# A Formally Verified IEEE 754 Floating-Point Implementation of Interval Iteration for MDPs

Bram Kohlen[1], Maximilian Schäffeler[2],
Mohammad Abdulaziz[3], Arnd Hartmanns[1], and Peter Lammich[1]

[1] University of Twente, The Netherlands b.kohlen@utwente.nl
[2] Technical University of Munich, Germany maximilian.schaeffeler@tum.de
[3] King's College London, United Kingdom

**Abstract.** We present an efficiently executable, formally verified implementation of interval iteration for MDPs. Our correctness proofs span the entire development from the high-level abstract semantics of MDPs to a low-level implementation in LLVM that is based on floating-point arithmetic. We use the Isabelle/HOL proof assistant to verify convergence of our abstract definition of interval iteration and employ step-wise refinement to derive an efficient implementation in LLVM code. To that end, we extend the Isabelle Refinement Framework with support for reasoning about floating-point arithmetic and directed rounding modes. We experimentally demonstrate that the verified implementation is competitive with state-of-the-art tools for MDPs, while providing formal guarantees on the correctness of the results.

## 1 Introduction

Probabilistic model checking (PMC) [3,4] is a formal verification technique for randomised systems and algorithms like wireless communication protocols [35], network-on-chip (NoC) architectures [47], and reliability and performance models [5]. Typical properties checked by means of PMC relate to *reachability probabilities*: What is the probability for a file to eventually be transmitted successfully [17]? Is the probability for a NoC router's queue to overflow within $c$ clock cycles below $10^{-5}$? What is the maintenance strategy that minimises service outages within a given cost budget [48,49]? The system models that PMC is applied to are specified in higher-level modelling languages such as Modest [11,23] or JANI [14] with a formal semantics in terms of (extensions of) Markov chains and Markov decision processes (MDPs) [10,45].

PMC delivers results with formal guarantees, typically that the computed and (unknown) true probabilities differ by at most a user-specified $\varepsilon$. PMC is thus well-suited for the design and evaluation of safety- and performance-critical systems. Over the past decade, however, we have witnessed several threats to the validity of PMC results. First and foremost, the most-used PMC algorithm, value iteration (VI), was shown to be *unsound*, i.e. produce arbitrarily wrong results for certain inputs [21]. Several sound replacements for VI were subsequently developed [22,29,46], yet their soundness proofs have so far been *pen-and-paper* style with room for human error. For example, the pseudocode for the *sound VI* algorithm as stated in [46] contains a subtle omission that only surfaces on 1 of the 78 models of the Quantitative Verification Benchmark Set (QVBS) [30]. This calls for *formal specifications of the algorithms* accompanied by *machine-checked correctness proofs*. Even correct algorithms, however, may be incorrectly implemented in today's manually-coded PMC tools. As a case in point, the implementation of the *interval iteration* algorithm [22] for expected rewards [7] in the mcsta model checker of the Modest Toolset [27] diverges on some inputs. We thus need *correct-by-construction implementations*, too.

VI-based algorithms are iterative numeric approximation schemes that need to be implemented via fixed machine-precision floating-point arithmetic to obtain acceptable performance [15,28]. This introduces approximation and rounding errors that in turn may lead to incorrect Boolean outputs [57]. An efficient solution is to carefully use the directed rounding modes provided by standard IEEE 754 floating-point implementations as in all of today's common CPUs [26], which however needs careful *reasoning about floating-point errors and rounding* in all formal proofs and correctness-preserving implementation strategies.

*Our contribution.* We present a solution to all of the above challenges based on the interval iteration (II) algorithm [22] for sound PMC on MDP models and the interactive theorem prover (ITP) Isabelle/HOL [44] with its Isabelle Refinement Framework (IRF) [39]:

- We formalise (i.e. model) II in Isabelle/HOL's logic and formally prove its correctness using Isabelle/HOL (Sect. 3), making II the first sound PMC algorithm for MDPs with machine-checked correctness.
- We extend the IRF with support for floating-point arithmetic, including directed rounding modes (Sect. 4.2), making it the first ITP-based algorithm refinement approach suitable for II and similar algorithms.
- Using the IRF, we refine the formalisation of II into efficient LLVM bytecode (Sects. 4.3 and 4.4), delivering the first correct-by-construction implementation of a PMC algorithm.
- We embed the code into mcsta, a competitive probabilistic model checker (Sect. 5). We experimentally evaluate the performance using the QVBS, showing that the verified implementation is efficient. Our formal proofs and the benchmark code are available as part of the *additional files*.

*State-of-the-art: Verification of Algorithms for MDPs.* A probabilistic model checker like mcsta performs preprocessing and transformation steps for both correctness and performance. Previously, the strongly connected component [31] and maximal end component decomposition [32] algorithms have been verified down to LLVM, replacing their previous unverified implementations inside mcsta by verified ones of comparable performance. These were fully discrete graph algorithms, however, that neither required reasoning about numerical convergence in their correctness proofs nor floating-point arithmetic in their refinement to an efficient implementation. With this work, we contribute an essential piece for the incremental replacement of unverified by verified algorithms for probabilistic reachability in mcsta's MDP model checking core.

Other work that is also relevant to our setting is the verification of iteration algorithms for MDPs: In Coq by Vajjha et. al. [55] and in Isabelle/HOL by Schäffeler and Abdulaziz [50,51]. In their work, they verified the classical version of value iteration and policy iteration, that optimise the expected discounted values, and a modified policy iteration algorithm for solving large, factored MDPs. We note that only Schäffeler and Abdulaziz [50,51] also verified practical implementations. However, since their implementations used infinite-precision arithmetic, they could not compete with state-of-the-art floating-point implementations. Thus, the work we present here is the first, up to our knowledge, where a full formal mathematical analysis of an algorithm, involving heavy usage of a formal mathematical probabilities library, is performed and a competitive floating-point implementation is also verified. Furthermore, from a formalisation-methodology perspective, we note that the correctness argument of II involves a substantial element of graph-theoretic reasoning, in addition to the reasoning about fixed points that is present in II and other verified MDP algorithms. This includes reasoning about connected components, acyclicity, and levels in a DAG, further complicating II's verification compared to other verified iteration algorithms.

*State-of-the-art: Verification of Floating-Point Algorithms.* That floating-point implementations deviate from the respective mathematical models of the algorithms is widely recognised as a problem. Bugs with potentially serious consequences were noted in the hardware and aerospace industry [25,43]. Due to the complexity of floating-point algorithms' behaviour, and the failure of testing to reliably catch bugs in those algorithms, there is a long tradition of applying formal methods to the verification of floating-point algorithms. This was done in verification systems like Z [9], HOL Light [24,25], PVS [13,42], and Coq [12,19]. Most of that previous work, however, focused on proving correctness of basic algorithms implemented in floating-point arithmetic. In contrast, we aim to do the correctness proofs on algorithms using real numbers, which we implement as floating-point numbers with directed rounding. This keeps our correctness proofs manageable while preserving interesting properties, even for complex programs.

A related line of work aims to prove correctness by providing error bounds. Tools like PRECiSA [54], FPTaylor [52], Real2Float [41], and Fluctuat [20] analyze the floating-point error propagation. They focus on determining the worst-case roundoff error. While more expressive than our approach, these tools have limited to no support for programs with complex control flow like the nested loops in the implementation of II.

The static analysis tool Astrée [16] is used in the aviation and automotive industry to check absence of runtime errors. While it supports programs using floating-point numbers, it cannot verify arbitrary correctness properties. Frama-C [36] has similar functionality but supports deductive verification. However, the properties supported are limited, e.g. that outputs lie in a given interval [37]. The situation is similar with other deductive verifiers like KeY [2], which can verify the absence of exceptional values like NaN and infinity [1].

## 2 Preliminaries

We now present the necessary background for the rest of the paper: we introduce Isabelle and the Isabelle Refinement Framework, followed by IEEE 754 floating-point numbers and Markov Decision Processes in Isabelle/HOL.

### 2.1 Isabelle/HOL

An *interactive theorem prover* (ITP) is a program that implements a formal mathematical system in which a user writes definitions and theorem statements, and constructs proofs from a set of axioms. To prove a theorem in an ITP, the user provides high-level steps, and the ITP fills in the details at the axiom-level.

We perform our formalization using the ITP Isabelle/HOL [44], which is a proof assistant for Higher-Order Logic (HOL). Roughly speaking, HOL can be seen as a combination of functional programming with logic. Isabelle is designed to be highly trustworthy: a small, trusted kernel implements the inference rules of the logic. Outside the kernel, a large set of tools implement proof automation and high-level concepts like algebraic data types. Bugs in these tools cannot lead to inconsistent theorems being proved, as the kernel refuses flawed proofs.

We aim to represent our formalization as faithfully as possible, but we have optimized the presentation for readability. The notation in Isabelle/HOL is similar to functional programming languages like ML or Haskell mixed with mathematical notation. Function application is written as juxtaposition: we write $f\ x_1\ \dots\ x_n$ instead of the standard notation $f(x_1,\ \dots\ , x_n)$. Recursive functions are defined using the **fun** keyword and pattern matching. For partial functions, we use the notation $f = (\lambda x \in X.\ g\ x)$, to explicitly restrict the domain of the function to $X$. Where required, we annotate types as $x :: type$.

Isabelle/HOL provides a keyword **locale** to define a named context with assumptions, e.g. an MDP with well-formedness assumptions [8]. Locales can be interpreted and extended in different contexts, e.g. a locale for MDPs can be instantiated for a specific MDP, which yields all theorems from within that locale.

### 2.2 Isabelle Refinement Framework

Our verification of II spans the mathematical foundations of MDPs, the implementation of optimized algorithms and data structures, and the low-level LLVM intermediate language [40]. To keep the verification effort manageable, we use a stepwise refinement approach: starting with an abstract specification, we incrementally add implementation details, proving that each addition preserves correctness, e.g. computing a fixed-point by iteration, or implementing MDPs by a sparse-matrix data structure. The former specifies the control-flow of a program, but the datatype remains the same. The latter we call *data refinement*.

This approach is supported by the Isabelle Refinement Framework (IRF) [39]. In the IRF, we define algorithms in the *nondeterministic result (nres) monad*, where a program either fails or produces a set of results. The notation $a \leq_R c$ denotes that every (non-deterministic) output of abstract program $a$ is related to an output of concrete program $c$ via the *refinement relation* $R$. In other words, $c$ is an implementation of $a$. If a refinement step does not involve data refinement, then we use $\leq := \leq_{R_{id}}$ where $R_{id}$ is the identity relation.

At the start of the refinement chain, we put our specification and at the end of the refinement chain, we aim to have an efficient LLVM program. Once sufficiently refined, the sepref [38] tool can automatically refine a program to LLVM. As $\leq_R$ is transitive, the LLVM program satisfies the specification.

### 2.3 Floating-Point Arithmetic

Our work uses the formalization of the IEEE 754 floating-point standard in Isabelle/HOL [58]. This library provides a generic type *(e,f) float* which resembles the scientific notation: $e$ is the number of bits for the *exponent*, and $f$ is the number of bits for the *fraction* (also known as mantissa).

We use the type *double = (11,52) float*. The floating-point format contains positive and negative numbers, as well as special values for $\pm\infty$ and *not a number (NaN)*. The function *valof :: (e,f) float → ereal* maps non-NaN floating-point numbers to *extended real numbers*, i.e., $\mathbb{R}\dot{\cup}\{-\infty, +\infty\}$. Finally, the formalization provides all standard floating-point instructions like addition, multiplication, and comparisons as well as intuitive predicates to identify special cases (e.g. *is_nan*).

### 2.4 Markov Decision Processes

*Markov Decision Processes* (MDPs) are widely used to model probabilistic systems with nondeterministic choices [45], e.g. in PMC, planning, operations research and reinforcement learning [6,53]. Intuitively, an *agent* interacts with an environment by choosing *actions* that, together with random elements, influence the *state* of the system. The agent has an *objective*, e.g. to reach certain states, and aims to choose actions that optimize the probability to achieve the objective. Many important concepts defined in this section are illustrated in Example 1.

Formally, a finite MDP is a pair $M = (S, K)$ where $S$ is a finite, non-empty set of *states*, and $K : S \to 2^{\mathcal{P}(S)}$ is the transition kernel. It maps every state to a finite, non-empty set of *actions* in the form of transition probabilities. $\mathcal{P}(S)$ denotes the set of probability measures on $S$, i.e. functions $p : S \to [0, 1]$ where $\sum_{s\in S} p(s) = 1$. Furthermore for $K$ is closed under $S$: Actions from $S$ lead to $S$.

Our formalization of II extends the *Markov Models* [33,34] formalization from the Isabelle/HOL library. MDPs are modeled with a generic type *'s mdpc* and a locale *Finite_MDP* that, in combination, contain the states, the transition kernel and well-formedness conditions (Locale 2.1). In the following, we abbreviate the projections *states M* and *actions M* as $S$ and $K$. The type of the states is *'s*, and the type of probability distributions over *'s* is *'s pmf*. For a *p :: 's pmf*, $set_{pmf}\ p$ denotes its support, i.e. the set of states with non-zero probability.

**locale** *Finite_MDP =* *(Locale 2.1)*
**fixes** *M :: 's mdpc* **and** $S$ **and** $K$
**defines** $S$ *= states M* **and** $K$ *= actions M*

**assumes** $S \neq \varnothing$ **and** *finite* $S$ **and** $\forall s.\ K\ s \neq \varnothing$ **and** $\forall s \in S.$ *finite* $(K\ s)$
**assumes** $\forall s \in S.\ (\bigcup a \in K\ s.\ set_{pmf}\ a) \subseteq S$

*Strategies* choose an action based on the visited states. This formalization works with *configurations*, which are pairs of states and strategies. A configuration is *valid* if the strategy selects only enabled actions and the state of the configuration is in $S$. $valid_{cfg}$ denotes the set of all valid configurations. Given a configuration and an MDP, the probability space of infinite traces $T\ cfg$ is constructed from the induced Markov Chain, where each state is a configuration.

MDP subcomponents play an important role in the analysis of II. A *sub-MDP* $M' = (S', K')$ consists of a subset of states $S' \subseteq S$ and a restricted kernel $K'$ where $\forall s \in S'.\ K'(s) \subseteq K(s)$. A sub-MDP is *strongly connected* if all states can reach each other via a sequence of actions. A closed, strongly connected sub-MDP is an *end component* (EC). A *maximal end component* (MEC) is an EC that is not a sub-MDP of another EC. Finally, *trivial MECs* are MECs with one state and no actions, and *bottom MECs* are MECs without an exit.

*Reachability.* In our PMC setting, the objective is to minimize or maximize the long-term reachability probabilities of a set of target states $U \subseteq S$. The value function $P_{cfg} :: 's \Rightarrow real$ gives the probability of reaching $U$ in the Markov Chain induced by the configuration $cfg$. Minimal and maximal reachability probabilities are denoted by $P_{\inf}$ and $P_{\sup}$ respectively. $P_{\inf}$ is defined as the infimum of $P_{cfg}$ over all valid configurations, $P_{\sup}$ is defined using the supremum. We also introduce the *Bellman optimality operators* $F_{\inf}$ and $F_{\sup}$ (Def. 2.1, $F_{\sup}$ omitted). For a state $s \in S$ and a value vector $v$, $F_{\inf}\ v\ s$ denotes the minimal expected value of $x$ after one step from $s$. The symbol $\bigsqcap$ denotes the infimum.

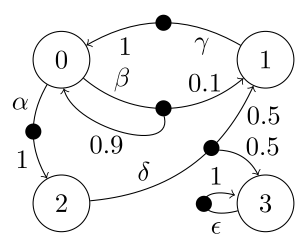


**Fig. 1.** A simple MDP with four states and five actions.

**definition** $F_{\inf}\ v = (\lambda s \in S.$ **if** $s \in U$ **then** *1* **else** $\bigsqcap a \in K\ s.\ \sum t \in set_{pmf}\ a.\ v\ t * pmf\ a\ t)$ *(Def. 2.1)*

The *least fixed point* (*lfp*) of $F_{\inf}$ is $P_{\inf}$. In other words, repeatedly applying $F_{\inf}$ to a lower bound of $P_{\inf}$ converges to $P_{\inf}$ in the limit. For II, we preprocess the MDP such that the *greatest fixed point* (*gfp*) of $F_{\inf}$ is the same as $P_{\inf}$, and then iterate $F_{\inf}$ on both a lower and an upper bound until they closely approximate $P_{\inf}$. The same holds for $F_{\sup}$ and $P_{\sup}$.

*Example 1.* Fig. 1 shows an MDP with four states, $S = \{0, 1, 2, 3\}$. The outgoing transitions from each state represent the actions in the MDP. Each transition leads to a black dot and branches into the successor states with corresponding probabilities. For example in state 0, $K(0) = \{\alpha, \beta\}$. The agent can choose $\alpha$ to move to state 2, or $\beta$ to have a 10% chance to move to state 1.

Let the target states $U = \{3\}$. The reachability probabilities are $P_{\inf}(2) = 0.5$ and $P_{\sup}(2) = 1$. The MDP has a single bottom MEC $\{3\}$ with action $\{\epsilon\}$, and a single trivial MEC $\{2\}$. The states $\{0, 1\}$ form a MEC with actions $\beta$ and $\gamma$.

## 3 Interval Iteration in Isabelle/HOL

The *interval iteration (II)* algorithm for MDPs is an iterative solution method for reachability problems based on value iteration. In contrast to standard value iteration, there is a simple and sound stopping criterion. We present our formalization of definitions and correctness proofs in Isabelle/HOL of II and preprocessing routines. Our formalization is based on the proofs in [22], we highlight the challenges encountered during formalization and point out differences in our formal proofs. In particular, we present a more elegant and much more precise proof of [22, Proposition 3]. Moreover, we simplify the definitions of the preprocessing steps significantly. In the following, all statements prefixed with **lemma** or **theorem** are formally verified in Isabelle/HOL. All theorems and definitions for $P_{\text{sup}}$ that are analogous to the ones for $P_{\text{inf}}$ are omitted here for brevity.

### 3.1 The Interval Iteration Algorithm

The idea of the II algorithm is to start with a lower and an upper bound on the true reachability probability and iterate the Bellman optimality operator $F_{\text{inf}}$ ($F_{\text{sup}}$) on both. Since the optimality operators are monotone, both sequences converge to a fixed point. On arbitrary MDPs, these fixed points are not necessarily the same. However, if the MDP is preprocessed to only contain MECs that are trivial or bottom MECs, both fixed points are equal to the optimal reachability probabilities.

For now, assume an arbitrary MDP with a single target state $s_+$ and an avoid state $s_-$, that are both sinks. As the initial lower bound $lb_0$, we take the function that assigns 1 to $s_+$ and 0 to all other states. The initial upper bound $ub_0$ assigns 0 to $s_-$ and 1 to all other states. The II algorithm computes the sequences $lb_{\text{inf}}\ n$ and $ub_{\text{inf}}\ n$, defined as the $n$-fold application of $F_{\text{inf}}$ to $lb_0$ and $ub_0$:

**definition** $lb_{\text{inf}}\ n = (F_{\text{inf}})^n\ lb_0$ **and** $ub_{\text{inf}}\ n = (F_{\text{inf}})^n\ ub_0$ *(Def. 3.1)*

It is an immediate consequence of the monotonicity of the Bellman optimality operators that the lower (upper) bounds are monotonically increasing (decreasing). Clearly, $lb_0$ is a lower bound and $ub_0$ is an upper bound of $P_{\text{inf}}$. Additionally, we formally derive that the Bellman optimality operators preserve upper and lower bounds in Lemma 3.1. These two properties of the abstract II algorithm are required for the refinement proof in Sect. 4.

**lemma** **assumes** $v \leq P_{\text{inf}}$ **shows** $F_{\text{inf}}\ v \leq P_{\text{inf}}$ *(Lemma 3.1)*
**lemma** **assumes** $v \geq P_{\text{inf}}$ **shows** $F_{\text{inf}}\ v \geq P_{\text{inf}}$

### 3.2 Reduced MDPs

II is only guaranteed to converge if the MDP only contains trivial or bottom MECs. We therefore need to preprocess the MDP before applying II. The preprocessing steps differ for $P_{\text{inf}}$ and $P_{\text{sup}}$, they are called *min-reduction* and *max-reduction*. In a first step, we extend the existing MDP formalization [33] with *strongly connected components* (SCCs) and bottom MECs (Def. 3.2). The states of an MDP that form trivial or bottom MECs are called *trivials* or *bottoms* respectively. We follow [22] and call an MDP *reduced* if all of its MECs are either trivial or bottom MECs.

**definition** *bmec M b* = *(Def. 3.2)*
*mec M b* $\wedge$ ($\forall s \in$ *states b. actions b s* = *K s*)

*Min-Reduction.* The min-reduction for MDPs transforms an arbitrary MDP into a reduced MDP with the same $P_{\text{inf}}$. The idea is that all non-trivial MECs (except $s_+$) can reach the target with probability 0: There exists a strategy that stays in the MEC forever and therefore never reaches $s_+$. Hence, all such MECs may be collapsed into a single absorbing state $s_-$. To formalize this transformation, we first define a function $red_{\text{inf}}$ (Def. 3.3) mapping the states, and then apply it to the MDP $M$ to obtain the reduced MDP $M_{\text{inf}}$ (Def. 3.4). Our formal definition is a substantial simplification compared to [22, Def. 4].

**definition** $red_{\text{inf}}\ s = $ **if** $s \in$ *trivials* $M \cup \{s_+\}$ **then** $s$ **else** $s_-$ *(Def. 3.3)*

**definition** $M_{\text{inf}} = $ *fix_loop* $s_-$ ($map_{mdpc}$ $red_{\text{inf}}$ $M$) *(Def. 3.4)*

The function $map_{mdpc}$ (defined in [34]) applies a function to every state of an MDP. If the function merges states, $map_{mdpc}$ merges the action sets. Finally, *fix_loop* $s_-$ replaces the actions at $s_-$ with a single self-loop. Fig. 2 displays the min-reduced version of the MDP from Fig. 1.

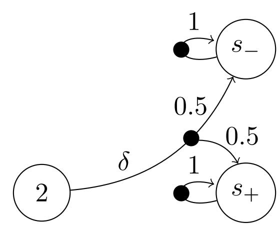

**Fig. 2.** Min-reduced version of the MDP in Fig. 1.

The formal proof of the fact that $M_{\text{inf}}$ is in fact a reduced MDP follows the original [21]. We also show that the transformation preserves $P_{\text{inf}}$. To distinguish the reachability probabilities of the original and the reduced MDP, we use the notation $P_{\text{inf}}$ for the original MDP and $M_{\text{inf}}.P_{\text{inf}}$ for the reduced MDP. Our proof is based on the fact that min-reduction preserves the finite-horizon probabilities $P^{\leq}_{\text{inf}}\ n$ (Lemma 3.2), i.e. the reachability probability in $n$ steps. Now, the main claim (Thm. 3.1, [22, Proposition 3]) is a direct consequence. Note that our proof is simpler and more precise than the original: in [22], the authors only claim without details that for every strategy in the reduced MDP, there exists a strategy in the original MDP with the same $P_{\text{inf}}$ and vice versa.

**lemma assumes** $s \in S$ **shows** $P^{\leq}_{\text{inf}}\ n\ s = M_{\text{inf}}.P^{\leq}_{\text{inf}}\ n\ (red_{\text{inf}}\ s)$ *(Lemma 3.2)*

**theorem assumes** $s \in S$ **shows** $P_{\text{inf}}\ s = M_{\text{inf}}.P_{\text{inf}}\ (red_{\text{inf}}\ s)$ *(Thm. 3.1)*

*Max-Reduction.* The $P_{\text{sup}}$ case can be handled similarly with a max-reduction. The procedure is more involved, as not all non-trivial MECs can be collapsed while preserving $P_{\text{sup}}$: A maximizing strategy might choose to leave a non-trivial MEC. We can, however,

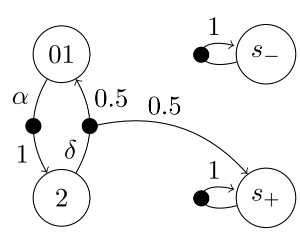

**Fig. 3.** Max-reduced version of the MDP in Fig. 1.

first collapse each MEC into a single state to obtain an MDP $M_{MEC}$. In a second step, we map the bottom MECs to $s_-$. Finally, we remove self-loops at all states except $s_+$ and $s_-$. Our formalization decomposes max-reduction [22, Def. 5] into multiple steps. This again simplifies both the definitions and proofs.

Collapsing the MECs into a single state is done by the function *the_mec*, the transformation preserves $P_{\sup}$ (Thm. 3.2). Our proof resembles the proof of [18, Theorem 3.8], however we have to work around the fact that the MDP formalization [33] only supports deterministic policies. Note that every state of $M_{MEC}$ now forms its own MEC. The correctness proof of the second phase of the reduction is similar to min-reduction. See Fig. 3 for the max-reduced version of the MDP from Fig. 1.

**theorem** **assumes** $s \in S$ *(Thm. 3.2)*
**shows** $P_{\sup}\ s = M_{MEC}.P_{\sup}\ (\mathit{the\_mec}\ M\ s)$

*Proof.* ($\leq$) As collapsing MECs only shortens paths in the MDP, the proof proceeds similarly to the one for min-reduction via finite-horizon probabilities.

($\geq$) Consider an optimal strategy $\pi_{MEC}$ in $M_{MEC}$, we need to show that there exists a strategy in $M$ with the same reachability probability. Every MEC $m$ of $M$ contains a state $s_m^\pi$ where the action selected by $\pi_{MEC}$ is enabled. Moreover, within a MEC, we can obtain a deterministic, memoryless strategy $\pi_m$ that reaches this state with probability 1. Thus we can construct a strategy in $M$ that behaves like $\pi_m$ within each MEC until $s_m^\pi$ is reached and then follows $\pi_{MEC}$. The reachability probability of this strategy in $M$ is the same as in $M_{MEC}$.

### 3.3 Reachability in Reduced MDPs

From now on we assume that we are working with a reduced, finite MDP $M$ where each state is a trivial or bottom MEC. We show that in such an MDP, over time any strategy reaches a bottom MEC almost surely. This is the key property that will then allow us to prove the convergence of II.

*Level Graph.* First, we build a level graph of the MDP, starting at the bottom MECs (Def. 3.5). At level $n+1$, we add all those states where every action has a successor in level $n$ or below. We define $I$ to be the greatest non-empty level of the level graph $G$, so $I$ is the number of steps that allows us to reach a bottom MEC from every state. We formally show that $G$ has the desired properties, i.e. it is acyclic and contains every state at exactly one level. The proofs in Isabelle/HOL require substantial reasoning about graph-theoretic properties, e.g. we need to show that every MDP contains a bottom MEC.

**fun** $G$ **where** *(Def. 3.5)*
$G\ 0 = \mathit{bottoms}\ M$
$G\ (n+1) = \textbf{let}\ G_{\leq n} = \bigcup i \leq n.\ G\ i\ \textbf{in}$
$\{s \in S \setminus G_{\leq n}.\ \forall a \in K\ s.\ G_{\leq n} \cap a \neq \varnothing\}$

*Reachability of BMECs.* We now show that intuitively, every strategy eventually descends through the levels of $G$. The rate at which a bottom MEC is encountered depends on $\eta$, the smallest probability of any transition in the MDP. At every step, the probability of descending a level wrt. $G$ is at least $\eta$. Hence we can show that for any valid configuration, the probability to reach the bottom MECs in $I$ steps is no less than $\eta^I$:

**lemma** **assumes** $cfg \in valid_{cfg}$ **shows** $\eta^I \leq P^{\leq}_{cfg}\ I$ *(Lemma 3.3)*

The value $P^{\leq}_{cfg}\ n$ denotes the finite-horizon reachability probability of the bottom MECs in $n$ steps under configuration *cfg*. Note that the lemma was originally stated for safety instead of reachability problems. We transform it using the well-known equivalence $\mathbb{P}^{\leq n}_{\pi}(\Diamond U) = 1 - \mathbb{P}^{\leq n}_{\pi}(\Box \neg U)$. For multiples $n$ of $I$, we obtain a stronger lower bound of $1 - (1 - \eta^I)^n$ (Thm. 3.3, [22, Proposition 1]). As $n$ increases, $(1 - \eta^I)^n$ converges to 0 and thus $P^{\leq}_{cfg}\ nI$ tends towards 1. Since we chose *cfg* arbitrarily, we almost surely reach a bottom MEC in the limit.

**theorem** **assumes** $cfg \in valid_{cfg}$ **shows** $1 - (1 - \eta^I)^n \leq P^{\leq}_{cfg}\ nI$ *(Thm. 3.3)*

### 3.4 Convergence of Interval Iteration

We assume a special form of reduced MDPs, where the only bottom MECs are the target state $\{s_+\}$ and avoid state $\{s_-\}$ that both are absorbing (Locale 3.1). The reduced MDPs from Sect. 3.2 are instances of this locale.

**locale** *MDP_Reach* = *Finite_MDP M* + *(Locale 3.1)*
  **assumes** $s_- \in S$ **and** $s_+ \in S$ **and**
    $\forall s \in S \setminus \{s_+,\ s_-\}.\ s \in \mathit{trivials}\ M$ **and**
    $K\ s_- = \{\mathit{return}_{pmf}\ s_-\}$ **and** $K\ s_+ = \{\mathit{return}_{pmf}\ s_+\}$

Towards a convergence proof of II, we show that the lower and upper bound sequences relate to finite-horizon reachability probabilities of both $s_+$ and $s_-$ (Lemma 3.4, [22, Lemma 4]). In this section, we indicate the target sets explicitly.

**lemma** **assumes** $s \in S$ *(Lemma 3.4)*
  **shows** $lb_{\inf}\ n\ s = P^{\leq}_{\inf}\ \{s_+\}\ n\ s$ **and** $ub_{\inf}\ n\ s = 1 - P^{\leq}_{\sup}\ \{s_-\}\ n\ s$

With Thm. 3.3 we can bound the distance between the sequences (Thm. 3.4). Note that convergence is in general only guaranteed if all computations are carried out with arbitrary precision arithmetic. In a floating-point setting, the convergence to a unique fixed point is not guaranteed. Still, this theoretical result motivates the usage of the II algorithm to optimally solve reachability problems on MDPs. In practice, on most instances the algorithm converges much faster than the theoretical bound suggests (see Sect. 5 for experimental results).

Finally, like [22], the theorem does not apply if all probabilities in the MDP are equal to one, i.e. no branching after an action is selected. In this case, the MDP is deterministic and is better solved with qualitative solution methods.

**theorem** *(Thm. 3.4)*
**assumes** $s \in S$ **and** $\epsilon > 0$ **and** $\eta \neq 1$ **and** $n \geq \lceil \log_{(1-\eta^I)} \epsilon \rceil * I$
**shows** $ub_{\inf}\ n\ s - lb_{\inf}\ n\ s \leq \epsilon$

*Proof.* As a first step, we show for all $i$:

$$
\begin{aligned}
ub_{\inf}\ iI\ s - lb_{\inf}\ iI\ s &= 1 - P^{\leq}_{\sup}\ \{s_-\}\ iI\ s - P^{\leq}_{\inf}\ \{s_+\}\ iI\ s && \textit{(Lemma 3.4)}\\
&\leq 1 - (P^{\leq}_{\inf}\ \{s_-\}\ iI\ s + P^{\leq}_{\inf}\ \{s_+\}\ iI\ s) && (P^{\leq}_{\inf} \leq P^{\leq}_{\sup})\\
&= 1 - P^{\leq}_{\inf}\ \{s_-, s_+\}\ iI\ s && \textit{(Disjoint events)}\\
&\leq (1 - \eta^I)^i && \textit{(Thm. 3.3)}
\end{aligned}
$$

Set $i = \lceil \log_{(1-\eta^I)} \epsilon \rceil$ and the theorem follows from monotonicity.

## 4 Refinement using Floating-Point Arithmetic

In the next step, we use the IRF to refine the abstract specification of II to an efficient LLVM implementation. In our executable version, we implement real numbers using IEEE 754 double precision floating-point numbers (floats). Due to dedicated hardware support on modern consumer processors, floats have superior performance compared to any other decimal format. However, the rounding errors inherent to floating-point arithmetic make the refinement tricky. We propose an approach based on directed rounding modes to refine reals to *upper bounding* or *lower bounding* floats. A further challenge is that during II, the rounding mode needs to be switched regularly. Most consumer CPUs set the rounding mode through a global flag, which is both time-consuming at runtime [26] and cumbersome to reason about in the IRF. To circumvent this, we use the AVX512 instruction set which supports operation-specific rounding modes.

We first introduce the sepref tool for refinement to LLVM, after which we present our extension of the IRF and sepref for floats. We use that to obtain correct-by-construction LLVM code for II.

### 4.1 The Sepref Tool

We use sepref [38] to automatically refine an algorithm to an LLVM program. The sepref tool provides a library of verified, reusable data structures. As these data structures may access the *heap*, we need to use *(separation logic) assertions* that extend refinement relations with a heap.

*Example 2.* We demonstrate how to automatically refine to LLVM using sepref through the following example.

**definition** $ls\_app\ x\ xs = xs\ @\ [x]$ **and** $half\_nat\ n = n\ div\ 2$
**definition** $apphalf\ n\ xs =$ **do** { **let** $n' = half\_nat\ n;$ **return** $ls\_append\ n'\ xs$ }
**lemma** $(rshift1, half\_nat) :: A_{size} \rightarrow A_{size}$
**lemma** $(arl\_app, ls\_app) :: [\lambda n\ xs.\ length\ xs < 2^{63}-1]\ A_{size} \rightarrow A^d_{arl} \rightarrow A_{arl}$
**lemma** $(arl\_apphalf, apphalf) :: [\lambda n\ xs.\ length\ xs < 2^{63}-1]\ A_{size} \rightarrow A^d_{arl} \rightarrow A_{arl}$

Line 1 defines *ls_app* (insert an element at the end of a list) and *half_nat* (divide a natural number in half). Both are used in the definition of *apphalf* in Line 2. The standard library of sepref has LLVM implementations for lists as array lists ($A_{arl}$), and natural numbers as 64-bit signed words ($A_{size}$). The $A$ indicates that these are assertions. For example, *half_nat* can be refined with the LLVM program *rshift1*, which performs an efficient bit shift to the right.

For sepref to use such refinements, they need to be in *parametric functor notation* as in Line 3. The lemma states that *rshift1* and *half_nat* are related as follows: if the inputs of *half_nat* (type *nat*) and *rshift1* (type *size*) are related via $A_{size}$, then the outputs are related via $A_{size}$. Line 4 provides a refinement of the append operation following the same principle, only now with two inputs. Moreover, the precondition *length xs* $< 2^{63} - 1$ limits the length of the input list. Finally, the superscript $^d$ indicates that the input is destructively updated. With all these rules registered to sepref, we can automatically refine *app_half*. We provide a signature so that sepref knows which data structure to use:

$$[\lambda n\ xs.\ length\ xs < 2^{63} - 1]\, A_{size} \rightarrow A^d_{arl} \rightarrow A_{arl}.$$

sepref automatically translates this into the LLVM program *arl_app_half* based on *rshift1* and *arl_append*. We also obtain the refinement relation in Line 5.

### 4.2 Floating-Point Extension of the Isabelle Refinement Framework

We extend the IRF with two data refinements to reason about floating-point arithmetic: real numbers to lower bounding (*lb*) or upper bounding (*ub*) floats. Since the *ub* case is mostly symmetric to the *lb* case, we focus on *lb* floats. We aim to construct a refinement relation that never produces NaN for the operations we support. NaN is incomparable, rendering it incompatible with a framework that reasons about bounds. Furthermore, the operations we support must preserve upper/lower bounds: the float $-2_f$ (subscript $f$ denotes floats) is a lower bound of 1, yet $-2_f * -2_f = 4_f$ is not a lower bound of $1 * 1 = 1$.

To resolve this, we only consider non-negative floats. We define the refinement relation $R_{lb} = \{(fl,r).\ valof\ fl \le r \wedge \neg is\_nan\ fl \wedge valof\ fl \ge 0\}$ which relates reals to *lb* floats, e.g. $(2_f, 3) \in R_{lb}$, but $(-2_f, -1) \notin R_{lb}$, as $-2_f$ is negative. Since floats are *pure*, e.g. they do not need to allocate memory on the heap. This means that the assertion $A_{lb}$ ignores the heap and is in essence $R_{lb}$.

We now present a non-exhaustive list of operations supported by our framework. We focus on the operations required for our use-case.

*Fused multiply-add.* The ternary operation $fma\ a\ b\ c = a * b + c$ (fused multiply-add) yields a smaller floating-point rounding error compared to separately multiplying and then adding. We name the AVX512 operation for *fma* with rounding mode *to_negative_infinity* *fma_avx_lb* and prove the following refinement:

**lemma** (*fma_avx_lb, fma*) :: $A_{lb} \rightarrow A^{\geq 0}_{lb} \rightarrow A_{lb} \rightarrow A_{lb}$ *(Lemma 4.1)*

This lemma says that if the inputs are *lb*, not NaN and non-negative, the output is also *lb*, not NaN and non-negative. Note that the result of $0_f * \infty_f$ is NaN. We resolve this with the stricter assertion $A_{lb}^{>0}$ that only allows positive finite floats. In the case of II, these are the transition probabilities from the input model.

*Comparison.* Comparisons are not preserved among equally bounding floats since floating-point errors can stack up arbitrarily. We can only preserve information partially by implementing them as mixed operations (e.g. comparing *lb* to *ub*).

**definition** *leq_sound a b* = **spec** ($\lambda r.\ r \longrightarrow a \leq b$) *(Lemma 4.2)*
**lemma** (*leq_double, leq_sound*) :: $A_{ub} \rightarrow A_{lb} \rightarrow A_{bool}$

We use **spec** to introduce non-determinism as follows: the operation must return *False* if $a > b$ and can return anything otherwise. Consider the following 2 cases. Case 1: $4_f$ is a *ub* of 2, and $3_f$ is an *lb* of 5, we have $2 \leq 5$ but $4_f \not\leq 3_f$; Case 2: $3_f$ is a *ub* of 2, and $4_f$ is an *lb* of 5, we have $2 \leq 5$ and $3_f \leq 4_f$. So a single comparisons of reals has two valid outcomes in our refinement relation. Similarly, by swapping rounding modes we get the converse specification.

We implement subtraction as a mixed operator for similar reasons (omitted).

*Min and max.* It is possible to refine the minimum (*min*) and maximum (*max*) operations directly using comparisons. We define the following refinement:

**definition** *min_double fl1 fl2* = **if** *fl1* $\leq$ *fl2* **then** *fl1* **else** *fl2* *(Lemma 4.3)*
**lemma** (*min_double, min*) :: $A_{lb} \rightarrow A_{lb} \rightarrow A_{lb}$

This refinement holds despite the fact that a comparison does not reveal anything about the bounding float number. Consider the following case: $4_f$ is a lower bound of 5 and $3_f$ is a lower bound of 6. *min_double* $4_f$ $3_f$ = $3_f$ is a lower bound of *min* 5 6 = 5, even though the floating-point implementation returns the first argument, while the definition on reals returns the second argument. The refinement works analogously for *ub* with *min* and *lb*/*ub* with *max*.

*Constants.* We provide the obvious refinements for the real number constants 0 and 1, which can be exactly represented as floating-point numbers.

### 4.3 Refinement of Interval Iteration

Using our floating-point extension to the IRF, we derive an implementation of II of the abstract specification from Sect. 3 using floats and conservative rounding. The IRF allows us to reuse the correctness proofs of the abstract specification, and reason about the correctness of the implementation in isolation. Through this separation of concerns we avoid directly proving the floating-point implementation correct.

The plot in Fig. 4 shows a fictive run of II on both reals and their refinement to bounding floats.

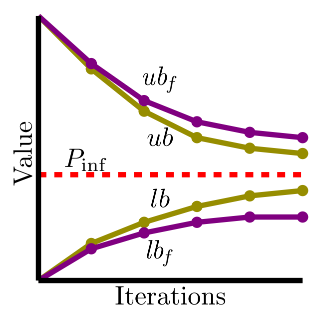


**Fig. 4.** The valuation of an MDP state over successive iterations: reals (green) vs. floats with safe rounding (violet). The dashed line marks the reachability probability.

In the long run, II converges to the dashed red line $P_{\text{inf}}$. The green lines denote the valuations of an MDP state using reals, $lb$ starting from $lb_0$ and $ub$ from $ub_0$. Implementing $lb$ with floats using $A_{lb}$ yields the violet line $lb_f$ (similarly for $ub$ using $A_{ub}$). Note that the deviations are exaggerated in this example. In practice, the errors are so small that no visual differences would appear in a to-scale plot.

Formally, the following specification states soundness of II, i.e. the outputs are lower and upper bounds of the reachability probability:

**definition** *ii_inf_spec M* = *(Def. 4.1)*
  **spec** $(\lambda(x,\ y).\ \forall s \in \mathit{states}\ M.\ x\ s \leq P_{\text{inf}}\ M\ s \wedge P_{\text{inf}}\ M\ s \leq y\ s)$

Despite the fact that convergence follows from Thm. 3.4, we have excluded it from the specification. As we will discuss in Sect. 4.5, this would yield a void statement for our implementation with floats. As a first step towards the refinement to LLVM, we define II in the nres-monad (the *sup* case is analogous):

1 **definition** *ii_gs_inf M L* = *(Def. 4.2)*
2   $x \leftarrow lb_0\ M;\ y \leftarrow ub_0\ M;\ i \leftarrow 0;\ \mathit{flag} \leftarrow \mathit{True};$
3   **while** $(i{+}{+} < L \wedge \mathit{flag})$ (
4     $(x,y) \leftarrow F^{gs}_{inf}\ M\ x\ y$
5     $\mathit{flag} \leftarrow$ **spec**$(\lambda x.\ \mathit{True})$)
6   **return** $(x,y)$

We define *ii_gs_inf* in Line 1. It takes as inputs an MDP $M$, and a maximal iteration count $L$ to guarantee termination. Line 2 initializes variables, most importantly the lower bounds $lb$ and upper bounds $ub$. The *flag* nondeterministically decides whether to abort the loop. This is sound, because *ii_inf_spec* is satisfied after any number of iterations. In each iteration, we first update the valuations according to a Gauss-Seidel variant of $F_{\text{inf}}$ (Line 4): we update $lb$ and $ub$ in-place, thereby we already use the updated values in the current iteration and converge faster.

The algorithm is now in a format ready for refinement proofs to LLVM. Using the setup from Sect. 3 and Lemma 3.1, it is straightforward to prove that the algorithm refines the specification:

**theorem** *ii_gs_inf M L* $\leq$ *ii_inf_spec M* *(Thm. 4.1)*

### 4.4 Refinement of the mcsta Data Structure

The motivation behind refining II to LLVM code is to embed it into the model checker mcsta from the Modest Toolset [27]. mcsta is an explicit-state probabilistic model checker that also supports quantitative model checking of MDPs. To avoid costly conversions of the MDP representation at runtime, we refine the *mdpc* data structure to the MDP data structure of mcsta. This is a two-step process: First, we do a data refinement from *mdpc* to the sparse-matrix representation used by mcsta based on HOL lists. Then, we use sepref to refine this data structure to LLVM. The sparse-matrix representation that we use is a 6-tuple:

$(St{::}nat\ list,\ Tr{::}nat\ list,\ Br{::}nat\ list,\ Pr{::}real\ list,\ u_a{::}nat,\ u_t{::}nat)$.

For each state, *St* contains an index to *Tr*, pointing to the first transition (action) of the state. Similarly, for each transition, *Tr* contains its first index *Br* and *Pr*, pointing to the first branch of the transition and its probability. Finally, *Br* contains the target of the branch, pointing to *St*. Additionally, $u_a$ and $u_t$ are the avoid and target states respectively. Example 3 illustrates this data structure.

*Example 3.* A possible representation of the MDP from Fig. 1 is $St = [0,2,3,4,5]$, $Tr = [0,1,3,4,6,7]$, $Br = [2,0,1,0,1,3,3]$, $Pr = [1.0,0.9,0.1,1,0.5,0.5,1.0]$. The index of the avoid and target state are stored in $u_a$ and $u_t$. State 0 has two actions: $\alpha$ and $\beta$ (transitions 0 and 1). Transition 1 (action $\beta$) has two branches, 1 and 2, that lead to state 0 and 1 with probability 0.9 and 0.1 respectively.

*Refinement Relation.* We relate the abstract MDP type *mdpc* to the concrete data structure of mcsta with the refinement relation $R_M$ (definition omitted). For example, $R_M$ contains a tuple of the MDP of Fig. 1 and Example 3 as well as with each other instance of the data structure in Sect. 4.4 along with the MDP it represents. These lists present in the Isabelle/HOL model of the data structure can be directly refined to arrays of 64-bit integers (signed for compatibility with mcsta). Through composition we obtain assertion $A_M$ that maps an abstract MDP to LLVM.

*Refinement of Operations.* We use sepref to refine the functions $lb_0$, $ub_0$, $F^{gs}_{inf}$ and *flag* that are used by II. Refining $lb_0$ and $ub_0$ to the concrete data structure is straightforward: we initialize an array and set entries to constants $0_f$ or $1_f$. The floating-point refinement of $F^{gs}_{inf}$ builds on *fma* and *min* (*max* for $F^{gs}_{sup}$) from Sect. 4.2. Finally, we implement *flag* as follows: we compare the upper and lower bound, and set the flag if the difference is less than $\varepsilon$, specified by the user. Using the above refinements for all operations in *ii_gs_inf*, we use sepref to obtain an LLVM algorithm *ii_gs_inf_llvm* within Isabelle/HOL.

### 4.5 Correctness Statement

For the final step in our proof, we combine all of our proofs into one theorem. We use the Hoare-triple because it allows is to write a concise correctness statement of a program while blending out all of the intermediary tools like the parametric functor notation. We show the correctness of program *ii_gs_inf_llvm* against the specification *ii_inf_spec*. The resulting triple combines Thm. 4.1 with the refinements from Sect. 4.4 through transitivity.

**theorem** *llvm_htriple* *(Thm. 4.2)*
$(A_{size}\ n\ n_i \star A_{size}\ L\ L_i \star A_{ub}\ \varepsilon\ \varepsilon_f \star A_M\ M\ M_i$
$\quad \star \uparrow(MDP\_Reach\ M \wedge n{+}1 < max\_size \wedge n = card\ (states\ M)))$
$(ii\_gs\_inf\_llvm\ L_i\ n_i\ \varepsilon_f\ M_i\ res_i)$
$(\lambda(lb_f,ub_f).\ \exists lb\ ub.\ A^{out}_{lb}\ lb\ lb_f \star A^{out}_{ub}\ ub\ ub_f$
$\quad \star \uparrow(\forall s \in states\ M.\ lb\ s \leq P_{\inf}\ M\ s \leq ub\ s))$

Lines 2 and 3 specify the preconditions, where Line 2 states the input data: $n$ and $L$ are natural numbers implemented as 64-bit words $n_i$ and $L_i$; $\varepsilon$ is a real number implemented as float $\varepsilon_f$ and $M$ is the MDP implemented as $M_i$. The separation conjunction $\star$ specifies that these implementations do not overlap on the heap. Line 3 is a boolean predicate lifted to separation logic using $\uparrow$. It states that $M$ satisfies locale *MDP_Reach* (Locale 3.1) and limits the number of states to the largest 64-bit number.

If these preconditions hold, a run of the algorithm (Line 4) yields the arrays $lb_f$ and $ub_f$ that satisfy the postconditions in Lines 5 and 6. These state that $lb_f$ is a lower-bound implementation of $lb$, which is in turn a lower bound of $P_{\text{inf}}$ (similarly for $ub$). Due to this last fact induced by the refinement, we cannot guarantee convergence of $lb_f$ and $ub_f$. However, we experimentally show that convergence is generally achieved with our implementation.

## 5 Experimental Evaluation

The LLVM code generator of the IRF exports the LLVM program *ii_gs_inf_llvm* for use in the LLVM compiler pipeline. Additionally, it generates a C header that makes it easy to embed the program into other software, such as mcsta.

*Integration with mcsta.* We integrate our verified implementation into mcsta, replacing the existing unverified interval iteration algorithm. This requires a small amount of unverified glue code to convert the MDP's probabilities into a floating-point representation: mcsta stores the probabilities in the MDP as 128-bit rationals (encoded as a pair of 64-bit integers representing the numerator and denominator). Our MDP data structure $M_i$ expects two floats per branch, representing lower and upper bounds of the rational probability. We convert the probabilities to 64-bit doubles by first converting the numerator and denominator to doubles and then performing two separate division operations, once rounding up and and once down. We assume that the input MDP as produced by the mcsta pipeline is well-formed. If there were a bug in the parser that produces MDPs that are not well-formed according to the precondition of the Hoare triple, we would lose the formal guarantees. As long as parts of the toolchain remain unverified, we rely on the correctness of the mcsta implementation for the preprocessing.

*Evaluation Questions.* We have formally verified in Isabelle/HOL that II with precise arithmetic computes lower and upper bounds ($lb$ and $ub$), and eventually converges to the reachability probability. However, two important questions remain: First, verified implementations tend to be orders of magnitude slower than unverified ones [50]. Since we verified an implementation with efficient numerics down to LLVM, we expect much faster runtimes. So how does our verified implementation compare to state-of-the-art tools in terms of performance? Second, the floating-point outputs ($lb_f$ and $ub_f$) provably provide conservative bounds. Can we experimentally confirm that the algorithm converges in practice?

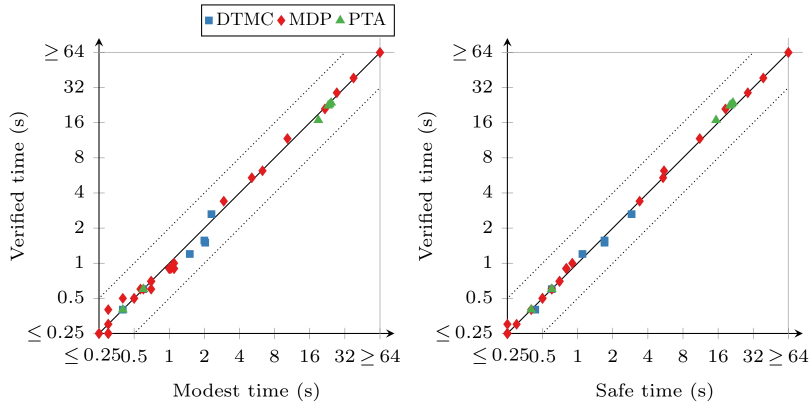


**Fig. 5.** Comparison of elapsed runtime for completing Interval Iteration

*Setup.* For the first question, we compare the runtime of our verified II implementation to its two unverified counterparts in mcsta: a C# implementation with standard rounding (*Modest* implementation) and a C implementation with safe rounding [26] (*Safe* implementation). The latter is similar to our verified LLVM implementation, also using AVX512 instructions for safe rounding.

We set the maximal iteration count to a high value ($10^7$) to ensure the computation is never terminated prematurely before convergence. While we anticipate all benchmarks to converge within fewer iterations, this upper limit provides a safeguard. We set the convergence threshold to $\varepsilon = 10^{-6}$. Once the lower and upper bound differ by less than $\varepsilon$, the *Verified*, *Safe*, and *Modest* implementations terminate. We use all DTMC, MDP and PTA models of the Quantitative Verification Benchmark Set (QVBS) [15] that contain $10^6$ to $10^8$ states and need at least two iterations to converge to $\varepsilon$. For our benchmarks, we consider both minimal and maximal reachability probabilities. In total, this yields a benchmark set of 49 benchmark instances. We execute all benchmarks on an Intel i9-11900K (3.5-5.3 GHz) system with 128 GB of RAM running Ubuntu Linux 22.04.

*Results.* Fig. 5 compares the runtimes of the three implementations. Each point $\langle x, y\rangle$ in a plot indicates that, on one benchmark instance, the implementation on the horizontal axis took $x$ seconds while the *Verified* implementation took $y$ seconds. We note that none of our benchmark instances reached the timeout of 10 minutes. The times are for running the II algorithm only and do not include other steps mcsta performs equally for all implementations such as state space exploration or min/max-reduction.

We see that the *Verified* implementation matches the unverified ones in terms of performance. Thus, as far as this benchmark set can show, the answer to the first question is that we have achieved comparable performance to a state-of-the-art unverified tool with a fully verified implementation. We also observed that the *Verified* implementation does not reach the maximal iteration count on any instance, i.e. it always converged up to $\varepsilon$, indicating that the second question can also be answered affirmatively.

Additionally, we did not find any significant discrepancies in the raw data output (Appendix A). The number of iterations to convergence is equal for all instances except for the Haddad-Monmege model [22] between *Verified* and *Modest*. The exception is not surprising, as this model is designed to converge very slowly, increasing the influence of floating-point errors. We also compared the computed results. Due to using different rounding modes, the results of *Verified* and *Modest* show differences well within $\varepsilon$. Despite the fact that *Verified* and *Safe* both implement safe rounding, their outputs still differ by minimal amounts on the order of $10^{-20}$. This may be caused by floating-point operations being non-associative, so a different order of operations may yield different outputs.

In terms of memory consumption, *Verified* and *Safe* use almost the same amount of RAM, but use on average 20% more RAM than *Modest*. Since the memory consumption is very similar for *Verified* and *Safe*, we suspect that these differences come from garbage collection effects caused by *Verified* and *Safe* being native code called from within a tool otherwise running in the managed C# runtime, as opposed to the purely-C# *Modest* implementation.

## 6 Discussion

We have formally verified the interval iteration algorithm in Isabelle/HOL. Our developments prove that the algorithm computes lower and upper bounds for the reachability probabilities (soundness) and converges to a single fixpoint (completeness). Furthermore, we show that soundness is preserved if we implement the algorithm using floating-point arithmetic with safe rounding. For this purpose, we used a principled refinement approach. We exploited the parametricity principle [56] of the IRF by consistently rounding our floating-point values in one direction. To make this practical, we equipped the sepref tool with reasoning infrastructure for floating-point numbers to generate an LLVM program. All our proofs culminate in a single statement, presented as a Hoare triple, leaving no gaps in the link between the specification and the implementation in LLVM.

Finally, we extract verified LLVM code from our formalization and embed it in the mcsta model checker of the Modest toolset. We experimentally verify that our implementation converges in practice and is competitive with manually implemented, optimised unverified counterparts. This is an important step towards a fully verified probabilistic model checking pipeline.

We also present our approach as an alternative to the bottom-up approach of building a verified model checker from scratch. With our top-down approach, the full functionality of the model checker is available to the user, possibly in cross-usage with verified components. Verified components are integrated with the model checker incrementally as drop-in replacements for unverified components, designed with competitive performance in mind.

*Verification vs. Certification for II.* An alternative to verifying II is to verify a *certifier* that, given the result from an unverified implementation of II plus a (compact) *certificate*, can efficiently check that the result is indeed correct. The advantage of certification is that the unverified implementation of II can be improved in an agile way independent of the certifier. Also since the certifier is (presumably) simpler than II, it should be easier to verify, including the verification of a high-performance implementation. One possible certification scheme for II is using the Park induction check of the optimistic value iteration algorithm [29], i.e. performing one iteration of II and checking if the lower/upper bound does not decrease/increase for any state. This could confirm that *some* fixed point lies between the bounds computed by II. This certification scheme would require as input (1) the final lower and upper bound for all states, (2) the full MDP, (3) *and a certificate that the MDP is reduced* that it also needs to check. Without the latter, one could not be sure that the certified fixed point corresponds to the reachability probability we want to compute (i.e. that it is the least fixed point). One challenge with this scheme is that, unlike our verified implementation of II, it cannot guarantee that *all* fixed points lie between the bounds. Also, since the entire reduced MDP is input to the certificate checker, the certificate checking performance might be fundamentally limited.

## References


1. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Deductive verification of floating-point java programs in KeY. In: Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. pp. 242–261 (2021). `https://doi.org/10.1007/978-3-030-72013-1_13`
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - the KeY Book - from Theory to Practice (2016)
3. Baier, C.: Probabilistic model checking. In: Dependable Software Systems Engineering, pp. 1–23. IOS Press (2016). `https://doi.org/10.3233/978-1-61499-627-9-1`
4. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Handbook of Model Checking, pp. 963–999. Springer (2018). `https://doi.org/10.1007/978-3-319-10575-8_28`
5. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Performance evaluation and model checking join forces. Communications of The Acm (9), 76–85 (2010). `https://doi.org/10.1145/1810891.1810912`
6. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
7. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In: 29th International Conference on Computer Aided Verification (CAV). pp. 160–180 (2017). `https://doi.org/10.1007/978-3-319-63387-9_8`
8. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: TYPES. pp. 34–50 (2003)
9. Barrett, G.: Formal methods applied to a floating-point number system. IEEE Transactions on Software Engineering (5), 611–621 (1989). `https://doi.org/10.1109/32.24710`

10. Bellman, R.: A Markovian decision process. Journal of Mathematics and Mechanics (5), 679–684 (1957)
11. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MoDeST: A compositional modeling formalism for hard and softly timed systems. IEEE Transactions on Software Engineering (10), 812–830 (2006). https://doi.org/10.1109/TSE.2006.104
12. Boldo, S., Melquiond, G.: Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In: 2011 IEEE 20th Symposium on Computer Arithmetic. pp. 243–252 (2011). https://doi.org/10.1109/ARITH.2011.40
13. Boldo, S., Munoz, C.: A high-level formalization of floating-point number in PVS. Tech. rep. (2006)
14. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: Quantitative model and tool interaction. In: 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 151–168 (2017). https://doi.org/10.1007/978-3-662-54580-5_9
15. Budde, C.E., Hartmanns, A., Klauck, M., Kretínský, J., Parker, D., Quatmann, T., Turrini, A., Zhang, Z.: On correctness, precision, and performance in quantitative verification (QComp 2020 competition report). In: 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA). pp. 216–241 (2020). https://doi.org/10.1007/978-3-030-83723-5_15
16. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In: Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. pp. 21–30 (2005). https://doi.org/10.1007/978-3-540-31987-0_3
17. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV). pp. 39–56 (2001). https://doi.org/10.1007/3-540-44804-7_3
18. de Alfaro, L.: Formal Verification of Probabilistic Systems. Ph.D. thesis, Stanford University, USA (1997)
19. De Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Transactions on Computers (2), 242–253 (2010). https://doi.org/10.1109/TC.2010.128
20. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. pp. 232–247 (2011). https://doi.org/10.1007/978-3-642-18275-4_17
21. Haddad, S., Monmege, B.: Reachability in MDPs: Refining convergence of value iteration. In: 8th International Workshop on Reachability Problems (RP). pp. 125–137 (2014). https://doi.org/10.1007/978-3-319-11439-2_10
22. Haddad, S., Monmege, B.: Interval iteration algorithm for mdps and imdps. Theoretical Computer Science pp. 111–131 (2018). https://doi.org/10.1016/J.TCS.2016.12.003
23. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Des. (2), 191–232 (2013). https://doi.org/10.1007/S10703-012-0167-Z

24. Harrison, J.: Formal verification at Intel. In: 18th Annual IEEE Symposium of Logic in Computer Science, 2003. pp. 45–54 (2003). https://doi.org/10.1109/LICS.2003.1210044
25. Harrison, J.: A Machine-Checked Theory of Floating Point Arithmetic. In: Theorem Proving in Higher Order Logics. pp. 113–130 (1999). https://doi.org/10.1007/3-540-48256-3_9
26. Hartmanns, A.: Correct probabilistic model checking with floating-point arithmetic. In: TACAS (2). pp. 41–59 (2022). https://doi.org/10.1007/978-3-030-99527-0_3
27. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 593–598 (2014). https://doi.org/10.1007/978-3-642-54862-8_51
28. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A practitioner's guide to MDP model checking algorithms. In: 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 469–488 (2023). https://doi.org/10.1007/978-3-031-30823-9_24
29. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: 32nd International Conference on Computer Aided Verification (CAV). pp. 488–511 (2020). https://doi.org/10.1007/978-3-030-53291-8_26
30. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 344–350 (2019). https://doi.org/10.1007/978-3-030-17462-0_20
31. Hartmanns, A., Kohlen, B., Lammich, P.: Fast verified sccs for probabilistic model checking. In: ATVA (1). pp. 181–202 (2023). https://doi.org/10.1007/978-3-031-45329-8_9
32. Hartmanns, A., Kohlen, B., Lammich, P.: Efficient formally verified maximal end component decomposition for mdps. In: FM (1). pp. 206–225 (2024). https://doi.org/10.1007/978-981-99-8664-4
33. Hölzl, J.: Markov chains and markov decision processes in isabelle/HOL. Journal of Automated Reasoning (3), 345–387 (2017). https://doi.org/10.1007/S10817-016-9401-5
34. Hölzl, J., Nipkow, T.: Markov models. Arch. Formal Proofs (2012)
35. Kamali, M., Katoen, J.P.: Probabilistic model checking of AODV. In: 17th International Conference on the Quantitative Evaluation of Systems (QEST). pp. 54–73 (2020). https://doi.org/10.1007/978-3-030-59854-9_6
36. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects Comput. (3), 573–609 (2015). https://doi.org/10.1007/S00165-014-0326-7
37. Kosmatov, N., Prevosto, V., Signoles, J.: Guide to Software Verification with Frama-c. Springer (2024)
38. Lammich, P.: Generating verified LLVM from isabelle/HOL. In: 10th International Conference on Interactive Theorem Proving (ITP). pp. 22:1–22:19 (2019). https://doi.org/10.4230/LIPICS.ITP.2019.22
39. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to hopcroft's algorithm. In: 3rd International Conference on Interactive Theorem Proving (ITP). pp. 166–182 (2012). https://doi.org/10.1007/978-3-642-32347-8_12
40. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: CGO. pp. 75–88 (2004)

41. Magron, V., Constantinides, G.A., Donaldson, A.F.: Certified roundoff error bounds using semidefinite programming. ACM Trans. Math. Softw. (4), 34:1–34:31 (2017). https://doi.org/10.1145/3015465
42. Miner, P.S.: Defining the IEEE-854 floating-point standard in PVS. Tech. rep. (1995)
43. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.A.: Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In: Computer Safety, Reliability, and Security. pp. 213–229 (2017). https://doi.org/10.1007/978-3-319-66266-4_14
44. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – a Proof Assistant for Higher-Order Logic. Springer (2002)
45. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley (1994). https://doi.org/10.1002/9780470316887
46. Quatmann, T., Katoen, J.P.: Sound value iteration. In: 30th International Conference on Computer Aided Verification (CAV). pp. 643–661 (2018). https://doi.org/10.1007/978-3-319-96145-3_37
47. Roberts, R., Lewis, B., Hartmanns, A., Basu, P., Roy, S., Chakraborty, K., Zhang, Z.: Probabilistic verification for reliability of a two-by-two network-on-chip system. In: 26th International Conference on Formal Methods for Industrial Critical Systems (FMICS). pp. 232–248 (2021). https://doi.org/10.1007/978-3-030-85248-1_16
48. Ruijters, E., Guck, D., Drolenga, P., Peters, M., Stoelinga, M.: Maintenance analysis and optimization via statistical model checking – evaluating a train pneumatic compressor. In: 13th International Conference on the Quantitative Evaluation of Systems (QEST). pp. 331–347 (2016). https://doi.org/10.1007/978-3-319-43425-4_22
49. Ruijters, E., Guck, D., van Noort, M., Stoelinga, M.: Reliability-centered maintenance of the electrically insulated railway joint via fault tree analysis: A practical experience report. In: 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 662–669 (2016). https://doi.org/10.1109/DSN.2016.67
50. Schäffeler, M., Abdulaziz, M.: Formally Verified Solution Methods for Infinite-Horizon Markov Decision Processes. In: The 37th AAAI Conference on Artificial Intelligence (AAAI) (2023). https://doi.org/10.1609/aaai.v37i12.26759
51. Schäffeler, M., Abdulaziz, M.: Formally Verified Approximate Policy Iteration. In: The 38th AAAI Conference on Artificial Intelligence (AAAI), To Appear (2025). https://doi.org/10.48550/arXiv.2406.07340
52. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. ACM Trans. Program. Lang. Syst. (1), 2:1–2:39 (2019). https://doi.org/10.1145/3230733
53. Sutton, R.S., Barto, A.G.: Reinforcement Learning – an Introduction. MIT Press (1998)
54. Titolo, L., Moscato, M., Feliu, M.A., Masci, P., Muñoz, C.A.: Rigorous Floating-Point Round-Off Error Analysis in PRECiSA 4.0. In: Formal Methods. pp. 20–38 (2025). https://doi.org/10.1007/978-3-031-71177-0_2
55. Vajjha, K., Shinnar, A., Trager, B.M., Pestun, V., Fulton, N.: CertRL: Formalizing convergence proofs for value and policy iteration in Coq. In: The 10th International Conference on Certified Programs and Proofs (CPP). pp. 18–31 (2021). https://doi.org/10.1145/3437992.3439927

56. Wadler, P.: Theorems for Free! In: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA 1989, London, UK, September 11-13, 1989. pp. 347–359 (1989). https://doi.org/10.1145/99370.99404
57. Wimmer, R., Kortus, A., Herbstritt, M., Becker, B.: Probabilistic model checking and reliability of results. In: 11th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS). pp. 207–212 (2008). https://doi.org/10.1109/DDECS.2008.4538787
58. Yu, L.: A formal model of IEEE floating point arithmetic. Arch. Formal Proofs (2013)

# A Appendix: Raw Data Analysis

We discuss the raw data of our experiments in our results section (Sect. 5). You can find the results of our experiments and the script that we used to analyze it in the artifact that we submitted along with this article. The artifact also contains everything that you need to run the experiments on your own Linux machine (support for AVX512 required). This appendix contains the output of the raw data analysis as you will obtain it by running our data analysis tool.

## A.1 Iteration count

*Comparing Verified and Safe* Identified 0 discrepancies in the iteration counts.

*Comparing Verified and Modest* Identified 1 discrepancy in the iteration counts:

| Name | #Verified | #Modest | Difference |
|---|---|---|---|
| haddad-monmege.20-0.7.target.mcsta.lp | 6879937 | 6879892 | 45 |

## A.2 Reachability Probabilities

*Comparing Verified and Safe.* Identified 11 discrepancies in the reachability probabilities:

| Name | P Verified | P Safe | Difference |
|---|---|---|---|
| zeroconf.1000-8-False.correct_max.mcsta.lp | 7.56652e-07 | 7.56652e-07 | 6.35275e-22 |
| echoring.100.MaxOffline1.mcsta.lp | 1.82736e-06 | 1.82736e-06 | 6.35275e-22 |
| echoring.100.MaxOffline2.mcsta.lp | 1.71838e-06 | 1.71838e-06 | 1.05879e-21 |
| echoring.50.MaxOffline2.mcsta.lp | 1.21025e-06 | 1.21025e-06 | 6.35275e-22 |
| echoring.100.MaxOffline3.mcsta.lp | 1.66364e-06 | 1.66364e-06 | 1.48231e-21 |
| echoring.50.MaxOffline1.mcsta.lp | 1.35879e-06 | 1.35879e-06 | 1.05879e-21 |
| echoring.50.MaxOffline3.mcsta.lp | 1.17584e-06 | 1.17584e-06 | 4.23516e-22 |
| zeroconf.20-8-False.correct_max.mcsta.lp | 1.63423e-07 | 1.63423e-07 | 1.32349e-22 |
| brp-pta.64-12-32-256.P_3.mcsta.lp | 2.4325e-07 | 2.4325e-07 | 1.05879e-21 |
| brp-pta.64-12-32-256.P_2.mcsta.lp | 2.72208e-07 | 2.72208e-07 | 1.05879e-21 |
| brp-pta.64-12-32-256.P_1.mcsta.lp | 2.72208e-07 | 2.72208e-07 | 1.21761e-21 |

*Comparing Verified and Modest.* Identified 37 discrepancies in the reachability probabilities:

| Name | P Verified | P Modest | Difference |
|---|---:|---:|---:|
| brp.2048-64.p2.mcsta.lp | 9.74704e-07 | 9.74704e-07 | 1.07785e-18 |
| haddad-monmege.20-0.7.target.mcsta.lp | 0.7 | 0.7 | 7.01599e-11 |
| brp.2048-64.p1.mcsta.lp | 9.74704e-07 | 9.74704e-07 | 1.08674e-18 |
| crowds.6-15.positive.mcsta.lp | 0.128655 | 0.128655 | -2.77556e-16 |
| crowds.6-20.positive.mcsta.lp | 0.120477 | 0.120477 | 6.93889e-17 |
| crowds.5-20.positive.mcsta.lp | 0.0860699 | 0.0860699 | 5.55112e-17 |
| consensus.6-2.c2.mcsta.lp | 0.29435 | 0.29435 | 1.11022e-16 |
| zeroconf.1000-8-False.correct_max.mcsta.lp | 7.56652e-07 | 7.56652e-07 | 2.0117e-21 |
| beb.4-8-7.LineSeized.mcsta.lp | 0.999885 | 0.999885 | 1.11022e-16 |
| csma.3-6.all_before_max.mcsta.lp | 0.998834 | 0.998834 | -4.44089e-16 |
| echoring.50.MinOffline3.mcsta.lp | 1.17584e-06 | 1.17584e-06 | 8.08916e-20 |
| consensus.6-2.disagree.mcsta.lp | 0.363644 | 0.363644 | -2.77556e-16 |
| echoring.50.MinOffline2.mcsta.lp | 1.21025e-06 | 1.21025e-06 | 8.34327e-20 |
| ij.20.stable.mcsta.lp | 0.999999 | 0.999999 | -2.55351e-15 |
| echoring.100.MaxOffline1.mcsta.lp | 1.82736e-06 | 1.82736e-06 | 2.82697e-19 |
| csma.3-4.all_before_max.mcsta.lp | 0.932446 | 0.932446 | -4.44089e-16 |
| zeroconf.20-8-False.correct_min.mcsta.lp | 1.54169e-07 | 1.54169e-07 | 1.32349e-22 |
| echoring.100.MaxOffline2.mcsta.lp | 1.71838e-06 | 1.71838e-06 | 3.14037e-19 |
| echoring.100.MinOffline3.mcsta.lp | 1.66364e-06 | 1.66364e-06 | 2.97097e-19 |
| echoring.50.MinFailed.mcsta.lp | 2.73838e-06 | 2.73838e-06 | 4.91279e-20 |
| echoring.100.MinOffline2.mcsta.lp | 1.71838e-06 | 1.71838e-06 | 3.12979e-19 |
| csma.3-4.some_before.mcsta.lp | 0.989522 | 0.989522 | -1.11022e-16 |
| beb.4-8-7.GaveUp.mcsta.lp | 0.000114562 | 0.000114562 | 1.35525e-20 |
| zeroconf.1000-8-False.correct_min.mcsta.lp | 1.16663e-07 | 1.16663e-07 | 1.32349e-22 |
| echoring.50.MaxOffline2.mcsta.lp | 1.21025e-06 | 1.21025e-06 | 8.4068e-20 |
| echoring.50.MinOffline1.mcsta.lp | 1.35879e-06 | 1.35879e-06 | 7.58094e-20 |
| echoring.100.MaxOffline3.mcsta.lp | 1.66364e-06 | 1.66364e-06 | 2.98579e-19 |
| echoring.100.MinOffline1.mcsta.lp | 1.82736e-06 | 1.82736e-06 | 2.82062e-19 |
| echoring.50.MaxOffline1.mcsta.lp | 1.35879e-06 | 1.35879e-06 | 7.68682e-20 |
| csma.3-4.all_before_min.mcsta.lp | 0.904691 | 0.904691 | -1.11022e-16 |
| echoring.50.MaxOffline3.mcsta.lp | 1.17584e-06 | 1.17584e-06 | 8.13152e-20 |
| zeroconf.20-8-False.correct_max.mcsta.lp | 1.63423e-07 | 1.63423e-07 | 3.17637e-22 |
| csma.3-6.all_before_min.mcsta.lp | 0.99715 | 0.99715 | -4.44089e-16 |
| echoring.100.MinFailed.mcsta.lp | 3.52431e-06 | 3.52431e-06 | 2.28699e-19 |
| brp-pta.64-12-32-256.P_3.mcsta.lp | 2.4325e-07 | 2.4325e-07 | 8.99973e-21 |
| brp-pta.64-12-32-256.P_2.mcsta.lp | 2.72208e-07 | 2.72208e-07 | 1.01115e-20 |
| brp-pta.64-12-32-256.P_1.mcsta.lp | 2.72208e-07 | 2.72208e-07 | 1.02173e-20 |

### A.3 Overview Peak Memory Usage

| Name | Verified | Safe | Modest | % vs. Safe | % vs. Modest |
|---|---|---|---|---|---|
| brp.2048-64.p2.mcsta.lp | 726 | 726 | 525 | 0 | 38.2857 |
| haddad-monmege.20-0.7.target.mcsta.lp | 66.3333 | 62 | 62 | 6.98925 | 6.98925 |
| brp.2048-64.p1.mcsta.lp | 728 | 728 | 526 | 0 | 38.403 |
| crowds.6-15.positive.mcsta.lp | 1345.67 | 1315 | 1017 | 2.33207 | 32.3173 |
| crowds.6-20.positive.mcsta.lp | 6079 | 6079.33 | 4576 | -0.00548306 | 32.8453 |
| crowds.5-20.positive.mcsta.lp | 1198 | 1178.33 | 892 | 1.66902 | 34.3049 |
| consensus.6-2.c2.mcsta.lp | 998.667 | 995 | 957 | 0.368509 | 4.35388 |
| zeroconf.1000-8-False.correct_max.mcsta.lp | 844 | 844 | 824 | 0 | 2.42718 |
| beb.4-8-7.LineSeized.mcsta.lp | 6771 | 6767.67 | 7065.33 | 0.0492538 | -4.16588 |
| csma.3-6.all_before_max.mcsta.lp | 36572.7 | 36570.7 | 29256.3 | 0.00546886 | 25.0077 |
| echoring.50.MinOffline3.mcsta.lp | 283 | 283 | 247.667 | 0 | 14.2665 |
| firewire.true-3-600.deadline.mcsta.lp | 377.667 | 374 | 380 | 0.980392 | -0.614035 |
| consensus.6-2.disagree.mcsta.lp | 995 | 995 | 1170 | 0 | -14.9573 |
| echoring.50.MinOffline2.mcsta.lp | 283 | 283 | 247.667 | 0 | 14.2665 |
| wlan_dl.3-80.deadline.mcsta.lp | 1234 | 1230 | 988.333 | 0.325203 | 24.8567 |
| ij.20.stable.mcsta.lp | 3430.67 | 3150.33 | 2078.67 | 8.89853 | 65.0417 |
| csma.3-6.some_before.mcsta.lp | 35332.3 | 35331.7 | 28087 | 0.00188688 | 25.796 |
| wlan_dl.5-80.deadline.mcsta.lp | 3669.33 | 3666 | 2928 | 0.0909256 | 25.3188 |
| wlan_dl.2-80.deadline.mcsta.lp | 485 | 482 | 430 | 0.622407 | 12.7907 |
| firewire.true-36-600.deadline.mcsta.lp | 38692 | 38692.7 | 31994 | -0.00172298 | 20.9352 |
| echoring.100.MaxOffline1.mcsta.lp | 406 | 406 | 312 | 0 | 30.1282 |
| csma.3-4.all_before_max.mcsta.lp | 686.667 | 683 | 497 | 0.536847 | 38.1623 |
| zeroconf.20-8-False.correct_min.mcsta.lp | 844 | 844 | 824 | 0 | 2.42718 |
| echoring.100.MaxOffline2.mcsta.lp | 427.667 | 428 | 312 | -0.0778816 | 37.0726 |
| echoring.100.MinOffline3.mcsta.lp | 405 | 405 | 311.333 | 0 | 30.0857 |
| echoring.50.MinFailed.mcsta.lp | 288.667 | 285 | 250 | 1.28655 | 15.4667 |
| echoring.100.MinOffline2.mcsta.lp | 428 | 427.667 | 312 | 0.0779423 | 37.1795 |
| csma.3-4.some_before.mcsta.lp | 633 | 633 | 460 | 0 | 37.6087 |
| beb.4-8-7.GaveUp.mcsta.lp | 6385 | 6385.67 | 6182 | -0.01044 | 3.28373 |
| zeroconf.1000-8-False.correct_min.mcsta.lp | 820.667 | 844 | 824 | -2.76461 | -0.404531 |
| echoring.50.MaxOffline2.mcsta.lp | 283 | 283 | 247.333 | 0 | 14.4205 |
| echoring.50.MinOffline1.mcsta.lp | 283 | 283 | 248 | 0 | 14.1129 |
| firewire.true-3-800.deadline.mcsta.lp | 712 | 712 | 554.667 | 0 | 28.3654 |
| echoring.100.MaxOffline3.mcsta.lp | 405 | 405 | 311.333 | 0 | 30.0857 |
| echoring.100.MinOffline1.mcsta.lp | 406 | 406 | 312.333 | 0 | 29.9893 |
| echoring.50.MaxOffline1.mcsta.lp | 283 | 283 | 248 | 0 | 14.1129 |
| csma.3-4.all_before_min.mcsta.lp | 683 | 683 | 497 | 0 | 37.4245 |
| echoring.50.MaxOffline3.mcsta.lp | 283 | 283 | 247.667 | 0 | 14.2665 |
| wlan_dl.4-80.deadline.mcsta.lp | 2611 | 2601.67 | 2076.67 | 0.358744 | 25.7303 |
| zeroconf.20-8-False.correct_max.mcsta.lp | 847.667 | 844 | 823.667 | 0.434439 | 2.9138 |
| wlan_dl.6-80.deadline.mcsta.lp | 3828 | 3824 | 3046 | 0.104603 | 25.673 |
| csma.3-6.all_before_min.mcsta.lp | 36567.7 | 36568.3 | 29255.7 | -0.00182307 | 24.9934 |
| echoring.100.MinFailed.mcsta.lp | 409 | 409 | 315.667 | 0 | 29.5671 |
| firewire-pta.30-7500.eventually.mcsta.lp | 1556.67 | 1553 | 1580 | 0.236102 | -1.47679 |
| brp-pta.64-12-32-256.P_3.mcsta.lp | 16474 | 16474 | 13904 | 0 | 18.4839 |
| brp-pta.64-12-32-256.P_2.mcsta.lp | 16362.3 | 16363 | 13791 | -0.00407423 | 18.645 |
| brp-pta.64-12-32-256.P_1.mcsta.lp | 16541.3 | 16537.3 | 13971.3 | 0.0241877 | 18.3948 |
| wlan-large.2.P_max.mcsta.lp | 1242 | 1242 | 1101 | 0 | 12.8065 |
| wlan-large.2.P_min.mcsta.lp | 1245.67 | 1242 | 1101 | 0.295223 | 13.1396 |

### A.4 Additional Memory Information

Version: *Verified*, Average Value: 5152.23, Number of Highest Entries: 38
Version: *Safe*, Average Value: 5144.72, Number of Highest Entries: 27
Version: *Modest*, Average Value: 4248.34, Number of Highest Entries: 4
Average % difference *Verified* vs. *Safe*: 0.4657451626657746

Average % difference *Verified* vs. *Modest*: 20.431173130083195