Symbolic Model Checking of DTMCs with Exact and Inexact Arithmetic

by

Ralf Wimmer Alexander Kortus Marc Herbstritt

Bernd Becker

AVACS Technical Report No. 30
August 2007
ISSN: 1860-9821

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Martin Fränzle, Ernst-Rüdiger Olderog,
Andreas Podelski, Reinhard Wilhelm
ATRs (AVACS Technical Reports) are freely downloadable from `www.avacs.org`

Author(s) contact: Ralf Wimmer (`wimmer@informatik.uni-freiburg.de`).

Symbolic Model Checking of DTMCs with Exact and Inexact Arithmetic*

Ralf Wimmer, Alexander Kortus, Marc Herbstritt, and Bernd Becker

Albert-Ludwigs-University, Freiburg im Breisgau, Germany
{wimmer | kortus | herbstri | becker}@informatik.uni-freiburg.de

Abstract. In formal verification, *reliable results* are of utmost importance. In model checking of digital systems, mainly incorrect implementations due to logical errors are the source of wrong results. In probabilistic model checking, numerical instabilities are an additional source for inconsistent results.
In this report, we experimentally analyze the impact of inexact floating-point arithmetic on the correctness of the result in the context of probabilistic model checking of discrete-time Markov chains. Inexact arithmetic performs implicitly *rounding* of values that are generated during the model checking process.
To enable a direct comparison to the state of the art, which relies on inexact floating-point arithmetic, we have implemented a prototypical probabilistic model checker that provides standard floating-point arithmetic as well as exact arithmetic.
During our experimental evaluation, we found practical examples where the use of inexact arithmetic produces unacceptable results. Two reasons for these problems are: (1) Rounding can change the structure of the system, and (2) rounding can change the truth-value of sub-formulae. Both issues can result in probabilities that are far away from the correct ones.
As a summary, this work reveals the demand for investigating reliable numerical methods within probabilistic model checking.

1 Introduction

While for traditional decision problems arising in computer science a binary true/false answer is intuitively understandable, in the domain of probabilistic systems the interpretation of probability values can be very cumbersome. But our nature is inherently stochastic and hence we have to cope with stochastic systems, e.g., discrete-time Markov chains (DTMCs). At the very end of any method for analyzing stochastic systems, there has to be a representation of probabilistic values within the computer. Typically, the architecture of today's computers provide a *floating-point* representation for real numbers, most prominently established by the IEEE 754 standard specification.

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

Model checking as a verification method enables the separation of the system model from the properties that specify the correct behavior of the system. Model checking has been investigated very deeply in the last 25 years and has become a mature verification methodology pushing forward the frontiers for both large industrial systems (e.g., microprocessors) and novel academic models (e.g., hybrid systems). In the last 10 years, stochastic model checking has been the focus of intense research and besides enormous advances w. r. t. probabilistic models and logics that can be handled algorithmically, it has also reached the usage within industrial applications.

There are several academic tools available for stochastic model checking. To the best of our knowledge, none of these tools makes use of exact arithmetic, but rely on inexact floating-point arithmetic. Also, to the best of our knowledge we are not aware of publications that discuss the impact of using inexact floating-point arithmetic on the correctness of the analysis result. Especially in the context of probabilistic model checking this topic is often euphemized by stating that the probabilistic values of the model are derived from natural observations which itself are inherently stochastic. But this argument does not give the permission to allow inadequate computations *after* the probabilistic values of the model were agreed to be the most accurate values available.

The main topic of this report is therefore to discuss the impact of using inexact and exact, resp., arithmetic while model checking probabilistic models. We will see that in some cases discrepancies appear that clearly reveal the demand for reliable numerical methods.

This report consists of the following parts: First, we review the basic definitions of discrete-time Markov chains (DTMCs), the logic PCTL, and the algorithm for model checking PCTL formulas on DTMCs. The next section is devoted to exact arithmetic. We show at which points of conventional model checkers inaccuracy is introduced and how this can be avoided. Then, we provide an experimental comparison of exact and floating-point arithmetic. In section 4 we investigate what can go wrong if we apply controlled rounding to enable a more compact symbolic representation of the Markov chain. The report closes with a conclusion and an outlook to future work.

2 Foundations of DTMC Model Checking

In this section we will briefly recall the definitions of discrete-time Markov chains (DTMCs), the model we will use in this report, and the logic PCTL, which is used for the specification of properties. We will also sketch the algorithms for checking if a DTMC exhibits a property specified in PCTL.

2.1 Discrete-time Markov Chains

One of the simplest models in probabilistic model checking are discrete-time Markov chains. They are essentially transition systems in which the transitions are labeled with the probability walk from its source state to the target state. This probability is independent of the way the state was reached (so-called Markov property).

Definition 1. *Let AP be a fixed set of atomic propositions. A discrete-time Markov chain (DTMC) is a tuple $M = (S, P, L)$ such that*

- *S is a finite, non-empty set of states,*
- *$P : S \times S \to [0, 1]$ is the matrix of transition probabilities, and*
- *$L : S \to 2^{AP}$ a labelling function which assigns each state the set of propositions that are satisfied in that state.*

P has to be a stochastic matrix, i. e. for each state $s \in S$ the condition $\sum_{s' \in S} P(s, s') = 1$ has to be satisfied.

A path of M is a finite or infinite sequence $\pi = s_0 s_1 s_2 \ldots$ of states such that $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$. We denote the i-th state of π by π^i (i. e. $\pi^i = s_i$) and the i-th prefix by $\pi{\uparrow}^i = s_0, s_1, \ldots, s_i$. The number of states on a finite path π is $|\pi|$. Path_s is the set of infinite paths of M starting in state s.

Let ω be a finite path. The basic cylinder of ω is defined as

$$\Delta(\omega) = \{\pi \in \mathrm{Path}_{\omega^0} \mid \pi{\uparrow}^{|\omega|} = \omega\}$$

Following Markov chain theory, we now define a probability space on the set of paths starting in state $s \in S$:

Definition 2. *Given a Markov chain $M = (S, P, L)$ and $s \in S$, we define a probability space*

$$\Psi^s = (\Delta(s), \Delta^s, \mathrm{Pr}_s)$$

such that

- *Δ^s is the σ-algebra generated by the empty set and the basic cylinders over s that are subsets of $\Delta(s)$.*
- *Pr_s is the uniquely induced probability measure which satisfies the following constraint: $\mathrm{Pr}_s(\Delta(s)) = 1$ and for all basic cylinders $\Delta(s, s_1, \ldots, s_n)$ over S:*

$$\mathrm{Pr}_s(\Delta(s, s_1, \ldots, s_n)) = P(s, s_1) \cdot P(s_1, s_2) \cdot \cdots \cdot P(s_{n-1}, s_n).$$

We now illustrate the main concepts in the following example:

Example 1. In figure 1, you can see a DTMC modelling a very simple communication protocol. First, an initialization is performed, then data blocks are sent and the process waits for an acknowledgment. This normal operation can be interrupted by an error which occurs with probability 0.1 (when sending) and 0.05 (when waiting for an acknowledgment). After an error, the initialization has to be repeated.

Let us consider the finite path $\pi = s_0 s_1 s_2 s_1 s_2$ which is taken if two data packets are transmitted without being interrupted by an error. It's probability is $\mathrm{Pr}_{s_0}(\pi) = 1.0 \cdot 0.9 \cdot 0.95 \cdot 0.9 \cdot 0.95 = 0.731025$.

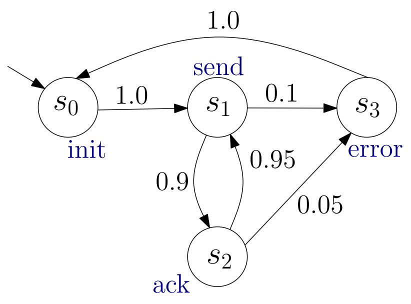


Fig. 1. A discrete-time Markov chain

2.2 Probabilistic Computation Tree Logic

After the formal introduction of the models, we still need a formal language to describe the properties which we want to verify. The most common language is probabilistic computation tree logic (PCTL), which was introduced by Hansson and Jonsson in [5].

In the following, we will briefly define syntax and semantics of PCTL before we turn to the model checking algorithms.

Definition 3 (Syntax of PCTL). *Let AP be a fixed set of atomic propositions, $a \in AP$, $\bowtie \in \{<, \leq, >, \geq\}$, $k \in \mathbb{N}$, and $p \in [0,1]$. PCTL state formulae are then given by*

$$\Phi ::= \text{true} \mid a \mid \neg\Phi \mid (\Phi \wedge \Phi) \mid \mathcal{P}_{\bowtie p}(\Psi)$$

where Ψ is a path formula. PCTL path formulae are created by the following grammar:

$$\Psi ::= X\Phi \mid \Phi\, U\, \Phi \mid \Phi\, U^{\leq k}\, \Phi.$$

Definition 4 (Semantics of PCTL). *Let $M = (S, s_0, P, L)$ be a DTMC, $a \in AP$, $\bowtie \in \{<, \leq, >, \geq\}$, $s \in S$, and $\phi, \phi_1, \phi_2, \psi$ PCTL (state/path) formulae. We define the satisfaction relation $\models$ recursively as follows:*

$s \models \text{true}$	*for all* $s \in S$
$s \models a$	*iff* $a \in L(s)$
$s \models \neg\phi$	*iff* $s \not\models \phi$
$s \models (\phi_1 \wedge \phi_2)$	*iff* $s \models \phi_1$ *and* $s \models \phi_2$
$s \models \mathcal{P}_{\bowtie p}(\psi)$	*iff* $\Pr(\{\pi \mid \pi \models \psi\}) \bowtie p$
$\pi \models X\phi$	*iff* $\pi^1 \models \phi$
$\pi \models \phi_1\, U\, \phi_2$	*iff* $\exists i \geq 0 : (\pi^i \models \phi_2 \wedge \forall j < i : \pi^j \models \phi_1)$
$\pi \models \phi_1\, U^{\leq k}\, \phi_2$	*iff* $\exists 0 \leq i \leq k : (\pi^i \models \phi_2 \wedge \forall j < i : \pi^j \models \phi_1)$.

We define the usual abbreviations for state formulae ϕ_1, ϕ_2, and ϕ:

$$\begin{aligned}
\text{false} &:= \neg\text{true} \\
(\phi_1 \vee \phi_2) &:= \neg(\neg\phi_1 \wedge \neg\phi_2) \\
\mathcal{P}_{\bowtie p}(F^{\leq k}\phi) &:= \mathcal{P}_{\bowtie p}(\text{true}\, U^{\leq k}\, \phi) \\
\mathcal{P}_{\bowtie p}(F\phi) &:= \mathcal{P}_{\bowtie p}(\text{true}\, U\, \phi) \\
\mathcal{P}_{\bowtie p}(G^{\leq k}\phi) &:= \neg\mathcal{P}_{\bowtie (1-p)}(F^{\leq k}\neg\phi) \\
\mathcal{P}_{\bowtie p}(G\phi) &:= \neg\mathcal{P}_{\bowtie (1-p)}(F\neg\phi).
\end{aligned}$$

2.3 Model Checking PCTL

Up to now, we have introduced discrete-time Markov chains as our system models and the logic PCTL for the description of desired properties. In this section we will show how to compute the states which satisfy a given PCTL formula. We will concentrate on the main principles which are necessary to understand where inaccuracies are introduced. For more details on PCTL model checking see e. g. [6, 7].

Like model checking for CTL, model checking for PCTL is based on recursively traversing the syntax tree of the formula bottom-up and computing the set Sat(ϕ) for each state sub-formula ϕ. This can be done for state formulae as follows (a denotes an atomic proposition; ϕ, ϕ_1, and ϕ_2, PCTL state formulae; ψ, a PCTL path formula; and $p \in [0, 1]$, a real number):

$$\begin{aligned}
\text{Sat}(\text{true}) &= S \\
\text{Sat}(a) &= \{s \in S \,|\, a \in L(s)\} \\
\text{Sat}(\neg\phi) &= S \setminus \text{Sat}(\phi) \\
\text{Sat}((\phi_1 \wedge \phi_2)) &= \text{Sat}(\phi_1) \cap \text{Sat}(\phi_2) \\
\text{Sat}(\mathcal{P}_{\bowtie p}(\psi)) &= \{s \in S \,|\, \Pr(s, \psi) \bowtie p\}
\end{aligned}$$

Hereby, $\Pr(s, \psi)$ denotes the probability $\Pr(\{\pi \in \text{Path}_s \,|\, \pi \models \psi\})$. The remaining task is consequently the computation of $\Pr(s, \psi)$. Depending on the path quantifier (X, $U^{\leq k}$, U), we distinguish three cases. For each we will point out that the main operation which has to be implemented is matrix-vector multiplication.

Next Quantifier (*X*) Given the set Sat(ϕ), the probability $\Pr(s, X\phi)$ can be computed as follows:

$$\Pr(s, X\phi) = \sum_{s' \in \text{Sat}(\phi)} P(s, s')$$

Let χ_ϕ be a vector with

$$\chi_\phi(s) = \begin{cases} 1 & \text{if } s \in \text{Sat}(\phi) \\ 0 & \text{otherwise} \end{cases}$$

and p_ψ the vector with $p_\psi(s) = \Pr(s, \psi)$. Then we can write:

$$p_{X\phi} = P \cdot \chi_\psi.$$

Hence, model checking for the X quantifier requires one matrix-vector multiplication.

Bounded Until Quantifier ($U^{\leq k}$) We can characterize the bounded until operator with the following recursive equation system:

$$\Pr(s, \phi_1 U^{\leq k} \phi_2) = \begin{cases} 1 & \text{if } s \in \mathrm{Sat}(\phi_2) \\ 0 & \text{if } s \notin \mathrm{Sat}(\phi_1) \text{ and } s \notin \mathrm{Sat}(\phi_2) \\ 0 & \text{if } k = 0 \text{ and } s \notin \mathrm{Sat}(\phi_2) \\ \sum\limits_{s' \in S} P(s, s') \cdot \Pr(s', \phi_1 U^{\leq (k-1)} \phi_2) & \text{otherwise.} \end{cases}$$

The intuition behind it is the following: if s satisfies ϕ_2, all paths starting in s fulfill $\psi := \phi_1\ U^{\leq k}\ \phi_2$; hence, the probability is 1. If s satisfies neither ϕ_1 nor ϕ_2, ψ cannot be satisfied on any path starting in s; accordingly, the probability is set to 0. The same holds if s does not satisfy ϕ_2 and $k = 0$, such that no further steps may be taken. Otherwise, we may walk one step and fulfill the formula in one step less.

For the computation of the probabilities, we set $p_{\phi_1 U^{\leq 0} \phi_2} = \chi_{\phi_2}$ and modify the matrix of transition probabilities as follows:

$$P'(s, s') = \begin{cases} 1 & \text{if } s \in \mathrm{Sat}(\phi_2) \\ 0 & \text{if } s \notin \mathrm{Sat}(\phi_1) \text{ and } s \notin \mathrm{Sat}(\phi_2) \\ P(s, s') & \text{otherwise.} \end{cases}.$$

With these modifications we can write

$$p_{\phi_1 U^{\leq i} \phi_2} = P' \cdot p_{\phi_1 U^{\leq i-1} \phi_2}.$$

The probabilities we are looking for are then given by

$$\Pr(s, \phi_1 U^{\leq k} \phi_2) = p_{\phi_1 U^{\leq k} \phi_2}.$$

We can conclude that we need k matrix-vector multiplication for the bounded until operator.

Unbounded Until Quantifier (U) The unbounded until operator can be characterized in a similar way as the bounded operator. But in this case the characterization leads to a linear equation system:

We partition the state space into three sets S^0, S^1, and $S^?$ such that $\Pr(s, \phi_1 U \phi_2) = 1$, if $s \in S^1$, and 0, if $s \in S^0$. S^0 contains all those states from which no path leads to a ϕ_2 state while passing only ϕ_1-states. Conversely, S^1 is the set of ϕ_1-states which do not have a path to a state in S^0. $S^?$ contains all states not in S^0 or S^1.

These sets can be computed as follows [6]:

$$
\begin{aligned}
S_f &= \{s \in S \mid s \notin \mathrm{Sat}(\phi_1) \wedge s \notin \mathrm{Sat}(\phi_2)\} \\
S_i &= \{s \in S \mid s \in \mathrm{Sat}(\phi_1) \wedge s \notin \mathrm{Sat}(\phi_2)\} \\
S^0 &= S_f \cup \{s \in S_i \mid \text{there exists no path in } S_i \text{ from } s \text{ to any } s' \in \mathrm{Sat}(\phi_2)\} \\
S^1 &= S_s \cup \{s \in S_i \mid \text{there exists no path in } S_i \text{ from } s \text{ to any } s' \in S^0\} \\
S^? &= S \setminus (S^0 \cup S^1)
\end{aligned}
$$

The probability that in state s, a path π is taken with $\pi \models \phi_1 U \phi_2$ is then given by the unique solution of the following system of linear equations:

$$
\Pr(s, \phi_1 U \phi_2) = \begin{cases} 1 & \text{if } s \in S^1 \\ 0 & \text{if } s \in S^0 \\ \sum\limits_{s' \in S} P(s, s') \cdot \Pr(s', \phi_1 U \phi_2) & \text{if } s \in S^?. \end{cases}
$$

The intuition behind this equation system is the same as in the case of bounded until. The difference is that we do not have to take a bound on the number of steps into account.

Such linear equation systems are usually solved using iterative methods like Jacobi, Gauß-Seidel or over-relaxation methods. The reason for using iterative methods instead of e. g. Gaussian elimination is the following: explicit model checkers use representations for the probability matrix which are optimized for sparse matrices. The use of direct solution methods destroys the sparseness of the probability matrix. Symbolic model checkers use iterative methods because they can exploit the compact symbolic representation better than direct methods which have to modify single elements of the matrix.

Since the Jacobi method can be implemented symbolically in a efficient way [7], we put our focus on this method.

The Jacobi Method Assume we have to solve a linear equation system of the form $Ax = b$:

$$
\sum_{i=0}^{|S|-1} A_{ij} x_j = b_i \qquad \text{for } i = 0, \ldots, |S| - 1
$$

with $A_{ii} \neq 0$ for all i. This equation can be rearranged as follows:

$$
x_i = b_i - \frac{1}{A_{ii}} \sum_{j \neq i} A_{ij} x_j
$$

The Jacobi method is based on using the values from the $(k-1)$-th iteration on the right-hand side to compute the new approximation:

$$
x_i^{(k)} = b_i - \frac{1}{A_{ii}} \sum_{j \neq i} A_{ij} x_j^{(k-1)}
$$

We split the matrix A into its diagonal D and a matrix $B = D - A$, i. e., D contains all diagonal elements of A and B all the negated non-diagonal elements. Then we can write

$$x^{(k)} = D^{-1} \cdot (B \cdot x^{(k-1)} + b).$$

Also in this case, the main operation to perform a single iteration is matrix-vector multiplication. This step is repeated until $\|x^{(k)} - x^{(k-1)}\|_{\mathcal{L}} < \varepsilon$ for a given $\varepsilon > 0$ and a norm $\mathcal{L}$ on $\mathbb{R}^{|S|}$.

2.4 Symbolic Methods for PCTL Model Checking

Algorithms which rely on an explicit representation of the system are naturally restricted to quite a small number of states. A method to overcome this problem is the usage of symbolic data structures. Their advantage is that their size does not directly depend on the size of the represented state space. For many practical examples, the size of the symbolic representation is much smaller than the explicit representation such that larger systems can be handled.

One of the most prominent symbolic data structure are binary decision diagrams (BDDs) [2]. We assume some familiarity of the reader with BDDs. For further information see e. g. Wegener's monograph on decision diagrams [1].

Since the details of how the model checking algorithms can be modified to exploit the symbolic representation efficiently are only of little importance for the understanding of the effects of inexact arithmetic, we refer the reader to Parker's PhD thesis [7] about probabilistic model checking.

3 Model Checking with Exact Arithmetic

We now turn our attention to the comparison of probabilistic model checking with exact and with floating-point arithmetic. We will first point out where inaccuracy is introduced during the model checking process and how it can be avoided. The effects of the floating-point arithmetic on some practical benchmarks are then evaluated.

3.1 Sources of Inaccuracy

To identify at which points inaccuracy is introduced during the model checking process, we had a close look at the model checker PRISM [3]. We have identified three major sources of inaccuracy. These are not restricted to a specific tool, but they are inherent to all state-of-the-art model checkers.

1. The floating-point arithmetic, which is used by all state-of-the-art model checkers for PCTL.
2. The termination criterion for solving the linear equation systems for the unbounded-until quantifier. E. g. for the Jacobi method, PRISM uses two termination criteria: The iteration terminates either if $\|x^{(k)} - x^{(k-1)}\|_\infty < \varepsilon$ for

a given constant $\varepsilon > 0$ or if $\frac{\|x^{(k)} - x^{(k-1)}\|_\infty}{\|x^{(k)}\|_\infty} < \varepsilon$. PRISM uses the second criterion and $\varepsilon = 10^{-6}$ as default.

The floating-point arithmetic is based on IEEE standard 754 [9] for 64 bit numbers. While the additions and multiplications are carried out with higher precision internally, the result of each arithmetic operation is rounded to fit into the 64-bit representation. About 15 decimal digits (51 binary digits) can be represented correctly. If the result is not representable as floating-point number with that precision the nearest representable number is chosen if it is unique. If the result lies exactly in the middle of two floating-point numbers, the one whose representation ends with "0" is chosen (round-to-nearest-even). We refer the reader to e.g. [8] for details on how the rounding for floating-point number works.

3. Another reason for inaccuracy is located in the BDD package. In most packages like Cudd [15], which is used by PRISM, there is a constant δ such that a new leaf with value v is only generated if there is no leaf with value v' and $|v - v'| \leq \delta$ for a constant $\delta > 0$. The value of δ is chosen to be in the order of the error by the floating-point arithmetic. Cudd uses $\delta = 10^{-12}$ as default.

3.2 Exact Arithmetic

By using exact arithmetic, we can eliminate the first and the third source of inaccuracy. The second source could only be eliminated by using a direct method like Gaussian elimination for solving linear equation systems. These direct methods, however, are very badly suited for a symbolic data representation because single entries in the matrix have to be manipulated. Thereby the compact representation cannot be exploited and—making it still worse—the structure of the matrix gets lost such that the size of the MTBDD explodes.

For these reasons we decided to eliminate only the rounding problems completely by using exact arithmetic for the computations. We only lessen the inaccuracy introduced by the termination criterion of the Jacobi method by choosing an extremely small constant ε, e. g. $\varepsilon = 2^{-100}$ in our experiments below.

3.3 Experimental Evaluation

We implemented a PCTL model checker for PCTL using C++ as programming language. We allow for switching from floating-point arithmetic to exact arithmetic and choosing the constants ε and δ by the user. This enables a direct comparison of exact arithmetic with floating-point arithmetic.

We use the GNU multiple precision arithmetic library (gmp) [10] for the exact arithmetic which supports rational numbers of arbitrary length. Their numerators and denominators are represented by a sequence of 64 bit numbers (called "limbs"). The library provides efficient implementations of all necessary operations.

PRISM uses internally the Cudd [15] library for OBDDs and MTBDDs. Since it is only able to deal with floating-point numbers which are stored in the leaves of the DDs, we had to modify the MTBDDs such that the gmp rationals are used as labels of the leaves and had to change most algorithms such that the corresponding gmp operations are used.

Furthermore, for exact arithmetic, we changed the constant δ which is used by Cudd internally to avoid the creation of leaves whose values differ only by less than the error induced by the floating-point arithmetic, to 0.

We used a selection of models that come with the PRISM distribution, namely a bounded retransmission protocol (`brp<N>_<MAX>`) [12] and a randomized self-stabilizing protocol (`herman<N>`) [11].

- `brp<N>_<MAX>`: This protocol tries to send a file in a number of chunks, but allows only a bounded number of retransmissions of each chunk in case of an error. The value $N \in \{16, 32, 64\}$ denotes the number of chunks and $MAX \in \{2, 3, 4, 5, 6\}$ the maximum allowed number of retransmissions of each chunk. The property [13] we checked is
  "Eventually the sender does not report a successful transmission."
  or more formally:
  $$\Pr(s_0, \text{true } U \neg \text{success}),$$
  where s_0 is the initial state of the system.
- `herman<N>`: A self-stabilizing protocol for a network of processes is a protocol which, when started from an arbitrary initial state, returns to a stable state without any outside intervention within some finite number of steps. We calculate the probability to reach a stable state within $k = 100$ steps, i. e.,
  $$\Pr(s_0, \text{true } U^{\leq k} \text{stable}).$$

Table 1 contains the number of states and transitions for each model, table 2 the experimental results thereof.

Table 1. Size of the models in terms of number of states and number of transitions.

Model	# States	# Transitions	Model	# States	# Transitions
brp_16_2	677	832	brp_16_3	886	1119
brp_16_4	1095	1406	brp_16_5	1304	1693
brp_16_6	1513	1980	brp_32_2	1349	1664
brp_32_3	1766	2239	brp_32_4	2183	2814
brp_32_5	2600	3389	brp_32_6	3017	3964
brp_64_2	2693	3328	brp_64_3	3526	4479
brp_64_4	4359	5630	brp_64_5	5192	6781
brp_64_6	6025	7932			
herman03	8	28	herman05	32	244
herman07	128	2188	herman09	512	19684
herman11	2048	1771485	herman13	8192	1594324
herman15	32768	14348908	herman17	131072	129140164

Table 2. Results for comparing inexact vs. exact arithmetic.

Model	# Leaves	Time	Memory	Result
brp_16_2	98	2.21	10725	0.00042333344377341790
	98	1.22	9407	0.0004233334**5332507531**
brp_16_3	130	3.96	11831	0.00001261776603623259
	130	1.56	10384	0.0000126177**7701679546**
brp_16_4	189	6.13	13007	0.00000037601158556078
	162	1.95	10873	0.0000003760**2213653487**
brp_16_5	249	8.29	14128	0.00000001120514716583
	194	2.31	10697	0.0000000112**1397044814**
brp_16_6	310	10.48	15606	0.00000000033391338724
	225	9.89	12632	0.0000000003**4014825664**
brp_32_2	215	7.94	13915	0.00084648767634221873
	194	3.04	10500	0.000846487**70344783594**
brp_32_3	329	13.01	16145	0.00002523537286444544
	258	4.07	11543	0.0000252353**9639424109**
brp_32_4	450	18.99	18652	0.00000075202302973685
	322	5.24	12105	0.0000007520**4403616045**
brp_32_5	575	23.96	22179	0.00000002241029420610
	386	5.88	15125	0.0000000224**3232895423**
brp_32_6	702	30.59	28049	0.00000000066782677437
	n. a.	n. a.	n. a.	n. a.
brp_64_2	496	26.04	23136	0.00169225881129823822
	386	9.94	14700	0.0016922588**4667877470**
brp_64_3	738	41.14	29594	0.00005047010890484727
	514	12.39	14869	0.0000504701**5148748653**
brp_64_4	986	58.45	33447	0.00001504045493935506
	642	15.49	18451	0.000015040**9070913174**
brp_64_5	1238	75.95	45097	0.00000004482058790997
	770	28.72	18807	0.0000000448**4327987946**
brp_64_6	1492	92.6	46901	0.00000000133565354830
	n. a.	n. a.	n. a.	n. a.
herman03	2	0	5739	1.00000000000000000000
	1	0	5716	1.00000000000000000000
herman05	4	0.06	5894	0.99999999999999999961
	4	0.01	5851	0.9999999999**7269473084**
herman07	9	0.43	6036	0.99999999912510300776
	25	0.22	5932	0.9999999991**1152470666**
herman09	23	2.87	6172	0.99999604484699721382
	21	1.39	6047	0.999996044846997**33398**
herman11	63	20.47	8295	0.99974392938338150787
	63	8.59	7044	0.9997439293833**7242054**
herman13	190	146.12	16795	0.99725437499022528290
	195	56.94	12080	0.9972543749**8551250393**
herman15	612	958.78	54535	0.98795298924500753115
	1847	391.88	43981	0.98795298924**165586563**
herman17	2056	6677.12	221506	0.96776426181745151251
	10117	2078.00	93572	0.967764261**90205311340**

In table 2, the first line of each model contains the data for the exact model checker, the second one the data of the symbolic floating-point engine. The columns have the following meaning:

Model Name of the model.
# States The number of states the model consists of.
# Transitions The number of transitions with non-zero probability.
# Leaves The number of different leaves needed to represent the vector of the probabilities that the path formula holds in a state.
Time The runtime of the model checking algorithm in seconds.
Memory The amount of memory used in kilobytes.
Result The probability computed by the model checking algorithm. It is the probability that a path, starting in the initial states of the system, is taken that satisfies the given PCTL path formula. The figures in which the exact and the inexact model checker differ are marked bold.

It can be observed that (1) exact arithmetic often creates more different values resulting in a higher number of leaves in the representation of the probability vector. There are a number of exceptions (see e. g. `herman17` where inexact arithmetic creates almost 5 times as many leaves as exact arithmetic). The reason for this artefact can be understood by the following example: Assume we are using the number system with two decimal digits and want compute $\frac{1}{3} \cdot 3 = 1$ and $\frac{2}{3} \cdot \frac{3}{2} = 1$, respectively. By using this inexact arithmetic, we obtain $0.33 \cdot 3.00 = 0.99$ and $0.67 \cdot 1.50 = 1.01$. Although the result should be the same, inexact arithmetic would create two different leaves in the MTBDD.

(2) Exact arithmetic needs more time due to expensive arithmetic operations, and (3) it consumes more memory because of the larger MTBDDs and the much bigger size of the rational numbers. (4) In spite of the inaccuracy introduced in every arithmetic operation, the result of the floating-point version is accurate within an error of 10^{-10}.

But there are two exceptions from these observations: `brp_32_6` and `brp_64_6`. When using $\varepsilon = 10^{-12}$ the algorithm turned out to be unstable so that the Jacobi algorithm did not terminate within 4 hours! Changing the value to 10^{-11} or 10^{-13}, however, produced results which followed the trend from the other benchmarks.

4 Inexact Arithmetic

After the evaluation of exact arithmetic we now turn our focus onto inexact arithmetic, i. e. the controlled rounding of values.

4.1 Idea

Since the results of the comparison of the exact model checker with the unmodified version of PRISM showed that the inaccuracies introduced by the floating-point arithmetic are relatively small, we tried to exploit the following idea:

The problem which restricts the success of symbolic methods in probabilistic model checking is the vector of probabilities $\Pr(\cdot, \psi)$ which has to be computed for each path formula. In the beginning it is sparse and contains only few different values. But in opposite to the probability matrix the vector is changed by every matrix-vector multiplication. This has the effect that the vector contains many different values already after few multiplications. Its MTBDD representation often grows so massively that it cannot be handled anymore.

The idea is to apply rounding to the vector entries, thereby reducing the number of distinct values. The hope is that this also reduces the MTBDD size significantly without losing so much precision that we cannot give any sensible result.

4.2 Implementation

We changed the model checking algorithm we have used for the comparison of exact and floating-point arithmetic in the last section such that each time a new leaf has to be created its value is rounded to the specified precision. Since the constant δ used to identify values which differ only very little has to become irrelevant, we set it to 0.

Furthermore, the constant ε of the termination criterion is adapted such that it is not smaller than the precision of the rounding mechanism (otherwise it would not terminate).

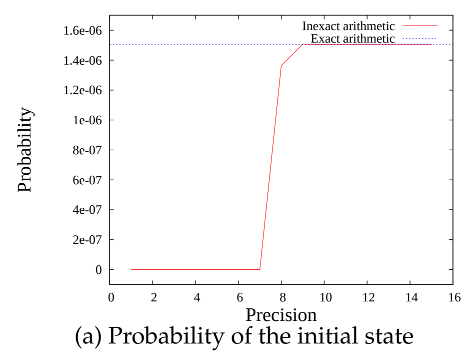
In the following, we denote the precision to which the numbers are rounded by σ.

4.3 Experimental Evaluation

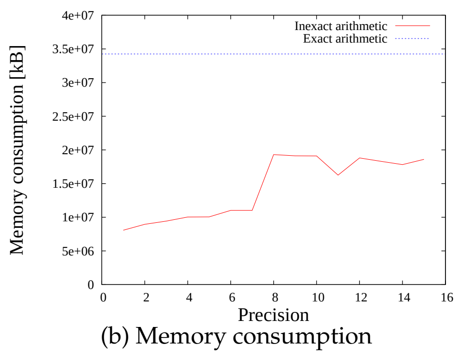
We applied the model checking tool to the examples, which we already used in the previous section, and increased the precision σ from 10^{-1} to 10^{-15} by an increment of 10^{-1}. Figures 2 and 3 show the results of the comparison to the version with exact arithmetic for two of the benchmarks (`brp_64_4` and `herman17`, respectively).

The precision grows exponentially along the x-axis; the value i means that the precision is 10^{-i}, i. e. that we round all values to i digits after the period. Part (a) of the figures depicts the probability for which the given path formula holds in the initial state. In part (b) the memory consumption in kilobytes is shown. The runtime of the model checking algorithm is displayed in part (c) and finally the number of distinct leaves required for the representation of the final probability vector is shown in part (d).

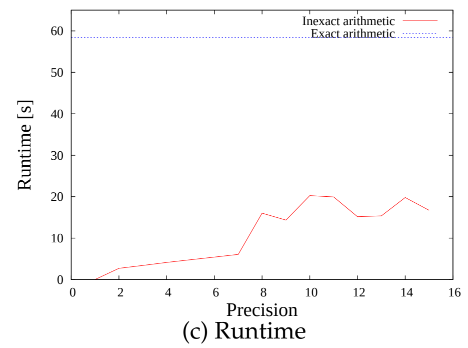
The following observations can be made: If the precision is very low (10^{-1} to 10^{-7} for `brp_64_4` and 10^{-1} to 10^{-5} for `herman17`, resp.) the number of different leaves is small, the same holds for the memory consumption and the runtime. The error introduced by the rounding is considerable: for `brp_64_4` the result is



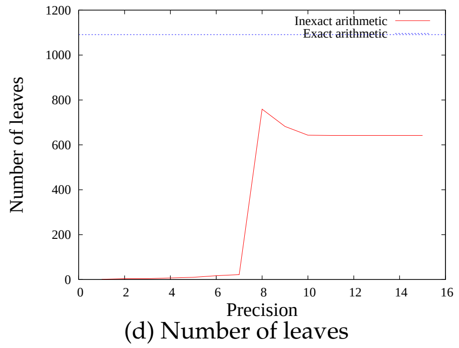
(a) Probability of the initial state



(b) Memory consumption



(c) Runtime



(d) Number of leaves

Fig. 2. Results for `brp_64_4` depending on the chosen precision

0 instead of $\sim 1.5 \cdot 10^{-6}$. It is much worse for `herman17`: the correct probability would be in the area of 0.96, but the result when using rounding is 0. The reason for this effect can be seen in the following example:

Example 2. Figure 4 shows a Markov chain with two states s_0 *and* s_1*. In its initial state* s_0 *the property* $a \, U \, b$ *holds with probability* 1 *as long as* $\gamma > 0$.

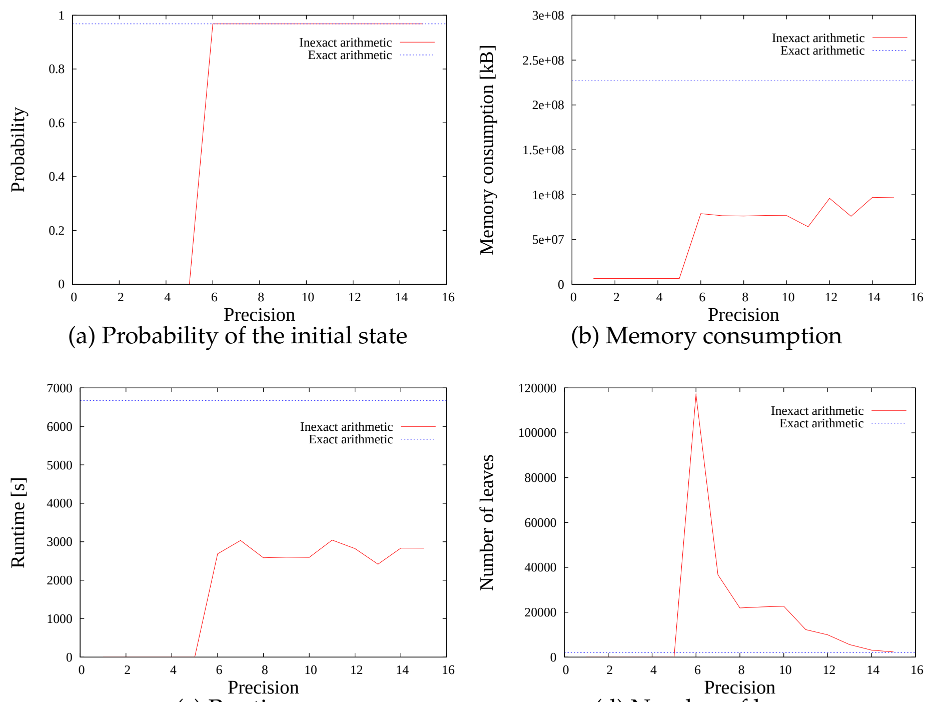
Assume now that we are working with a precision σ *and* $0 < \gamma < \frac{1}{2}\sigma$*. Then* γ *is rounded to* 0 *and* $1 - \gamma$ *to* 1*. The result is a Markov chain with two disconnected states such that it is not possible anymore to go from* s_0 *to* s_1*. This means, the property* $a \, U \, b$ *does not hold. The inexact model checker returns probability* 0.

This problem does not only appear when transition probabilities become 0 due to rounding. The same phenomenon can also arise depending on the formula to be checked. This is illustrated in the next example:

Example 3. Assume, we want to compute the probability with which states of the DTMC in figure 5 satisfy the formula

$$c \, U \, \mathcal{P}_{>0.5}(a \, U \, b).$$

Since $\Pr(s_1, a \, U \, b) = 0.5 + 0.25 \cdot \gamma > 0.5$*, the property is satisfied in state* s_0 *with probability* 1.



(a) Probability of the initial state (b) Memory consumption

(c) Runtime (d) Number of leaves

Fig. 3. Results for `herman17` depending on the chosen precision

Now consider the case that we are working with a precision σ *and that* $\sigma \leq \gamma < 2\sigma$*. Then the structure of the DTMC is not affected by rounding the probabilities, but the probability* $0.5 + 0.25 \cdot \gamma$ *that the formula* $a\,\mathcal{U}\,b$ *is satisfied in state* s_1 *is rounded downwards to* 0.5*. This means that* $\mathcal{P}_{>0.5}(a\,\mathcal{U}\,b)$ *is not satisfied in state* s_1 *and hence the probability that* s_0 *satisfies* $c\,\mathcal{U}\,\mathcal{P}_{>0.5}(a\,\mathcal{U}\,b)$ *becomes* 0*.*

The picture changes if the precision is chosen high enough: The probability of the initial state converges quickly to the exact precision. The memory consumption and the runtime also grow, but always stays below the values of the version with exact arithmetic. The number of leaves is ambiguous: Sometimes it is less than the number of leaves in the exact version, sometimes it is much larger (see e. g. `herman17` with precision $\sigma = 10^{-6}$). But for both benchmarks the number of leaves is convergent with increasing precision.

5 Conclusion

In this report we have experimentally analyzed the effect of exact and inexact arithmetic in the context of PCTL model checking for discrete-time Markov chains.

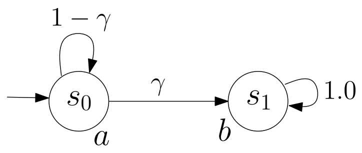


Fig. 4. Worst case error depending on the system

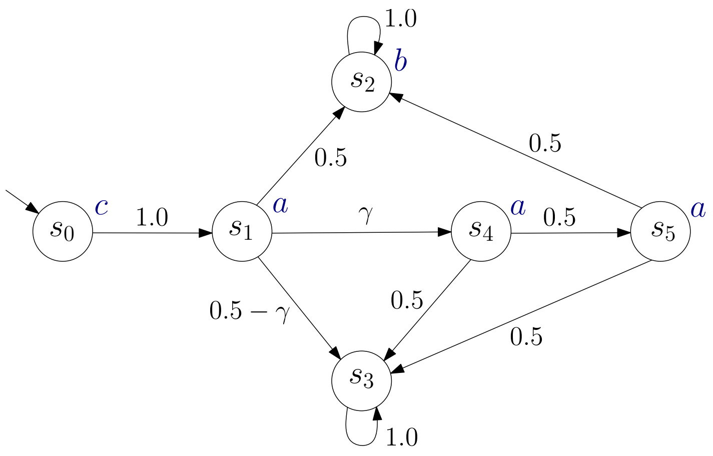


Fig. 5. Worst case error depending on the formula

We have first compared the results of a model checker with *exact* arithmetic with those of a model checker which uses the standard IEEE floating-point arithmetic. The evaluation mostly met our expectations: the deviation of the floating-point results were in the order of magnitude of the rounding errors and the error introduced by the termination criterion. The exception is that the numerical instabilities prevented the termination of the algorithm in two cases.

The next step was to investigate the effects of controlled rounding to a given precision. We have seen practical examples where inexact arithmetic produced completely wrong results: probability 0 instead of 0.96. We have tried to find reasons for these discrepancies: one is that the structure of the system under consideration can be changed due to rounding of probabilities. Another source of problems comes from the formula that is to be checked: If the probability with which some sub-formulae hold is near the given bound, rounding can have great impacts on the probability of the complete formula.

It is therefore of utmost importance to obtain reliable *certificates* for the correctness of the results since due to rounding errors completely wrong answers can be given by the model checker. Such certificates can for example be counterexamples [17] in the case that the formula is not satisfied. Another option is to use exact or interval arithmetic [16] to compute safe approximations of the probabilities.

References

1. Wegener, I.: Branching Programs and Binary Decision Diagrams – Theory and Applications. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA (2000)
2. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691
3. Kwiatkowska, M.Z., Norman, G., Parker, D.: Prism 2.0: A tool for probabilistic model checking. In: 1st International Conference on Quantitative Evaluation of Systems (QEST), Enschede, The Netherlands, IEEE Computer Society (2004) 322–323
4. de Alfaro, L., Kwiatkowska, M.Z., Norman, G., Parker, D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS). Vol. 1785 of Lecture Notes in Computer Science., Berlin, Germany, Springer (2000) 395–410
5. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing **6**(5) (1994) 512–535
6. Ciesinski, F., Größer, M.: On probabilistic computation tree logic. In: Validation of Stochastic Systems. Vol. 2925 of Lecture Notes in Computer Science., Springer (2004) 147–188
7. Parker, D.: Implementation of Symbolic Model Checking for Probabilistic Systems. PhD thesis, University of Birmingham, Great Britain (2002)
8. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys **23**(1) (1991)
9. IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee, American National Standards Institute: IEEE standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985. IEEE Computer Society Press, Silver Spring, MD 20910, USA (1985)
10. GNU Multiple Precision Arithmetic Library: Release 4.2.1, `http://gmplib.org/` (2007)
11. Herman, T.: Probabilistic self-stabilization. Information Processing Letters **35**(2) (1990) 63–67
12. Helmink, L., Sellink, M., Vaandrager, F.: Proof-checking a data link protocol. In: International Workshop on Types for Proofs and Programs (TYPES'93). Vol. 806 of Lecture Notes in Computer Science., Springer (1994) 127–165
13. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: Proceedings of the Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV). Vol. 2165 of Lecture Notes in Computer Science., Springer (2001) 39–56
14. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Information and Computation **88**(1) (1990)
15. Somenzi, F.: CUDD: CU Decision Diagram Package Release 2.4.1. University of Colorado at Boulder (2005)
16. Fecher, H., Leucker, M., Wolf, V.: Don't know in probabilistic systems. In: 13th International SPIN Workshop on Model Checking Software. Vol. 3925 of Lecture Notes in Computer Science., Vienna, Austria, Springer (2006) 71–88
17. Han, T., Katoen, J.P.: Counterexamples in probabilistic model checking. In: 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS). Vol. 4424 of Lecture Notes in Computer Science., Braga, Portugal, Springer Verlag (2007) 60–75