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

ANONYMOUS

No Institute Given

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

MA

I think the focus for CAV should be on:
A) Whether the formalisation of II is mathematically interseting, e.g. new proofs, or insights
B) How the formalisation of II refines the methodolgy developed for formalising iteration/DP algorithms, and how much of that is reusable in the future
C) The verification of fp arithmetic here and how that is hard in general, any methodological gains compared to Moscato, for instance, that can be reused.
D) Methdological gains from connecting FP verification to mathemtaically interesting algo's like II – that would have been hard w/o step-wise refinement

Probabilistic model checking (PMC) [6,7] is a formal verification technique for randomised systems and algorithms such as wireless communication protocols [41], network-on-chip (NoC) architectures [54], or reliability and performance models [8]. Typical properties checked by means of PMC relate to *reachability probabilities*: What is the probability for a file to eventually be transmitted successfully [20]? 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 [56,55]? The system models that PMC is applied to are specified in higher-level modelling languages such as Modest [14,28] or JANI [17] with a formal semantics in terms of (extensions of) Markov chains and Markov decision processes (MDPs) [13,52].

PMC delivers results with formal guarantees, typically that the computed and (unknown) true probabilities differ by at most a user-specified ε. 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 [26]. Several sound replacements for VI were subsequently developed [27,53,34], 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 [53] contains a subtle mistake that only surfaces on 1

of the 78 models of the Quantitative Verification Benchmark Set (QVBS) [35]. 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 [27] for expected rewards [10] in the mcsta model checker of the Modest Toolset [32] 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 [18,33]. This introduces approximation and rounding errors that in turn may lead to incorrect boolean outputs [63]. A solution lies in the careful use of the directed rounding modes provided by standard IEEE 754 floating-point implementations as in all of today's common CPUs [31], 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 [27] for sound PMC on MDP models and the interactive theorem prover (ITP) Isabelle/HOL [51] with its Isabelle Refinement Framework (IRF) [46]:

- We formalise (i.e. model) II in Isabelle/HOL's logic and formally prove its correctness in 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 (Sect. 6), showing that the verified implementation is efficient.

MA

I can shorten the state-of-the-art significantly.

State-of-the-art: Verification of PMC Algorithms. A probabilistic model checker like mcsta performs preprocessing and transformation steps for both correctness and performance. Previously, the strongly connected component [36] and maximal end component decomposition [37] 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.

State-of-the-art: Verification of Probabilistic Algorithms. The verification of probabilistic algorithms in theorem provers is a well-studied, albeit challenging, task. This includes both, algorithms that use randomness [22,5] and probabilistic analysis of otherwise classical algorithms [62,57,2]. Current research themes include improving methodologies for modelling such algorithms (e.g. the Giry monad [24,23]) and improving ways to reason about these algorithms when modelled as programs (e.g. quantitative separation logic [60,42]). The closest related pieces of work formally verified the VI algorithm: In Coq by Vajjha et. al. [62] and in Isabelle/HOL by Schäffeler and Abdulaziz [57]. In their work, they verified the classical version of VI that optimises the expected discounted values.

In almost all previous formal randomised analyses of algorithms, the authors the authors do not verify efficient implementations. This is due to the sheer difficulty of performing such analyses, as they need substantial formal mathematical libraries on analysis, probabilities, asymptotics, and transition systems. An exception is Schäffeler and Abdulaziz [57], who formally verified a practical implementation of VI. However, since their implementation used infinite precision arithmetic, it 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 probabilistic mathematical analysis is performed and a competitive floating-point implementation is also verified.

State-of-the-art: Verification of Floating-Point Algorithms. The verification of numerical algorithms is an active field of research. For example, the work of Immler et al. [40] which formalizes and verifies the initial value problem of ordinary differential equations. Their work also extracts exectuable ML code based on arbitrary-precision floating-point numbers that are truncated in-between computations for efficiency reasons. While highly important work, this does not scale for PMC, which is why we look at IEEE-754 floating-point numbers that scale much better due to dedicated hardware on most consumer CPU's. The fact that such floating-point implementations of algorithms deviate from the respective mathematical models of algorithms is widely recognised as a problem. Examples of bugs with potentially serious consquences were noted in the hardware and aerospace industry [30,50]. 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 formal systems like Z [12], HOL Light [30,29], PVS [49,16], and Coq [15,21]. Most of that previous work, however, focused on proving correctness of fundamental 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 [61], FPTaylor [58], Real2Float [48] and Fluctuat [25] 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 e.g. nested loops in the implementation of II.

The industrial-strength static analysis tool Astrée [19] 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 [43] has similar functionality but supports deductive verification. However, it is restricted in its use, e.g. that outputs lie in a given interval [44]. Similarly for other deductive verifiers like KeY [3], which can verify the absence of exceptional values like Nan and infinity [1].

2 Preliminaries

MA

No need to use definition blocks. Might save some space

MA

No need for this long motivation of MDPs for CAV

MA

Perhaps the example could be minimised or even removed. I think a very light review of MDPs is needed here since we can assume they know a lot about MDPs.

MA

Section Isabelle/HOL can be made much shorter as I think we should minimise Isabelle syntax. We can add to the appendix some description of Isabelle's syntax along with more extensive listings.

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 definitions and theorem statements are written, and proofs are constructed from a set of axioms (derivation rules). To prove a theorem in an ITP, the user provides high-level steps of a proof, and the ITP fills in the details at the level of axioms.

We perform our formalization using the ITP Isabelle/HOL [51], 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 $f\ x_1\ \ldots\ x_n$ instead of the standard notation $f(x_1,\ \ldots\ ,x_n)$. 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 [11]. 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 [47]. 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) [46]. 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 $\leq_{R_{id}}$ where R_{id} is the identity relation.

At the end of the refinement chain, we aim to have an efficient LLVM program. Once sufficiently refined, the sepref [45] tool can automatically refine a program to LLVM. As $\leq_R$ is transitive, the specification holds for this LLVM program as well.

MS

which specification, this is mentioned the first time here

BK

Note sure if I understood you correctly, but we talk about the specification in the beginning of the section, do you think we need to make it clearer?

2.3 Floating-Point Arithmetic

Our work uses the formalization of the IEEE 754 floating-point standard in Isabelle/HOL [64]. This library provides a generic type *(e,f) float*, where e is the number of bits for the *exponent*, and f is the number of bits for the *fraction* (also known as mantissa). Here, we use the type *double* = *(11,52) float*. The IEEE floating point representation contains positive and negative numbers, as well as special values for $\pm\infty$ and *not a number (NaN)*. The function *valof :: (e,f) float* $\to$ *ereal* maps non-NaN floating point numbers to *extended real numbers*, i.e., $\mathbb{R}\dot{\cup}\{-\infty, +\infty\}$. Moreover, the formalization provides intuitive predicates to identify special cases (e.g. *is_nan*), as well as all standard floating-point instructions like addition, multiplication, and comparisons.

2.4 Markov Decision Processes

Markov Decision Processes (MDPs) are widely used to model probabilistic systems with nondeterministic choices [52], e.g. in PMC, planning, operations research and reinforcement learning [59,9]. 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 avoid certain states, and therefore needs to choose actions such that the probability of achieving the objective is optimized. Many important concepts defined in this section are illustrated in Example 1. We work with the following formal definition:

Definition 1 (MDP). *A finite MDP is a pair* $M = (S, K)$ *where*

- S *is a finite and non-empty set of* states.
- K *is the* transition kernel*, a function* $K : S \to 2^{\mathcal{P}(S)}$ *that 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* $s \in S$*,* $K(s)$ *is closed w.r.t.* S*: for all* $a \in K(s)$*,* $a(s') > 0$ *implies* $s' \in S$*.*

Our formalization of II builds on an existing MDP formalization from the Isabelle/HOL library *Markov Models* [39,38]. 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, cf. Def. 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 *'s pmf* is the type of discrete distributions over *'s*. For a distribution *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 $\forall s \in S.\ (\bigcup a \in K\ s.\ set_{pmf}\ a) \subseteq S$

A *strategy* chooses an action based on the current state and the history of visited states. In the formalization, we work 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 part of the state space. The set of all valid configurations is denoted by $valid_{cfg}$. 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.

The study of MDP subcomponents plays 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 $\forall s \in S'.\ K'(s) \subseteq K(s)$. M' is called *closed* if for all $a \in K'(s)$, we have $a(s') > 0$ implies $s' \in S'$. A sub-MDP is *strongly connected* if every state is reachable from every other state via a sequence of actions. A closed and strongly connected sub-MDP is an *end component*. A *maximal end component* (MEC) is an end component that is not a sub-MDP of any other end component. Finally, *trivial MECs* are MECs with one state and no actions, and a *bottom MEC* is a MEC from which no other MEC is reachable.

Reachability In our PMC setting, the objective is to optimize the strategy 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

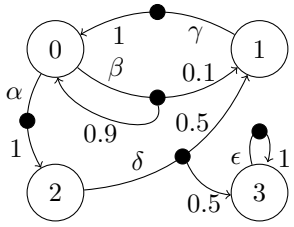

Fig. 1. A simple MDP with four states and five actions.

of P_{cfg} over all valid configurations (P_{sup} is defined analogously). 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, $F_{\text{inf}}\ v\ s$ denotes the minimal expected value of x after one step from s. The symbol $\bigsqcap$ denotes the infimum.

The *least fixed point* (*lfp*) of F_{inf} is P_{inf}, so elementary fixed point theory tells us that iterating a lower bound of P_{inf} leads to P_{inf} in the limit (the same holds for F_{sup} and P_{sup}). To perform II, we need to preprocess the MDP such that the *gfp* of F_{inf} is the same as P_{inf}, and then iterate F_{inf} on both a lower and an upper bound to compute an approximation of P_{inf}.

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

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 α to move to state 2, or β to have a 10% chance to move to state 1.

Let the target states $U = \{3\}$. The reachability probabilities are $P_{\text{inf}}(2) = 0.5$ and $P_{\text{sup}}(2) = 1$. 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 β and γ.

3 Interval Iteration in Isabelle/HOL

The *interval iteration (II)* algorithm for MDPs [27] 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 formalize II and preprocessing routines and prove their correctness in Isabelle/HOL.

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_{inf} (F_{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 maximal end components (MECs) that are trivial or bottom MECs, both fixed points are equal to the optimal reachability probabilities. For now, it is sufficient to assume that the MDP has a single target state s_+ and a single 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. Next, we define the iterated lower and upper bounds $lb_{\text{inf}}\ n$ and $ub_{\text{inf}}\ n$ as the n-fold application of the Bellman optimality operator F_{inf} to the initial lower and upper bounds lb_0 and ub_0:

MA

Here I think the Isabelle notation looks very similar to the maths notation, so it is accessible. However, I think using maths notation will save a lot of space. IMO the formalisation should be mentioned if it found something wrong/different in the maths or if it is difficult for a fundamental reason.
E.g. proof of thm 4.3 is discussed as it is different from the standard proof, but it is different as the formalisation supports deterministic policies. Maybe it should be highlighted if the restriction is mathematically interesting or is it a formalisation quirk.
A lot of the theorem statements are there for no clear reason, as they are not use in later proofs, and nothing about their proofs is mentioned. Maybe in this case they should be mentioned briefly and not as theorem statements...

MA

I hate to repeat this: we should minimise listings here and rather state what where the challenges of formalising definitions and only display a listing if it makes that clearer. I think this should especially be done if the challenges are independent of Isabelle, and more about formal vs. informal, rather than some isabelle-specific difficulty. You could reuse a similar strategy as in the AAAI papers: review a concept (e.g. MDP), and at the same time mention how it is formalised.

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_{inf}. Additionally, we formally derive that the Bellman optimality operators preserve upper and lower bounds in Lemma 3.1.

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

These two are the only properties of the abstract II algorithm that we need for the refinement proof in Sect. 4. Last, in Thm. 3.1 we show that after any number of iterations, II indeed computes bounds for the reachability probabilities. The theorems for P_{sup} are all analogous.

theorem $lb_{\text{inf}}\ n \leq P_{\text{inf}}$ **and** $P_{\text{inf}} \leq ub_{\text{inf}}\ n$ *(Thm. 3.1)*

3.2 Reduced MDPs

II is only guaranteed to converge if the MDP is reduced, i.e. all MECs are trivial or bottom MECs. We therefore need to preprocess the MDP before applying II. The preprocessing steps differ for P_{inf} and P_{sup}, they are called *min-reduction* and *max-reduction*.

As a first step, we extend the existing MDP formalization [38] with SCCs (strongly connected components) and bottom MECs (Def. 3.2). The states of an MDP that form trivial or bottom end components are called *trivials* or *bottoms* respectively. We follow [27] and call an MDP *reduced* if all of its MECs are either trivial or bottom MECs. Reduced MDPs are relevant because the gfp of the Bellman operator of a reduced MDP is equal to its lfp.

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

Min-Reduction We formalize the min-reduction algorithm for MDPs: it transforms an arbitrary MDP with a single target state s_+ into a reduced MDP with the same minimal reachability probabilities. The main insight is that for a minimal reachability problem, all non-trivial MECs other than s_+ can be assigned probability 0: there exists a strategy that almost surely forever stays in the MEC 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_{inf} (Def. 3.3) mapping the states, and then apply it to the MDP M to obtain the reduced MDP M_{inf} (Def. 3.4).

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

definition $M_{\text{inf}} = \textit{fix\_loop}\ s_-\ (\textit{map}_{\textit{mdpc}}\ \textit{red}_{\text{inf}}\ M)$ *(Def. 3.4)*

The function $\textit{map}_{\textit{mdpc}}$ (defined in [39]) applies a function to every state of an MDP. If the function merges states, $\textit{map}_{\textit{mdpc}}$ merges the action sets. Finally, *fix_loop* s_- replaces the actions at s_- with a single self-loop. See Fig. 2 for the min-reduced version of the MDP from Fig. 1.

We now prove that the transformation preserves P_{inf}. The formal proof of the fact that M_{inf} is in fact a reduced MDP follows the original [26]. To distinguish the reachability probabilities of the original and the reduced MDP, we use the notation P_{inf} for the original MDP and $M_{\text{inf}}.P_{\text{inf}}$ for the reduced MDP. Our proof that the min-reduction algorithm preserves P_{inf} is based on the fact that it 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. 3.2, [27, Proposition 3]) is a direct consequence. Note that our proof is simpler and more precise than the original: in [27], the authors only argue that for every strategy in the reduced MDP there exists a strategy in the original MDP with the same P_{inf} and vice versa.

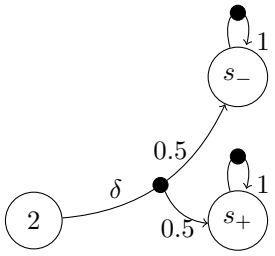

Fig. 2. Min-reduced MDP from Fig. 1.

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

theorem **assumes** $s \in S$ **shows** $P_{\text{inf}}\ s = M_{\text{inf}}.P_{\text{inf}}\ (\textit{red}_{\text{inf}}\ s)$ *(Thm. 3.2)*

Max-Reduction Maximal reachability probabilities can similarly be handled with a max-reduction, but the procedure is more involved. Not all non-trivial MECs can be collapsed into a single state, as P_{sup} would not be preserved: A maximizing strategy might choose to leave a non-trivial MEC. We can, however, 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_-. See Fig. 3 for the max-reduced version of the MDP from Fig. 1.

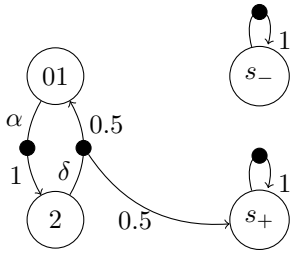

Fig. 3. Max-reduced MDP from Fig. 1.

Collapsing the MECs into a single state is done by the function *the_mec*, the transformation preserves P_{sup} (Thm. 3.3). Our proof resembles the proof of [4, Theorem 3.8], however we have to work around the fact that the MDP formalization only supports deterministic policies [38]. Note that every state of M_{MEC} now forms its own MEC. The second part of the reduction is similar to the min-reduction.

theorem **assumes** $s \in S$ *(Thm. 3.3)*
shows $P_{\text{sup}}\ s = M_{MEC}.P_{\text{sup}}\ (\textit{the\_mec}\ M\ s)$

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

($\geq$) Consider an optimal strategy π_{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^π where the action selected by π_{MEC} is enabled. Moreover, within a MEC, we can obtain a deterministic, memoryless strategy π_m that reaches this state with probability 1. Thus we can construct a strategy in M that behaves like π_m within each MEC until s_m^π is reached and then follows π_{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. Thus I is the smallest 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.

fun G **where** *(Def. 3.5)*
$G\ 0 = \mathit{bottoms}\ M$
$G\ (n+1) =$ **let** $G_{\leq n} = \bigcup i \leq n.\ G\ i$ **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 to reach a bottom MEC. The rate at which a bottom MEC is encountered depends on the smallest probability of any transition in the MDP, called η. At every step, the probability of descending a level wrt. G is at least η. Hence we can show that for any valid configuration, the probability to reach the bottom MECs in I steps is no less than η^I:

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

The value $P^{\leq}_{\mathit{cfg}}\ n$ 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. 3.4, [27, Proposition 1]). As n increases, $(1-\eta^I)^n$ converges to 0 and therefore $P^{\leq}_{\mathit{cfg}}\ nI$ tends towards 1. Since we chose *cfg* arbitrarily, this means that we almost surely reach a bottom MEC in the limit.

theorem **assumes** $\mathit{cfg} \in \mathit{valid}_{\mathit{cfg}}$ **shows** $1 - (1-\eta^I)^n \leq P^{\leq}_{\mathit{cfg}}\ nI$ *(Thm. 3.4)*

3.4 Convergence of Interval Iteration

We now assume a special form of reduced MDPs, where the only bottom MECs are $\{s_+\}$ and $\{s_-\}$ (Locale 3.1). The MDP has a single target state s_+ and a single avoid state s_-, that both are absorbing: $return_{pmf}\ s$ is the probability mass function that assigns probability 1 to state s. 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** $\forall s \in S \setminus \{s_+, s_-\}.\ s \in trivials\ M$ **and**
$K\ s_- = \{return_{pmf}\ s_-\}$ **and** $K\ s_+ = \{return_{pmf}\ s_+\}$

Towards a convergence proof of II, we first show that the lower and upper bound sequences relate to finite-horizon reachability probabilities of both s_+ and s_- (Lemma 3.4, [27, Lemma 4]). In this section, we indicate the target sets explicitly. Again, we only present results for minimal probabilities, the results for maximal probabilities are analogous.

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

Combined with Thm. 3.4 we can give a bound on the difference between both II sequences (Thm. 3.5). Note that this convergence result in general only holds if all computations are carried out with arbitrary precision arithmetic. In our concrete implementation, i.e. 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. 6 for experimental results).

Finally, the theorem is not applicable if all probabilities in the MDP are equal to one, i.e. there is no branching after an action is selected. In this case, the MDP is deterministic and is better solved with qualitative solution methods. Note that [27] also assume $\eta < 1$.

theorem *(Thm. 3.5)*
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:

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

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 competitive performance. 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 (ub)* or *lower bounding (lb)* 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 [31] and cumbersome to reason about in the IRF. Instead, 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 [45] to automatically refine an algorithm to an LLVM program. sepref contains a library of data structures and operations that it can automatically refine to LLVM. 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 we use assertions in the IRF with the following example.

```
1  definition ls_app x xs = xs @ [x] and half_nat n = n div 2
2  definition apphalf n xs = do { let n' = half_nat n; return ls_append n' xs }
3  lemma (rshift1, half_nat) :: A_size → A_size
4  lemma (arl_app, ls_app) :: [λn xs. length xs < 2^63−1] A_size → A^d_arl → A_arl
5  lemma (arl_apphalf, apphalf) :: [λn xs. length xs < 2^63−1] A_size → A^d_arl → 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 registered in the format of Line 3. The lemma states that *rshift1* and *half_nat* are related as follows: if the inputs of *half_nat* (a natural number) and *rshift1* (of 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:

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

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 *lb* floats and real numbers to *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 $R_{lb} = \{(fl,r).\ valof\ fl \leq r \wedge \neg is\_nan\ fl \wedge valof\ fl \geq 0\}$ 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 allocated memory. This means that the assertion A_{lb} ignores the heap and is in essence R_{lb}.

We now present the operations supported by our framework. Note that this presentation is not exhaustive, but we focus on the operations required for our use-case.

Fused multiply-add The ternary operation $fma\ a\ b\ c = a * b + c$ represents fused multiply-add. Compared to separately multiplying and adding, it yields a smaller floating-point rounding error. 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*) :: $A_{lb} \to A^{>0}_{lb} \to A_{lb} \to A_{lb}$ *(Lemma 4.1)*

This lemma states that for all inputs that 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 problem with the stricter assertion $A^{>0}_{lb}$ that only allows positive finite floats. In the case of II, the transition probabilities from the input model satisfy this assertion.

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*) :: $A_{ub} \to A_{lb} \to A_{bool}$

Similarly, by swapping rounding modes we get complete instead of sound comparisons. The specification of a sound but incomplete comparison is defined using the **spec** function: 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 for two identical comparisons of reals, we have two implementations with different outcomes. Similarly, subtraction can also only be implemented as a mixed operator.

Min & 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*) :: $A_{lb} \to A_{lb} \to A_{lb}$

This refinement holds despite the fact that a comparison does not reveal anything about the bounding floating point 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 for *ub* and *max* proceeds analogously.

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

4.3 Refinement of Interval Iteration

Using our floating-point extension to the IRF, we now derive an implementation of II that implements 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.

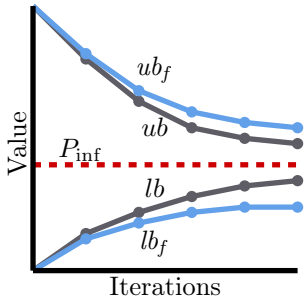

Fig. 4. The valuation for an MDP state over successive iterations: reals (grey) vs. floats with safe rounding (blue). The dashed red line marks the reachability probability.

MS
is this visible in the print version (bw)?

The plot in Fig. 4 shows a fictive run of II on both reals and floats using the refinement relation A_{lb}. In the long run, II converges to the dashed red line P_{inf}. The grey 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 blue line lb_f (similarly for *ub* using A_{ub}). Note that the deviantions 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 $(\lambda(x, y).\ \forall s \in states\ M.\ x\ s \leq P_{\text{inf}}\ M\ s \wedge P_{\text{inf}}\ M\ s \leq y\ s)$

Despite the fact that convergence follows from Thm. 3.5, 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 ← lb0 M; y ← ub0 M; i ← 0; flag ← True;
3    while (i++ < L ∧ flag) (
4      (x,y) ← F^gs_inf M x y
5      flag ← spec(λx. 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* signals early termination before L iterations, which is sound because the specification *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_{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* $\leq_{R_{id}}$ *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 [32]. 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 model the MDP data structure of mcsta as a 6-tuple in Isabelle/HOL:

(*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 into *Tr*, pointing to the first transition (action) of the state. Similarly, for each transition, *Tr* contains an index to *Br* and *Pr*, pointing to the first branch of the transition and its probability. Finally, *Br* contains indices pointing back to *St*, respresenting the target state of the branch. Additionally, u_a, 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 values of u_a and u_t contain the index of the avoid and target state.

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 along 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 ε, to be 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

The final step is to prove that the LLVM algorithm *ii_gs_inf_llvm* adheres to the specification *ii_inf_spec* by combining the refinement in Thm. 4.1 with the refinements implicitly described in Sect. 4.4 through transitivity. Our correctness statement takes the form of a Hoare triple.

```
1  theorem llvm_htriple                                              (Thm. 4.2)
2    (A_size n n_i ⋆ A_size L L_i ⋆ A_ub ε ε_i ⋆ A_M M M_i
3      ⋆ ↑(MDP_Reach M ∧ n+1 < max_size ∧ n = card (states M)))
4    (ii_gs_inf_llvm L_i n_i ε_i M_i res_i)
5    (λ(lb_f,ub_f). ∃lb ub. A^out_lb lb lb_f ⋆ A^out_ub ub ub_f
6      ⋆ ↑(∀s ∈ states M. lb s ≤ P_inf M s ≤ 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; ε is a real number implemented as float ε_i and M is the MDP implemented as M_i. The separation conjunction $\star$ specifies that these implementations are allocated to disjoint places 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_{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 Implementation

We use the LLVM code generator of the IRF to export the LLVM program *ii_gs_inf_llvm* for use in the LLVM compiler pipeline. Additionally, it generates a C header that allows easy embedding into other software, e.g. mcsta.

Integration with mcsta We integrate our verified implementation into the mcsta pipeline, replacing the existing unverified interval iteration algorithm. When integrating the library, we aim to have as little error-prone glue code as possible. mcsta stores the MDP with real-valued probabilities as 128-bit rationals (a pair of 64-bit longs representing the numerator and denominator). For the MDP data structure M_i, we need to obtain two floats that are lower and upper bounds of the rational probability. We convert the probabilities to 64-bit doubles by directly converting the numerator and denominator to doubles and performing a division twice, once rounding up and once rounding down.

We assume that the input MDP as produced by the mcsta pipeline is well-formed. If there is a bug in the parser that produces MDPs that are not well-formed, we lose our formal guarantees. Until we have a fully verified toolchain, this remains a risk.

Interpretation of the output Thm. 4.2 states that performing finitely many iterations using precise arithmetic provides us with lower and upper bounds (lb and ub) on the actual reachability probability. However, the implementation provides outputs as floats lb_f and ub_f which are conservative bounds: They provably bound the solution. However, whereas we can prove that lb and ub converge, this does not hold for lb_f and ub_f. We experimentally validate that we still converge up to ε by running a broad set of benchmarks.

6 Experiments

We experimentally evaluate the performance of our implementation on standard benchmark MDPs. The goals of this evaluation are twofold: 1) compare the running times of the verified with an unverified implementation and 2) evaluate whether convergence still generally occurs in the floating-point setting. We argue that 1) is important to demonstrate since verified implementations tend to be orders of magnitude slower than unverified ones [57]. Moreover, 2) is important because our verified implementation does not formally guarantee convergence.

For runtime speed, we compare our verified implementation to the manual implementation in mcsta. There are two (unverified) variants of this algorithm in mcsta: one with standard rounding (*Modest* implementation) and a variant with safe rounding [31] (*Safe* implementation). The latter uses AVX512 instructions for safe rounding, similar to our verified LLVM implementation. We set the convergence threshold to $\varepsilon = 10^{-6}$, which is the standard in mcsta.

We use all DTMC, MDP and PTA models of the QVBS of QCOMP [18] with between 10^6 and 10^8 states that need at least two iterations to converge

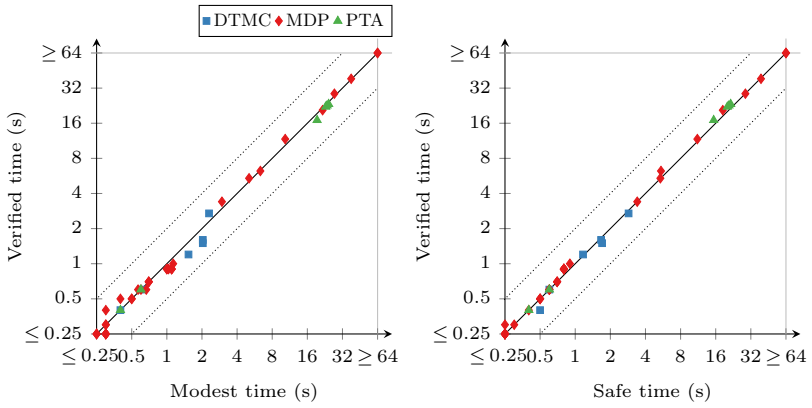

Fig. 5. Comparison of runtime to complete the Interval Iteration routine

to ε. We consider both minimal and maximal reachability probabilities for our benchmarks. In total, this yields a benchmark set of 49 benchmark instances. We ran all benchmarks on an Intel i9-11900K at 3.5GHz with 128GB of RAM.

The plot on the left in Fig. 5 compares the *Verified* implementation to the *Modest* implementation, while the plot on the right compares to the *Safe* implementation. Our data shows that the performance is consistent between the different implementations. The raw data (omitted here) shows that the *Safe* and *Verified* implementation perform the same number of iterations on each instance and converge up to ε. We conclude that the differences between implementations are minimal. In summary, we note that 1) we have replaced an unverified implementation with a verified one at no or negligible cost and 2) that our algorithm still converges for practical use-cases.

7 Conclusion

We 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 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 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.

In a next step, we plan to build a complete verified II backend of mcsta. The missing part is an efficient verified implementation of the transformations to reduced MDPs. For this purpose, we aim to build on recent advancements that provide a verified and efficiently executable MEC decomposition algorithm in Isabelle/HOL [37].

References

1. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Deductive verification of floating-point java programs in key. In: Groote, J.F., Larsen, K.G. (eds.) 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. Lecture Notes in Computer Science, vol. 12652, pp. 242–261. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_13, https://doi.org/10.1007/978-3-030-72013-1_13
2. Abdulaziz, M., Madlener, C.: A Formal Analysis of RANKING. In: The 14th Conference on Interactive Theorem Proving (ITP) (2023). https://doi.org/10.48550/arXiv.2302.13747
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6, http://dx.doi.org/10.1007/978-3-319-49812-6
4. de Alfaro, L.: Formal verification of probabilistic systems. Ph.D. thesis, Stanford University, USA (1997), https://searchworks.stanford.edu/view/3910936
5. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in Coq. Science of Computer Programming (Jun 2009). https://doi.org/10.1016/j.scico.2007.09.002
6. Baier, C.: Probabilistic model checking. In: Esparza, J., Grumberg, O., Sickert, S. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series – D: Information and Communication Security, vol. 45, pp. 1–23. IOS Press (2016). https://doi.org/10.3233/978-1-61499-627-9-1
7. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 963–999. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_28
8. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Performance evaluation and model checking join forces. Commun. ACM **53**(9), 76–85 (2010). https://doi.org/10.1145/1810891.1810912
9. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)

10. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In: Majumdar, R., Kuncak, V. (eds.) 29th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 10426, pp. 160–180. Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_8
11. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: TYPES. Lecture Notes in Computer Science, vol. 3085, pp. 34–50. Springer (2003)
12. Barrett, G.: Formal methods applied to a floating-point number system. IEEE Transactions on Software Engineering (May 1989). https://doi.org/10.1109/32.24710
13. Bellman, R.: A Markovian decision process. Journal of Mathematics and Mechanics **6**(5), 679–684 (1957)
14. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MoDeST: A compositional modeling formalism for hard and softly timed systems. IEEE Trans. Software Eng. **32**(10), 812–830 (2006). https://doi.org/10.1109/TSE.2006.104
15. Boldo, S., Melquiond, G.: Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In: 2011 IEEE 20th Symposium on Computer Arithmetic (Jul 2011). https://doi.org/10.1109/ARITH.2011.40
16. Boldo, S., Munoz, C.: A high-level formalization of floating-point number in PVS. Tech. rep. (2006)
17. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: Quantitative model and tool interaction. In: Legay, A., Margaria, T. (eds.) 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 10206, pp. 151–168 (2017). https://doi.org/10.1007/978-3-662-54580-5_9
18. 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: Margaria, T., Steffen, B. (eds.) 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA). Lecture Notes in Computer Science, vol. 12479, pp. 216–241. Springer (2020). https://doi.org/10.1007/978-3-030-83723-5_15
19. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Sagiv, S. (ed.) 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. Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_3, https://doi.org/10.1007/978-3-540-31987-0_3
20. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: de Alfaro, L., Gilmore, S. (eds.) Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV). Lecture Notes in Computer Science, vol. 2165, pp. 39–56. Springer (2001). https://doi.org/10.1007/3-540-44804-7_3
21. De Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Transactions on Computers (2010). https://doi.org/10.1109/TC.2010.128
22. Eberl, M., Haslbeck, M.W., Nipkow, T.: Verified Analysis of Random Binary Tree Structures. J. Autom. Reason. (2020). https://doi.org/10.1007/S10817-020-09545-0

23. Eberl, M., Hölzl, J., Nipkow, T.: A Verified Compiler for Probability Density Functions. In: Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (2015). https://doi.org/10.1007/978-3-662-46669-8_4
24. Giry, M.: A categorical approach to probability theory. In: Categorical Aspects of Topology and Analysis (1982). https://doi.org/10.1007/BFb0092872
25. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D.A. (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6538, pp. 232–247. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_17, https://doi.org/10.1007/978-3-642-18275-4_17
26. Haddad, S., Monmege, B.: Reachability in MDPs: Refining convergence of value iteration. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) 8th International Workshop on Reachability Problems (RP). Lecture Notes in Computer Science, vol. 8762, pp. 125–137. Springer (2014). https://doi.org/10.1007/978-3-319-11439-2_10
27. Haddad, S., Monmege, B.: Interval iteration algorithm for mdps and imdps. Theor. Comput. Sci. **735**, 111–131 (2018). https://doi.org/10.1016/J.TCS.2016.12.003
28. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Des. **43**(2), 191–232 (2013). https://doi.org/10.1007/S10703-012-0167-Z
29. Harrison, J.: Formal verification at Intel. In: 18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings. (Jun 2003). https://doi.org/10.1109/LICS.2003.1210044
30. Harrison, J.: A Machine-Checked Theory of Floating Point Arithmetic. In: Theorem Proving in Higher Order Logics (1999). https://doi.org/10.1007/3-540-48256-3_9
31. Hartmanns, A.: Correct probabilistic model checking with floating-point arithmetic. In: TACAS (2). Lecture Notes in Computer Science, vol. 13244, pp. 41–59. Springer (2022)
32. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 8413, pp. 593–598. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_51
33. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A practitioner's guide to MDP model checking algorithms. In: Sankaranarayanan, S., Sharygina, N. (eds.) 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 13993, pp. 469–488. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_24
34. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: Lahiri, S.K., Wang, C. (eds.) 32nd International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 12225, pp. 488–511. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_26
35. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: Vojnar, T., Zhang, L. (eds.) 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 11427, pp. 344–350. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_20

36. Hartmanns, A., Kohlen, B., Lammich, P.: Fast verified sccs for probabilistic model checking. In: ATVA (1). Lecture Notes in Computer Science, vol. 14215, pp. 181–202. Springer (2023)
37. Hartmanns, A., Kohlen, B., Lammich, P.: Efficient formally verified maximal end component decomposition for mdps. In: FM (1). Lecture Notes in Computer Science, vol. 14933, pp. 206–225. Springer (2024)
38. Hölzl, J.: Markov chains and markov decision processes in isabelle/hol. J. Autom. Reason. **59**(3), 345–387 (2017). https://doi.org/10.1007/S10817-016-9401-5
39. Hölzl, J., Nipkow, T.: Markov models. Arch. Formal Proofs **2012** (2012), https://www.isa-afp.org/entries/Markov_Models.shtml
40. Immler, F., Hölzl, J.: Numerical analysis of ordinary differential equations in isabelle/hol. In: Beringer, L., Felty, A.P. (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 7406, pp. 377–392. Springer (2012). https://doi.org/10.1007/978-3-642-32347-8_26, https://doi.org/10.1007/978-3-642-32347-8_26
41. Kamali, M., Katoen, J.P.: Probabilistic model checking of AODV. In: Gribaudo, M., Jansen, D.N., Remke, A. (eds.) 17th International Conference on the Quantitative Evaluation of Systems (QEST). Lecture Notes in Computer Science, vol. 12289, pp. 54–73. Springer (2020). https://doi.org/10.1007/978-3-030-59854-9_6
42. Kaminski, B.L.: Advanced Weakest Precondition Calculi for Probabilistic Programs. Ph.D. thesis, RWTH Aachen University, Germany (2019), http://publications.rwth-aachen.de/record/755408
43. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/S00165-014-0326-7, https://doi.org/10.1007/s00165-014-0326-7
44. Kosmatov, N., Prevosto, V., Signoles, J.: Guide to Software Verification with Frama-C. Springer (2024)
45. Lammich, P.: Generating verified LLVM from isabelle/hol. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving (ITP). LIPIcs, vol. 141, pp. 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPICS.ITP.2019.22
46. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to hopcroft's algorithm. In: Beringer, L., Felty, A.P. (eds.) 3rd International Conference on Interactive Theorem Proving (ITP). Lecture Notes in Computer Science, vol. 7406, pp. 166–182. Springer (2012). https://doi.org/10.1007/978-3-642-32347-8_12
47. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. p. 75–88. San Jose, CA, USA (Mar 2004)
48. Magron, V., Constantinides, G.A., Donaldson, A.F.: Certified roundoff error bounds using semidefinite programming. ACM Trans. Math. Softw. **43**(4), 34:1–34:31 (2017). https://doi.org/10.1145/3015465, https://doi.org/10.1145/3015465
49. Miner, P.S.: Defining the IEEE-854 floating-point standard in PVS. Tech. rep. (1995)
50. 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 (2017). https://doi.org/10.1007/978-3-319-66266-4_14
51. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)

52. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994). https://doi.org/10.1002/9780470316887, https://doi.org/10.1002/9780470316887
53. Quatmann, T., Katoen, J.P.: Sound value iteration. In: Chockler, H., Weissenbacher, G. (eds.) 30th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 10981, pp. 643–661. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_37
54. 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: Lluch-Lafuente, A., Mavridou, A. (eds.) 26th International Conference on Formal Methods for Industrial Critical Systems (FMICS). Lecture Notes in Computer Science, vol. 12863, pp. 232–248. Springer (2021). https://doi.org/10.1007/978-3-030-85248-1_16
55. Ruijters, E., Guck, D., Drolenga, P., Peters, M., Stoelinga, M.: Maintenance analysis and optimization via statistical model checking – evaluating a train pneumatic compressor. In: Agha, G., Houdt, B.V. (eds.) 13th International Conference on the Quantitative Evaluation of Systems (QEST). Lecture Notes in Computer Science, vol. 9826, pp. 331–347. Springer (2016). https://doi.org/10.1007/978-3-319-43425-4_22
56. 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. IEEE Computer Society (2016). https://doi.org/10.1109/DSN.2016.67
57. Schäffeler, M., Abdulaziz, M.: Formally verified solution methods for markov decision processes. In: AAAI. pp. 15073–15081. AAAI Press (2023)
58. 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. **41**(1), 2:1–2:39 (2019). https://doi.org/10.1145/3230733, https://doi.org/10.1145/3230733
59. Sutton, R.S., Barto, A.G.: Reinforcement learning – an introduction. Adaptive computation and machine learning, MIT Press (1998)
60. Tassarotti, J., Harper, R.: A separation logic for concurrent randomized programs. Proc. ACM Program. Lang. (2019). https://doi.org/10.1145/3290377
61. 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 (2025). https://doi.org/10.1007/978-3-031-71177-0_2
62. 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) (2021). https://doi.org/10.1145/3437992.3439927
63. Wimmer, R., Kortus, A., Herbstritt, M., Becker, B.: Probabilistic model checking and reliability of results. In: Straube, B., Drutarovský, M., Renovell, M., Gramata, P., Fischerová, M. (eds.) 11th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS). pp. 207–212. IEEE Computer Society (2008). https://doi.org/10.1109/DDECS.2008.4538787
64. Yu, L.: A formal model of IEEE floating point arithmetic. Arch. Formal Proofs **2013** (2013), https://www.isa-afp.org/entries/IEEE_Floating_Point.shtml