Efficient Formally Verified Maximal End Component Decomposition for MDPs

Arnd Hartmanns, Bram Kohlen(✉), and Peter Lammich

University of Twente, Enschede, The Netherlands
b.kohlen@utwente.nl

Abstract. Identifying a Markov decision process's maximal end components is a prerequisite for applying sound probabilistic model checking algorithms. In this paper, we present the first mechanized correctness proof of a maximal end component decomposition algorithm, which is an important algorithm in model checking, using the Isabelle/HOL theorem prover. We iteratively refine the high-level algorithm and proof into an imperative LLVM bytecode implementation that we integrate into the Modest Toolset's existing mcsta model checker. We bring the benefits of interactive theorem proving into practice by reducing the trusted code base of a popular probabilistic model checker and we experimentally show that our new verified maximal end component decomposition in mcsta performs on par with the tool's previous unverified implementation.

1 Introduction

Model checking [12] is a verification technique that determines the validity of properties specified as temporal logics formulae on formal models of systems ranging from hardware circuits [6,13] and concurrent programs [21] to cyber-physical systems [15,45]. The model's semantics is traditionally some form of transition system [3]. Extended model checking approaches deal with, for example, real-time systems using a timed automata semantics [1,7], or probabilistic systems [2] using Markov chains or Markov decision processes (MDP) [5,47]. Given the often safety- or mission-critical nature of the systems being model-checked, the correctness of the model checker is of utmost importance.

As of today, however, few model checkers themselves are formally verified, and none of those is widely used. The Cava LTL model checker [8,18], for example, is fully verified, from algorithmic correctness all the way down to a correct implementation. Yet, for the same purpose, Spin [31] remains the tool of choice for practitioners despite being unverified. This is because Cava supports only a fragment of the Promela input language [44], and is much slower due

Authors are listed in alphabetical order. This work was supported by the European Union's Horizon 2020 research and innovation programme under Marie Skłodowska-Curie grant agreement 101008233 (mission), the Interreg North Sea project storm_safe, NWO grant OCENW.KLEIN.311, NWO VIDI grant VI.Vidi.223.110 (TruSTy) and NWO grant OCENW.M.21.291 (vespa).

A. Platzer et al. (Eds.): FM 2024, LNCS 14933, pp. 206–225, 2025.
https://doi.org/10.1007/978-3-031-71162-6_11

to its purely functional-programming implementation, while Spin's algorithms and code have been highly optimised. Similarly, the fully verified Munta model checker for timed automata [55] is significantly slower than the de-facto standard tool UPPAAL [4], despite Munta's refinement resulting in Standard ML code that uses imperative elements such as arrays to obtain better performance.

While initiatives like Cava and Munta constitute major achievements in interactive theorem proving (ITP) research, they have not managed to bring the benefits of ITP into verification practice. Their approach towards the goal of a fully-verified model checker is top-down: Create a new tool from scratch, necessarily starting (and ultimately remaining) with a limited scope that prevents practical adoption. In addition, they are limited by the technology available in their time for refining abstract algorithms into executable code.

We instead propose a bottom-up approach: Starting from an existing model checker that is competitive and has an established user base, replace its unverified code by provably correct implementations component-by-component. In this way, the tool is not immediately fully verified, but the trusted code base is reduced step-by-step. Crucially, by exploiting recent advances in refinement technology [39,41] that deliver highly-efficient LLVM bytecode, our verified replacement components perform similarly to the unverified originals implemented in e.g. C or C#. The incremental approach is thus "invisible" to the users, leading to an immediate adoption of the benefits of ITP in verification practice.

Our contributions are to formalise an algorithm for *maximum end component* (MEC) decomposition in MDPs with Isabelle/HOL, prove its correctness, and iteratively refine the abstract algorithm to imperative code and data structures in LLVM bytecode. We integrate the resulting verified implementation into an existing probabilistic model checker and experimentally show that it performs on par with the previous unverified implementation.

A MEC is a subset of the states of an MDP for which a strategy exists that remains within the MEC with probability 1. In an MDP with nontrivial MECs, the Bellman operator used in sound numeric algorithms for probabilistic model checking (PMC) for indefinite-horizon properties [22,26,48] has multiple fixed points, leading to divergence [22] and/or breaking the algorithm's correctness proof [26]. Eliminating or later deflating [17] the MECs of an MDP is thus a necessary step in PMC. To the best of our knowledge, ours is **the first mechanical formalisation and correctness proof of MEC decomposition**. We use the Isabelle Refinement Framework [42] to refine our algorithm down to LLVM code which we integrate into an existing model checker. We target mcsta of the Modest Toolset [25]. Its performance is competitive [9], and it has been used for various case studies by different teams of researchers [24,50,53]. Our verification and refinement of MEC decomposition constitutes a critical step on the long-term bottom-up path towards a fully-verified probabilistic model checker, laying the foundation for verifying the actual numeric algorithm as the next step. MEC decomposition is also used in probabilistic planning [57] as part of the FRET[1]

[1] Here, end components are called "traps"; FRET is "find, revise, eliminate traps" [33].

approach [33,51], and can be generalised from MDP to stochastic games where it is equally necessary for sound algorithms [17]. Our work can thus be transferred to tools in these areas.

Our MEC decomposition algorithm, or *MEC algorithm* for short, follows the standard approach [3, Algorithm 47]: (i) find all strongly connected components (SCCs) of the MDP's graph, (ii) identify all bottom SCCs as MECs and remove them, (iii) delete all transitions with nonzero probability to leave an SCC, and (iv) repeat until no more states remain. After defining MDPs and MECs in Sect. 2, we present the algorithm, our formalisation in Isabelle/HOL, and our correctness proof in Sect. 3. We introduce the efficient data structures for the implementation in Sect. 4. We had earlier verified Gabow's SCC-finding algorithm and refined it into efficient LLVM code for mcsta [28]. We were able to integrate the SCC algorithm's high-level correctness proof into our MEC algorithm formalisation with minor technical adaptations. However, the SCC algorithm could assume the graph to be static, whereas the MEC algorithm iteratively changes the MDP graph. We thus need a new data structure that allows deleting states and transitions, which we describe together with the corresponding refinement proofs in Sect. 4. In this part, we also extended proofs and refinement relations for the SCC-finding aspect due to an extended data structure. In Sect. 5, we describe the LLVM code generation and integration into mcsta. By adopting mcsta's existing MDP representation, we minimise costly glue code and transformations or copies of the data. This is important for the scalability and performance of our end result, which we experimentally show in Sect. 6.

Related Work. Certification is an alternative to verification: A formally verified certifier checks the results of an unverified tool. This requires a practical certification mechanism and the support of the unverified model checker. Formally verified certification tools that work on significant problem sizes exist for e.g. timed automata model checking [54,56] and SAT solving [29,40].

Probabilistic models have been the subject of ITP work before. Notably, there are some formalisations of MDPs and the value iteration algorithm in Isabelle/HOL [30] and Coq [52], but executable code does not appear to have been extracted from these proofs. Additionally, there is a formalisation of value iteration for discounted expected rewards [43] which extracts Standard ML code from the proof. We note that MEC decomposition is not necessary in the discounted case, thus [43] and the many current works in machine learning/artificial intelligence based on reinforcement learning typically avoid the problem.

The standard MEC decomposition approach computes SCCs. SCC-finding algorithms have been formalised with various tools, including Isabelle/HOL [38], Coq [46], and Why3 [11]. Of these, only [38] extracted executable code, which however performed poorly. Our earlier verification and high-performance refinement of Gabow's SCC-finding algorithm [28] built upon ideas from [38]. An asymptotically faster MEC algorithm has been proposed [10]. It combines SCC-finding with a lock-step depth-first search. The algorithm has not been adopted by PMC tools so far, likely due to its implementation complexity.

2 Background

We introduce MDPs and MECs in the context of probabilistic model checking, then explain the refinement-based approach to program verification that we use.

Probabilistic Model Checking. Let $[0,1] \subseteq \mathbb{R}$ be the interval of real numbers from 0 to 1 and 2^X the power set of X. A (discrete) *probability distribution* over X is a function $\mu\colon X \to [0,1]$ where $\sum_{x \in X} \mu(x) = 1$ that has countable support $Sp(\mu) = \{\, v \mid \mu(v) > 0 \,\}$. *Dist(X)* is the set of probability distributions over X.

Definition 1. *A Markov decision process (MDP) is a pair* (S, K) *where* S *is a finite set of* states *and* K *is the* kernel *of type* $S \to 2^{Dist(S)}$.

An MDP models the interaction of an agent with a random environment: In current state u, the agent makes a decision, i.e. non-deterministically chooses a distribution $\mu \in K(u)$. The environment then updates the current state by sampling μ. By repeating this process, we trace a *path* with a certain probability. A *strategy* represents an agent's decisions of which distribution to pick next based on the path traced so far. Combining an MDP and a strategy removes all non-determinism, resulting in a Markov chain on which a probability measure over paths can be defined in the standard way [3]. We characterise interesting sets of paths via *properties*; for this work, we are particularly interested in *reachability*:

Definition 2. *Given sets* $A, T \subseteq S$*, a* reachability property *is an LTL formula* $\neg A \,\mathsf{U}\, T$ *(characterising the set of paths that do not visit* avoid *states (A) before a* target *state (T) which is visited eventually). Under a given strategy, the probability of satisfying a property is the probability mass of the (measurable) set of paths satisfying that property.*

There is a strategy that *minimises* and one that *maximises* the probability of satisfying $\neg A \,\mathsf{U}\, T$ [3], which induce the minimum/maximum reachability probabilities.

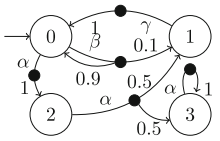

Fig. 1. MDP (S, K)

Example 1. Figure 1 shows an MDP with $S = \{\, 0, 1, 2, 3 \,\}$. The edges represent K where α, β and γ label the non-deterministic choices followed by the probability mass of each state. The minimum probability to satisfy $\neg\{\, 1 \,\} \,\mathsf{U}\, \{\, 3 \,\}$ is 0 for the strategy that always chooses β and γ. The maximum probability is 0.5 by choosing α twice. After this, we are either in target state 3 or in avoid state 1.

The edges of an MDP kernel are $Edges(K) = \{\, (u, v) \mid \exists \mu \in K(u)\colon \mu(v) > 0 \,\}$. A *sub-MDP* of (S, K) is a pair (C, D) where $C \subseteq S$ and $D(u) \subseteq K(u)$.

Definition 3. *Given an MDP* (S, K)*, an* end component *(EC) [14] is a sub-MDP* (C, D) *such that* $C \times C \subseteq Edges(D)^*$ *(it is strongly connected) and* $(u, v) \in Edges(K) \wedge u \in C \Rightarrow v \in C$ *(it is closed). A* maximal *end component (MEC) is an EC that is not a sub-MDP of another EC.*

SCCs are weaker than MECs: They are maximal strongly connected *subsets of states* rather than closed sub-MDPs. In other words, for every state, there exists a strategy such that the next state is in the SCC with probability > 0, while for a MEC the probability is 1. MECs play an essential role in sound algorithms for evaluating reachability probabilities: Collapsing the MECs (i.e. replacing every MEC by a single state that collects all edges out of the MEC) guarantees a single fixed point for these algorithms. We find MECs through a graph analysis that requires the computation of SCCs. *Graph analysis* means that we only need to know whether probabilities are non-zero, i.e. we work with the *MDP structure* that maps state u to its set of supports $\{ Sp(\mu) \mid \mu \in K(u) \} \subseteq 2^S$. We call elements of the outer set *transitions* and elements of the inner sets *branches*.

Example 2. In the MDP structure for Fig. 1, state 0 is mapped to $\{ \{ 0, 1 \}, \{ 2 \} \}$. The MDP has two SCCs: $\{ 0, 1, 2 \}$ and $\{ 3 \}$. Set $\{ 0, 1 \}$ is not an SCC as it is not maximal. There are three MECs: $\{ 0, 1 \}$, $\{ 2 \}$, and $\{ 3 \}$. While state 2 has an edge to 1, it is not in the same MEC as it cannot go back with probability 1.

We also use models that are *Markov automata* (MA) [16] and *probabilistic timed automata* (PTA) [37]. Untimed reachability on a MA can be checked on its embedded MDP, while PTA can be converted to MDP using e.g. digital clocks [36].

Verification by Refinement. We aim for efficient verified executable code. This requires reasoning about the high-level behaviour of algorithms as well as about lower-level concepts like efficient data structures. To keep these independent concerns separate, we use an iterative *refinement* approach:

We represent the algorithm with the *nondeterministic result* (nres) monad of the Isabelle Refinement Framework (IRF) [42]. It has two possible states: *result* and *fail*. The former captures the set of outputs of all non-deterministic behaviours (e.g. picking an element of a set) of a program while the latter occurs if any behaviour of the program fails (e.g. non-termination). For abstract program A and concrete program C, the *refinement relation* $C \leq \Downarrow R\, A$ holds iff each result of C relates to a result of A via relation R. If A fails, then C always refines it. We use predefined relations like R_{size} and R_{bool} to relate natural numbers and booleans to 64 and 1 bit words, respectively, or $br\ \alpha\ I = \{(c,a).\ a{=}\alpha\ c \wedge I\ c\}$ to build a relation from *abstraction function* α which converts concrete data to abstract data and *invariant* I that holds if the data is in valid form. We use notation $(C,A) \in [\lambda\ a_1 ... a_n.\ P\ a_1 ... a_n]\ R_1 \to \ldots \to R_n \to R$ for

$P\ a_1 ... a_n \Longrightarrow (c_1, a_1) \in R_1 \ldots \Longrightarrow (c_n, a_n) \in R_n \Longrightarrow (C\ c_1 ... c_n) \leq \Downarrow\ R\ (A\ a_1 ... a_n)$

where P is a precondition over the abstract program. To refine e.g. addition of natural numbers $(a + b)$ to addition of 64-bit words, we need the precondition $a + b \leq 2^{63} - 1$; the maximal value of 64-bit signed words.

As they are transitive, we can compose refinements. The final step is an automatic refinement to a model of LLVM using the sepref tool [39]. It uses *assertions* of separation logic [49] to map data structures to concrete memory contents; e.g. A'_{size} and A'_{bool} map 64 and 1 bit words to memory, respectively,

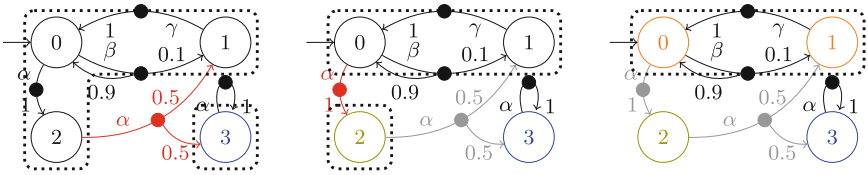

Fig. 2. An execution of the MEC algorithm using 3 iterations.

and A_{list} maps a list to memory using a heap. We combine relations and assertions through composition; e.g. $A_{size} = R_{size}\ O\ A'_{size}$ maps natural numbers to memory.

Example 3. We show an example of a bitset, abstractly represented as a *nat set*. We implement operation *sget*, which tests whether a value is in a bitset.

$(*1*)$ *sget bs i* $= i \in bs$ $\quad$ *bs_get bs i* $\equiv$ $(bi\ !\ (i\ div\ 64)\ !!\ (i\ mod\ 64))$
$(*2*)$ *bs_α bs* $=$ *Collect* $(\lambda\ i.\ bitset\_get\ i\ bs \wedge i < 64 * length\ bs)$
bs_inv n bs $= (n \leq 64 * length\ bs)$ $\quad R_{bs} = br\ bs\_\alpha\ bs\_inv$
$(*3*)$ $(bs\_get,\ sget) \in R_{bs}\ n \to (R_{nbn}\ n) \to R_{bool}$
$(*4*)$ $(bs\_geti,\ bs\_get) \in [\lambda(i,l).\ i<length\ l * 64]\ A_{size} \to A_{list} \to A_{bool}$
$(*5*)$ $A_{bs} = R_{bs}\ O\ A_{list}$ $\quad (bs\_geti,\ sget) \in (A_{nbn}\ n) \to A_{bs} \to A_{bool}$

Here, we (1) define an abstract function *sget*, which is a membership test, and an implementation *bs_get* over *bs::64 word list* (i.e. a list of 64-bit binary words) and an index *i::nat* . This function obtains the i-th bit in the sequence: $bs\ !\ j$ obtains the j-th word and $x\ !!\ k$ the k-th bit in word x. We then (2) relate *64 word list* to a *nat set* using R_{bs}; we provide bs_α as abstraction function to convert *64 word list*s to *nat set*s, and *bs_inv* as invariant that makes sure that n values fit in our bitset. Next, (3) we prove refinement of *sget* to *bs_get*. $R_{nbn}\ n$ maps all values up to n to themselves. (4) Function *bs_geti* is an LLVM program *automatically generated* by sepref and refines *bs_get*. The precondition guarantees that the index is in bounds. Finally (5) through composition we obtain A_{bs} that maps a *nat set* to a bitset on the heap. This allows sepref to generate LLVM code for every occurence of *sget*. Note that we simplified the notation e.g. to match the relation refinement and we omitted notation for (non-)destructive heap access.

3 Correctness of the MEC Algorithm

The standard MEC algorithm iteratively culls the MDP as follows: (1) Calculate the SCC decomposition of the current MDP, (2) find the SCCs that are MECs, and (3) remove the found MECs, and remove all transitions with branches to a different SCC. Figure 2 shows 3 iterations of this algorithm on the MDP of Fig. 1. SCCs are marked by dotted lines, states of SCCs that are MECs are coloured, and culled branches in the current/previous iteration are red/gray. This algorithm is loosely based on those of [3,10] where [10] uses an attractor

computation to remove more states per iteration for which we were unable to find an efficient implementation while [3] excludes step 2, not identifying MECs early, which means computations may be repeated on them. Our approach is based on the existing code in mcsta, which includes step 2 and omits the attractor computation.

3.1 Abstract MDP Structure

We represent the MDP structure as mapping each state to a list of lists of states, i.e. it is of type $'a\ mdp\_K = 'a \Rightarrow 'a\ list\ list$. We chose a list-based representation over a set-based one for straightforward compatibility with our earlier SCC implementation [28] while still being abstract enough for our purposes. The MEC algorithm takes the states ($S_0 :: 'v\ set$) and the MDP kernel ($K_0 :: 'v\ mdp\_K$) as parameters. We use Isabelle/HOL's locale mechanism for general constructs. A locale creates a block in which user-specified assumptions hold. We define an MDP locale with some natural well-formedness assumptions:

definition *closed_mdp* $S\ K \equiv \forall u \in S.\ \forall\ a \in set\ (K\ u).\ set\ a \subseteq S$
locale *mdp* = **fixes** $S :: 'v\ set$ **and** $K :: 'v\ mdp\_K$ +
assumes *1:* $u \in S \Longrightarrow [] \notin set\ (K\ u)$ **and** *2: finite* S **and** *3: closed_mdp* $S\ K$

This locale states that transitions have at least one branch, the state space is finite, and the MDP is closed, i.e. all transitions starting in S end in S.

3.2 Specification

Let *sc* $S\ K$ denote that the MDP is strongly connected. Given *mdp* $S\ K$:

sub_mdp $S_1\ K_1\ S_2\ K_2 \equiv S_1 \subseteq S_2 \wedge (\forall u \in S_1.\ set\ (K_1\ u) \subseteq set\ (K_2\ u))$
is_ec $S'\ K' \equiv S' \neq \{\} \wedge$ *sub_mdp* $S'\ K'\ S\ K \wedge$ *closed_mdp* $S'\ K' \wedge$ *sc* $S'\ K'$
is_mec $S'\ K' \equiv$ *is_ec* $S'\ K' \wedge (\nexists S''\ K''.$ *is_ec* $S''\ K'' \wedge$ *psub_mdp* $S'\ K'\ S''\ K'')$

where *psub_mdp* is the proper *sub_mdp*. Here, *sub_mdp* $S_1\ K_1\ S_2\ K_2$ holds if (S_1,K_1) is a sub-MDP of (S_2,K_2). We allow reorderings and (de)duplications of the transitions as they do not alter the MDP structure. An EC is a strongly connected, closed sub-MDP with at least one state. A MEC is an EC that is not a proper sub-MDP of another EC. With these definitions, we *specify* MEC algorithms as those that return a list with the MECs of the input MDP structure:

definition *compute_MEC_spec* $\equiv$ **spec** $(\lambda\ r.\ set\ r = \{S'\ |S'\ K'.$ *is_mec* $S'\ K'\ \})$

3.3 Abstract Algorithm

We now *define* the MEC algorithm, focusing on its abstract, high-level behaviour; we refine this to concrete data structures in Sect. 4. The definition in Isabelle is:

```
1  compute_MEC ≡ do {
2      let (M,S,K) = op_init_mdp (S₀,K₀);
3      (M,S,K) ← while (compute_MEC_invar S₀ K₀) op_states_non_empty
4              (λ(M,S,K). do {
5              C ← compute_sccs (M,S,K);
6              (C,V) ← identify_mecs C (M,S,K);
7              (M,S,K) ← cull_graph (C,V) (M,S,K);
8              return (M,S,K)
9      }) (M,S,K);
10     return (op_get_mecs (M,S,K))
```

We initialise the loop state in line 2 as an empty list M to store the MECs and $S = S_0$ and $K = K_0$. We bundle this data into one tuple so that we can refine them through a single assertion in Sect. 4.2. We iterate as long as there are states for which we have not found a MEC in line 3. We then perform the three-step process described earlier: We (1) compute C in line 5 such that *scc_list* $C\ S\ K$ holds (i.e. C is a distinct list of all SCCs of the graph structure of S and K). We then (2) obtain list V which contains all SCCs of C that are also MECs in line 6. We finally (3) remove MECs and transitions between different SCCs in line 7. At the end of the program, we extract M which contains the MECs.

These operations are defined by high-level behaviour; e.g. for *cull_graph*:

cull_graph $(C,V)\ (M,S,K) \equiv$ **spec**
$(\lambda\ (M',S',K').\ M' = M@V \wedge S' = S - \bigcup(set\ V) \wedge culled\_edges\ C\ V\ K\ K')$

We elided the definition of predicate *culled_edges* which holds if K' only contains the transitions in K whose branches all remain within the same SCC as their source. Also, *cull_graph* adds the identified MECs to M and removes them from S.

These definitions are still far from an efficient implementation. We first refine each operation to a control flow (definitions elided). The operations of that control flow are implemented in the respective data structures in Sect. 4. The SCC algorithm has been refined separately in [28].

Invariant. We define the following invariant for the main loop of the algorithm:

locale *compute_MEC_invar* = *mdp* $S_0\ K_0$ **for** $S_0\ K_0\ (M,S,K)$ +
assumes *1:* $S \cap \bigcup(set\ M) = \{\}$ **and** *2:* $S_0 = \bigcup(set\ M) \cup S$
and *3: pairwise disjnt* $(set\ M)$ **and** *4: distinct* M **and** *5: sub_mdp_of* $S\ K\ S_0\ K_0$
and *6: mdp* $S\ K$ **and** *7: is_mec* $S'\ K' \Longrightarrow S' \in set\ M \vee sub\_mdp\_of\ S'\ K'\ S\ K$
and *8:* $S \neq \{\} \Longrightarrow mdp\_def.is\_mec\ S\ K\ S'\ K' \Longrightarrow is\_mec\ S'\ K'$
and *9:* $S' \in set\ M \Longrightarrow \exists K'.\ is\_mec\ S'\ K'$ **and** *10:* $scc\_list\ C\ S\ K \Longrightarrow a \in set\ (K_0\ u)$
$\Longrightarrow S' \in set\ C \Longrightarrow u \in S' \Longrightarrow \forall v \in set\ a.\ v \in S' \Longrightarrow a \in set\ (K\ u)$

It states that (1) the states (S) and MECs (M) are disjoint and (2) cover the original statespace. Also, (3,4) M is pairwise disjoint and contains no duplicates. The current graph structure is (5) a sub-MDP of the input and (6) an MDP itself. Further, (7) all MECs are either in M or in the current graph structure, (8) a MEC of the current graph structure is a MEC of the original one, (9) each element of M is a MEC, and (10) transitions in the original graph structure within

one SCC are preserved in the current one. We have proven that the invariant is preserved throughout the while-loop and if S is empty the specification holds.

Termination is guaranteed as every non-empty graph has at least one *bottom-SCC* (BSCC), i.e. an SCC with no outgoing edges. Our algorithms finds MECs by identifying BSCCs; we find at least one MEC per iteration and remove it from the state space. Since the state space is finite, we necessarily terminate.

4 Data Structures and Refinement

The next step is to define the data structures to efficiently implement the abstract operations specified in Sect. 3.3. For input and output, we formalize the data structures that mcsta uses, so that we can integrate our implementation without costly conversions. Using the IRF, the refinement is done modularly, and in multiple steps to structure the correctness proof and keep it manageable.

4.1 Supplementary Data Structures

We introduce auxiliary data structures that are part of mcsta's data structure:

Intervals. In mcsta, intervals of natural numbers $\{l..<h\}$ are represented as a single 64 bit word, where the 20 most significant bits encode the length, and the 44 remaining bits encode the starting point l. Like in [28], we express this refinement in two levels: the relation A_{sn} relates a 64 bit word to a pair (n, i) of type $sn = nat \times nat$, and the functions $sn\_intv\ (n,i) = \{i..<i+n\}$ and $ls\_intv\ (n,i) = [i..<i+n]$ represent these as set and list, respectively.

Disjoint Nat Set List. Our implementation requires a map from states to indices of MECs or SCCs. Low-valued indices are MECs while high-valued ones are SCCs. Abstractly, we represent this as two lists of sets of states such that each state occurs at most once. We highlight some operations here:

type_synonym *dslt = nat set list* × *nat set list*
d_empty :: dslt **where** *d_empty* = ([],[])
d_count1 :: dslt ⇒ *nat* **where** *d_count1*(*xs*,*ys*) = *length xs*
d_move1 :: dslt ⇒ *nat* ⇒ *nat* ⇒ *dslt* **where** *d_move1* (*xs*,*ys*) *v i* =
((*map* (λ *x*. *x* − {*v*}) *xs*)[*i*:=(*xs* ! *i*) ∪ {*v*}], (*map* (λ *y*. *y* − {*v*}) *ys*))

Operation *d_empty* constructs a tuple of empty lists, *d_count1* returns the length of the first list, and *d_move1 ds v i* moves state v into index i of the first list (removing it from anywhere else if necessary). Every operation on the first list (with suffix 1) has a corresponding operation on the second one (with suffix 2). We omit some further operations for this data structure.

We implement the data structure as an array map that maps values to the index of the set that they are in. This means that we flatten the two lists into one map. We introduce a bound L which is the maximal size of the first list.

Indices $i < L$ represent indices to sets in the first list; indices $i \geq L$ represent index $i - L$ in the second list. Values that are not in any set get a -1 entry. We capture this mapping in assertion A_{dslt}.

Example 4. Let $L = 3$ and $N = 5$. Then A_{dslt} maps the abstract *nat dslt* $([\{1\}, \{\}, \{2\}], [\{0, 4\}])$ to array $a = [3, 0, 2, -1, 3]$: We have $a[1] < L$ so value 1 must be in the first list; since $a[1] = 0$, we find value 1 in the set at index 0. We also have $a[4] \geq L$ so value 4 is in the second list. Since $a[4] = 3$, we find it at index $3 - L = 0$. Lastly, we have $a[3] = -1$, which means that value 3 is not in any of the sets.

4.2 The mcsta Data Structure

The mcsta data structure is a tuple $SS = (St, Tr, Br, Av, Ta)$. St, Tr and Br represent the states, transitions, and branches of the MDP structure, respectively. Additionally, Av and Ta are sets representing the avoid and target states of the reachability property being verified (corresponding to sets A and T of Def. 2). We define a relation R_{Mdi} that relates our model of the mcsta data structure to an MDP structure:

$S_0\_\alpha\ N = \{0..<N\}$ $\quad K_0\_\alpha\ N\ SS :: nat\ mdp\_K\ (*\ elided\ *)$
$MG_0\_\alpha\ N\ SS = (S_0\_\alpha\ N,\ K_0\_\alpha\ N\ SS)$

locale *Md_input_inv* = *mdp* S_0::*nat set* K_0 **fixes** N (St,Tr,Br,Av,Ta) +
assumes *1:* $N = length\ St$ **and** *2:* $Av \subseteq \{0..<N\}$ **and** *3:* $Ta \subseteq \{0..<N\}$
and *4:* $i < length\ St \Longrightarrow uid\ (St!i) \leq length\ Tr$ **and** *5:* $i < length\ Tr \Longrightarrow$
$uid\ (Tr!i) \leq length\ Br$ **and** *6:* $i < length\ Br \Longrightarrow Br!i < length\ St$
and *7:* $i < length\ St \Longrightarrow cnt\ (St!i) > 0$ **and** *8:* $i < length\ Tr \Longrightarrow cnt\ (Tr!i) > 0$
and *9:* $i < length\ Tr \Longrightarrow j < i \Longrightarrow sn\_intv\ (Tr\ !\ i) \cap sn\_intv\ (Tr\ !\ j) = \{\}$
and *10:* $S_0 = S_0\_\alpha\ N$ **and** *11:* $K_0 = K_0\_\alpha\ N\ (St,Tr,Br,Av,Ta)$

$R_{Mdi}\ S_0\ K_0\ N = br\ (MG_0\_\alpha\ N)\ (Md\_input\_inv\ S_0\ K_0\ N)$

The states of an MDP in mcsta are numbered from 0 to $N-1$. K_0_α derives the kernel from the data structure as follows: St and Tr are lists of intervals (represented as tuples, see Sect. 4.1) and Br is a list of state indices. If $St\ !\ v = (n,i)$, the next n transitions starting at index i in Tr belong to state v. This means that a transition is an index $i \leq t < n + i$. Similarly, $Tr\ !\ t$ is a tuple pointing to an interval of indices on Br. A branch is thus an index $i \leq b < n + i$ and $Br\ !\ b$ is the target state of the branch. Furthermore, if $v \in Av \vee v \in Ta$, we ignore all outgoing edges. The invariant states the following: (1) it fixes the number of states to N for the bounds calculations in sepref. It states that (2,3) our target and avoid states are a subset of S_0. It also states that (4,5,6) St points to valid indices on Tr, Tr points to valid indices on Br and Br points to a valid indices on St, (7,8) St and Tr do not contain empty intervals, (9) transitions cannot overlap, and (10,11) the input MDP structure remains constant. The relation R_{Mdi} relates the input MDP structure to the concrete data structure. Using

sepref and composition, we obtain the according assertion A_{Mdi} . We refine the concrete data structure to LLVM using the IRF standard library and the supplementary data structures from Sect. 4.1: We implement *St* and *Tr* as lists of bit-packed intervals, *Br* as a list of 64-bit values, and *Av* and *Tr* as bitsets.

Example 5. One possibility to represent Fig. 1 is $St = [(2,0),(1,2),(1,3),(1,3)]$, $Tr = [(1,0),(2,1),(1,3),(2,4),(1,6)]$, $Br = [2,0,1,0,3,1,3]$. For state 0 we have $St!0 = (2,0)$, i.e. it has 2 successors (α and β) starting at index 0. Similarly, for transition 1 (corresponding to β in this case) we have $Tr!1 = (2,1)$, which means that this transition has 2 branches starting at index 1 in *Br* (i.e. state $Br!1 = 0$ and $Br!2 = 1$). Note that we have not defined *Av* and *Ta* yet as these are dependent on the property. If we assume the property of Example 1 then $Av = \{1\}$ and $Ta = \{3\}$. These translate to the bitsets ...0010 and ...1000 respectively, which removes the outgoing transitions from those states (not visualized).

mcsta directly passes this data to our implementation. However, as we have seen in Sect. 3.3, our algorithm needs to be able to efficiently remove states and transitions. The data structure that we have presented so far cannot implement this functionality efficiently.

Cullable MDP Structure. The implementation of *op_init_mdp* from Sect. 3.3 supplements the input data structure with the *dslt* data structure from Sect. 4.1, which is a tuple of lists of sets of states. States in the first list of the tuple are removed while states in the second one are not. Furthermore, a transition starting in some state v is "activated" if all branches of that transition are within the same set. If any branch connects different sets, the transition is deactivated. With this approach, we place states of the same SCC into the same set, disabling transitions between SCCs in the process. Additionally, we use the tuple structure to distinguish between MECs in the first list and SCCs in the second, which means that it eventually stores the MEC decomposition.

is_act u t = *t*∈*sn_intv* (*St*!*u*) ∧ (∀*b*∈*sn_intv* (*Tr*!*t*) *b* ⟶ *d_eqset Mm u* (*Br*!*b*))
S_α N Mm = {*v*. *v* < *N* ∧ (∀ *i* < *length* (*fst Mm*). *v* ∉ (*fst Mm*) ! *i*)}
K_α N (*SS,Mm*) :: *nat mdp_K* (∗ *elided* ∗)
MSK_α N (*SS,Mm*) = (*fst Mm,S_α N Mm,K_α N* (*SS,Mm*))

locale *Md_mdp_cullable_inv* = *Md_mdp_input_inv* S_0 K_0 *N* (*St,Tr,Br,Av,Ta*) + *mdp* S_0 K_0 **for** S_0::*nat set* **and** K_0 *N* (*St,Tr,Br,Av,Ta,Mm,Nr*) + ...

With *is_act* we test if a transition is activated by checking that all branches are in the same set (*d_eqset* which is a *dslt* operation) as the source state. We use this to derive the culled kernel *K_α* which contains exactly the activated transitions of K_0_α. *S_α* omits the states that have been identified as a MEC. The MECs are stored in the first list of *Mm* directly. Variable *Nr* is the number of remaining states, i.e. for which no MEC has been identified. We use this for implementing the termination criterion. We omit the definition of the invariant which mainly concerns well-formedness of *Mm*. This data structure allows us to

efficiently implement *cull_graph* from Sect. 3.3 by putting states from the same SCC into the same set. This update is straightforward to implement as it merely involves updating the value of unfinished states in the map to the corresponding index of the SCC, which is also stored in a map for the SCC algorithm.

Example 6. Assuming the middle situation in Fig. 2, consider the input data from Example 5 and additionally $Mm = ([\{3\}, \{2\}], [\{0, 1\}])$. For state 0, β is activated since its branches (to 0 and 1) are in the same set as the source (0). However, α branches to 2 which is in another set, so the transition was deleted.

4.3 Filter List

Given the number of states N and the number of MECs M, we have $M \leq N$. This is essential for our bounds calculation: Since the ID of a MEC is represented as a 64-bit value, we need to bound M. We require a "dense" indexing for the MECs, i.e. they must be numbered from 0 to $M - 1$ efficiently. This way, we can do our bounds calculation solely using N, which we know a priori. We do so by iterating over all states, and if we find any transitions leaving the SCC, the SCC is not a MEC and we filter it. We implemented a filter set and filter list to implement this memory- and time-efficiently.

The filter list is an extension of any data structure representing a list. On the abstract layer, it is of type *'a list* × *'a list* where the first list of the pair is the original list that we want to filter and the second one is the filtered variant. Concretely, it is of type *'v list* × *nat option list* × *nat* . The first list (*xs*) is the original list, the second list (*ids*) is a map containing indices, and the natural number is a counter representing the length of the filtered list *c*. The list *ids* is the core of this data structure. It maps an index *i* of the unfiltered list to *None* if *xs* ! *i* is filtered or to *Some j* if *xs* ! *i* is at index *j* in the filtered list. The filter set is similar to the filter list but *ids* either maps to *None* (entry is filtered) or *Some 0* (entry is unfiltered). We then convert the filter set to a list by assigning a unique index to each unfiltered entry.

Example 7. Assume unfiltered list [*a,b,c,d*] out of which we want to filter *a* and *c*. Abstractly, we have ([*a,b,c,d*],[*b,d*]). Its concrete implementation is the triple ([*a,b,c,d*],[*None,Some 0,None,Some 1*],*2*). Since *a* (index 0) and *c* (index 2) are filtered, we have *ids* ! *0* = *ids* ! *2* = *None*. Similarly, we find *b* (index 1) at index 0 in the filtered list. Therefore *ids* ! *1* = *Some 0*.

5 Code Generation and Integration

Using the algorithm of Sect. 3 and the data structures of Sect. 4, we derive an LLVM program *Md_compute_MEC* using sepref. Through transitivity of the refinement relation, we can show that this program refines the specification from Sect. 3.2. The IRF provides the setup to extract a separation logic Hoare triple from our correctness proof. Let $\star$ be the separation conjunction. Then we obtain:

theorem *Md_compute_MEC_htriple: llvm_htriple* (
(∗1∗) A_{size} *N ni* ⋆ A_{Mdi} *N* S_0 K_0 (S_0, K_0) *mdpi*
(∗2∗) ⋆ *ll_pto mdpi p_mdpi* ⋆ *ll_pto anything resp*
(∗3∗) ⋆ (*mdp* S_0 K_0 ∧ $N < 2^{62}$ ∧ $S_0 = \{0..<N\}$))
(∗4∗) (*Md_compute_MEC ni p_mdpi resp*)
(∗5∗) (λ_. *EXS M resi.*
(∗6∗) A_{size} *N ni* ⋆ A_{Mdi} *N* S_0 K_0 (S_0, K_0) *mdpi* ⋆ A_{Mdo} *N N M resi*
(∗7∗) ⋆ *ll_pto resi resp* ⋆ *ll_pto mdpi p_mdpi*
(∗8∗) ⋆ (*set M* = {S' |S' K'. *mdp.is_mec* S_0 K_0 S' K' }))

where A_{Mdo} is derived from A_{dslt} mapping only its first list to memory given that the second list is empty. The precondition consists of several parts: (1) The input consists of a value N representing the number of states with its 64 bit representation *ni* and an input MDP structure (S_0, K_0) which is represented in memory by *mdpi*. (2) We are provided a pointer to the MDP structure (*p_mdpi*) and one to an address where we can store our output. (3) (S_0, K_0) is an MDP structure that has fewer than 2^{62} states and a dense numbering S. Given these preconditions we (4) run our program *Md_compute_MEC* with the specified input parameters. We then get (5) a MEC decomposition M and its representation in memory *resi* such that (6) the input parameters are preserved and we additionally obtain the MEC decomposition, (7) the provided pointer (*resp*) points to that decomposition and (8) M is the set of MECs.

The IRF has built-in functionality to translate *Md_compute_MEC* to LLVM code with a header file, which can be called as an external function from mcsta. Note that we use indirection through pointers to avoid problems with different ABIs when passing structures as parameters or return values. It is invoked as:

export_llvm *Md_compute_MEC* **is**
void compute_MEC(*modest_size_t, modest_input_mdp_t* ∗, *mec_output_t* ∗)

5.1 Compatibility with mcsta

We refer to the verified LLVM code as the *verified* implementation and to the pre-existing C# implementation in mcsta as the *integrated* implementation. While the data format of the verified implementation is compatible with mcsta, there were some important differences that we fix using glue code for post-processing:

First, collapsing the MECs for interval iteration, which is currently not verified, requires the MECs to be sorted in exploration order. The algorithm we formalised does not do that out of the box and we are not aware of an algorithm that does preserve this order. That is why we decided to reorder the MEC indices as a post-processing step.

Second, the integrated algorithm groups all target states into one collapsed target state and does the same for avoid states. The verified algorithm puts each target and avoid state in its own MEC. Both approaches are correct, but the verified algorithm therefore calculates at least as many MECs as the integrated one. We considered formalising this collapsing of states in our proofs, but we

decided against it as it would complicate them. Since we decided for the post-processing approach for reordering, we included the latter as well.

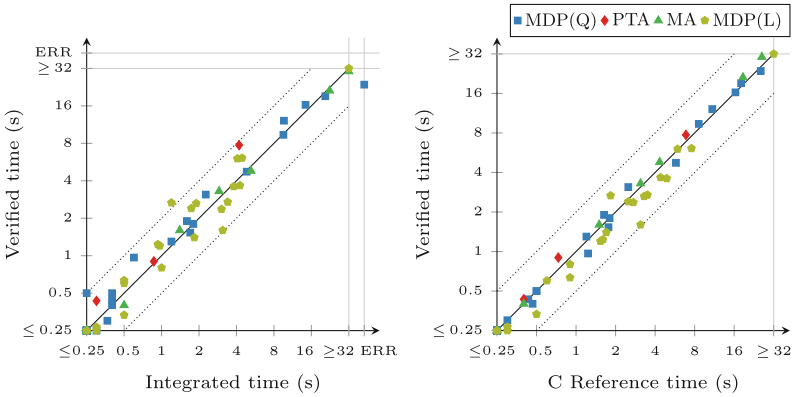

Fig. 3. Comparison of runtime to complete the MEC decomposition routine

6 Experimental Evaluation

We have embedded the verified implementation into the mcsta tool of the Modest Toolset. Since it uses mcsta's regular input and output data structures, we do not need any expensive conversions and minimal glue code with negligible runtime. Furthermore, we implemented a *reference* implementation in C++ that we manually optimised. We now compare the performance of these two and the integrated implementation.

6.1 Experimental Setup

We use all applicable benchmarks (i.e. all MDP models, PTA models transformed into their digital clocks MDP, and MA transformed into the embedded MDP for untimed properties) from the Quantitative Verification Benchmark Set (QVBS) [27], which however rarely contain any nontrivial MECs. MEC decomposition is still necessary since we do not know a priori whether nontrivial MECs exist in a model and the algorithm may still require multiple iterations to obtain this result. To study the performance when nontrivial MECs exist, we adapt a benchmark set for long-run average rewards (LRA) from [20]. We test one reachability property per model to trigger the MEC algorithm and inflated the parameters to challenge the implementations. This benchmark set contains the *mer* Mars rover case study from [19], the *sensors* case study from [34], and other models from the PRISM benchmarks. We added parameters where sensible to allow scaling the model. We also created MDP adaptations of the stochastic

games originally hand-crafted to contain interesting MEC structures for the evaluation of [35]. This gave us 61 benchmark instances to test our implementation on. We aimed to benchmark models between 500,000 and 100 million states. Smaller models terminate too quickly to benchmark while larger models run out of memory. We ran all benchmarks on an Intel Core i7-12700H system with 32 GB of RAM running Linux Mint 21.3.

6.2 Results

We ran each benchmark three times and report the averages of those runs. Figure 3 compares the wall clock runtime, with the left scatter plot comparing the verified to the integrated implementation and the right comparing the verified to the reference implementation. Each dot is a pair of runtime values for one benchmark instance. We distinguish benchmarks for PTA, MA and MDP from the QVBS (Q) or the LRA benchmarks (L). With our setup, we found that our verified implementation performs on par with the reference and integrated implementations, with a slight edge for the integrated implementation, but with little optimisation potential. We observe that this pattern also seems to hold independently of the type of model. One noteworthy outcome is the fact that the integrated implementation crashes for one instance whereas the reference and verified implementations do not. This is caused by the integrated implementation requiring more memory. We compared peak memory usage (working set) of the verified and integrated implementations. While this approach may be influenced by external factors like garbage collection, it can still provide a useful indication of relative memory consumption. Peak memory was higher for the integrated implementation in 42 out of the 61 instances. On average, the integrated implementation used about 8.2% more memory than the verified implementation. In isolated instances it reached up to 36.4% more. In comparison, the verified implementation used at most 28.4% more than the integrated implementation for isolated instances. The instance that crashed (*tireworld* with $n = 45$) lies on the verge of what a laptop with 32 GB of RAM can process: Peak memory reached almost 31 GB for this instance using the verified algorithm.

7 Conclusion

We have formally verified a MEC decomposition algorithm in Isabelle/HOL. As far as we know, this is the first such formalization. We have refined this algorithm down to LLVM and generated efficient executable code which we embedded into the mcsta probabilistic model checker of the Modest Toolset. This is a step towards a fully verified model checking toolchain. We aim to replace algorithms in the toolchain piece by piece, monitoring the performance impact in each step. Where previous attempts at formally verified model checkers have not been competitive in terms of performance and functionality, our approach yields comparable performance to manual implementations. Additionally, if desired, cross-usage with other (unverified) functionality is possible. While the performance of our

verified implementation is comparable to the integrated implementation, it is a clear improvement over the integrated implementation in terms of memory usage.

Future Work. Comparisons with the manual implementations suggest that the verified implementation does not have a lot of optimization potential. We consider it more useful to focus on other algorithms at this point. One candidate is the improved MEC algorithm by Chatterjee et al. [10], which has a better theoretical complexity than our implementation; deriving a competitive implementation from this would be highly relevant. Another candidate is the interval iteration algorithm [22] which uses the MEC algorithm as a pre-processing step. An efficient implementation of interval iteration requires a representation of real or rational numbers with low overhead. Unverified implementations rely on IEEE floating-point values (floats) which are suitable for high-performance computations but come with rounding errors [23]. This requires an extension of the IRF in order to refine real numbers to floats and reason about rounding.

Data availability. The proofs and benchmarks presented in this paper are archived and available at https://doi.org/10.4121/3f2a4539-e69b-4d16-b665-530c1abddfbc [32].

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
2. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Handbook of Model Checking, pp. 963–999. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_28
3. Christel Baier and Joost-Pieter Katoen. Principles of Model Checking. MIT Press (2008)
4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
5. Bellman, R.: A Markovian decision process. J. Math. Mech. **6**(5), 679–684 (1957)
6. Biere, A., Van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: Stewart, D., Weissenbacher, G., (eds.) 2017 International Conference on Formal Methods in Computer Aided Design FMCAD, p. 9. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102233
7. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Ouaknine, J., Worrell, J.: Model checking real-time systems. In: Handbook of Model Checking, pp. 1001–1046. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_29
8. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. J. Autom. Reasoning **60**(1), 3–21 (2018). https://doi.org/10.1007/s10817-017-9418-4
9. Budde, C.E., et al.: On correctness, precision, and performance in quantitative verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12479, pp. 216–241. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-83723-5_15

10. Chatterjee, K., Henzinger, M.: Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In: Randall, D. (ed.) 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1318–1336. SIAM (2011). https://doi.org/10.1137/1.9781611973082.101
11. Chen, R., Lévy, J.-J.: A semi-automatic proof of strong connectivity. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 49–65. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_4
12. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
13. Clarke, E., Mishra, B.: Automatic verification of asynchronous circuits. In: Clarke, E., Kozen, D. (eds.) Logic of Programs 1983. LNCS, vol. 164, pp. 101–115. Springer, Heidelberg (1984). https://doi.org/10.1007/3-540-12896-4_358
14. Alfaro,L.: Formal verification of probabilistic systems. PhD thesis, Stanford University, USA (1997). https://searchworks.stanford.edu/view/3910936
15. Doyen, L., Frehse, G., Pappas, G.J., Platzer, A.: Verification of hybrid systems. In: Handbook of Model Checking, pp. 1047–1110. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_30
16. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom, pp. 342–351. IEEE Computer Society (2010). https://doi.org/10.1109/LICS.2010.41
17. Eisentraut, J., Kelmendi, E., Křetínský, J., Weininger, M.: Value iteration for simple stochastic games: Stopping criterion and learning algorithm. Inf. Comput. 285(Part), 104886 (2022). https://doi.org/10.1016/J.IC.2022.104886
18. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_31
19. Feng, L., Kwiatkowska, M., Parker, D.: Automated learning of probabilistic assumptions for compositional reasoning. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 2–17. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19811-3_2
20. Grover, K., Weininger, M., Kretinsky, J.: QComp LRA results. Zenodo (2023). https://doi.org/10.5281/zenodo.8219191
21. Gupta, A., Kahlon, V., Qadeer, S., Touili, T.: Model checking concurrent programs. In: Handbook of Model Checking, pp. 573–611. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_18
22. 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
23. Hartmanns, A.: Correct probabilistic model checking with floating-point arithmetic. In: TACAS 2022. LNCS, vol. 13244, pp. 41–59. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_3
24. Hartmanns, A.: An overview of Modest models and tools for real stochastic timed systems. In: Dubslaff, C., Luttik, B. (eds.) 5th Workshop on Models for Formal Analysis of Real Systems (MARS), vol. 355 EPTCS, pp. 1–12 (2022). https://doi.org/10.4204/EPTCS.355.1

25. Hartmanns, A., Hermanns, H.: The Modest Toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_51
26. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 488–511. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_26
27. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 344–350. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_20
28. Hartmanns, A., Kohlen, B., Lammich, P.: Fast verified SCCs for probabilistic model checking. In: André, É., Sun, J. (eds.) 21st International Symposium on Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 14215, pp. 181–202. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-45329-8_9
29. Heule, M., Hunt, W., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 269–284. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_18
30. 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
31. Holzmann, G.J.: Software model checking with SPIN. Adv. Comput. **65**, 78–109 (2005). https://doi.org/10.1016/S0065-2458(05)65002-4
32. Kohlen, B., Hartmanns, A., Lammich, P.: Artifact for the paper "Efficient formally verified maximal end component decomposition for MDPs". 4TU.ResearchData (2024). https://doi.org/10.4121/3f2a4539-e69b-4d16-b665-530c1abddfbc
33. Kolobov, A., Mausam, M., Weld, D., Geffner, H.: Heuristic search for generalized stochastic shortest path MDPs. In: Bacchus, F., Domshlak, C., Edelkamp, S., Helmert, M. (eds.) 21st International Conference on Automated Planning and Scheduling (ICAPS). AAAI, (2011). http://aaai.org/ocs/index.php/ICAPS/ICAPS11/paper/view/2682
34. Komuravelli, A., Păsăreanu, C.S., Clarke, E.M.: Assume-guarantee abstraction refinement for probabilistic systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 310–326. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_25
35. Křetínský, J., Ramneantu, E., Slivinskiy, A., Weininger, M.: Comparison of algorithms for simple stochastic games. Inf. Comput., 289(Part), 104885 (2022). https://doi.org/10.1016/J.IC.2022.104885
36. Kwiatkowska, M., Norman, G., Parker, D. et al.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods Syst. Des., 29(1), 33–78, (2006). https://doi.org/10.1007/s10703-006-0005-2
37. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theor. Comput. Sci. **282**(1), 101–150 (2002). https://doi.org/10.1016/S0304-3975(01)00046-9
38. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 325–340. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_21
39. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Harrison,J., 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

40. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2020). https://doi.org/10.1007/s10817-019-09525-z
41. Lammich, P.: Refinement of parallel algorithms down to LLVM. In: Andronick, J., Moura, L. (eds.) 13th International Conference on Interactive Theorem Proving (ITP). LIPIcs, vol. 237, pages 24:1–24:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.24
42. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_12
43. Schäffeler, M., Abdulaziz, M.: Formally verified solution methods for Markov decision processes. In: 37th AAAI Conference on Artificial Intelligence, pp. 15073–15081 (2022). https://doi.org/10.1609/aaai.v37i12.26759
44. Neumann, R.: Using Promela in a fully verified executable LTL model checker. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 105–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_7
45. Platzer, A.: Logical Foundations of Cyber-Physical Systems. (2018). https://doi.org/10.1007/978-3-319-63588-0
46. Pottier, F.: Depth-first search and strong connectivity in Coq. In: Vingt-sixièmes journées francophones des langages applicatifs (JFLA) (2015)
47. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics. Wiley (1994). https://doi.org/10.1002/9780470316887
48. Quatmann, T., Katoen, J.-P.: Sound value iteration. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 643–661. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_37
49. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings, pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
50. Roberts, R., et al.: Probabilistic verification for reliability of a two-by-two network-on-chip system. In: Lluch Lafuente, A., Mavridou, A. (eds.) FMICS 2021. LNCS, vol. 12863, pp. 232–248. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85248-1_16
51. Steinmetz, M., Hoffmann, J., Buffet, O.: Goal probability analysis in probabilistic planning: exploring and enhancing the state of the art. J. Artif. Intell. Res. **57**, 229–271 (2016). https://doi.org/10.1613/JAIR.5153
52. Vajjha, K., Shinnar, A., Trager, B., Pestun, V., Fulton, N.: CertRL: formalizing convergence proofs for value and policy iteration in Coq. In: Hritcu, C., Popescu, A. (eds.) 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP), pp. 18–31. ACM, (2021). https://doi.org/10.1145/3437992.3439927
53. van den Berg, F., Remke, A., Haverkort, B.R.: iDSL: automated performance prediction and analysis of medical imaging systems. In: Beltrán, M., Knottenbelt, W., Bradley, J. (eds.) EPEW 2015. LNCS, vol. 9272, pp. 227–242. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23267-6_15
54. Wimmer, S., Herbreteau, F., van de Pol, J.: Certifying emptiness of timed Büchi automata. In: Bertrand, N., Jansen, N. (eds.) FORMATS 2020. LNCS, vol. 12288, pp. 58–75. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57628-8_4
55. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 61–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_4

56. Wimmer, S., Mutius, J.: Verified certification of reachability checking for timed automata. In: TACAS 2020. LNCS, vol. 12078, pp. 425–443. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_24
57. Younes, H.L., Littman, M.L., Weissman, D., Asmuth, J.: The first probabilistic track of the international planning competition. J. Artif. Intell. Res. **24**, 851–887 (2005). https://doi.org/10.1613/JAIR.1880