Directed and Heuristic Counterexample Generation for Probabilistic Model Checking - A Comparative Evaluation

Husain Aljazzar Matthias Kuntz Florian Leitner-Fischer Stefan Leue
Computer and Information Science
University of Konstanz, Germany

ABSTRACT

The generation of counterexamples for probabilistic model checking has been an area of active research over the past five years. Tangible outcome of this research are novel directed and heuristic algorithms for efficient generation of probabilistic counterexamples, such as K^* and XBF. In this paper we present an empirical evaluation of the efficiency of these algorithms and the well-known Eppstein's algorithm. We will also evaluate the effect of optimisations applied to Eppstein, K^* and XBF. Additionally, we will show, how information produced during model checking can be used to guide the search for counterexamples. This is a first step towards automatically generating heuristic functions. The experimental evaluation of the various algorithms is done by applying them to one case study, knwon from the literature on probabilistic model checking and one case study taken from the automotive industry.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Software/Program Verification; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and SearchHeuristic methods

General Terms

Algorithms, Reliability, Verification

1. INTRODUCTION

A characteristic of embedded systems is that they consist of hardware and software components, where the software component controls the behavior of the hardware component. In embedded automotive systems, for instance, the software is often attached to electronic control units, which then control the behavior of hardware components such as actuators or sensors. A typical example for such a system is an airbag control system, which we have described in more detail in [4].

QUOVADIS '10, May 3 2010, Cape Town, South Africa

Dependability analysis for this class of embedded systems, such as for instance probabilitstic Failure Mode and Effect Analysis (pFMEA) [17, 4], is hence inevitably also software dependability analysis. It has been shown that stochastic model checking [13] is an effective tool to support system dependability analysis, including pFMEA. While standard stochastic model checking tools such as PRISM [20] and MRMC [22] can determine whether or not a given stochastic model satisfies a property specification given by a stochastic temporal logic formula, they cannot provide diagnostic information in case the property is not satisfied. Such diagnostic information, also referred to as counterexample, is however crucial to determine which sequences of events carry a large amount to probability mass that lead to a property violation, such as reaching an undesired system state. Counterexample generation is a quintessential part of probabilistic systems analysis. It forms the basis of counterexample-guided abstraction refinement (CEGAR) in stochastic model checking [19]. In [6] we demonstrate how to use visualization techniques in order to support the human user in analyzing counterexamples to localize causal factors of errors, which also greatly enhanced our pFMEA work [4].

1.1 Related Work

In percursory work [5, 3, 9, 8, 11, 16, 18, 14, 25] we and others have addressed this shortcoming by devising algorithms that compute counterexamples for various stochastic models and different stochastic temporal logics.

Some approaches consider counterexamples to be a *set of computation paths* that need to be enumerated until a specified probability threshold is exceeded, in which case this set is a witness for the property violation. The algorithmic problem to be solved here is a k-shortest paths Problem (KSP), where the shortness of the paths to be collected is interpreted as the probability mass carried by the selected execution paths. This approach has been chosen in [18] to compute counterexamples for Deterministic Time Markov Chains (DTMCs) and for Continuous Time Markov Chains (CTMCs). We chose a similar approach to compute counterexamples for Markov Decision Processes (MDPs) [3, 9]. Another group of approaches considers a counterexample to be given by a *subgraph of the stochastic model*, referred to as diagnostic subgraph, and not by sets of execution paths. We have chosen this approach in [8] to compute counterexamples for DTMCs and CTMCs. The path enumerating approaches rely on the use of variants Eppstein's algorithm [15] for solving the KSP. The modification of Eppstein's algorithm proposed in [21] performs better in practice, and when we men-

tion implementations of Eppstein in this paper we mean to refer to the implementation according to [21]. We have proposed K* as an alternative KSP algorithm in [7, 3, 10] which is an on-the-fly algorithm. This means that it is not necessary to store the complete MDP model in main memory, but that the parts to be explored will be generated on demand. Also, K* is heuristics guided. Both enhancements give K* a performance advantage over Eppstein when the models to be analyzed are very large, as we have shown in [9].

The subgraph computation approach that we have proposed relies on the use of an extended Best-First algorithm (XBF) which is an extension of the well-known Best-First search strategy (BF) [24]. XBF explores the state space of the model on-the-fly searching for diagnostic paths. This approach has the advantage of giving a much more compact representation of the counterexample than the path enumeration approach, since counterexamples in stochastic model checking can contain millions of paths, in the worst case. Due to the nature of MDPs the subgraph approach cannot be used for the generation of counterexamples for MDPs. Notice that the path enumeration approach allows us to directly compute the probability of every path in the counterexample, and hence the counterexample in total. For the subgraph based approach it is necessary to use a probabilitstic model checker to determine the counterexample probability.

Other approaches use regular expressions [14] to represent counterexamples, a similar approach was taken in [25] where counterexamples were interpreted as pairs of words.

1.2 Contributions

In this paper we present the idea of the X-optimization technique, which is to add all counterexample traces found by a KSP based approach into a diagnostic subgraph. Using this technique we gain the advantage of the diagnostic subgraph approach as yielding a compact representation of counterexamples. We refer to the algorithms K* and Eppstein combined with X-optimization as XK* and XEppstein.

A further idea presented in this paper is that data which was produced during the model checking process, can later on be used as a heuristic to guide the counterexample search.

The objective of this paper is to present a comprehensive experimental evaluation of the above described algorithmic approaches by applying them to a number of case studies well known from the literature. In particular, the contributions of this paper are:

- We illustrate the advantage of using K* as an on-the-fly algorithm compared to Eppstein.
- We demonstrate the advantage of adding the X-optimization to KSP approaches by comparing XK* with K* and XEppstein with Eppstein.
- We compare KSP algorithms, specially K* and XK*, with XBF.
- We show how information that is produced during the model checking process, can be used as heuristic to guide the search for an counterexample.

1.3 Structure of the Paper

The remainder of the paper is structured as follows: In Section 2 we describe the foundations of our counterexample generation approaches. The case study of an embedded controller system is presented in Sec. 3. The results of our experimental evaluation of the different approaches are presented in 4. We conclude in Sec. 5.

2. COUNTEREXAMPLE GENERATION

2.1 Notion of Counterexamples

For the sake of simplicity, we consider here upper bounded properties which require the probability of a property offending behavior not to exceed a certain upper probability bound. In CSL [2, 13] such properties can be expressed by formulas of the form $P_{\leq p}(\varphi)$, where φ is path formula specifying undesired behavior of the the system. Any path which starts at the initial state of the system and which satisfies φ is called a *diagnostic path*. A counterexample for an upper bounded property is a set X of diagnostic paths such that the accumulated probability of X violates the probability constraint $\leq p$.

A good counterexample is easy to comprehend and provides the needed diagnostic information, hence the quality of the counterexample is dependent on the size of the set of diagnostic paths, since a small number of paths is easier to comprehend then a large number.

2.2 Generation of Counterexamples

In the following subsections, we discuss the algorithms used for the generation of the counterexamples.

2.2.1 eXtended Best-First

eXtended Best-First (XBF) is an extension of the well-known Best-First search strategy (BF) [24]. XBF explores the state space of the model on-the-fly searching for diagnostic paths. It does not explicitly compute the set X of diagnostic paths forming the counterexample. Instead it computes a sub-graph of the state space of the model which covers this set, called *diagnostic sub-graph*. The diagnostic sub-graph is selected incrementally. Once the selected diagnostic sub-graph covers enough diagnostic paths so that the accumulated probability exceeds the given upper probability bound, XBF terminates and produces the diagnostic sub-graph as a counterexample. A more comprehensive description on how XBF can be used to generate counterexamples can be found in [3].

2.2.2 K-Shortest-Paths

A k-shortest-paths (KSP) based approach considers counterexamples to be a set of computation paths that need to be enumerated until a specified probability threshold is exceeded, in which case the property is violated.

The KSP based approaches have several advantages over an XBF approach, such as:

1. An XBF based approach includes diagnostic traces which are suboptimal or which indicate an insignificant probability among the selected diagnostic subgraph. On the other hand, a KSP approach allows more control over which diagnostic traces are added to the counterexample improving the quality of the counterexamples.

2. Some counterexample analysis could benefit from the individual listing of diagnostic traces.

Meanwhile, KSP based methods for counterexample generation have a big scalability drawback as shown in the experiments in Section 4. This drawback has mainly two reasons. The first reason is that classical KSP algorithms, such as Eppstein's algorithm, require that an exhaustive search is performed initially on the state transition graph in order to determine the shortest path tree. The use of K* allows us to overcome this problem. The other reason is that a KSP based approach has to individually deliver all diagnostic traces which are needed to produce the counterexample. This results in a performance bottleneck in KSP based methods since the number of diagnostic traces needed to form a counterexample can be enormous [18] whereas the counterexample provided by a XBF based approach includes all diagnostic traces induced by the selected diagnostic subgraph. We illustrate this effect using the following example.

EXAMPLE 1. *Let R be a diagnostic trace which includes one run of some cycle. If a KSP based approach finds R, then it adds the probability of R to the counterexample probability. However, an XBF based approach would add the probability of all diagnostic traces induced by the states and transitions of R. All diagnostic traces which run the cycle for an arbitrary number of times are included. XBF includes the total probability of these traces at once, while a KSP approach concerns itself with enumerating them one-by-one.*

2.2.3 X-Optimization for KSP

The *X-optimization* technique that we now describe mitigates this problem, although it does not completely solve it. The idea of this optimization technique is to add all diagnostic traces found by a KSP based approach into a diagnostic subgraph, which is then considered as the counterexample. The probability of the counterexample is computed using a stochastic model checker whenever the diagnostic subgraph grows by q per cent (in our experiments $q = 20\%$). Using this technique we gain the advantage of diagnostic subgraphs, as being a compact representation of counterexamples. We also avoid the drawback of XBF of adding diagnostic traces which indicate insignificant probability. We refer to the algorithms K* and Eppstein combined with X-optimization as XK* and XEppstein.

2.2.4 Heuristics for Counterexamples for MRMC

The *Markov Reward Model Checker* (MRMC) [22] is a model checker for DTMCs and CTMCs. MRMC operates completely on the explicit-state space unlike PRISM, which uses symbolic data structures. This makes it easier for our explicit-state search algorithms to benefit from the information which MRMC attaches to the states during the model checking. The search algorithm can use this information to guide the search. MRMC computes a probability vector $\bar{p}$ when it is used to check a property $\mathcal{P}_{\bowtie p}(\varphi)$ on a Markov chain $\mathcal{M}$. The vector $\bar{p}$ indicates the probability of all paths which start at s and satisfy φ for each state s. Formally, it holds that $\bar{p}(s) = Pr_s^{\mathcal{M}}(\varphi)$. This probability indicates the likelihood of reaching a target state starting from s. The vector $\bar{p}$ can be used to guide the search algorithm along the states which lead to target states with high probability. We can easily do that by using $\bar{p}$ as a heuristic function. We set $h = \bar{p}$ in the evaluation function of XZ. It is important to note that this heuristic will be available as a side product of the model checking step without additional computational cost. It is also produced automatically, without any human intervention.

3. CASE STUDIES

3.1 Embedded Control System

This case study models an embedded control system based on the one presented in [23]. The system consists of a main processor (**M**), an input processor (**I**), an output processor (**O**), 3 sensors and two actuators. The input processor I reads data from the sensors and forwards it to M. Based on this data, M sends instructions to the output processor O which controls both actuators according to the received instructions. This model, which we refer to as *Embedded*, is translated by PRISM into a CTMC $\mathcal{C}$ comprising 8,548 states and 36,041 transitions (model parameter MAX_COUNT = 8).

Various failure modes, such as the failure of I, M or O, or sensor or actuator failures, can lead to a shutdown of the system. We want to check, whether the probability that the system shuts down within one hour, does not exceed the probability p. One hour corresponds to 3,600 time units as we take one second as the basic time unit in our model. In CSL, this property reads as $\mathcal{P}_{\leq p}(true\ U^{\leq 3,600}\ down)$. For the initial state, PRISM computes the actual probability of this property to be 3.0726 E-4. This required approximately 0.3 seconds and 314.4 KB. We applied the various algorithms in order to generate counterexamples for the upper-bounded property $\mathcal{P}_{\leq 3.0726\,E-4}\ (true\ U^{\leq 3,600}\ down)$.

3.2 NVRAM Manager

This case study models the *non volatile random access memory manager (NVRAM Manager)* from the AUTOSAR automotive open system architecture [1]. The NVRAM module [12] provides the central memory management of AUTOSAR control units and ensures the data storage and maintenance of non volatile data according to the individual requirements in an automotive environment. An example for such a requirement is to handle concurrent access to the memory. A job scheduler with one queue for write- and one for read-jobs, is the central part of the NVRAM Manager. The application can add new jobs through application programming interface (API) calls. The job processing is controlled by a cyclic operating system task, which can be concurrent to the API calls of the application.

We require that the probability of a read overflow, that is an overflow of the queue that handles the read jobs, does not exceed p. In CSL, this property reads as $\mathcal{P}_{\leq p}(true\ U^{\leq 10}\ overflow\_read = 1)$. Using PRISM, we obtain 0.1694 as the actual probability bound for the initial state of the model. We applied the various algorithms in order to generate counterexamples for the upper-bounded property $\mathcal{P}_{\leq 0.1694}(true\ U^{\leq 10}\ overflow\_read = 1)$.

4. EXPERIMENTAL EVALUATION

We assess the different counterexample generation algorithms using the following criteria. In the experiments we first record runtime and memory consumption as a function of the counterexample probability mass. The higher the probability mass, the more search effort is needed, and this then gives us a good feeling for the computational efficiency of the algorithms. Notice that hence the computational effort depends on the probability bound given in the property

specification. We then assess the counterexample quality by relating increasing counterexample probability with the counterexample size. We measure the counterexample size by counting the numbers or states and transitions forming the diagnostic subgraph for XBF, and the unique numbers of states and transitions encountered in the counterexample path set for K*. The smaller the counterexample, the easier it is for a human to comprehend, and hence we assume that smaller counterexamples are of higher quality.

All experiments presented in this section were made on a machine with an Intel Core i5 CPU (2.67 GHz)[1] and 8 GB RAM. Java code was executed by the Java runtime environment JDK 1.6.

4.1 Counterexamples Generation KSP versus XBF for PRISM

4.1.1 Embedded Control System

We applied the algorithms XBF, K*, XK*, Eppstein and XEppstein to the *Embedded Controller System* model from Section 3 in order to generate counterexamples. We considered here the same property $\mathcal{P}_{\leq 3.0726\,E-4}\ (true\ U^{\leq 3{,}600}\ down)$ as described in Section 3. Remember that 3.0726 E-4 is actually the total probability $Pr(true\ U^{\leq 3{,}600}\ down)$. In the experiments we do not use heuristic estimates in either XBF nor in K* or XK*. In other words, the trivial heuristic function $h = 1$ is used.

The results of this experiment are illustrated in Figures 1, 2, 3 and 4. The X-axis indicates the probability of the counterexample which has been selected so far. Figures 1 and 2 show the runtime and memory effort needed by the various algorithms. Figures 3 and 4 show, for each algorithm, the size of portion of the state transition graph which has been explored to provide a counterexample carrying a certain amount of probability, as well as the size of the counterexample.

The performance gap between XBF and KSP algorithms is relatively small. The reason for this is that the property probability of this model is distributed across a relatively small number of diagnostic paths. Therefore, KSP algorithms were capable of selecting them quickly. We can observe that Eppstein incurs high overhead costs, in both runtime and memory consumption, before it delivers a counterexample. This is explained by the fact that Eppstein's algorithm has to explore the entire state transition graph before it can deliver the first diagnostic path. Note that this observation applies to both Eppstein and XEppstein. Meanwhile, K* and XK* start to provide counterexamples much earlier after exploring considerably smaller portions of the state transition graph. This, of course, reduces the memory consumption of both K* and XK*.

The X-optimization resulted both in better runtime behavior and a minor advantage in the memory consumption, that is, when comparing XK* with K* and XEppstein with Eppstein.

Although the KSP algorithms performs well for this model, the performance of XBF is still the best. It takes the least time and consumes the least memory. It also explores the smallest section of the state transition graph. The KSP algorithms, in particular XK* and XEppsteins, provide smaller counterexamples than XBF, as shown in Figure 4.

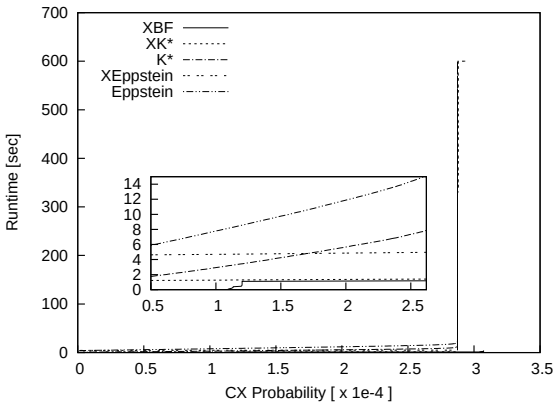

Figure 1: Counterexample generation for $\mathcal{P}_{\leq 3.0726\,E-4}$ $(true\ U^{\leq 3{,}600}\ down)$ on Embedded – runtime

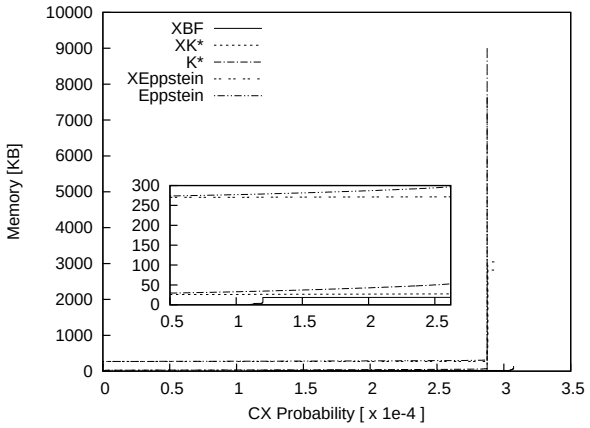

Figure 2: Counterexample generation for $\mathcal{P}_{\leq 3.0726\,E-4}$ $(true\ U^{\leq 3{,}600}\ down)$ on Embedded – memory consumption

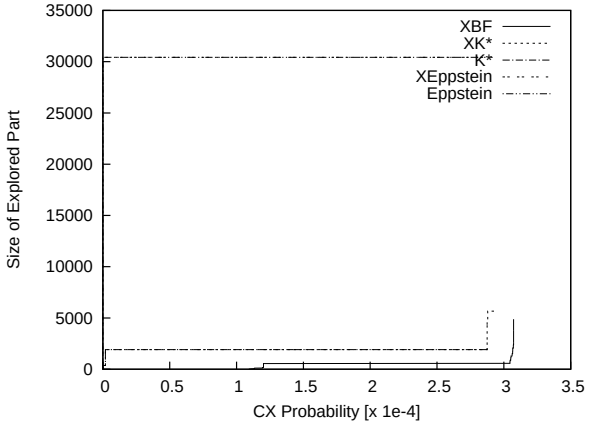

Figure 3: Counterexample generation for $\mathcal{P}_{\leq 3.0726\,E-4}$ $(true\ U^{\leq 3{,}600}\ down)$ on Embedded – exploration effort

4.1.2 NVRAM Manager

[1]The CPU had got 4 cores, but we do not exploit them because our implementation does not employ parallelism.

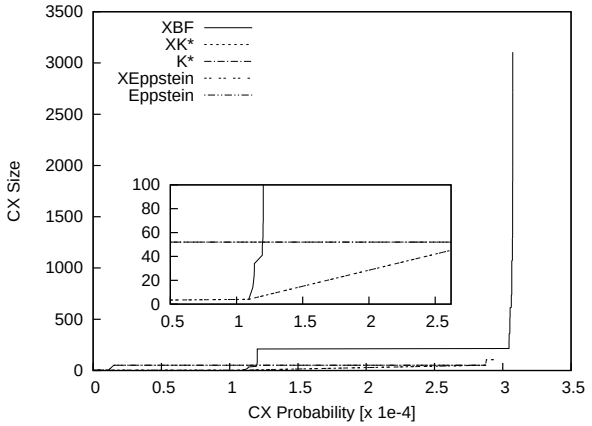

Figure 4: Counterexample generation for $\mathcal{P}_{\leq 3.0726\,E-4}$ $(true\ U^{\leq 3,600}\ down)$ on Embedded – counterexample quality

Here we consider the case study of the *NVRAM Manager* from Section 3.2. We use the algorithms XBF, K*, XK*, Eppstein and XEppstein to generate a counterexample for the property $P_{\leq 0.1694}(true\ U^{\leq 10}\ overflow\_read = 1)$. We recall that 0.1694 is the total probability $Pr(true\ U^{\leq 10}\ overflow\_read = 1)$.

The results of the counterexample generation are illustrated in Figures 5, 6, 7 and 8. The X-axis indicates the probability of the counterexample which has been selected so far. Figures 5 and 6 show the runtime and memory effort needed by the various algorithms. Figures 7 and 8 show, for each algorithm, the size of portion of the state transition graph which has been explored to provide a counterexample carrying a certain amount of probability and the size of the counterexample.

By comparing the curves of K* and Eppstein and their respective X-optimizations in Figures 5 and 6, the benefit of the X-optimization is obvious. The comparison also shows that there is no significant difference between K* and Eppstein, respectively XK* and XEppstein. This is due to the fact, that K* and XK* have to explore about the same amount of the state transition graph as Eppstein and XEppstein, which can be seen in Figure 7

Meanwhile, we see that XBF clearly outmatches K*. The reason for this is that K* was overloaded by the enormous number of diagnostic paths necessary to achieve a significant counterexample probability. Although we significantly improved the efficiency of both K* and Eppstein by using the X-optimization, XBF is still more efficient. However, the counterexamples provided by XK* or XEppstein have about the same size than those provided by XBF, as illustrated in Figure 8. This means that the counterexample quality is equal.

4.2 Counterexamples Generation KSP versus XBF for MRMC

4.2.1 Embedded Control System

We examine the Embedded model from Section 3 after converting it to MRMC. We applied XBF with and without heuristic (XBF and UXBF), K* with and without heuristic (K* and UK*), K* with and without X-Optimization (XK* and XUK*), and Eppstein with and without X-optimization

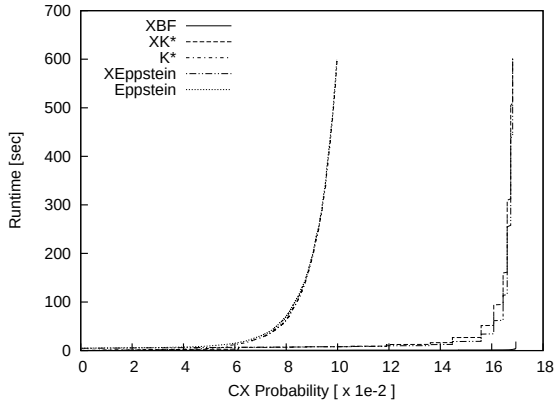

Figure 5: Counterexample generation for $P_{\leq 0.1694}(true\ U^{\leq 10}\ overflow\_read = 1)$ on NVRAM – runtime

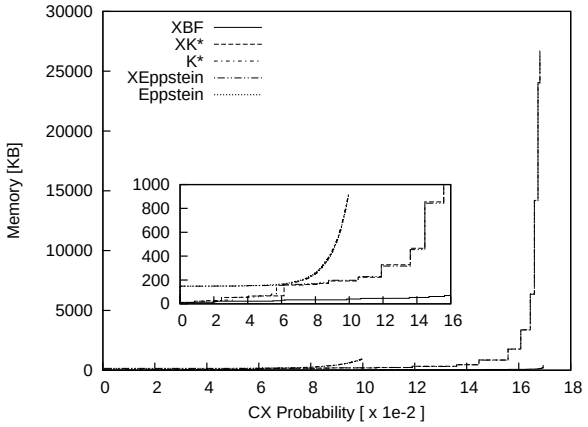

Figure 6: Counterexample generation for $P_{\leq 0.1694}(true\ U^{\leq 10}\ overflow\_read = 1)$ on NVRAM – memory consumption

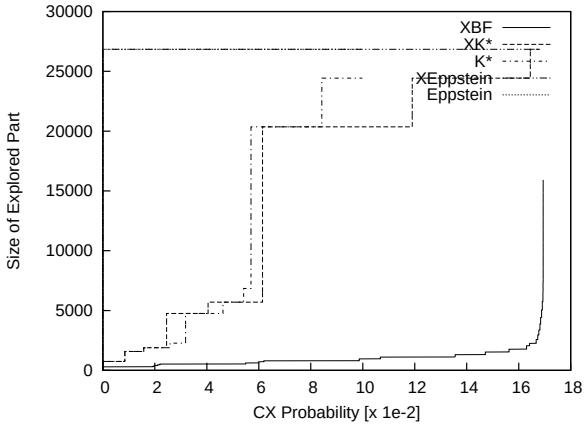

Figure 7: Counterexample generation for $P_{\leq 0.1694}(true\ U^{\leq 10}\ overflow\_read = 1)$ on NVRAM – exploration effort

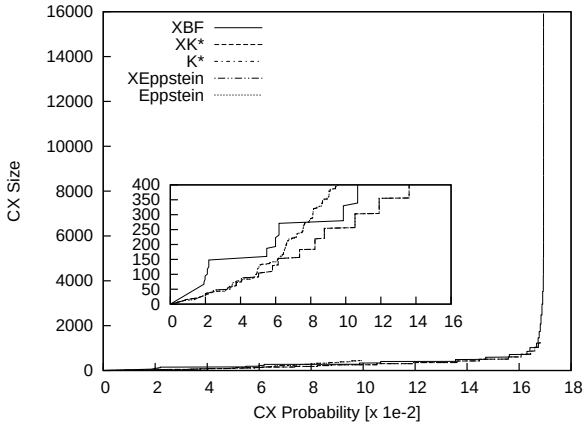

Figure 8: Counterexample generation for $P_{\leq 0.1694}(true\ U^{\leq 10}\ overflow\_read = 1)$ on NVRAM – exploration effort

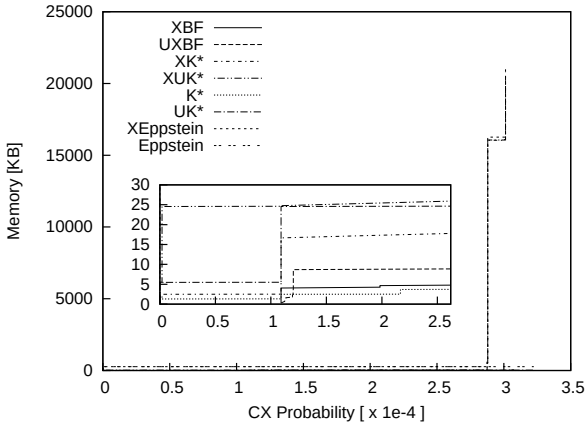

Figure 10: Counterexample generation for $\mathcal{P}_{\leq 3.0726\,E-4}\ (true\ U^{\leq 3,600}\ down)$ on Embedded (MRMC) – memory consumption

(XEppstein and Eppstein) in order to generate counterexamples.

We considered here the same property $\mathcal{P}_{\leq 3.0726\,E-4}\ (true\ U^{\leq 3,600}\ down)$ as described in Section 3 and that was used for the evaluation with Prism 4.1. Remember that 3.0726 E-4 is actually the total probability $Pr(true\ U^{\leq 3,600}\ down)$

The results are shown in Figures 9, 10, 11 and 12. The X-axis indicates the probability of the counterexample which has been selected so far. Figures 9 and 10 show the runtime and memory effort needed by the various algorithms. Figures 11 and 12 show, for each algorithm, the size of portion of the state transition graph which has been explored to provide a counterexample carrying a certain amount of probability and the size of the counterexample.

We observe that the heuristic caused a minor disadvantage in terms of runtime if it is used in combination with XK* (see Figure 9). The heuristic improved both the memory consumption and counterexample quality for K* and XBF as shown in Figures 10 and 12. XBF is still superior in terms of runtime and memory, but both Eppstein and K* provide counterexamples with a higher quality, that is a smaller size.

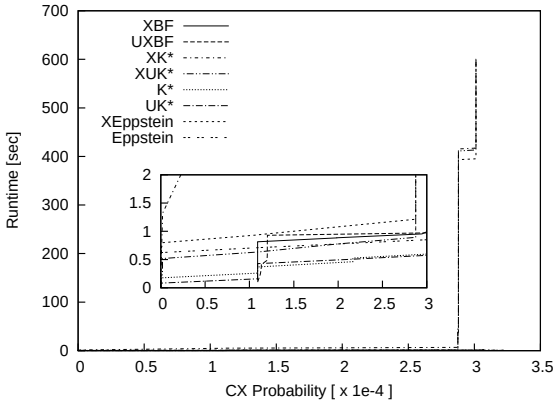

Figure 9: Counterexample generation for $\mathcal{P}_{\leq 3.0726\,E-4}\ (true\ U^{\leq 3,600}\ down)$ on Embedded (MRMC) – memory consumption

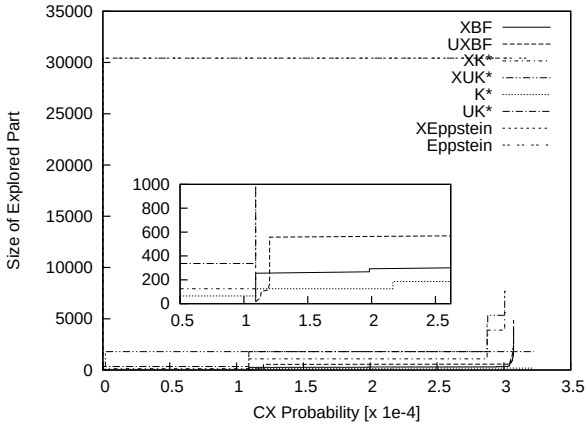

Figure 11: Counterexample generation for $\mathcal{P}_{\leq 3.0726\,E-4}\ (true\ U^{\leq 3,600}\ down)$ on Embedded (MRMC) – exploration effort

4.2.2 NVRAM Manager

We examine the NVRAM Manager model from Section 3.2 after converting it to MRMC. We applied XBF with and without (UXBF) heuristic, K* with and without (UK*) heuristic, K* with and without X-Optimization (XK* and XUK*), and Eppstein with and without X-optimization (XEppstein) in order to generate counterexamples.

The results are shown in Figures 13, 14, 15 and 16. The X-axis indicates the probability of the counterexample which has been selected so far. Figures 13 and 14 show the runtime and memory effort needed by the various algorithms. Figures 15 and 16 show, for each algorithm, the size of portion of the state transition graph which has been explored to provide a counterexample carrying a certain amount of probability and the size of the counterexample.

Figure 13 shows, that the use of the X-optimization improved the runtime of K* and Eppstein significantly. The use of the heuristic improved the memory consumption for K* and XK* as can be seen in Figure 14. The reason behind the observation that the heuristic does not lead to a no-

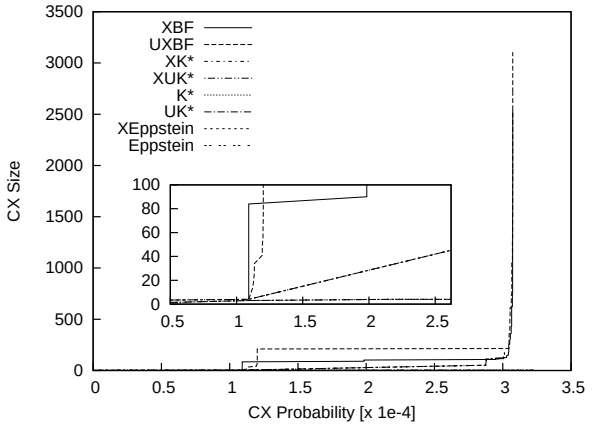

Figure 12: Counterexample generation for $\mathcal{P}_{\leq 3.0726\,E-4}$ $(true\ U^{\leq 3,600}\ down)$ on Embedded (MRMC) – counterexample quality

ticeable improvement in runtime is that K* was overloaded by the enormous number of diagnostic paths necessary to achieve a significant counterexample probability and therefore had to explore the same size of the state transition graph as Eppstein (see Fig. 15).

In terms of runtime and memory XBF is still superior to all other algorithms. In terms of quality, that is counterexample size (Fig. 16), the heuristic improves the quality of the counterexample generated by XBF. K* and Eppstein do deliver counterexamples with about the same quality.

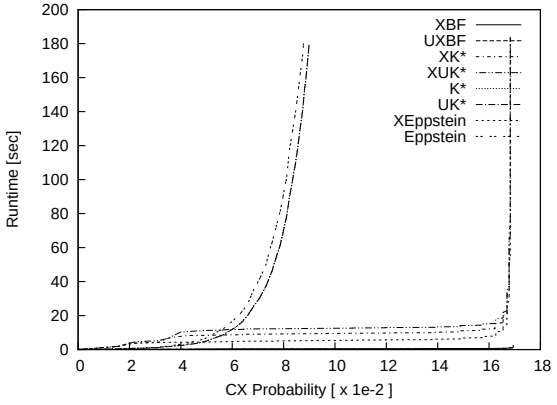

Figure 13: Counterexample generation for $P_{\leq 0.1694}(true\ U^{\leq 10}\ overflow\_read = 1)$ on NVRAM (MRMC) – runtime

5. CONCLUSION

Counterexample generation methods based on KSP search sometimes provide smaller counterexamples. However, conventional KSP algorithms, like Eppstein's algorithm, do not scale to models with realistic size. Our directed KSP algorithm K* represents an efficient and scalable alternative for these algorithms. Nevertheless, counterexample generation methods based on XBF are much more efficient and scalable than KSP based methods, including K*. XBF takes the least time and consumes the least memory.

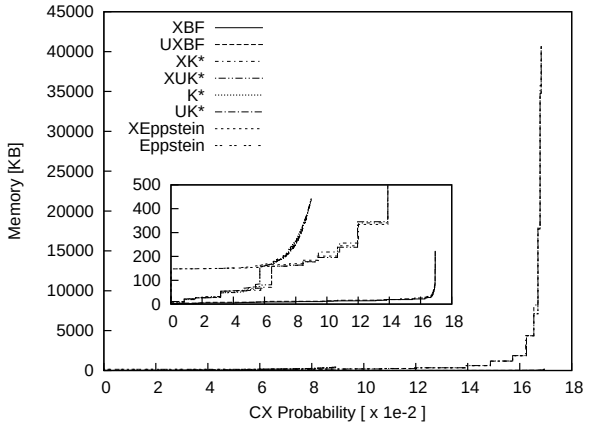

Figure 14: Counterexample generation for $P_{\leq 0.1694}(true\ U^{\leq 10}\ overflow\_read = 1)$ on NVRAM (MRMC) – memory consumption

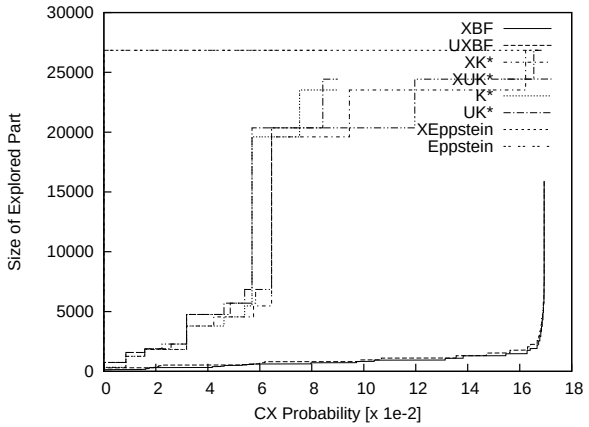

Figure 15: Counterexample generation for $P_{\leq 0.1694}(true\ U^{\leq 10}\ overflow\_read = 1)$ on NVRAM (MRMC) – exploration effort

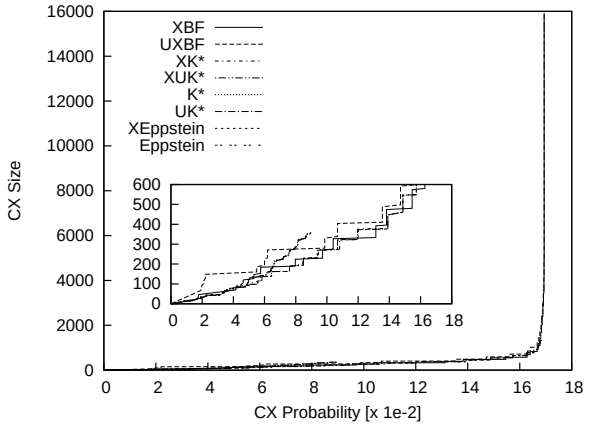

Figure 16: Counterexample generation for $P_{\leq 0.1694}(true\ U^{\leq 10}\ overflow\_read = 1)$ on NVRAM (MRMC) – counterexample quality

It was also shown that the MRMC heuristic can improve the runtime, memory consumption and counterexample quality for K* and XBF. It should be noted that the heuristic was generated automatically as a by-product of the model checking steps that anyway had to be performed during the counterexample computation. Whether the use of heuristics is beneficial depends to a large extent on the model and the informedness of the heuristic itself. At this point we are unable to propose any general guidelines regarding the use of specific heuristics.

Future research will address techniques for the automatic generation of counterexamples when no model checking steps need to be performed that one could take advantage from. We are also working towards extending the counterexample generation framework to stochastic models annotated by rewards.

Acknowledgements.

The idea of using $\bar{p}$ to compute a heuristic function arose during a personal discussion with Tinting Han and Joost-Pieter Katoen.

6. REFERENCES

[1] www.autosar.org, checked 16.12.2009.

[2] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time Markov chains. In Rajeev Alur and Thomas A. Henzinger, editors, *Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 269–276, New Brunswick, NJ, USA, 1996. Springer Verlag LNCS.

[3] H. Aljazzar. *Directed Diagnostics of System Dependability Models*. PhD thesis, University of Konstanz, 2009. http://kops.ub.uni-konstanz.de/volltexte/2009/9188/.

[4] H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, and S. Leue. Safety analysis of an airbag system using probabilistic fmea and probabilistic counterexamples. *Quantitative Evaluation of Systems, International Conference on*, 0:299–308, 2009.

[5] H. Aljazzar and S. Leue. Extended directed search for probabilistic timed reachability. In *FORMATS '06*, LNCS vol. 4202, pages 33–51. Springer, 2006.

[6] H. Aljazzar and S. Leue. Debugging of dependability models using interactive visualization of counterexamples. In *QEST '08*. IEEE Computer Society Press, 2008.

[7] H. Aljazzar and S. Leue. K*: A directed on-the-fly algorithm for finding the k shortest paths. Technical Report soft-08-03, Univ. of Konstanz, Gemany, March 2008. submitted for publication.

[8] H. Aljazzar and S. Leue. Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Transactions on Software Engineering*, 2009.

[9] H. Aljazzar and S. Leue. Generation of counterexamples for model checking of markov decision processes. *Quantitative Evaluation of Systems, International Conference on*, 0:197–206, 2009.

[10] H. Aljazzar and S. Leue. K*: Heuristics-guided, on-the-fly k shortest paths search. 2010. Submitted for publication. Under review.

[11] M. E. Andrés, P. R. D'Argenio, and P. van Rossum. Significant Diagnostic Counterexamples in Probabilistic Model Checking. In *Haifa Verification Conference*, volume 5394 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2008.

[12] AUTOSAR GbR. Requirements on Memory Services v. 2.2.1. www.autosar.org/download/AUTOSAR_SRS_MemoryServices.pdf, checked 16.12.2009.

[13] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transions on Software Engineering*, 29(7), 2003.

[14] B. Damman, T. Han, and J.-P. Katoen. Regular Expressions for PCTL Counterexamples. In *Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 179–188. IEEE Computer Society, 2008.

[15] D. Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.

[16] H. Fecher, M. Huth, N. Piterman, and D. Wagner. Hintikka games for PCTL on labeled Markov chains. In *Fifth International Conference on the Quantitative Evaluaiton of Systems (QEST 2008)*, pages 169–178, Washington, DC, USA, 2008. IEEE Computer Society.

[17] L. Grunske, R. Colvin, and K. Winter. Probabilistic model-checking support for fmea. In *QEST '07: Proceedings of the Fourth International Conference on Quantitative Evaluation of Systems*, pages 119–128, Washington, DC, USA, 2007. IEEE Computer Society.

[18] T. Han, J.-P. Katoen, and B. Damman. Counterexample generation in probabilistic model checking. *IEEE Trans. Softw. Eng.*, 35(2):241–257, 2009.

[19] H. Hermanns, B. Wachter, and L. Zhang. Probabilistic CEGAR. In *Computer Aided Verification, 20th International Conference, CAV 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2008.

[20] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *TACAS'06*, LNCS vol. 3920, pages 441–444. Springer, 2006.

[21] V. M. Jiménez and A. Marzal. A lazy version of eppstein's shortest paths algorithm. In *WEA '03*, LNCS vol. 2647, pages 179–190. Springer, 2003.

[22] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov Reward Model Checker. In *QEST '05*, pages 243–244. IEEE Computer Society, 2005.

[23] J. Muppala, G. Ciardo, and K. Trivedi. Stochastic reward nets for reliability prediction. *Communications in Reliability, Maintainability and Serviceability*, 1(2):9–20, July 1994.

[24] J. Pearl. *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addision–Wesley, 1986.

[25] M. Schmalz, D. Varacca, and H. Völzer. Counterexamples in Probabilistic LTL Model Checking for Markov Chains. In *Proceedings of the 20th International Conference on Concurrency Theory*, volume 5710 of *Lecture Notes in Computer Science*, pages 587 – 602. Springer, 2009.