Joachim Klein, Christel Baier, Philipp Chrszon, Marcus Daum, Clemens Dubslaff, Sascha Klüppelholz, Steffen Märcker, David Müller

Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic Büchi automata

Erstveröffentlichung in / First published in:

International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. Springer Link. ISSN 1433-2787.

DOI: https://doi.org/10.1007/s10009-017-0456-3

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

Advances in Probabilistic Model Checking with PRISM: Variable Reordering, Quantiles and Weak Deterministic Büchi Automata

Joachim Klein, Christel Baier, Philipp Chrszon, Marcus Daum, Clemens Dubslaff, Sascha Klüppelholz, Steffen Märcker, David Müller*

Institute of Theoretical Computer Science
Technische Universität Dresden, 01062 Dresden, Germany

Abstract. The popular model checker PRISM has been successfully used for the modeling and analysis of complex probabilistic systems. As one way to tackle the challenging state explosion problem, PRISM supports symbolic storage and manipulation using multi-terminal binary decision diagrams for representing the models and in the computations. However, it lacks automated heuristics for variable reordering, even though it is well known that the order of BDD variables plays a crucial role for compact representations and efficient computations. In this article we present a collection of extensions to PRISM. First, we provide support for automatic variable reordering within the symbolic engines of PRISM and allow users to manually control the variable ordering at a fine-grained level. Second, we provide extensions in the realm of reward-bounded properties, namely symbolic computations of quantiles in Markov decision processes and, for both the explicit and symbolic engines, the approximative computation of quantiles for continuous time Markov chains as well as support for multi-reward-bounded properties. Finally, we provide an implementation for obtaining minimal weak deterministic Büchi automata for the obligation fragment of Linear Temporal Logic (LTL), with applications for expected accumulated reward computations with a finite horizon given by a co-safe LTL formula.

* This is a post-peer-review, pre-copyedit version of an article published in the International Journal on Software Tools for Technology Transfer. The final authenticated version is available online at: `https://doi.org/10.1007/s10009-017-0456-3`. The authors are supported by the DFG through the collaborative research centre HAEC (SFB 912), the Excellence Initiative by the German Federal and State Governments (cluster of excellence cfaed), the Research Training Groups QuantLA (GRK 1763) and RoSI (GRK 1907), and the DFG-projects BA-1679/11-1 and BA-1679/12-1.

1 Introduction

The prominent probabilistic model checker PRISM [47, 37, 48] provides support for modeling and the analysis of discrete-time Markov chains (DTMC) and Markov decision processes (MDP) as well as continuous-time Markov chains (CTMC) against temporal logical specifications. While the behavior of Markov chains is purely probabilistic, MDPs exhibit both probabilistic and nondeterministic choices. The typical task addressed within the analysis of MDPs is to compute a scheduler that resolves all the nondeterministic choices and that maximizes (or minimizes) expected values or the probability of a given path property.

One prominent approach to cope with the well-known combinatorical state-explosion problem in model checking is the use of symbolic methods, such as those based on binary decision diagrams (BDDs) [10, 43]. Various BDD-variants have been studied and implemented in tools for the quantitative analysis of probabilistic systems, see, e.g., [25, 3, 27, 47, 29, 45, 33, 40, 12, 28]. For its symbolic engines, PRISM uses algorithms relying on a multi-terminal binary decision diagram (MTBDD) [2, 22] representation of the model. Using these data structures, a symbolic representation of the Markovian models can be obtained and the required steps in the analysis (e.g., graph analysis on the transition matrix, matrix-vector multiplications, etc) can be carried out by manipulating the MTBDDs without requiring an explicit representation of the model. In particular this allows the analysis of models with extremely large state spaces that are infeasible to handle with explicit methods, provided that the structure of the model permits a compact MTBDD representation.

Overall, PRISM provides four different engines for the analysis of DTMCs, CTMCs and MDPs. The Explicit engine uses an explicit representation of the reachable state space for the system analysis, whereas the Mtbdd engine completely builds on MTBDD-based symbolic

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

representation. Additionally, there are two semi-symbolic engines, called HYBRID and SPARSE that rely on the symbolic data structures and combine them with explicit representations and computations during the numerical computations. HYBRID combines the MTBDD-based representation of the transition matrix with an explicit representation of the state value solution vector [35] during the iterative computations. This is motivated by the observation that the MTBDD representation of such state value vectors can become quite large during such computations, even for models that have a compact MTBDD representation themselves. The SPARSE engine constructs an explicit, sparse matrix from the MTBDD-based transition matrix for numerical computations and performs computations using this explicit representation. While the three (semi-)symbolic engines rely internally on the infrastructure of the C-based CUDD library [52] for MTBDD storage and manipulation, the EXPLICIT engine is fully implemented in Java.

The efficiency of each of the four engines crucially depends on the concrete model, its structure and size as well as the objective(s) under consideration. For each of the engines there are situations where it can show its particular strength. While the EXPLICIT engine is particularly well-suited for initial prototype implementations, it is desirable to have support for the symbolic engines as well to allow the handling of models where the state space exceeds what can be explicitly represented. In practice, it is thus of considerable interest that novel analysis techniques are implemented both for the EXPLICIT and (semi-)symbolic engines.

It is well known that the ordering of the (MT)BDD variables plays a crucial role for obtaining a compact representation of the model and for model-checking performance. Within PRISM the order of the BDD variables is fully determined by the order in which the individual modules and the state variables are written down in the model file. Hence, the only influence on the order of BDD variables is by changing the order of module definitions and variable declarations. In our previous work, when applying PRISM for the analysis of complex systems, in particular for models for which explicit approaches were infeasible (e.g., [16]), we often had to manually swap the modules and variable definitions for finding a better ordering. While care has been taken to use a sensible variable order derived from the structure of elements in the model description [47], PRISM lacks any support for automatically finding a good variable order using dynamic reordering techniques such as sifting [49,46], which are routinely employed in symbolic model checkers for non-probabilistic systems (e.g., [13]).

Contributions. This article presents several enhancements of PRISM. Motivated by observed limitations in practice, our aim here is to provide an increase in the efficiency of the symbolic engines and extend the functionality to allow the analysis of a wider range of models and properties.

First, we address the issue of variable ordering in PRISM by adding support for the *automated variable reordering* of the MTBDD-based model representation by enabling CUDD's implementation of group sifting. This is complemented by extensions of PRISM's input modeling language that allow to rearrange and interleave the order of the bits of state variables within the same module as well as (the bits) of state variables of different modules. The impact of the automated reordering has been evaluated using examples from the PRISM benchmark suite [38] and in the context of the symbolic quantile computations.

Our second contribution are extensions of PRISM in the area of reward-bounded properties. This includes computation schemes for *cost-* or *reward-bounded reachability properties* in DTMCs and MDPs. Additionally, we provide implementations in the MTBDD, HYBRID and SPARSE engines for computing *quantiles* for reward-bounded reachability properties in MDPs, complementing the prototypical implementation in the EXPLICIT engine presented in [4] and report on the results of comparative experimental studies. Furthermore, we also present an implementation, in all four engines, for the approximative computation of quantiles for time-bounded reachability properties in CTMCs using the algorithm proposed in [5].

Finally, we augment the existing implementation for Linear Temporal Logic (LTL) model checking in PRISM by a specialized treatment for the obligation fragment of (LTL) [14]. While full LTL requires the use of complex acceptance conditions such as Rabin acceptance to construct a deterministic ω-automata used for probabilistic model checking of LTL path properties, the obligation fragment allows a conversion to *weak deterministic Büchi automata* (WDBA). Büchi acceptance conditions are structurally simple and require that a given set of states is visited infinitely often. Crucially, WDBA enjoy as well the property that they can be efficiently minimized to a minimal WDBA with at most as many states as any other ω-regular automaton for the same language, which is in general not the case for ω-automata with more complex acceptance conditions or structure. We have implemented the translation and minimization algorithms [14,41] to obtain this minimal automaton. As the probability computations are carried out in the product model formed from the original model and the deterministic automaton obtained from the LTL formula, the size of the automaton significantly impacts the efficiency of the analysis and a specialized treatment of this fragment is worthwhile. We also apply the translation to minimal WDBA in the context of expected accumulated reward computations with a finite horizon as given by a co-safe LTL formula [34,39], where the regular structure of the obtained automaton is crucial for correctness.

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

Outline. Section 2 presents our new approaches for variable reordering in PRISM. Section 3 summarizes the main features of our implementations for cost/reward-bounded properties and quantiles, while Section 4 presents our implementation for obtaining weak deterministic Büchi automata for the obligation fragment of LTL and computing expected accumulated rewards with co-safety formulas. Throughout the article, we assume a basic familarity with the concepts and notations used in probabilistic model checking and PRISM and refer to [21] for a tutorial-style introduction.

For further details (implementation, experiments) see `http://wwwtcs.inf.tu-dresden.de/ALGI/PUB/STTT/` We are collaborating with PRISM's authors to eventually integrate our extensions into the main version and would like to thank David Parker for fruitful discussions. [1]

2 Automatic variable reordering in PRISM

Here, we will briefly describe the relevant infrastructure in PRISM for dealing with variable ordering. The MTBDD variable ordering of the symbolic model representation is determined by the order of module and variable definitions in the PRISM model file. Fig. 1 sketches the general schema.[2] In a first block, MTBDD variables for nondeterministic choices are allocated. This includes a unary encoding of the synchronizing actions (i.e., one MTBDD variable for each action), scheduling variables (one MTBDD variable indicating that a given module is active) as well as several bits for representing local choices, e.g., between alternative commands for the same synchronizing action. Then, two blocks of extra variables are preallocated to serve in later model transformations, e.g., during a product construction with a deterministic ω-automaton for LTL model checking. For each individual bit of a state variable in the model, two MTBDD variables are allocated, one serving in the representation of the rows and one for the columns of the transition matrix. The MTBDD variables for representing the possible values of the (integer-valued) state variables are allocated in the order in which they appear in the PRISM model file, with each state variable forming a block of row/column pairs. The bits for each state variable are ordered from most-significant to least-significant. Global state variables are treated as if they were contained in a single module located before the "real" modules.

The arrows in Fig. 1 indicate the extent of the influence that can be applied to the variable ordering by syntactically reordering the PRISM source file: At the highest level, the order of the modules can be changed. Additionally, inside each module, the order of the definition of the state variables can be changed. Note that such changes of the ordering in the PRISM model file do not lead to any semantic changes in the model, but can lead to cosmetic changes, e.g., in the order of the states for exported models. To complement the manual, trial-and-error approach for finding a good order in the model file, we detail our automatic approach in the next section.

2.1 Automatic variable reordering using group sifting

PRISM internally relies on the CUDD (MT)BDD library [52] for the management of a set of BDDs that arise during probabilistic model checking. CUDD provides implementations of several heuristics for (dynamic) variable reordering which in principle should be available to be used by PRISM. Unfortunately, the implementation of PRISM heavily relies on the assumption that the variable ordering of the MTBDD does not change at all. The order of the MTBDD variables is assumed to correspond with the order of the respective variables in the underlying PRISM model, i.e., that the variable index (logical index) and the variable level (index in the current variable order) need to agree. Eliminating this restriction on the variable order would require a substantial refactoring of PRISM's infrastructure, touching many parts of the implementation.

Our approach presented in this section makes automatic variable reordering available to a PRISM user while avoiding any substantial refactoring of PRISM's infrastructure. First, a symbolic, MTBDD-based representation of the model is built by PRISM as usual. After the model is built, we trigger the group sifting reordering heuristic [49, 46] via the CUDD library, using several variable grouping constraints that will be detailed later. After this reordering, the MTBDD-based model representation violates PRISM's assumptions, which renders further computations in PRISM impossible. Thus, we perform an analysis of the variable ordering found by group sifting and translate the changes in variable locations back to the source level of the PRISM model. This way, we obtain a syntactically reordered PRISM model, where the placement of the PRISM modules and state variables reflects the calculated variable ordering. Our implementation then allows using this reordered model directly after the reordering computation via the following trick: After reordering, we delete the MTBDDs of the model and reset the variable ordering in CUDD to the one that PRISM expects, where each variable index corresponds to the variable level in the BDD. Then, we build the BDDs for the model a second time, this time using the syntactically reordered PRISM model. We thus obtain the reordered model again, but now with the underlying assumptions of PRISM intact, allowing to use the full PRISM machinery. This approach provides transparent

[1] This article is an extended version of the TACAS'16 paper [31].

[2] The depicted scheme corresponds to the default ordering for the Hybrid and Sparse engines. There are subtle differences when using the Mtbdd engine, discussed in Appendix A. Additionally, standard PRISM preallocates only extra *state* variables, mainly for the product with deterministic automata. To support generic symbolic model transformations, we also preallocate choice variables, i.e., for fresh actions in the transformed MDP.

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

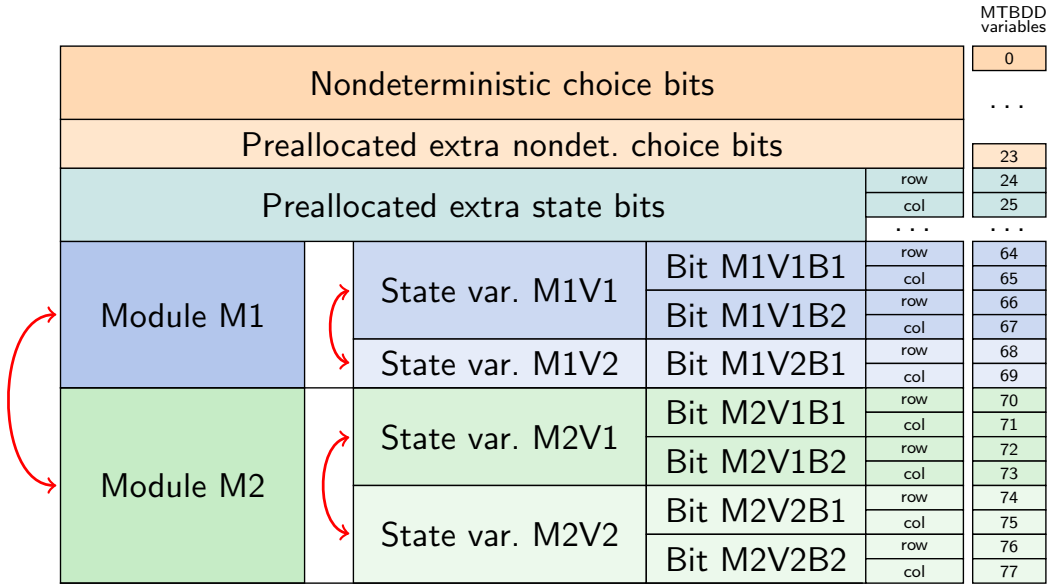

Fig. 1. Schema for the standard variable ordering used by PRISM. The arrows indicate the effect of syntactic reordering in the PRISM model file on the variable order.

and convenient access to the reordering functionality to the user. Additionally, we also support exporting the reordered model to a file, which can then be used in future PRISM runs. This way, the time for reordering can be amortized over multiple model-checking runs.

For this approach to work, it is crucial that we are able to seamlessly convert between the reordered variable ordering obtained after sifting and the variable order that is induced by syntactically reordering the elements of the PRISM model file. To achieve this, we introduce appropriate groups of MTBDD variables represented by a tree structure and used in the groups sifting. The grouping reflects the structure of the given model file: Each PRISM module forms a group of BDD variables that can be reordered as a block. This corresponds to syntactically changing the order of modules in the model file. Additionally, inside each module, the MTBDD variables for each state variable form another group. Reordering those groups corresponds to changing the order of the variable declarations inside a PRISM module. The remaining variables, e.g., those for nondeterministic choices remain in fixed positions. Hence, the above approach allows for creating all variable orders that can result from permutations of modules and state variables within the PRISM model file. In the next section we show how a more fine-grained control can be achieved.

2.2 Bit-level control over the variable order using views

Although it is well known that for some operators, e.g., the addition of two integers, an efficient representation relies on the interleaving of the individual bit-variables, there is no way of interleaving the individual bits of multiple state variables in PRISM up to now.

```
module M
   s_bit_2 : [0..1];
   s_bit_1 : [0..1];
   s_bit_0 : [0..1];

   s : view (s_bit_2,s_bit_1,s_bit_0)
          <=> [2..7] init 3;

   [inc] s<7 -> 1:(s'=s+1);
endmodule
```

Fig. 2. Defining a view `s` with data domain $(2,7)$ from three single-bit state variables.

Our implementation provides the option of syntactically "exploding the bits" of all the state variables in a PRISM model file: Each multi-bit state variable s is replaced with the appropriate number of single-bit variables s_i. To keep this transformation simple and transparent to the user we introduce a syntactic enhancement of the PRISM modeling language called a *view*. A view forms a virtual variable s over bit variables s_j. This virtual variable can be used in guards and updates of transition definitions just as ordinary variables. Hence, exploding the bits does not affect any of the transition definitions given in the model file.

As an example, consider the PRISM module in Fig. 2. Here, the virtual state variable `s` with an integer data domain of $2 \leqslant \mathtt{s} \leqslant 7$ requires three bits to represent all values, as internally integer variables are encoded by first subtracting the lower bound of the data domain (2 is internally represented as 0, etc.). The actual storage is provided by the three single-bit state variables `s_bit_`i. The order of the single-bit state variables in the view

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

definition determines their use in the encoding, with the most-significant bit appearing first. As can be seen, the virtual view variable `s` is being used just like a standard PRISM state variable.

Note that "exploding the bits" of a PRISM model file alone will not change the variable ordering and MTBDD representation, as the encoding and ordering of the newly introduced single-bit state variables correspond to the standard encoding used for the original variables. When applying the automatic reordering detailed in the previous section to an "exploded" model file, the individual bits of the state variables can be now be sifted and interleaved, as their grouping is removed. However, the MTBDD variables are still restricted from crossing module boundaries. We detail how to remove this restriction in the next section.

2.3 Interleaving state variables of different modules

To overcome the limitation that state variables cannot be interleaved across modules our implementation provides the option of "globalizing" all state variables in a PRISM model file: Each state variable inside a PRISM module is moved from the module to become a global variable, while keeping the order they appeared in the original model file. Realizing this requires to loosen some restrictions on the use of global variables imposed by PRISM. In standard PRISM, global variables cannot be updated in synchronous actions, as this has the potential of resulting in conflicting updates from multiple modules. We removed this restriction, as in our setting only the "previous owner" of a variable, i.e., the module in which the variable was initially declared, will update the global variable in the transformed model. This ensures that there can be no conflicting updates introduced by globalizing variables. Our implementation supports such global variable updates for similar situations as well, i.e., where it is apparent by a syntactic inspection that no conflicting updates can happen.

The options for exploding the bits and globalizing the variables can be used separately and in a combined fashion (cf. Fig. 3) and the resulting model yields a starting point for group sifting. This way, fine-grained control of the variable ordering for all state variables in the model becomes possible. Within the following section we will evaluate our implementation by means of a number of case studies.

2.4 Benchmarking automatic variable reordering of PRISM models

To explore the effect of automatic variable reordering using our implementation, we performed benchmarks using the DTMC, CTMC and MDP models in the PRISM benchmark suite [38]. The models are parameterized in various parameters, affecting both the number of states and the size of the MTBDD representation. In total, we performed benchmarks with 208 model instances (70 DTMCs, 70 CTMCs, 68 MDPs). We present here statistics for the "top" initial variable ordering [47] used by default in the HYBRID engine. Results using the default variable ordering of the MTBDD engine were roughly similar.

Before "explode bits" and "globalize":

```
module M1
  x : [0..3] init 0;
 [a] true ->   0.5:(x'=0)
             + 0.5:(x'=y);
endmodule
module M2
  y : [0..3] init 0;
 [a] true -> 1:(y'=0);
 [b] y<3  -> 1:(y'=y+1);
endmodule
```

After "explode bits" and "globalize":

```
global x_bit_1 : [0..1];
global x_bit_0 : [0..1];
global y_bit_1 : [0..1];
global y_bit_0 : [0..1];
global x :
  view (x_bit_1,x_bit_0) <=> [0..3] init 0;
global y :
  view (y_bit_1,y_bit_0) <=> [0..3] init 0;
module M1
  [a] true -> 0.5:(x'=0) + 0.5:(x'=y);
endmodule
module M2
 [a] true -> 1:(y'=0);
 [b] y<3  -> 1:(y'=y+1);
endmodule
```

Fig. 3. Example of both "exploding bits" and "globalizing variables" for a PRISM model file, before and after.

Fig. 4 presents statistics for the basic case, i.e., reordering without any syntactic transformations beforehand. Similar plots for reordering with the "globalize variables" (Sec. 2.3) and "explode bits" (Sec. 2.2) transformations being applied can be found in the appendix.[3] In the plots, the model instances are grouped by their base model. The size of the MTBDD refers to the number of nodes in the shared MTBDD structure storing the various individual MTBDDs. Those individual MTBDDs represent the model in PRISM, i.e., its transition matrix, a 0/1-version of the transition matrix representing the underlying graph structure of the model, the set of reach-

[3] The benchmarks for reordering were carried out on a machine with two Intel Xeon L5630 4-core CPUs at 2.13GHz and 192GB RAM, with a timeout of 1 hour and a CUDD memory limit of 10GB. The max-growth factor of CUDD was set to 2, i.e., allowing a doubling in MTBDD size before sifting is abandoned.

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

able states, representations for the transition and state rewards.

As can be seen in the second plot from the top in Fig. 4, the automatic reordering was able to achieve significant reductions for many of the model instances. As a particularly striking example, the reordering was very effective for the "mapk-cascade" model, a CTMC: For the instance with parameter $N = 8$, the MTBDD size was reduced from 1,478,511 nodes to 96,718 nodes, a reduction of more than 90%. The time for building the symbolic representation of this model instance was reduced from 124 s to less than 2 s for the reordered model. Most of the time, the reduction in the MTBDD size is accompanied by a reduction in the time needed for building the MTBDDs for the reordered model. The major outlier to this are several instances of the "crowds" model, where the time for building the reordered model was substantially worse compared with the original model. Our investigation revealed that this is due to the point in time at which our reordering is performed, i.e., after the symbolic transition matrix has been restricted to the reachable part of the state space, which is the symbolic representation that is then used for the actual model checking. The reordering heuristic thus produced a variable order tailored for this state space and which is not particularly suitable for the representation of the individual, not yet restricted parts of the model used during the building phase. This is a classic example of the case where an asynchronous reordering, i.e., continuously adapting the variable ordering during the construction phase, would be helpful.

In general, the time for reordering tends to be related to the size of the MTBDD before reordering, as expected. As noted above, even substantial reordering times might be worthwhile, as the reordered model can be stored and subsequently reused multiple times, profiting, e.g., from the reduced build time and more compact symbolic representation.

There were three models ("brp", "nand" and "poll"), where instances exhibited an overall reduction in the size of the MTBDD, but an increase in the size of the MTBDD for the transition matrix alone (in all cases the increase was less than 10%). This is explained by the fact that the reordering operates on the whole shared MTBDD data structure and thus does not necessarily optimize all the individual MTBDD functions that are stored.

We have also benchmarked the effect of our syntactic transformations on the automatic reordering and present here (Table 1) some notable examples. For further, detailed statistics we refer to the appendix. As already seen in Fig. 4, the "tandem" model has no reduction in MTBDD size when it is reordered as-is. However, when the state variables are "exploded", reordering becomes profitable, with additional reductions when combined with the "globalize variables" transformations. Globally, for every model instance from the benchmark suite, at least one of the variants achieved some reduction. As is to be expected, no variant is uniformly best. Consider the statistics for the "cluster" model in Table 1. For $N = 32$, "exploding" and "globalizing" are individually successful, but in combination lead to only minor reductions. For $N = 256$ and $N = 512$, the combination of "exploding" and "globalizing" becomes more successful, but the standard reordering leads to the most reductions. For "kanban" with $t = 6$, "globalizing" alone leads to worse reductions than reordering on the standard model. As can be seen, it remains an area of experimentation to select the reordering variant that is a good fit for a particular model and model instance.

As a good first assumption, the time for model checking tends to be related in general to the compactness of the symbolic representation. We present here statistics for the impact of the automatic reordering on the model checking time for two examples from the PRISM benchmark suite (Tables 2 and 3). The results are presented for the variant of reordering that provided the best reduction of the BDD size. In case of the "egl" case study this was the combination of "globalize" and "explore bits", whereas for the "fms" case study using "explode bits" by itself turned out to yield the highest reduction. For each instance, the tables present statistics for the number of reachable states, the size of the MTBDD for the transition/rate matrix after reordering, the reduction in size of the MTBDD representation of the matrix due to reordering and the time spent for reordering. Additionally, the tables depict the time for computing the respective query: once for the original model (without reordering) and once for the reordered model. As can be seen, the more compact MTBDD representation corresponds here to a reduction in the model checking time as well, with the largest relative improvement achieved for N=9 in the "fms" case study. It has to be kept in mind that the reordering time can be amortized over multiple runs of the model checker and multiple queries by reusing the reordered model.

However, as is usual in symbolic methods using BDDs, a more compact representation of the model does not necessarily guarantee an improvement in model-checking time. One possible reason for this can be that a variable order most suitable for representing the model might not be suitable for efficiently storing the value vectors that arise in the numerical algorithms as well. Similarly, a good order for the fully constructed model (restricted to the reachable state space) may be inefficient during the compositional construction phase where the different parts of the model have to be represented individually and are then composed. However, in the cases where our heuristic does work well, the improvements can be quite significant and help substantially in making large models tractable. In the next section, we will report on significant reductions in model-checking time in the context of quantile computations on the basis of reordered models.

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

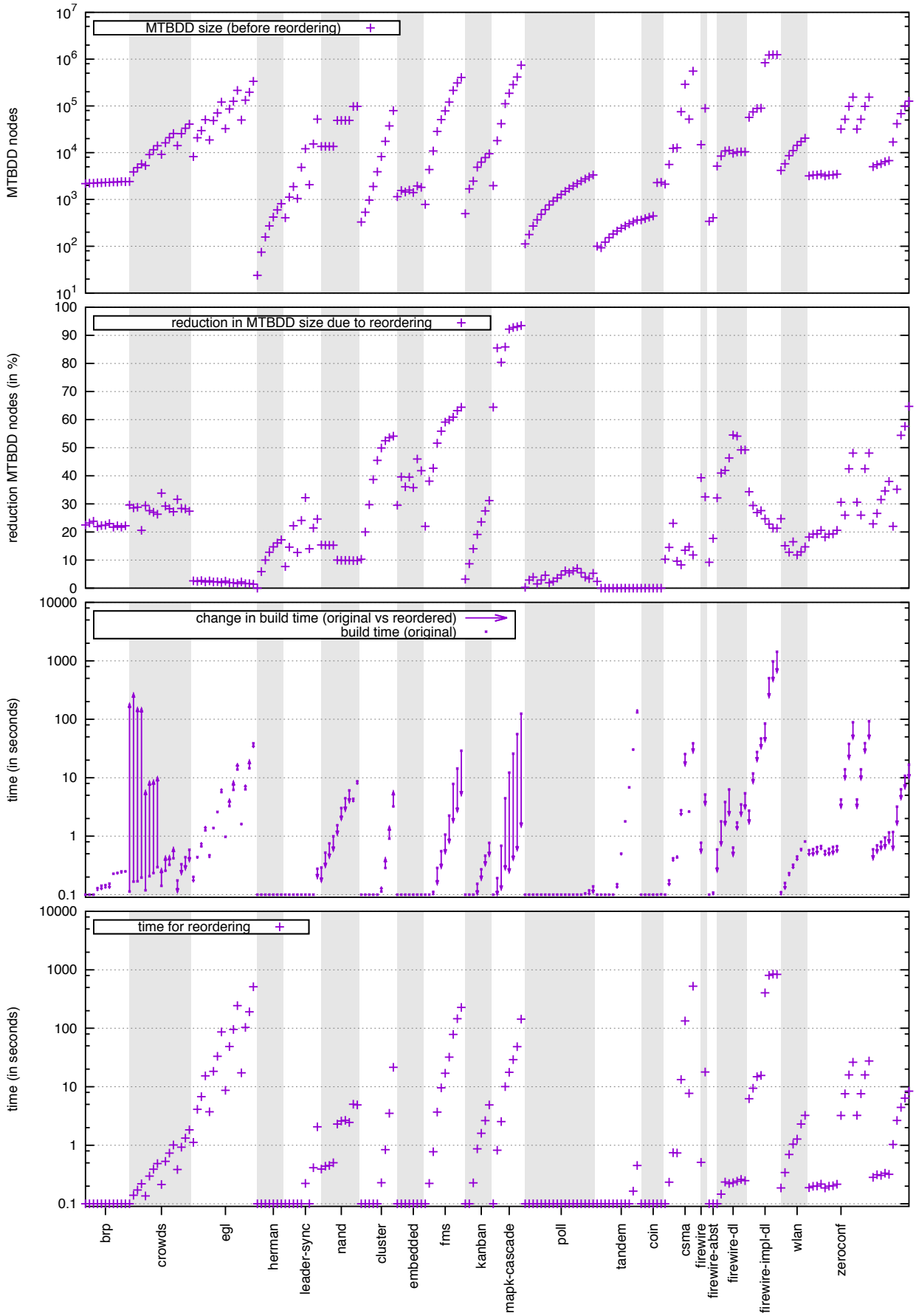

Fig. 4. Statistics for reordering without syntactic transformations: The number of MTBDD nodes before reordering, the reduction (larger numbers represent more reduction) in the number of MTBDD nodes, the change in time for building the model (before/after reordering) and the time spent reordering. Times below 0.1 seconds are clipped to 0.1 for visualization purposes. There was one timeout, reordering the "csma4_6" instance (30 minutes of the 1 hour timeout spent on building, with 3,589,198 nodes).

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

Table 1. Selected statistics for the reduction achieved using reordering on the standard model instance and where the "explode bits" and "globalize variables" transformations were applied. In the last column, both transformations are applied. For reference, the MTBDD size before reordering is included as well. For full details, see appendix.

Model instance		MTBDD before	reduction in % standard	explode	globalize	exp.+glob.
tandem	c=255	4 917	0.0	26.0	0.0	35.1
tandem	c=4095	103 233	0.0	35.7	0.0	64.3
cluster	N=32	7 391	45.5	47.9	52.2	8.3
cluster	N=256	61 749	53.6	42.0	24.2	39.8
cluster	N=512	129 740	54.1	42.0	26.2	47.5
kanban	t=6	14 001	27.5	34.0	1.2	32.3

Table 2. Statistics for the "egl" case study with "unfairA" query (L=8), MTBDD engine and "top" initial variable ordering. Reordering with "globalize" and "explode bits".

Instance	States	reordered MTBDD	reduction in %	t_{reorder}	t_{query} original	t_{query} reordered
N=5	156 670	21 797	56.9	9.0 s	2.5 s	1.3 s
N=10	317 718 526	72 001	40.1	39.5 s	14.2 s	8.9 s
N=15	486 405 046 270	146 352	31.9	97.8 s	39.1 s	29.4 s
N=20	663 005 511 548 926	238 028	28.8	196.4 s	82.5 s	57.6 s

Table 3. Statistics for the "fms" case study with "productivity" query, HYBRID engine. Reordering with "explode bits".

Instance	States	reordered MTBDD	reduction in %	t_{reorder}	t_{query} original	t_{query} reordered
N=5	152 712	20 681	59.4	4.1 s	11.3 s	8.7 s
N=6	537 768	30 228	61.3	7.2 s	50.2 s	39.6 s
N=7	1 639 440	40 610	66.4	12.5 s	168.1 s	142.2 s
N=8	4 459 455	72 908	66.1	22.8 s	637.2 s	496.2 s
N=9	11 058 190	102 514	66.8	49.6 s	3148.2 s	1434.6 s

3 Reward-bounded reachability and quantiles

Models in PRISM can be annotated with rewards (non-negative values) specifying the costs or the gain for visiting certain states or taking certain transitions. PRISM provides implementations of algorithms for reasoning about expected rewards, but lacks support for computing the probabilities for reward-bounded path properties, except for the special case of step-bounded formulas (see [21] for a tutorial-style introduction to the underlying concepts, notations and algorithms).

Reward-bounded path properties are used, for example, for reasoning about resource-constraints (e.g., "what is the probability of reaching the goal within some energy budget" in the case of upper bounds) or for requiring a certain amount of utility (e.g., "what is the probability of finishing a task while having served at least a certain number of customers" in the case of lower bounds).

3.1 Reward-bounded reachability

We have extended PRISM with support for the computation of (extremal) probabilities of cost-/reward-bounded reachability path formulas for DTMCs and MDPs with non-negative integer rewards. For this, the standard reachability operator $\Diamond \Phi$ ("eventually some state satisfying state formula Φ is reached") is augmented with a reward bound, i.e., of the form $\Diamond^{\leq r} \Phi$ with reward bound r ("eventually some state satisfying Φ is reached while accumulating no more than r reward along the way"). Here, the accumulated reward for a given path fragment in the model corresponds to the sum of the rewards that are assigned to the states and actions that comprise the path fragment. Lower reward bounds, i.e., of the form $\Diamond^{\geq r} \Phi$, are supported as well.

For MDPs, this results in queries such as $\Pr^{\max}(\Diamond^{\leq r} \Phi)$ for a reward bound r and state formula Φ, which asks: "what is the maximal probability of reaching some state satisfying Φ while accumulating at most reward r", where the maximum is taken over all schedulers, i.e., resolutions of the non-determinism in the MDP.

For a single reward bound and $\Pr^{\max}$, we can use the iterative computation of the values $x_{s,r} = \Pr_s^{\max}(\Diamond^{\leq r} \Phi)$ for reward bounds $r = 1, 2, 3, \ldots, r$. This yields the building block of the algorithm for the computation of quantiles as proposed in [4] and discussed in the next section. For the $\Pr^{\min}$ operator (asking for minimal probabilities) and for Pr in the case of a DTMC, the analogous computations can be performed. For conjunctions of multiple

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

reward bounds and step bounds [1], we rely on a synchronous product of the MDP (or DTMC) with a finite automaton that tracks the accumulated reward. We can hence reduce the case of multiple bounds to the base case with only a single step bound or reward bound. This is implemented for the explicit and the (semi-)symbolic engines.[4] As the case of a reward-bounded simple path formula (i.e., a single eventually, until, globally or release operator with state formula operands) can easily be reduced to a (possibly negated) reachability question via simple transformations, those kinds of reward-bounded path formulas are supported as well.

3.2 Computing quantiles in MDPs

In our recent work [55,4], we addressed the computation of quantiles in Markov models for probability constraints on reward-bounded reachability formulas. The prototypical implementation based on PRISM's Explicit engine used in [4] for experimental studies has been refined and extended with implementations for the Mtbdd, Hybrid and Sparse engines. In what follows we describe important details of this implementation. We consider here MDPs and reward functions $rew\colon S \times Act \to \mathbb{N}_{\geqslant 0}$, mapping state-action pairs (s,α) to the non-negative integer reward $rew(s,\alpha)$. The considered type of quantiles (for details we refer to [4]) stand for optimal reward thresholds that guarantee that the maximal or minimal probability of a reward-bounded reachability path formula meets some probability bound. Examples are

$$\min\{\, r \in \mathbb{N} \,:\, \Pr^{\max}(\Diamond^{\leq r}\, \Phi) > p \,\}$$
$$\max\{\, r \in \mathbb{N} \,:\, \Pr^{\min}(\Diamond^{\geq r}\, \Phi) > p \,\}$$

where r can be seen as a parametric reward bound, Φ is a state formula and p a rational probability bound. The first of the two queries above could, for example, stand for the question "in the best case, what is the minimal amount of energy r that is required to ensure that some goal state Φ is reached with at least probability p". Quantiles thus yield a useful concept for a cost-utility analysis, allowing reasoning over some or all schedulers. The approach for computing quantiles as proposed in [4] consists of a two-step process. A precomputation step determines all states $s \in S$ for which the quantile exists, i.e., is finite. In the simplest case, this amounts to the computation of the maximal probability for unbounded reachability. In other cases, the computation requires the analysis of zero-reward and positive-reward end components [4]. For the remaining states, an iterative approach is used, which we illustrate here for a quantile of the form $\min\{\, r \in \mathbb{N} \,:\, \Pr^{\max}(\Diamond^{\leq r}\, \Phi) > p \,\}$ where we suppose the MDP has a unique initial state s_0. Successively, the values $x_{s,r} = \Pr_s^{\max}(\Diamond^{\leq r}\, \Phi)$ for $r = 1, 2, 3, \ldots$ are computed for all states $s \in S$ until some r with $x_{s_0,r} > p$ is reached, using the equation $x_{s,r} = \max\{A_s, B_s\}$ with

$$A_s = \max_{\alpha \in Act(s),\, rew(s,\alpha)=0} \sum\nolimits_{t \in S} P(s,\alpha,t) \cdot x_{t,r}$$
$$B_s = \max_{\alpha \in Act(s),\, rew(s,\alpha)>0} \sum\nolimits_{t \in S} P(s,\alpha,t) \cdot x_{t,r-rew(s,\alpha)}$$

where $P(s,\alpha,t)$ is the probability of reaching state t when action α is chosen in state s. For states satisfying Φ, $x_{s,r}$ is set to 1 for all r. The values A_s, handling the zero-reward actions, are computed using value iteration. The values B_s, handling the positive-reward actions, are determined by inserting the previously calculated values $x_{t,i}$ for $i < r$. For the other quantile variants, similar computations are performed [4]. The time complexity of this approach is pseudo-polynomial. Given the PSPACE-hardness result by [24], no polynomial-time algorithm can be expected.

3.3 Symbolic computation of quantiles in MDPs

We have extended PRISM with implementations for the computation of quantiles with the Mtbdd, Hybrid and Sparse engines, following the general approach outlined above. For the precomputation step, we rely on the PRISM machinery allowing for the computation of maximal/minimal probabilities for unbounded path formulas. The implementation for calculating (maximal) end components has been adapted by appropriate (symbolic) model transformations such that states in positive-reward end components and zero-reward end components can be identified.

The implementations for the Mtbdd, Hybrid and Sparse engines use tailored approaches for the iterative computation of the values $x_{s,r}$, $r = 1, 2, 3, \ldots$ until the probability threshold p is reached.

3.3.1 Iterative computation in the Mtbdd engine

The computed values $x_{s,r}$ are stored symbolically, using one MTBDD per bound r for representing the functions $x_r\colon S \to \mathbb{Q}$. For the computation of $x_{s,r+1}$, the positive reward fragment of the MDP is handled first, computing the MTBDD $B\colon S \to \mathbb{Q}$, i.e., the values B_s mentioned above, representing the result of choosing the "best" positive-reward actions. Here, all state-action pairs with identical reward value are handled simultaneously. Consequently, this symbolic approach tends to be most efficient if there are many state-action pairs, but few distinct reward values in the model.

For the handling of the zero-reward fragment in the MDP, we employ a modified version of PRISM's standard symbolic value iteration for computing extremal reachability probabilities in MDPs. The procedure is enriched with an additional comparison against the optimal values

[4] As the quantile algorithm for reward-bounded path formulas in the symbolic engines is currently only implemented for MDPs, reward-bounded reachability computations for DTMCs convert all reward bounds into counters in the state space.

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

among the positive reward actions as stored in B. Hence, in each iteration and for each state, either the currently considered zero-reward action is chosen or the optimal positive-reward action represented by the value stored in B.[5]

3.3.2 Iterative computation in the Hybrid engine

The Hybrid engine [47] relies on an explicit storage of the value vectors, combined with a symbolic storage of the transition matrix and a recursive procedure which traverses the symbolic matrix to compute, e.g., the matrix-vector product. To handle MDPs, for each distinct action a separate symbolic transition matrix is maintained, and each action is handled in turn.

For our quantile computations, we maintain separate matrices for the positive-reward fragment and for the zero-reward fragment. We additionally store explicit vectors for the previously computed values $x_{s,r}$ for the relevant values of r, i.e., those values that are still needed because they could still be reached by one of the positive-reward transitions. The handling of the positive-reward fragment then amounts to a modified recursion procedure, additionally tracking the relevant reward value. This way, once the terminal case in the recursion, i.e., the discovery of the probability $P(s, \alpha, t)$, is reached, we can lookup the relevant value $x_{t,r-rew(s,\alpha)}$ (see equation for B_s).

The subsequent computation for handling the zero-reward fragment then consists of an adapted variant of PRISM's Hybrid value iteration for extremal reachability probabilities, with the equivalent additional comparison with the optimal action in the positive-reward fragment as described above for the Mtbdd engine.

As an optimization, PRISM's Hybrid algorithms convert some of the lower levels of the MTBDDs for the matrices to sparse matrices, as this speeds up the very frequent bottom cases in the recursion at a negligible memory impact [47]. We support this in the quantile computations as well for the zero-reward fragment. For the positive-reward fragment, this optimization is used only if the rewards depend solely on the states or solely on the actions, as otherwise the computation of the corresponding reward during the recursion gets more complicated.

3.3.3 Iterative computation in the Sparse engine

The Sparse engine [47] of PRISM constructs an explicit sparse matrix for the MDP from the MTBDD representation and maintains explicit value vectors for the numerical algorithms.

Our quantile computations for the Sparse engine is structurally very similar to the one for Hybrid: We maintain a sparse matrix for the positive-reward and zero-reward fragments of the MDP. In addition, we maintain a sparse representation of the rewards assigned to each (s, α) pair, similar to PRISM's approach for computing expected reachability rewards. Again, we also store the relevant values $x_{s,r}$ from previous iterations. Then, for the positive-reward fragment, a parallel traversal of the matrix and the structure for the rewards obtains the values $P(s, \alpha, t) \cdot x_{t,r-rew(s,\alpha)}$. For the zero-reward fragment, we again use a modified variant of Sparse's value iteration.

3.4 Benchmarks for quantile computations in MDPs

To perform benchmarking of our implementation, we have reused several models and quantile queries that were first considered in [4] for benchmarking our implementation for PRISM's Explicit engine. We present here (Table 4) statistics for some noteworthy model instances. For further statistics and details on the models and quantile queries we refer to Appendix C. [6]

For the "Self-stabilizing algorithm" case study, our Hybrid and Sparse implementations tend to outperform the Explicit implementation, particularly due to the efficient model building phase. For the Mtbdd engine, however, this is an example where the numerical computation phase takes a long time, even though the model can be represented as a compact MTBDD. This is mostly due to the non-compact symbolic representation of the symbolic state value vectors during the numerical iterations.

In contrast, for the "asynchronous leader election" case study, the Mtbdd implementation becomes competitive for the larger model instances. The Explicit implementation is hampered here by the time spent for building the model, which becomes infeasible for the largest instance.

For the first "energy aware job scheduling" case study, eventually the Mtbdd engine becomes competitive for large instances. Here, it is interesting that the time t_{query} tends to be smaller for Explicit compared to Sparse. This is due to some additional heuristics used in the numerical quantile computations in the Explicit engine that have not yet been implemented in the Sparse engine. For the second "energy aware job scheduling" case study, all three (semi-)symbolic engines outperform the Explicit engine, which is mostly due to some implementation inefficiencies related to Java data structures in the precomputation phase of the quantile computations (e.g., 1735.5 s).

[5] As demonstrated in [26], the termination criterion for detecting convergence in the iterative numerical computations used by all engines of PRISM and by other probabilistic model checkers can lead to imprecise results for certain models. This is due to the convergence check succeeding before the overall result has converged with a sufficient precision. In the quantile calculations, we currently follow the standard PRISM approach. In separate work, we are currently working on an implementation of the fix proposed in [26]. As the relevant parts of the quantile computations are similar, those can then be adapted easily as well.

[6] The benchmarks for the quantile computations were carried out on a machine with two Intel E5-2680 8-core CPUs at 2.70 GHz with 384GB of RAM running Linux.

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

Table 4. Quantile computations for selected case studies, with statistics for the model size (reachable state space, MTBDD size of symbolic transition matrix) and times spent for model building and computing the quantile query (in seconds). The "iter." column depicts the number of overall iterations in the quantile computation.

N	States	MTBDD size	iter.	Explicit		symbolic quantile computations: Hybrid		Sparse		Mtbdd	
				t_{build}	t_{query}	t_{build}	t_{query}	t_{build}	t_{query}	t_{build}	t_{query}
Self-stabilizing algorithm (Israeli/Jalfon), N processes (query Q1)											
11	2 047	433	144	0.5s	0.2s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	1.1s
15	32 767	729	271	1.7s	3.5s	< 0.1s	1.6s	< 0.1s	1.3s	< 0.1s	74.1s
18	262 143	993	392	8.7s	37.0s	< 0.1s	22.1s	< 0.1s	18.4s	< 0.1s	1 338.9s
Asynchronous leader election, N processes (query Q3)											
7	2 095 783	180 383	13	62.1s	25.7s	4.5s	97.7s	4.5s	52.1s	4.5s	156.6s
8	18 674 484	392 093	13	894.0s	371.7s	17.3s	997.5s	17.3s	404.6s	17.3s	685.6s
9	167 748 115	868 257	14	–	–	68.1s	11 914.3s	68.3s	4 125.5s	68.2s	3 532.8s
Energy-aware job scheduling, N processes (query Q7)											
5	6 079 533	187 458	302	332.0s	365.2s	6.3s	1 493.9s	6.3s	404.8s	6.2s	776.0s
6	44 072 357	507 805	416	4 610.4s	2 623.2s	18.5s	19 408.2s	18.3s	5 003.7s	18.5s	2 815.7s
Energy-aware job scheduling, N processes (query Q8)											
5	3 049 471	25 363	13	72.6s	311.0s	0.5s	67.3s	0.5s	25.7s	0.5s	116.7s
6	7 901 694	38 911	15	226.8s	1 960.5s	0.9s	209.4s	0.9s	81.5s	0.9s	345.3s

Overall, it can be seen that the different implementations have their strengths and weaknesses, depending on the concrete model and situation. Interestingly, our quantile computation implementation in the Hybrid engine was not able to demonstrate its usual strengths, i.e., in situations where the Sparse engine becomes infeasible but the hybrid approach can still outperform the Mtbdd engine. This is partially due to the fact that our experiments were run with a large amount of memory, the additional memory requirements due to the quantile computation (storage of the values $x_{s,i}$ for a sufficiently large number of previous iterations). Additionally, we noticed that, for the larger models, a relatively large amount of time is spent for the setup of the data structures for the hybrid computations. We believe that in this area there is still substantial opportunity for efficiency gains.

For the "energy-aware job scheduling" case study, the computations were carried out in a reordered model, using the methods presented in Section 2. This led to a significant decrease in MTBDD size and computation times. For instance, for (Q7) and N=6 we observed a reduction in the size of the transition matrix of 78.2 % and the quantile computation (Sparse) took 6 201.2 s in the original model instead of 5 003.7 s in the reordered model. Similar results were observed for the Mtbdd and Hybrid engines.

Quantiles in feature-oriented systems. The concept of features provides an elegant way to specify families of systems: Features encapsulate additional functionalities that influence the behaviors of a given base system. The members of the family usually share a lot of common behaviors, such that symbolic representations may yield significantly smaller representations of the family (see, e.g., [54,16,11]). Such a smaller representation is also beneficial for analysis, i.e., using a family-based analysis approach, where the family is represented in one single model and the analysis of all systems contained is performed in a single run.

In previous work [16], we carried out experiments on an energy-aware server product line eServer, illustrating the benefits of symbolic representations in probabilistic product-line verification. There, we also showed that variable orderings have a crucial impact on family-based analysis performance. However, due to the lack of a symbolic quantile implementation, an energy-utility analysis of eServer had to be postponed as future work.

In Table 5, we summarize statistics for the computation of quantiles on two instances of eServer, becoming possible due to our symbolic implementation. We computed the minimal amount of energy required to guarantee in 95% of the cases a certain percentage of the time without any package drop. The table shows the impact of our four reorder mechanisms on the model size and the quantile computation time. We only included the results for the Mtbdd engine, as the other engines struggled with the size of the model and reached a timeout after one day. Within all computations, 1476 quantile iterations were required. Interestingly, although the model presented in [16] already used heuristics to find good initial variable orderings, the fully automatic reorder mechanisms presented here allow for a further significant reduction of the model size and a speedup of the analyses[7].

[7] Comparing with the experiments for eServer presented in [31], our Mtbdd implementation for quantile computations features an improved handling of the zero-reward fragment of the MDP, which more than halved the query times.

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

Table 5. Quantile computations for eServer, with statistics for the reachable state space and MTBDD size of the transition matrix, reduction, time for reordering, time building the model, and computing the quantile query (in seconds).

		States	MTBDD nodes	Reduction in %	t_{reorder}	MTBDD t_{build}	MTBDD t_{query}
Instance 1	original	145 984 112	63 619	-	-	8.0s	393.9s
	reordered	"	39 716	37.6	2.7s	5.9s	297.5s
	with explode	"	33 858	46.8	6.0s	6.8s	291.2s
	with globalize	"	35 769	43.8	2.2s	4.6s	286.6s
	with exp.+glob.	"	29 339	53.9	2.2s	6.1s	270.7s
Instance 2	original	441 704 832	139 136	-	-	21.6s	1 812.0s
	reordered	"	72 145	48.1	62.4s	12.7s	2 552.6s
	with explode	"	65 798	52.7	24.2s	13.2s	2 149.9s
	with globalize	"	42 857	69.2	5.0s	8.3s	2 617.8s
	with exp.+glob.	"	35 718	74.3	6.2s	9.0s	2 028.7s

3.5 Computing quantiles in CTMCs

Quantiles for continuous-time Markov chains can be defined in a similar way as for discrete Markov chains. However, in the case of CTMCs we have to consider trajectories rather than paths, in which case the quantile can be a real number (rather than an integer) and min and max in the definitions need to be replaced with inf and sup, respectively. As an example, the quantile

$$\inf\{\, t \in \mathbb{R} \,:\, \Pr_s(\, \Diamond^{\leqslant t} \Phi \,) \geqslant p \,\}$$

asks for the "smallest" time-bound t such that the probability of reaching states satisfying Φ within the given time is at least p starting from some fixed state s.

We present here an implementation of the simple approximation scheme for quantiles proposed in [5]. As the basic building block, we rely on the existing implementation in PRISM for computing probabilities in CTMCs for time-bounded reachability with a concrete time bound t. As a first step, we check that the quantile is finite using a precomputation. Then, an *exponential search* is used to determine the smallest $i \in \mathbb{N}$ such that $\Pr_s(\, \Diamond^{\leqslant 2^i} \Phi \,) \geqslant p$. If $i \geqslant 1$, then we perform a *binary search* to determine some value $t \in [2^{i-1}, 2^i]$ such that $\Pr_s(\, \Diamond^{\leqslant t - \frac{\varepsilon}{2}} \Phi \,) < p$ and $\Pr_s(\, \Diamond^{\leqslant t + \frac{\varepsilon}{2}} \Phi \,) \geqslant p$ for some user-defined $\varepsilon > 0$. Then, t is indeed an ε-approximation of the quantile $\inf\{\, t \in \mathbb{R} \,:\, \Pr_s(\, \Diamond^{\leqslant t} \Phi \,) \geqslant p \,\}$. In the case where the exponential search succeeds immediately with $i = 0$, we perform a similar binary search in the interval $[0, 1]$.

Using PRISM's realization of the uniformization approach to compute approximations for time-bounded reachability probabilities in CTMCs, we have implemented this scheme in both the Explicit and the symbolic engines, supporting quantiles with upper or lower time bounds on simple path formulas. PRISM currently does not support the computation of reward-bounded reachability probabilities in CTMCs, but once support is added (e.g., using the duality between reward- and time-bounds [6]), our implementation can be easily adapted to handle quantiles for reward-bounded reachability properties in CTMC with positive rewards.

To give a brief example of the performance of this scheme, we consider here an instance of the "tandem" case study from the PRISM benchmark suite, with parameter c set to 10. To obtain an ε-approximation of the quantile value for a precision of $\varepsilon = 10^{-6}$ and the quantile

$$\inf\{\, t \in \mathbb{R} \,:\, \Pr_s(\, \Diamond^{\leqslant t} \text{"network becomes full"} \,) \geqslant 0.1 \,\},$$

the exponential search requires 14 threshold computations for finding the upper bound 4096, which is then refined by the binary search using 31 additional threshold computations to obtain the result $t = 2954.281344$. The overall computation time was 56.079 s (Explicit), 4001.817 s (MTBDD), 33.609 s (Hybrid) and 17.625 s (Sparse). To provide a sense for the computation time for a single value of t, the computation of the probability for the result of the quantile computation, i.e.,

$$\Pr_s(\, \Diamond^{\leqslant t} \text{"network becomes full"} \,) \text{ for } t = 2954.281344,$$

takes 1.626 s (Explicit), 118.819 s (MTBDD), 0.993 s (Hybrid) and 0.528 s (Sparse).

The computation of time-bounded probabilities for CTMCs in PRISM relies on the computation of a truncated infinite sum using the uniformized DTMC (see, e.g., [7,47]), where the number of summands is chosen depending on a user-supplied value for the desired precision. In general, it can be expected that computations with a coarser precision require fewer iterations in the computation of the sum. We have thus experimented with an approach that gradually refines the precision for the probability computations: We start with a coarse precision, e.g., allowing imprecision of 0.1. As long as we get definitive results for the probability threshold computations when taking the imprecision of the result value into account, we remain at the same precision. If we get an inconclusive result, i.e., the threshold p lies inside the possible values when taking the imprecision into account, we refine the precision, e.g., by dividing by ten. Our experiments indicate that the potential savings in runtime due to the coarser precision tend to be negated by the required additional probability computations in the refinement step when encountering an inclusive result.

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

A reason for this is that the already carried out computations for the same time bound but with the coarser precision can not be reused. As a consequence, we suggest the integration of the threshold check into the computation of the time-bounded reachability probabilities in PRISM using an adaptive precision as future work: During the computation of the sum, a periodic check against the threshold could be carried out, taking the achieved bound on the precision into account and returning early if it can be conclusively determined that the threshold is satisfied or can never be satisfied.

4 Minimal weak deterministic Büchi automata and expected accumulated reward for co-safe LTL formulas

For handling complex path formulas given in Linear Temporal Logic (LTL), the standard approach in probabilistic model checking (e.g., [56]) is to perform a synchronous product of the model with a deterministic ω-automaton (see, e.g., [23] for an overview) obtained from the path formula and perform the analysis in this product. This approach is implemented in PRISM as well, relying on a Java reimplementation of the LTL2DSTAR tool [30]. Additionally, PRISM supports the use of external LTL to automata translators, which allows the use of state-of-the-art translators such as RABINIZER [19,32]. As the state space of the product model consists of the (reachable part) of the state spaces of the original model and of the automaton, the number of states of the automaton can have a decisive impact on model checking time and even the feasibility of analysis. In particular, as the numerical analyses carried out in the product model tend to be costly, even small reductions in automaton size can have a large impact.

Inspired by a similar implementation in the LTL to automaton translator of SPOT [17] of the translation proposed in [14], we have implemented a specialized treatment for formulas in the syntactic obligation fragment of LTL, relying on the construction of a weak deterministic Büchi automaton (WDBA) with a subsequent minimization step. Obligation formulas are, roughly speaking, boolean combinations of safety ("nothing bad happens") and co-safety ("eventually something good happens") formulas, for details see, e.g., [14]. The syntactic obligation fragment of LTL are then all formulas where it can be easily determined, by a syntactic inspection, that they belong to the obligation fragment and can thus be represented by a WDBA. A WDBA is a deterministic Büchi automaton such that, for every strongly connected component (SCC) of the automaton, either all states in the SCC are accepting states or all states in the SCC are non-accepting states. The algorithm of [14] relies on the standard powerset construction on the nondeterministic Büchi automaton with a subsequent analysis of the SCCs to determine whether all states in a SCC should be made accepting or not accepting. As described in [41], WDBA can be minimized to yield a minimal WDBA, which has at most as many states as any other ω-regular automaton for the same language (assuming that acceptance is defined on the states). The minimization procedure relies on two steps: At first, the WDBA is transformed into a normal form, and then standard minimization for deterministic finite automata (DFA) is applied. The normalization consists essentially of deciding whether states that are not contained in a cycle should be accepting or not. Since these states cannot be visited infinitely often, the normalization does not affect the accepted language of an WDBA. As DFA minimization, we use Brzozowski's minimization algorithm [9].

We have tested our implementation against automata generated by other tools using SPOT's ltlcross testing tool for equivalence. From the 94 benchmark formulas from [20,53,18], 44 formulas were detected to be syntactic obligation formulas by our implementation. For these 44 formulas, the constructed minimal automaton was smaller than the one generated by PRISM's LTL2DSTAR implementation in 30 cases, and was smaller than the one produced by RABINIZER in 31 cases. Comparing the cases with the largest reduction, the minimal WDBA had 6 states versus 17 states for the automaton using PRISM's default translation and the minimal WDBA had 4 states versus 30 states for the automaton obtained from RABINIZER. A more detailed experimental evaluation, including a comparison with probabilistic model checking approaches that try to avoid full determinization such as using unambiguous Büchi automata [8] or limit-deterministic automata [51], remains future work.

Expected accumulated reward for co-safe LTL. [39] addresses the problem of computing extremal expected accumulated rewards on a finite horizon which is given in terms of a co-safe LTL formula. We provide here an informal description. For details see [39]. The co-safety (sometimes also called guarantee) fragment of LTL is characterized by the existence of *good prefixes*. Formally, an LTL formula φ is said to be co-safe iff each (infinite) word in the language of φ has a finite prefix σ such that all infinite extensions of σ satisfy φ. Each such word σ is called a good prefix for φ. Intuitively, once a good prefix has been consumed, the formula can not be falsified anymore. It is known that one can construct a deterministic finite automaton (DFA) recognizing exactly the minimal good prefixes [34]. For a path in a DTMC, the accumulated reward for a co-safe LTL formula φ is then defined as the accumulated reward where reward accumulation happens only as long as the corresponding run in the DFA for φ has not yet reached an accepting state, i.e., where the path prefix is not yet a good prefix.

For the algorithmic treatment, the crucial step is the construction of the DFA. The additional steps rely on standard transformations to an automaton product and reduction to the problem of expected accumulated reward until an accepting state of the DFA is reached

Final edited form was published in "International Journal on Software Tools for Technology Transfer. 2018, 20 (2), S. 179–194. ISSN 1433-2787.
https://doi.org/10.1007/s10009-017-0456-3

in the product, i.e., an (extremal) expected reachability reward computation.

As the co-safe fragment of LTL is a subset of the obligation fragment, we can use the implementation described above to obtain such a DFA. Formally, an LTL formula is syntactically co-safe if, converted to positive normal form (negation only on the atomic propositions), it only uses the temporal operators $\Diamond$ (eventually), $\mathcal{U}$ (until) and $\bigcirc$ (next step). As the WDBA we construct is minimal, we can easily identify the set of acceptance states in a suitable DFA on the same automaton structure by a simple graph analysis.

Once the DFA is obtained, it can then be used by PRISM to compute (extremal) accumulated expected rewards for syntactically co-safe LTL formulas, for DTMCs, MDPs and CTMCs in all four engines. When translating the formula to a DFA in this context, it is crucial that PRISM's model-based optimizations in automata constructions (which are sound in the context of probability computations) are not used. For example, PRISM usually checks whether the satisfaction set for an atomic proposition (AP) matches the full state space of the model and simplifies the LTL formula by replacing the AP with `true`, e.g., simplifying $\varphi = \bigcirc\bigcirc a$ to $\varphi' = \bigcirc\bigcirc \texttt{true} \equiv \texttt{true}$, where φ results in reward accumulation for two steps while the simplified formula φ' would erroneously result in no reward accumulation due to the modified set of good prefixes. A similar issue arises for the identification of APs with the same satisfaction set. We thus disable these kinds of optimizations in this context.

5 Conclusion

This article presented several enhancements for PRISM. In particular, we have demonstrated a significant reduction in both time- and memory consumption gained through our implementation of automated variable reordering and its support for more fine-grained user influence on the order. Our implementation of quantile computations for reward-bounded reachability properties in MDPs for the MTBDD, HYBRID and SPARSE engines complements the implementation in the EXPLICIT engine and allows computations where the memory requirements of an explicit representation become too large, such as in the ESERVER product-line cases study. Additionally, we have implemented support to compute approximations of quantiles in CTMCs for time-bounded properties. Our translation to weak deterministic Büchi automata for the syntactic obligation fragment of LTL guarantees the minimal size of the obtained deterministic automata and yields the base for obtaining deterministic finite automata to be used in the computation of (extremal) accumulated rewards for co-safe LTL path formulas.

Future work. In the area of automatic variable reordering, it would be interesting to support more structured reordering: Often, models are obtained from templates with parameterization, e.g., specifying the number of copies of certain modules in the model. By swapping the variables of all copies simultaneously, it might be possible to discover good initial variable orders from instances with few copies and apply these to instances with more copies. This approach would also be interesting when the aim is to apply symmetry reduction [36,15], as all copies would remain symmetrical. While our syntactic transformations provide very fine-grained reordering for the state variables, it would be interesting to have the option of adding back some restrictions or hints for the reordering by annotating the variable declarations in the PRISM model. This would allow to state preferences which variable should be kept together, etc.

An orthogonal approach to obtain a good variable ordering are static variable ordering approaches, i.e., where heuristics are used to extract a good initial variable ordering from the structure of the model description. These approaches can be quite successful, see, e.g., [50,42,44], and it would be interesting to have automatic support in PRISM. Additionally, our extensions of the PRISM model language make it possible to easily specify and use a particular variable ordering at a much finer-grained level of control than possible before. This should also be beneficial for static ordering approaches. In addition, our benchmark results serve as an indication that it would be worthwhile to attempt a refactoring of PRISM to remove the variable order assumptions and add support for asynchronous reordering.

References

1. S. Andova, H. Hermanns, and J.-P. Katoen. Discrete-time rewards model-checked. In *Proc. Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*, pages 88–104. Springer, 2003.
2. R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.
3. C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Z. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In *Proc. International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*, pages 430–440, 1997.
4. C. Baier, M. Daum, C. Dubslaff, J. Klein, and S. Klüppelholz. Energy-utility quantiles. In *Proc. NASA Formal Methods (NFM'14)*, volume 8430 of *LNCS*, pages 285–299. Springer, 2014.
5. C. Baier, C. Dubslaff, J. Klein, S. Klüppelholz, and S. Wunderlich. Probabilistic model checking for energy-utility analysis. In *Horizons of the Mind. A Tribute to Prakash Panangaden - Essays Dedicated to Prakash Panangaden on the Occasion of His 60th Birthday*, volume 8464 of *LNCS*, pages 96–123. Springer, 2014.
6. C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. On the logical characterisation of performability properties. In *Proc. International Colloquium on Automata,*

Languages and Programming (ICALP'00), volume 1853 of *LNCS*, pages 780–792. Springer, 2000.

7. C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. Software Eng.*, 29(6):524–541, 2003.
8. C. Baier, S. Kiefer, J. Klein, S. Klüppelholz, D. Müller, and J. Worrell. Markov chains and unambiguous Büchi automata. In *Proc. Computer Aided Verification (CAV'16), Part I*, volume 9779 of *LNCS*, pages 23–42. Springer, 2016.
9. J.A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical Theory of Automata*, 12:529–561, 1963.
10. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
11. P. Chrszon, C. Dubslaff, S. Klüppelholz, and C. Baier. Family-based modeling and analysis for probabilistic systems - Featuring ProFeat. In *Proc. Fundamental Approaches to Software Engineering (FASE'16)*, volume 9633 of *LNCS*, pages 287–304. Springer, 2016.
12. G. Ciardo, A. S. Miner, and M. Wan. Advanced features in SMART: the stochastic model checking analyzer for reliability and timing. *SIGMETRICS Performance Evaluation Review*, 36(4):58–63, 2009.
13. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proc. Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.
14. C. Dax, J. Eisinger, and F. Klaedtke. Mechanizing the powerset construction for restricted classes of ω-automata. In *Proc. Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *LNCS*, pages 223–236. Springer, 2007.
15. A. F. Donaldson, A. Miller, and D. Parker. Language-level symmetry reduction for probabilistic model checking. In *Proc. Quantitative Evaluation of Systems (QEST'09)*, pages 289–298. IEEE, 2009.
16. C. Dubslaff, C. Baier, and S. Klüppelholz. Probabilistic model checking for feature-oriented systems. *Transactions on Aspect-Oriented Software Development*, 12:180–220, 2015.
17. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proc. Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *LNCS*, pages 122–129. Springer, 2016.
18. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. International Conference on Software Engineering (ICSE'99)*, pages 411–420. ACM, 1999.
19. J. Esparza and J. Kretínský. From LTL to deterministic automata: A Safraless compositional approach. In *Proc. Computer Aided Verification (CAV'14)*, volume 8559 of *LNCS*, pages 192–208. Springer, 2014.
20. K. Etessami and G. Holzmann. Optimizing Büchi automata. In *Proc. International Conference on Concurrency Theory (CONCUR'00)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167, 2000.
21. V. Forejt, M. Z. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In *Proc. School on Formal Methods for the Design of Computer, Communication and Software Systems, Formal Methods for Eternal Networked Software Systems (SFM'11)*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.
22. M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.
23. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.
24. C. Haase and S. Kiefer. The odds of staying on budget. In *Proc. Automata, Languages, and Programming (ICALP'15)*, volume 9135 of *LNCS*, pages 234–246. Springer, 2015.
25. G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(12):1479–1493, 1996.
26. S. Haddad and B. Monmege. Reachability in MDPs: Refining convergence of value iteration. In *Proc. International Workshop on Reachability Problems (RP'14)*, volume 8762 of *LNCS*, pages 125–137. Springer, 2014.
27. V. Hartonas-Garmhausen, S. V. A. Campos, and E. M. Clarke. ProbVerus: probabilistic symbolic model checking. In *Proc. Formal Methods for Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 96–110, 1999.
28. M. Heiner, C. Rohr, M. Schwarick, and A. A. Tovchigrechko. MARCIE's secrets of efficient model checking. *Transactions on Petri Nets and Other Models of Concurrency*, 11:286–296, 2016.
29. H. Hermanns, M. Z. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming*, 56(1-2):23–67, 2003.
30. J. Klein and C. Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theoretical Computer Science*, 363(2):182–195, 2006.
31. J. Klein, C. Baier, P. Chrszon, M. Daum, C. Dubslaff, S. Klüppelholz, S. Märcker, and D. Müller. Advances in symbolic probabilistic model checking with PRISM. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*, volume 9636 of *LNCS*, pages 349–366. Springer, 2016.
32. Z. Komárková and J. Kretínský. Rabinizer 3: Safraless translation of LTL to small deterministic automata. In *Proc. Automated Technology for Verification and Analysis (ATVA'14)*, volume 8837 of *LNCS*, pages 235–241. Springer, 2014.
33. M. Kuntz and M. Siegle. CASPA: symbolic model checking of stochastic systems. In *Proc. Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB'06)*, pages 465–468. VDE Verlag, 2006.
34. O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
35. M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: a hy-

brid approach. *Software Tools for Technology Transfer*, 6(2):128–142, 2004.
36. M. Z. Kwiatkowska, G. Norman, and D. Parker. Symmetry reduction for probabilistic model checking. In *Proc. Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 234–248. Springer, 2006.
37. M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
38. M. Z. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *Proc. Quantitative Evaluation of Systems (QEST'12)*, pages 203–204. IEEE, 2012. Website: `https://github.com/prismmodelchecker/prism-benchmarks/`.
39. B. Lacerda, D. Parker, and N. Hawes. Optimal and dynamic planning for Markov decision processes with co-safe LTL specifications. In *Proc. Conference on Intelligent Robots and Systems (IROS'14)*, pages 1511–1516. IEEE, 2014.
40. K. Lampka. *A symbolic approach to the state graph based analysis of high-level Markov reward models*. PhD thesis, Universität Erlangen-Nürnberg, 2007.
41. C. Löding. Efficient minimization of deterministic weak omega-automata. *Information Processing Letters*, 79(3):105–109, 2001.
42. V. Maisonneuve. Automatic heuristic-based generation of MTBDD variable orderings for PRISM models. Internship report, ENS Cachan & Oxford University, 2009. `http://www.prismmodelchecker.org/papers/vivien-bdds-report.pdf`.
43. K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
44. J. Meijer and J. van de Pol. Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis. In *Proc. NASA Formal Methods (NFM'16)*, volume 9690 of *LNCS*, pages 255–271. Springer, 2016.
45. A. S. Miner and D. Parker. Symbolic representations and analysis of large probabilistic systems. In *Validation of Stochastic Systems - A Guide to Current Research*, volume 2925 of *LNCS*, pages 296–338, 2004.
46. S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proc. Computer-Aided Design (ICCAD'95)*, pages 74–77. IEEE, 1995.
47. D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.
48. PRISM model checker. Website: `http://www.prismmodelchecker.org/`.
49. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. Computer-Aided Design (ICCAD'93)*, pages 42–47. IEEE, 1993.
50. M. Schwarick and M. Heiner. CSL model checking of biochemical networks with interval decision diagrams. In *Proc. Computational Methods in Systems Biology (CMSB'09)*, volume 5688 of *LNCS*, pages 296–312. Springer, 2009.
51. S. Sickert, J. Esparza, S. Jaax, and J. Kretínský. Limit-deterministic büchi automata for linear temporal logic. In *Proc. Computer Aided Verification (CAV'16), Part II*, volume 9780 of *LNCS*, pages 312–332. Springer, 2016.
52. F. Somenzi. CUDD: Colorado University decision diagram package. Website: `http://vlsi.colorado.edu/~fabio/CUDD/`.
53. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Proc. Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.
54. T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1s):6:1–6:45, 2014.
55. M. Ummels and C. Baier. Computing quantiles in Markov reward models. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS'13)*, volume 7794 of *LNCS*, pages 353–368. Springer, 2013.
56. M. Y. Vardi. Probabilistic linear-time model checking: An overview of the automata-theoretic approach. In *Proc. AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 265–276. Springer, 1999.