true));
    assert(result->isExplicitQuantitativeCheckResult());
    // Use that we know that the model checker produces an explicit quantitative result
    auto quantRes = result->asExplicitQuantitativeCheckResult<double>();

    return quantRes[*model->getInitialStates().begin()] > 0.5;
}
```

Fig. 4 Using the C++ interface (with STORM version 1.6.2). Please notice that we have omitted the necessary includes. An annotated version for the latest version is given in the starter project

```
import stormpy as sp

def check(path_to_model, property_str):
  program = sp.parse_prism_program(path_to_model)
  props = sp.parse_properties(property_str, program)
  model = sp.build_model(program, props)
  result = sp.model_checking(model, props[0])
  return result.at(model.initial_states[0]) > 0.5
```

Fig. 5 Using STORMPY 1.6.2

STORMPY[9]. We exemplify the ease of use in Fig. 5. The code is equivalent to Fig. 4. Using Python may induce some runtime penalty, but it enables a flexible access to the main functionality of STORM. We stress that the code is powerful enough to drive also larger projects, e.g., the parameter synthesis tool PROPHESY [40] relies on STORMPY. We provide a minimal working example to build your own Python tool based on STORMPY as a template repository[10].

6 Architecture

In this section, we report on some internal aspects of STORM. In particular, we aim to address how we realized performance and modularity. Naturally, we cannot go into the details of the various algorithms. Rather, we discuss some design choices that will help a user to feel more familiar with the code base.

6.1 Logical structure

The root directory of STORM contains—among others—sources and resources. The latter contains the logic for the configuration routines as well as various third-party dependencies. The sources are divided into various libraries and executables. The core functionality is found in the storm library. Inside that library, one finds data structures for the representation of matrices, models, expressions, modeling languages, as well as the model checking engines and solvers, which are discussed below. Besides this library, there are libraries for parsing, handling parametric models, and handling various modeling formalism such as GSPNs and DFTs. All libraries depend on the core storm library. Moreover, most libraries are accompanied by executables that provide adequate command line interfaces.

6.2 Models

STORM features two different in-memory representations of Markov models. First, it can use sparse matrices, an explicit representation form that uses memory roughly proportional to the number of transitions with nonzero probability. Sparse matrices are suited for small- and moderate-sized models and allow for fast operations also on models with irregular structure. Secondly, STORM can store models symbolically using MTBDD, cf. Sect. 4.3. The MTBDDs are built from the model description directly. While it is possible to go from MTBDDs to the explicit representation, the other direction is not (efficiently) possible. While MTBDDs often store a model compactly, typical operations for the analysis of models yield a growth in the MTBDDs and are therefore often slow. All models can be built representing the reachability probability with floating point arithmetic, exact rational numbers, or rational functions.

6.3 Model checking engines

STORM's engines are built around the two model representations. The sparse engine exclusively uses the sparse

[9] Available from http://stormchecker.org/stormpy or via the python package index at https://pypi.org/project/stormpy/.

[10] http://stormchecker.org/stormpy/starter-project.

Table 3 Overview of engines and supported features in STORM

Engine	Supported features
sparse, dd-to-sparse, automatic	All models and properties
dd	DTMC, MDP
hybrid	DTMC, CTMC, MDP, MA
exploration, abstraction–refinement	Reachability on DTMC and MDP

matrix-based representation. It first constructs the matrix representation of the state space by exploring the reachable state space specified in the modeling language and then analyzes the model using one of the many (standard, numerical) approaches, which are encapsulated as solvers (see below). While the exploration engine also uses sparse matrices, it uses ideas from reinforcement learning to avoid exploring all reachable states [23]. Instead, it proceeds in an "on-the-fly" manner and explores those parts of the system that appear to be most relevant to the verification task.

The next two engines use MTBDDs as their primary form of representation. Except for the concrete in-memory representation, the dd engine is the counterpart to the sparse engine in the sense that model building and verification is done on the very same representation and no translation takes place. STORM's hybrid engine tries to avoid the costly numerical operations on MTBDDs by transforming only parts of the system that are relevant for the considered property into a sparse matrix representation[11].

The dd-to-sparse engine is similar, but performs the translation independent of the property. This can be useful when multiple properties are to be checked on the same model or when symbolic bisimulation minimization is applied. In the latter case, the quotient model will directly be constructed in a sparse matrix representation.

The abstraction–refinement engine implements the technique described in Sect. 4.4 and is able to compute bounds for both minimal and maximal reachability probabilities for (infinite) MDPs.

Given simple features of the input PRISM or JANI model (such as the number of parallel automata or the average variable range), the automatic engine automatically selects reasonable settings for STORM. The current implementation uses a decision tree with 30 leaf nodes and a height of 7. It has been generated with the tool SCIKIT- LEARN [106] using training data from experiments on the QComp benchmark set [60]. To avoid over-fitting, the automatic choice only selects either

- the sparse engine,
- the sparse engine with exact model checking and rational arithmetic (cf. Sect. 4.1),
- the hybrid engine, or
- the dd-to-sparse engine with symbolic bisimulation minimization (cf. Sect. 4.3).

Support for queries and model descriptions. Table 3 provides an overview of the models and queries supported by each engine. The sparse engine supports all model checking queries present in STORM and all DTMCs, CTMCs, MDPs, MAs, and POMDPs described in PRISM or JANI. The engine can be paired with sound or exact model checking as in Sect. 4.1. However, exact arithmetic does not support time-bounded properties in CTMCs and MAs as these involve exponentials. Many advanced features such as cost-bounded reachability and multi-objective model checking are only implemented in the sparse engine. The dd-to-sparse engine can often make use of these implementations, as well. The support within other engines is more limited. The dd engine does not support continuous-time models (considered too slow) and POMDPs (typically sufficiently small). The exploration engine and the abstraction–refinement engine are both limited to reachability queries on discrete-time models. Moreover, some advanced features of the JANI language (indexed assignments, nontrivial system compositions) currently cannot be translated into DDs. The automatic engine falls back to the sparse engine if the input model is not supported by the predicted configuration.

6.4 Solvers

Probably the most outstanding trait of STORM's architecture is the concept of *solvers*. Ultimately, many tasks related to (probabilistic) verification revolve around solving subproblems. For example, computing reachability probabilities or expected costs in a DTMC reduces to solving a system of linear equations. Similarly, for an MDP a system of equations needs to be solved, with the difference that the equations are Bellman equations involving minima and maxima. However, these are by no means the only kinds of problems appearing in probabilistic verification.

Figure 6 illustrates some functionalities of STORM which have a dependency to one or more solvers. For example, (explicit) model building employs SMT solving. As the initial states of symbolic models (e.g., PRISM or JANI) are given by the satisfying assignments of an expression, STORM uses SMT

[11] This approach corresponds to PRISM's sparse engine and is not to be confused with the latter's hybrid engine, which is to be classified as "more symbolical."

Table 4 Solvers STORM provides out of the box

Solver type	Available solvers
Linear equations (sparse)	EIGEN, GMM++, Gaussian elimination*, native*
Linear equations (MTBDD)	CUDD, SYLVAN
Bellman equations (sparse)	EIGEN, GMM++, native*
Bellman equations (MTBDD)	CUDD, SYLVAN
Stochastic games (sparse)	native*
Stochastic games (MTBDD)	CUDD, SYLVAN
(MI)LP	GUROBI, GLPK
SMT	Z3, MATHSAT, SMT- LIB [17]

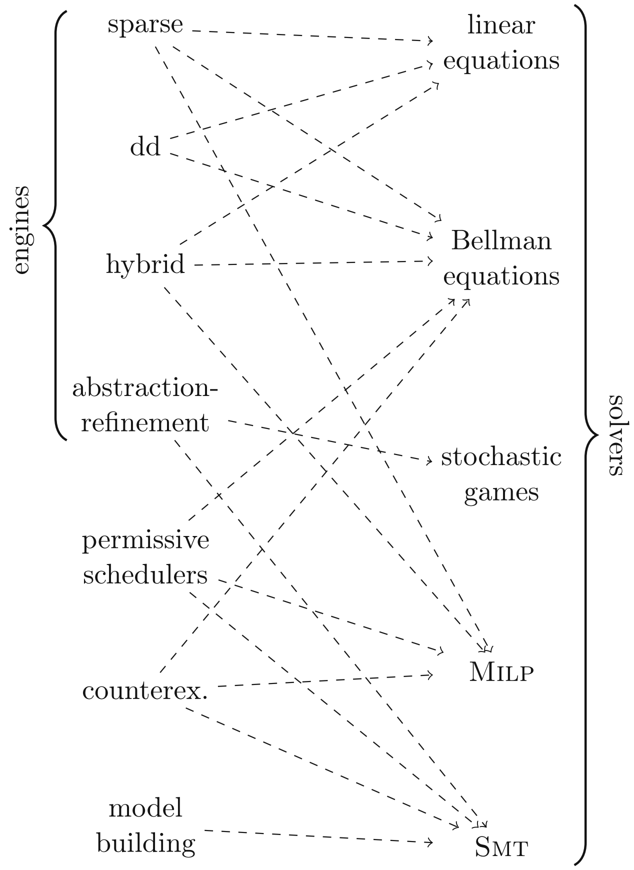


Fig. 6 Most important solvers used by STORM

solvers to enumerate the possible initial states. Similarly, the extraction of the abstract model from the symbolic model (as presented in Sect. 4.4) in the abstraction refinement engine crucially depends on enumerating satisfying assignments and therefore SMT solvers. As yet another example, consider the synthesis of high-level counterexamples as in Sect. 4.6. Here, one of the offered techniques relies on the solution of a MILP while the other uses SMT solvers.

Two of the main goals in the development of STORM were the ability to exchange central building blocks (like solvers) and to benefit from (re)using high-performance implementations provided by other libraries. It therefore offers abstract interfaces for the solver types mentioned above that are oblivious to the underlying implementation. Offering these interfaces has several key advantages. First, it provides easy and coherent access to the tasks commonly involved in probabilistic model checking. Secondly, it enables the use of dedicated state-of-the-art high-performance libraries for the task at hand. More specifically, as the performance characteristics of different backend solvers can vary drastically for the same input, this permits choosing the best solver for a given task. Licensing problems are avoided, because implementations can be easily enabled and disabled, depending on whether or not the particular license fits the requirements. Finally, implementing new solver functionality is easy and can be done without detailed knowledge of the global code base. This flexibility allows to keep STORM up to date with new state-of-the-art solvers.

For each of the solver interfaces, several actual implementations exist. For example, STORM currently has four implementations (each of them with a range of further options) of the linear equation solver interface for problems given as sparse matrices: One is based on GMM++, one is based on EIGEN [56], one uses its native internal data structures and algorithms for numerical algorithms and another one is based on Gaussian elimination [38]. Table 4 gives an overview over the currently available implementations. Here, all solvers that are purely implemented in terms of STORM's data structures and do not use libraries are marked with an asterisk to indicate that they are "built in."

To realize the support for DD-based representations of systems, STORM relies on two different libraries: CUDD [113] and SYLVAN [118]. While the former is very well established in the field, the latter is more recent and tries to make use of modern multi-core CPU architectures by parallelizing costly operations. The parallelization comes at the price of more expensive bookkeeping and in general CUDD performs better if there are many operations on smaller DDs, while SYLVAN is faster when fewer operations on larger DDs are involved. STORM implements an abstraction layer on top of the two libraries that uses static polymorphism. This way, it is possible to write code that is independent of the underlying library and does not incur runtime costs.

6.5 Technicalities

By far the largest part (over 170,000 lines of code) of STORM is written in the C++ programming language and extensively uses template meta-programming. This has several positive and negative implications. On the one hand, it serves the purpose of high performance for several reasons. First, C++ allows fine-grained control over implementation details like memory allocations. Secondly, C++ templates allow code to be heavily reused while maintaining performance as the static polymorphism enables type-dependent optimizations at compile time. Large parts of the code are written agnostic of the data type (floating point, rational number, or even rational functions) and only the core parts are specialized based on the data type. As this happens at compile time, no runtime cost is incurred. Finally, we observe that many high-performance solvers and data structure libraries that are well suited for the context of (probabilistic) verification are written in C or C++ (and also partially make use of template meta-programming), such as

- SMT solvers (Z3 [45], MATHSAT [34], SMT- RAT [36]),
- LP solvers (GUROBI [57], GLPK[12]),
- linear algebra libraries (GMM++[13], EIGEN [56]),
- DD libraries (CUDD [113], SYLVAN [118]), and
- rational arithmetic libraries (CARL [36], GMP[14]).

Choosing C++ as the language for STORM therefore allows easy and fast interfacing with these solvers. On the other hand, the advantages come at a price. Advanced templating patterns can be difficult to understand and increase compile times significantly.

7 Evaluation

This section contains an empirical evaluation of some key functionalities of STORM. Furthermore, we recap results of QComp 2019 [60] and QComp 2020 [26] to emphasize the competitiveness of STORM.

7.1 Setup and methodology

We consider the set of 100 benchmark instances that were selected in QComp 2019 and 2020 [26,60]. Each instance consists of a symbolic model description and a property specification from the *Quantitative Verification Benchmark Set (QVBS)* [72]. If available, we consider model descriptions in the PRISM language. Otherwise, the model is build from the JANI description. For a better comparison across STORM's engines, we did *not* employ the techniques from Sect. 4.9 to solve DFTs. Since STORM has no native support for PTA, we used the tool MOCONV (part of the MODEST TOOLSET[15] [67]) to translate PTAs into MDPs. For four instances either MOCONV did not support the PTA or STORM did not support the output of MOCONV. We therefore restrict our evaluation to the remaining 96 benchmark instances.

For each instance, the task is to solve the corresponding model checking query within a time limit of 30 minutes and a memory limit of 12 GB. The results are compared to the reference results provided by the QVBS. If the relative difference between these values is greater than 10^{-3}, the result is considered *incorrect*. This setup coincides with the setup of QComp 2019. All experiments were run on 4 cores of an Intel® Xeon® Platinum 8160 Processor. We measure the wall-clock runtimes (including model building and model checking) for all experiments. Notice that this machine is more powerful than the QComp 2019 machine.

For our evaluation, we consider STORM version 1.6.2 in *seven different configurations* comprising

- the main engines of STORM: sparse, hybrid, and dd,
- symbolic bisimulation (bisim) with sparse quotient (Sect. 4.3),
- sound and exact model checking within the sparse engine (Sect. 4.1), and
- the automatic engine.

Whenever the invoked model checking method is sound (i.e., provides precision guarantees), the precision of STORM is set to 10^{-3} (relative). Otherwise, STORM's default precision 10^{-6} (relative) is used. We select SYLVAN [118] as DD-library, and set its memory limit to 4 GB. We also consider a "fastest" configuration that takes the best result from the seven configurations, i.e., a configuration which runs all seven configurations and terminates whenever the fastest terminates (and further runs the seven configurations independently on different machines).

All benchmark files, log files, and replication scripts are available at [77].

7.2 Results

Table 5 summarizes the outcomes of our experiments. The seven columns refer to the seven configurations as described

[12] https://www.gnu.org/software/glpk/.

[13] http://getfem.org/gmm.html.

[14] https://gmplib.org/.

[15] http://www.modestchecker.net/.

Table 5 Outcomes of experiments on 96 benchmark instances

	sparse	hybrid	dd	bisim	sound	exact	automatic
#solved	73	67	40	59	73	43	84
#not supp.	0	11	42	7	0	14	0
#time-outs	3	5	4	8	3	12	3
#mem-outs	16	11	8	20	18	27	7
#incorrect	4	2	2	2	2	0	2
#fastest$_{+1\%}$	19	21	9	14	8	3	40
#fastest$_{+50\%}$	39	46	16	26	27	4	78

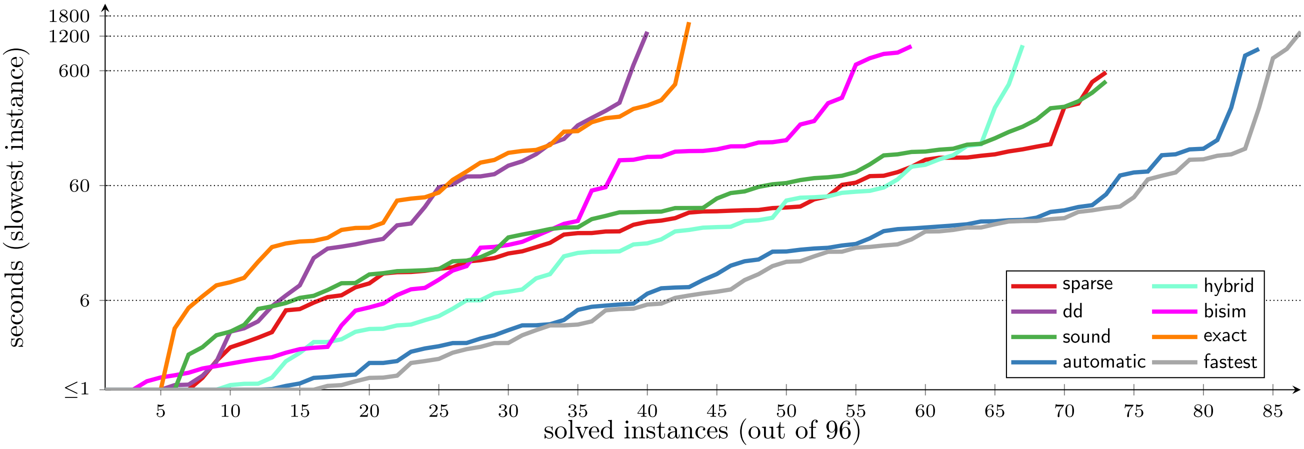


Fig. 7 Runtime comparison of STORM's key features

above. In the first row, we indicate how many of the 96 considered instances were correctly solved for each configuration. The subsequent rows indicate the number of not supported instances[16], the number of times the time or memory limit was exceeded, respectively, and the number of incorrect results[17] that were obtained. Observe that these rows always sum to 96.

For the "fastest" configuration, we obtain 87 solved instances and 0 incorrect results. The next rows (after the horizontal line) show how often each configuration was either the fastest among the tested ones or only 1% (50%) slower than the fastest one, i.e., terminated within 101% (150%) of the fastest configuration.

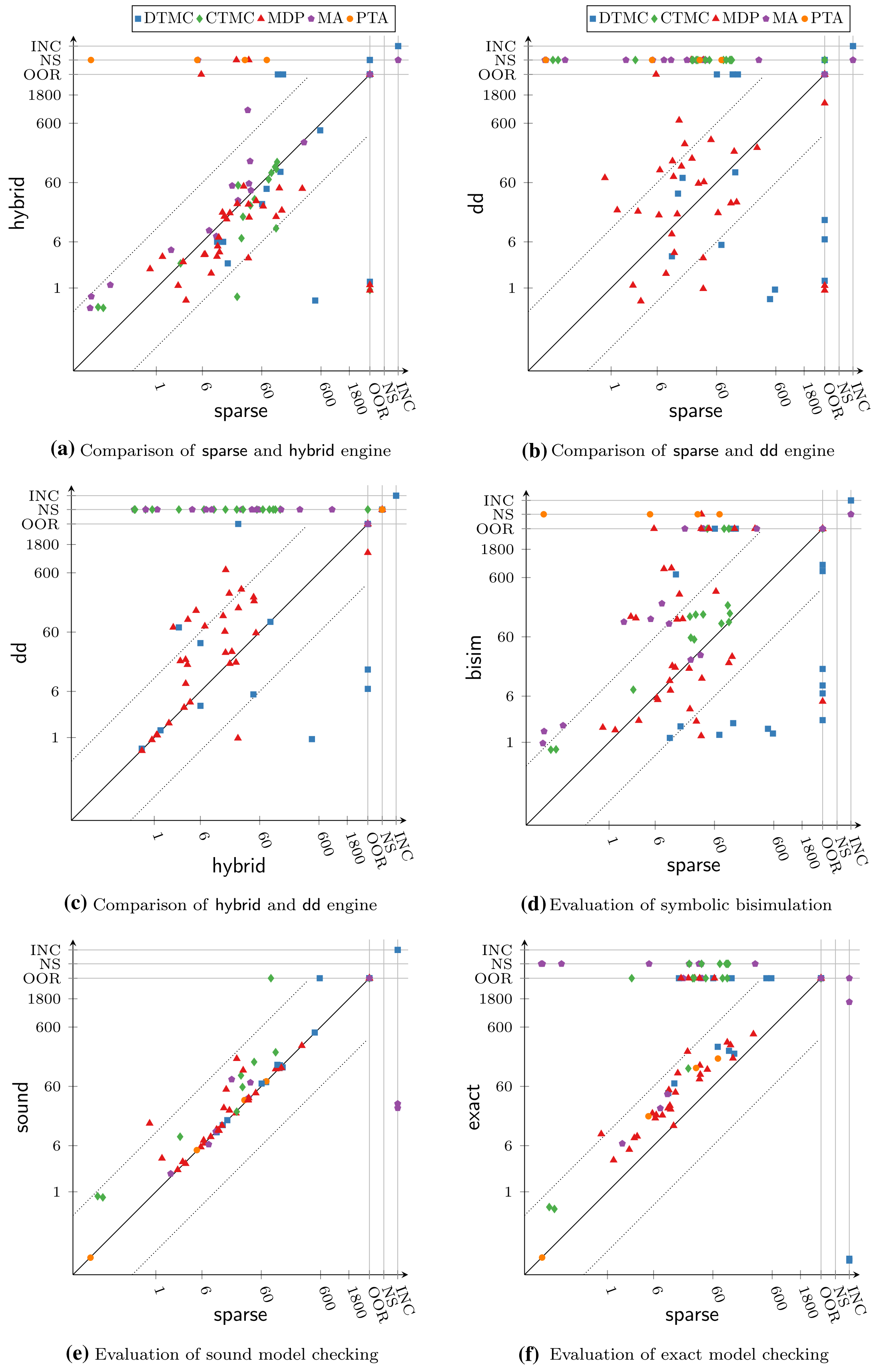
We further compare the runtimes of the different engines and features in Fig. 7. The shown quantile plot expresses how many benchmark instances (measured on the x-axis) *each* were solved in at most the time given on the y-axis. In other words, the point $\langle x, y\rangle$ is contained in the quantile plot for configuration c if the *maximal* runtime when using c on the x fastest solved instances (for c) is y seconds. Time and memory outs, incorrect results, and unsupported experiments may skew the lines of the affected configurations as all these outcomes do *not* count as solved. Besides the seven considered configurations, we also depict the runtime obtained by the fastest engine or feature for each individual benchmark.

Finally, we compare the configurations of STORM one by one and give the results in Figs. 8 and 9. Each point in the depicted scatter plots indicates the runtimes of the two compared configurations for one benchmark instance. The type (DTMC, CTMC, MDP, MA, or PTA) of the verification task is indicated by means of different marks. The scatter plots use logarithmic scales on both axes and indicate speed-ups of 10 by means of dotted lines. If an experiment ran out of resources (time or memory), was not supported, or yielded an incorrect result, we draw the point on separate lines, labeled OOR, NS, and INC, respectively. We compare the engines (sparse, hybrid, and dd) with each other in Fig. 8a–c. Symbolic bisimulation, sound, and exact model checking are compared with the sparse engine (the default of STORM) in Fig. 8d–f. For the comparison with sound model checking, we do not depict benchmark instances where the default method is already sound. Figure 9a, b compares the automatic engine with the sparse engine and with the fastest configuration, respectively.

More detailed results of our experiments can be found on http://stormchecker.org/benchmarks.

[16] Observe that sparse, sound, and automatic support all queries. For details on the other configurations see Sect. 6.3.

[17] The incorrect results are the consequence of imprecise floating points or algorithms that do not guarantee sound results, see Sect. 4.1.



(a) Comparison of sparse and hybrid engine

(b) Comparison of sparse and dd engine

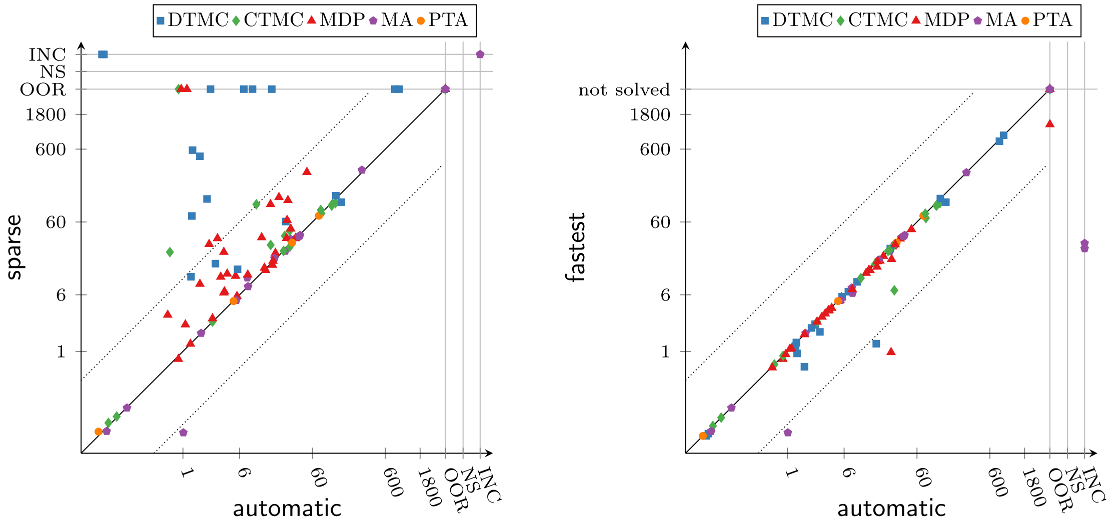
(c) Comparison of hybrid and dd engine

(d) Evaluation of symbolic bisimulation

(e) Evaluation of sound model checking

(f) Evaluation of exact model checking

Fig. 8 Comparison of engines and features of STORM



(a) Comparison of automatic and sparse engine **(b)** Comparison of automatic engine and fastest configuration

Fig. 9 Comparison of engines and features of STORM (continued)

7.3 Discussion

Comparing the three main engines of STORM (sparse, hybrid, and dd), the sparse engine was the most versatile engine during our experiments since it supports all 96 instances and successfully solved the majority (73) of them, outperforming the other two engines. However, looking at Fig. 7 we see that the other engines are competitive. The automatic engine often manages to pick the "right" configuration for a given benchmark and thus almost matches the performance of the (notional) fastest configuration. As indicated in Fig. 8, several instances could only be solved using symbolic techniques based on the hybrid or the dd engine. We emphasize that the benchmark selection can have a strong impact when comparing the engines of STORM because the symbolic engines are strongly reliant on the model structure. Moreover, many benchmarks are not supported by the hybrid and/or the dd engine which skews the lines in Fig. 7.

Symbolic bisimulation was extremely effective on models with a concise DD-based representation and a small bisimulation quotient. The export into a sparse quotient allows STORM to make use of the versatility of the sparse engine.

In Fig. 8e, we see that the overhead for sound model checking is often negligible. As mentioned above, we invoke classical model checking (such as value iteration) with the default precision parameters (10^{-6} relative precision), whereas sound model checking is invoked with the actual precision requirements (10^{-3} relative precision), yielding speed-ups for some instances.

Exact model checking is comparably costly. The use of exact (infinite precision) arithmetic induces increasingly larger number representations. Moreover, approximative, numerical solution methods cannot be applied. However, on a few instances where numerical methods do not work well, exact model checking was superior to the remaining configurations.

Figure 9a, b shows that—for this benchmark set—the automatic engine improved the runtime of the sparse engine in many cases and that there were only a few instances where it was outperformed by the (notional) fastest engine.

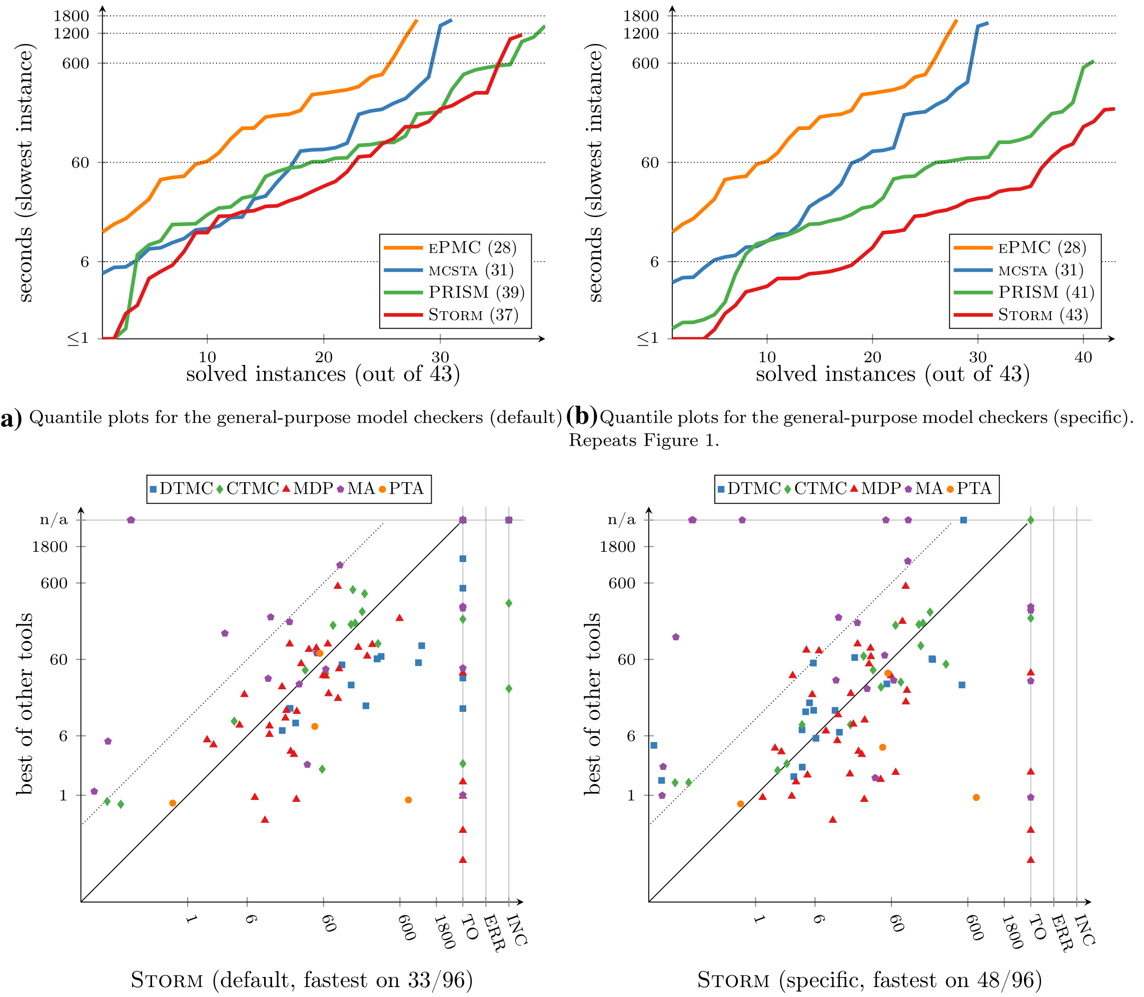
7.4 Summary from QComp 2019

We briefly recap the results of QComp 2019, focusing on the performance evaluation. For further details, we refer to the competition report [60].

The experimental setup of QComp 2019 (benchmark selection, precision requirements, time and memory limits, etc.) coincides with our setup as detailed above, except that

- a different machine was used, and
- STORM was considered in version 1.3.0.

Each tool was executed in two different modes: once with default settings (which for STORM coincides with using the sparse engine) and once with benchmark specific settings. For the latter mode, the participants could provide a tailored tool invocation for each individual benchmark instance. For STORM, this was realized by empirically determining the fastest configuration for a given instance, where we considered the configurations sparse, hybrid, dd, bisim, sound, and exact (as above).



(a) Quantile plots for the general-purpose model checkers (default) **(b)** Quantile plots for the general-purpose model checkers (specific). Repeats Figure 1.

(c) Runtime of Storm (default) compared with the best results. **(d)** Runtime of Storm (best config) compared with the best results.

Fig. 10 Performance of STORM compared with other state-of-the-art model checkers. All figures are taken from [60] licensed under Creative Commons Attribution 4.0 International License: http://creativecommons.org/licenses/by/4.0/

Figure 10 depicts the performance results of QComp 2019 that are relevant for STORM. The quantile plots in Fig. 10a, b compare STORM with the other participating general-purpose probabilistic model checkers EPMC [62], MCSTA [68], and PRISM [93] using the default and specific modes, respectively. STORM supported 96 of the 100 considered benchmark instances, whereas EPMC, PRISM, and MCSTA supported 63, 58, and 86 instances, respectively. For the quantile plots, only the 43 instances that were supported by all 4 tools were taken into account. In particular, all benchmarks are given in PRISM language since PRISM does not support JANI. The scatter plots in Fig. 10c, d compare STORM with the best of the other 8 participating tools. A point above the solid diagonal line indicates that on the corresponding instance, STORM was the fastest tool among all participants.

Considering the results for the default mode in Fig. 10a, STORM is the strongest competitor of the other three tools. However, the performance results of STORM and PRISM are very close to each other. For instance-specific invocations (Fig. 10b), STORM clearly outperformed all its competitors. The scatter plots show that STORM performed best among all tools for 1/3 of the supported benchmarks in default mode and 1/2 of the supported benchmarks in specific mode.

7.5 Outlook to QComp 2020

Since QComp 2019 further progress of the participating tools has been made. For example, new and efficient model checking techniques for MDPs and MAs have been implemented in MCSTA [27,71]. QComp 2020 [26] captures some of these changes and gives a special emphasis to the correctness of the results produced by the tools. In contrast to the 2019 edition, the performance evaluation is divided in six tracks. The tracks consider the same benchmark set but impose different correctness requirements ranging from *exact* results to *often ε-correct* results. Among all nine participants, STORM has been the only tool that implements supporting algorithms for *all* tracks and has proven competitiveness in each of them. More details to QComp 2020 can be found in its competition report [26].

We remark that both QComp 2019 and QComp 2020 necessarily only provide a snapshot of the tool landscape at the time of the evaluation. A repetition of the evaluation of QComp with newer tool versions can yield different results.

8 Conclusion

This paper presented the state-of-the-art probabilistic model checker STORM. We have discussed its main distinguishing features and described how it can be used for rapid prototyping of new algorithms and tools. Key aspects of STORM are its modularity, its accessibility through a Python interface, its various modeling formalisms, as well as the functionalities that go beyond the standard probabilistic model checking algorithms. We believe that its modularity, careful crafting of the most time-consuming operations, and our experience with earlier in-house developed model checkers, have led to a tool that is competitive to existing probabilistic model checkers. STORM provides an effective and efficient platform for future-proof developments in probabilistic model checking. It is open access and publicly available from http://stormchecker.org. A major challenges will be to keep up with the rapid progress in the field. This does not only involve the implementation of new algorithms, but also involve constantly revising existing code fragments.

Acknowledgements We would like to thank all additional contributors to STORM, in particular Philipp Berger, David Korzeniewski, Jip Spel, as well as Dimitri Bohlender, Alexander Bork, Harold Bruintjes, Michael Deutschen, Timo Gross, Thomas Heinemann, Thomas Henn, Tom Janson, Jan Karuc, Joachim Klein, Gereon Kremer, Sascha V. Kurowski, Manuel S. Weiand, and Lukas Westhofen. We would like to thank Tom van Dijk for his support regarding SYLVAN.
Furthermore, we would like to thank the users for being interested, and for the many kind bug-reports that helped us improving the tool.
This work has been supported by the ERC Advanced Grant 787914 (FRAPPANT) and the DFG RTG 2236 "UnRAVeL." S. Junges would like to acknowledge funding from NSF grants 1545126 (VeHICaL) and 1646208, the DARPA Assured Autonomy program, Berkeley Deep Drive, and by Toyota under the iCyPhy center.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Ábrahám, E., Becker, B., Dehnert, C., Jansen, N., Katoen, J.P., Wimmer, R.: Counterexample generation for discrete-time Markov models: An introductory survey. In: SFM, LNCS, vol. 8483, pp. 65–121. Springer (2014)
2. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018)
3. Alur, R., Henzinger, T.A., Vardi, M.Y.: Theory in practice for system design and verification. SIGLOG News **2**(1), 46–51 (2015)
4. Amato, C., Bernstein, D.S., Zilberstein, S.: Optimizing fixed-size stochastic controllers for POMDPs and decentralized POMDPs. Auton. Agent. Multi-Agent Syst. **21**(3), 293–320 (2010)
5. Andova, S., Hermanns, H., Katoen, J.P.: Discrete-time rewards model-checked. In: FORMATS, LNCS, vol. 2791, pp. 88–104. Springer (2003)
6. Ashok, P., Chatterjee, K., Daca, P., Kretínský, J., Meggendorfer, T.: Value iteration for long-run average reward in Markov decision processes. In: CAV (1), LNCS, vol. 10426, pp. 201–221. Springer (2017)
7. Åström, K.: Optimal control of Markov processes with incomplete state information. J. Math. Anal. Appl. **10**(1), 174–205 (1965)
8. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Model-checking continous-time Markov chains. ACM Trans. Comput. Log. **1**(1), 162–170 (2000)
9. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Handbook of Model Checking, pp. 963–999. Springer (2018)
10. Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M.Z., Ryan, M.: Symbolic model checking for probabilistic processes. In: ICALP, LNCS, vol. 1256, pp. 430–440. Springer (1997)
11. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Softw. Eng. **29**(6), 524–541 (2003)
12. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
13. Baier, C., Klein, J., Klüppelholz, S., Märcker, S.: Computing conditional probabilities in Markovian models efficiently. In: TACAS, LNCS, vol. 8413, pp. 515–530. Springer (2014)
14. Baier, C., Klein, J., Klüppelholz, S., Wunderlich, S.: Maximizing the conditional expected reward for reaching the goal. In: TACAS (2), LNCS, vol. 10206, pp. 269–285 (2017)
15. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: interval iteration for Markov decision processes. In: CAV (1), LNCS, vol. 10426, pp. 160–180. Springer (2017)
16. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011)
17. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.5. Tech. rep., Dep. of Computer Science, The University of Iowa (2015). www.smt-lib.org
18. Bauer, M.S., Mathur, U., Chadha, R., Sistla, A.P., Viswanathan, M.: Exact quantitative probabilistic model checking through rational search. In: FMCAD, pp. 92–99. IEEE (2017)
19. Bork, A., Junges, S., Katoen, J., Quatmann, T.: Verification of indefinite-horizon POMDPs. CoRR abs/2007.00102 (2020)

20. Boudali, H., Crouzen, P., Stoelinga, M.: A compositional semantics for dynamic fault trees in terms of interactive Markov chains. In: ATVA, LNCS, vol. 4762, pp. 441–456. Springer (2007)
21. Boudali, H., Crouzen, P., Stoelinga, M.: Dynamic fault tree analysis using input/output interactive Markov chains. In: DSN, pp. 708–717. IEEE Computer Society (2007)
22. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability and performance analysis of extended AADL models. Comput. J. **54**(5), 754–775 (2011)
23. Brázdil, T., Chatterjee, K., Chmelik, M., Forejt, V., Kretínský, J., Kwiatkowska, M.Z., Parker, D., Ujma, M.: Verification of Markov decision processes using learning algorithms. In: ATVA, LNCS, vol. 8837, pp. 98–114. Springer (2014)
24. Braziunas, D., Boutilier, C.: Stochastic local search for POMDP controllers. In: AAAI, pp. 690–696. The MIT Press (2004)
25. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. In: TACAS (2), LNCS, vol. 10206, pp. 151–168 (2017)
26. Budde, C.E., Hartmanns, A., Klauck, M., Kretínský, J., Parker, D., Quatmann, T., Turini, A., Zhang, Z.: On correctness, precision, and performance in quantitative verification (QComp 2020 competition report). In: ISoLA, LNCS. Springer (2020). (To Appear)
27. Butkova, Y., Hartmanns, A., Hermanns, H.: A Modest approach to modelling and checking Markov automata. In: QEST, LNCS, vol. 11785, pp. 52–69. Springer (2019)
28. Butkova, Y., Wimmer, R., Hermanns, H.: Long-run rewards for Markov automata. In: TACAS (2), LNCS, vol. 10206, pp. 188–203 (2017)
29. Calder, M., Vyshemirsky, V., Gilbert, D.R., Orton, R.J.: Analysis of signalling pathways using continuous time Markov chains. Trans. Comput. Syst. Biol. VI LNCS **4220**, 44–67 (2006)
30. Ceska, M., Hensel, C., Junges, S., Katoen, J.P.: Counterexample-driven synthesis for probabilistic program sketches. In: FM, LNCS, vol. 11800, pp. 101–120. Springer (2019)
31. Chadha, R., Viswanathan, M.: A counterexample-guided abstraction-refinement framework for Markov decision processes. ACM Trans. Comput. Log. **12**(1), 1:1–1:49 (2010)
32. Chatterjee, K., Chmelik, M., Davies, J.: A symbolic SAT-based algorithm for almost-sure reachability with small strategies in POMDPs. In: AAAI, pp. 3225–3232. AAAI Press (2016)
33. Chatterjee, K., Doyen, L., Henzinger, T.A.: Qualitative analysis of partially-observable Markov decision processes. In: MFCS, LNCS, vol. 6281, pp. 258–269. Springer (2010)
34. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: TACAS, LNCS, vol. 7795, pp. 93–107. Springer (2013)
35. Condon, A.: On algorithms for simple stochastic games. In: Advances in Computational Complexity Theory. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 13, pp. 51–71. DIMACS/AMS (1990)
36. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: SAT, LNCS, vol. 9340, pp. 360–368. Springer (2015)
37. Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite-state probabilistic programs. In: FOCS, pp. 338–345. IEEE Computer Society (1988)
38. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: ICTAC, LNCS, vol. 3407, pp. 280–294. Springer (2004)
39. Dehnert, C., Jansen, N., Wimmer, R., Ábrahám, E., Katoen, J.P.: Fast debugging of PRISM models. In: ATVA, LNCS, vol. 8837, pp. 146–162. Springer (2014)
40. Dehnert, C., Junges, S., Jansen, N., Corzilius, F., Volk, M., Bruintjes, H., Katoen, J.P., Ábrahám, E.: Prophesy: a probabilistic parameter synthesis tool. In: CAV (1), LNCS, vol. 9206, pp. 214–231. Springer (2015)
41. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A storm is coming: a modern probabilistic model checker. In: CAV (2), LNCS, vol. 10427, pp. 592–600. Springer (2017)
42. Dehnert, C., Katoen, J.P., Parker, D.: SMT-based bisimulation minimisation of Markov models. In: VMCAI, LNCS, vol. 7737, pp. 28–47. Springer (2013)
43. Delgrange, F., Katoen, J., Quatmann, T., Randour, M.: Simple strategies in multi-objective MDPs. In: TACAS (1), LNCS, vol. 12078, pp. 346–364. Springer (2020)
44. de Alfaro, L.: How to specify and verify the long-run average behavior of probabilistic systems. In: LICS, pp. 454–465. IEEE Computer Society (1998)
45. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS, LNCS, vol. 4963, pp. 337–340. Springer (2008)
46. Dräger, K., Forejt, V., Kwiatkowska, M.Z., Parker, D., Ujma, M.: Permissive controller synthesis for probabilistic systems. Logical Methods Comput. Sci. **11**, 2 (2015)
47. Dugan, J.B., Bavuso, S.J., Boyd, M.: Fault trees and sequence dependencies. In: Proceedings of RAMS, pp. 286–293. IEEE (1990). 10.1109/ARMS.1990.67971
48. Eisentraut, C., Hermanns, H., Katoen, J.P., Zhang, L.: A semantics for every GSPN. In: Petri Nets, LNCS, vol. 7927, pp. 90–109. Springer (2013)
49. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: LICS, pp. 342–351. IEEE Computer Society (2010)
50. Etessami, K., Kwiatkowska, M.Z., Vardi, M.Y., Yannakakis, M.: Multi-objective model checking of Markov decision processes. Logical Methods Comput. Sci. **4**, 4 (2008)
51. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: TACAS, LNCS, vol. 6605, pp. 112–127. Springer (2011)
52. Forejt, V., Kwiatkowska, M.Z., Parker, D.: Pareto curves for probabilistic model checking. In: ATVA, LNCS, vol. 7561, pp. 317–332. Springer (2012)
53. Fredlund, L.: The timing and probability workbench: a tool for analysing timed processes. Tech. Rep. 49, Uppsala University (1994)
54. Ghadhab, M., Junges, S., Katoen, J.P., Kuntz, M., Volk, M.: Safety analysis for vehicle guidance systems with dynamic fault trees. Rel. Eng. Syst. Saf. **186**, 37–50 (2019)
55. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE, pp. 167–181. ACM (2014)
56. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010)
57. Gurobi Optimization, L.: Gurobi optimizer reference manual (2019). http://www.gurobi.com
58. Haddad, S., Monmege, B.: Reachability in MDPs: refining convergence of value iteration. In: RP, LNCS, vol. 8762, pp. 125–137. Springer (2014)
59. Hahn, E.M., Hartmanns, A.: A comparison of time- and reward-bounded probabilistic model checking techniques. SETTA LNCS **9984**, 85–100 (2016)
60. Hahn, E.M., Hartmanns, A., Hensel, C., Klauck, M., Klein, J., Kretínský, J., Parker, D., Quatmann, T., Ruijters, E., Steinmetz, M.: The 2019 comparison of tools for the analysis of quantitative formal models- (QComp 2019 competition report). In: TACAS (3), LNCS, vol. 11429, pp. 69–92. Springer (2019)
61. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. STTT **13**(1), 3–19 (2011)
62. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasMc: A web-based probabilistic model checker. In: FM, LNCS, vol. 8442, pp. 312–317. Springer (2014)

63. Han, T., Katoen, J.P., Damman, B.: Counterexample generation in probabilistic model checking. IEEE Trans. Softw. Eng. **35**(2), 241–257 (2009)
64. Hansen, E.A.: Solving POMDPs by searching in policy space. In: UAI, pp. 211–219. Morgan Kaufmann (1998)
65. Hansson, H., Jonsson, B.: A framework for reasoning about time and reliability. In: RTSS, pp. 102–111. IEEE Computer Society (1989)
66. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Asp. Comput. **6**(5), 512–535 (1994)
67. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: TACAS, LNCS, vol. 8413, pp. 593–598. Springer (2014)
68. Hartmanns, A., Hermanns, H.: Explicit model checking of very large MDP using partitioning and secondary storage. In: ATVA, LNCS, vol. 9364, pp. 131–147. Springer (2015)
69. Hartmanns, A., Junges, S., Katoen, J.P., Quatmann, T.: Multi-cost bounded reachability in MDP. In: TACAS (2), LNCS, vol. 10806, pp. 320–339. Springer (2018)
70. Hartmanns, A., Junges, S., Katoen, J.P., Quatmann, T.: Multi-cost bounded tradeoff analysis in MDP. JAR (2020)
71. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: CAV (2), LNCS, vol. 12225, pp. 488–511. Springer (2020)
72. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS (1), LNCS, vol. 11427, pp. 344–350. Springer (2019)
73. Hartonas-Garmhausen, V., Campos, S.V.A., Clarke, E.M.: ProbVerus: probabilistic symbolic model checking. In: ARTS, LNCS, vol. 1601, pp. 96–110. Springer (1999)
74. He, J., Seidel, K., McIver, A.: Probabilistic models for the guarded command language. Sci. Comput. Program. **28**(2–3), 171–192 (1997)
75. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof-checking a data link protocol. In: TYPES, LNCS, vol. 806, pp. 127–165. Springer (1993)
76. Hensel, C.: The probabilistic model checker Storm: symbolic methods for probabilistic model checking. Ph.D. thesis, RWTH Aachen University, Germany (2018)
77. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker storm: evaluation results and replication package (2020). https://doi.org/10.5281/zenodo.3571209
78. Hermanns, H., Katoen, J.P., Meyer-Kayser, J., Siegle, M.: A Markov chain model checker. In: TACAS, LNCS, vol. 1785, pp. 347–362. Springer (2000)
79. Holzmann, G.J.: Mars code. Commun. ACM **57**(2), 64–73 (2014)
80. Horák, K., Bosanský, B., Chatterjee, K.: Goal-HSVI: heuristic search value iteration for goal POMDPs. In: IJCAI, pp. 4764–4770. ijcai.org (2018)
81. Junges, S., Ábrahám, E., Hensel, C., Jansen, N., Katoen, J.P., Quatmann, T., Volk, M.: Parameter synthesis for Markov models. CoRR abs/1903.07993 (2019)
82. Junges, S., Jansen, N., Dehnert, C., Topcu, U., Katoen, J.P.: Safety-constrained reinforcement learning for mdps. In: TACAS, LNCS, vol. 9636, pp. 130–146. Springer (2016)
83. Junges, S., Jansen, N., Seshia, S.A.: Enforcing almost-sure reachability in pomdps. CoRR abs/2007.00085 (2020)
84. Junges, S., Jansen, N., Wimmer, R., Quatmann, T., Winterer, L., Katoen, J.P., Becker, B.: Finite-state controllers of POMDPs using parameter synthesis. In: UAI, pp. 519–529. AUAI Press (2018)
85. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artif. Intell. **101**(1–2), 99–134 (1998)
86. Katoen, J.P.: The probabilistic model checking landscape. In: LICS, pp. 31–45. ACM (2016)
87. Katoen, J.P., Kemna, T., Zapreev, I.S., Jansen, D.N.: Bisimulation minimisation mostly speeds up probabilistic model checking. In: TACAS, LNCS, vol. 4424, pp. 87–101. Springer (2007)
88. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Perform. Eval. **68**(2), 90–104 (2011)
89. Klein, J., Baier, C., Chrszon, P., Daum, M., Dubslaff, C., Klüppelholz, S., Märcker, S., Müller, D.: Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic büchi automata. STTT **20**(2), 179–194 (2018)
90. Kwek, S., Mehlhorn, K.: Optimal search for rationals. Inf. Process. Lett. **86**(1), 23–26 (2003)
91. Kwiatkowska, M.Z., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: a hybrid approach. In: TACAS, LNCS, vol. 2280, pp. 52–66. Springer (2002)
92. Kwiatkowska, M.Z., Norman, G., Parker, D.: Game-based abstraction for Markov decision processes. In: QEST, pp. 157–166. IEEE Computer Society (2006)
93. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV, LNCS, vol. 6806, pp. 585–591. Springer (2011)
94. Kwiatkowska, M.Z., Norman, G., Parker, D.: Probabilistic verification of Herman's self-stabilisation algorithm. Formal Asp. Comput. **24**(4–6), 661–670 (2012)
95. Kwiatkowska, M.Z., Norman, G., Segala, R.: Automated verification of a randomized distributed consensus protocol using cadence SMV and PRISM. In: CAV, LNCS, vol. 2102, pp. 194–206. Springer (2001)
96. Lanotte, R., Maggiolo-Schettini, A., Troina, A.: Parametric probabilistic transition systems for system design and analysis. Formal Asp. Comput. **19**(1), 93–109 (2007)
97. Larsen, K.G., Legay, A.: Statistical model checking: past, present, and future. In: ISoLA (1), LNCS, vol. 9952, pp. 3–15 (2016)
98. Lovejoy, W.S.: Computationally feasible bounds for partially observed Markov decision processes. Oper. Res. **39**(1), 162–175 (1991)
99. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and related stochastic optimization problems. Artif. Intell. **147**(1–2), 5–34 (2003)
100. Marsan, M.A., Conte, G., Balbo, G.: A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. ACM Trans. Comput. Syst. **2**(2), 93–122 (1984)
101. Meuleau, N., Kim, K., Kaelbling, L.P., Cassandra, A.R.: Solving POMDPs by searching the space of finite policies. In: UAI, pp. 417–426. Morgan Kaufmann (1999)
102. Norman, G., Parker, D., Zou, X.: Verification and control of partially observable probabilistic systems. Real-Time Syst. **53**(3), 354–402 (2017)
103. Norris, J.R.: Markov Chains. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge (1998)
104. Olmedo, F., Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J.P., McIver, A.: Conditioning in probabilistic programming. ACM Trans. Program. Lang. Syst. **40**(1), 4:1–4:50 (2018)
105. Pajarinen, J., Peltonen, J.: Periodic finite state controllers for efficient POMDP and DEC-POMDP planning. In: NIPS, pp. 2636–2644 (2011)
106. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
107. Puterman, M.L.: Markov Decision Processes. Wiley, New York (1994)

108. Quatmann, T., Dehnert, C., Jansen, N., Junges, S., Katoen, J.P.: Parameter synthesis for Markov models: faster than ever. ATVA LNCS **9938**, 50–67 (2016)
109. Quatmann, T., Junges, S., Katoen, J.P.: Markov automata with multiple objectives. In: CAV (1), LNCS, vol. 10426, pp. 140–159. Springer (2017)
110. Quatmann, T., Katoen, J.P.: Sound value iteration. In: CAV (1), LNCS, vol. 10981, pp. 643–661. Springer (2018)
111. Ruijters, E., Stoelinga, M.: Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Rev. **15**, 29–62 (2015)
112. Segala, R., Lynch, N.A.: Probabilistic simulations for probabilistic processes. Nord. J. Comput. **2**(2), 250–273 (1995)
113. Somenzi, F.: CUDD 3.0.0. http://vlsi.colorado.edu/~fabio/CUDD/html/. Also available at https://github.com/ivmai/cudd
114. Spel, J., Junges, S., Katoen, J.P.: Are parametric Markov chains monotonic? In: ATVA, LNCS, vol. 11781, pp. 479–496. Springer (2019)
115. Sullivan, K.J., Dugan, J.B., Coppit, D.: The galileo fault tree analysis tool. In: FTCS, pp. 232–235. IEEE Computer Society (1999)
116. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state programs. In: FOCS, pp. 327–338. IEEE Computer Society (1985)
117. Volk, M., Junges, S., Katoen, J.P.: Fast dynamic fault tree analysis by model checking techniques. IEEE Trans. Ind. Inform. **14**(1), 370–379 (2018)
118. van Dijk, T.: Sylvan: multi-core decision diagrams. Ph.D. thesis, University of Twente, Enschede, Netherlands (2016)
119. van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. STTT **20**(2), 157–177 (2018)
120. Wachter, B.: Refined probabilistic abstraction. Ph.D. thesis, Saarland University (2011)
121. Wimmer, R.: Symbolische Methoden für die probabilistische Verifikation: Zustandsraumreduktion und Gegenbeispiele. In: Ausgezeichnete Informatikdissertationen, LNI, vol. D-12, pp. 271–280. GI (2011)
122. Wimmer, R., Jansen, N., Vorpahl, A., Ábrahám, E., Katoen, J.P., Becker, B.: High-level counterexamples for probabilistic automata. In: QEST, LNCS, vol. 8054, pp. 39–54. Springer (2013)
123. Wimmer, R., Kortus, A., Herbstritt, M., Becker, B.: Probabilistic model checking and reliability of results. In: DDECS, pp. 207–212. IEEE Computer Society (2008)
124. Winkler, T., Junges, S., Pérez, G.A., Katoen, J.: On the complexity of reachability in parametric markov decision processes. In: CONCUR, LIPIcs, vol. 140, pp. 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
125. Winterer, L., Junges, S., Wimmer, R., Jansen, N., Topcu, U., Katoen, J.P., Becker, B.: Motion planning under partial observability using game-based abstraction. In: CDC, pp. 2201–2208. IEEE (2017)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.