
Article:
Evangelidis, Alexandros, Vázquez, Gricel and Gerasimou, Simos (2026) Accelerating Policy Synthesis in Large-Scale MDPs via Hierarchical Adaptive Refinement. Proceedings of the ACM on Software Engineering. 196. ISSN: 2994-970X (In Press)

https://doi.org/10.1145/3808203

Accelerating Policy Synthesis in Large-Scale MDPs via Hierarchical Adaptive Refinement

ALEXANDROS EVANGELIDIS*, University of York, United Kingdom
GRICEL VÁZQUEZ, University of York, United Kingdom
SIMOS GERASIMOU, Cyprus University of Technology, Cyprus and University of York, United Kingdom

Software-intensive systems, such as software product lines and robotics, utilise Markov decision processes (MDPs) to capture uncertainty and analyse sequential decision-making problems. Despite the usefulness of conventional policy synthesis methods, they fail to scale to large state spaces. Our approach addresses this issue and accelerates policy synthesis in large MDPs by dynamically refining the MDP and iteratively selecting the most fragile MDP regions for refinement. This iterative procedure offers a balance between accuracy and efficiency, as refinement occurs only when necessary. We formally show that the composed policy is near-optimal under standard assumptions, with error bounded by the local solver tolerance and boundary mismatch. Across diverse case studies and MDPs up to 1M states, we demonstrate that our approach achieves up to 2× speedup over PRISM, offering a competitive solution for real-world policy synthesis in large MDPs.

CCS Concepts: • **Theory of computation → Markov decision processes**; • **Computing methodologies** → *Planning under uncertainty*.

Additional Key Words and Phrases: Probabilistic Model Checking, Policy Synthesis, Hierarchical Refinement

ACM Reference Format:
Alexandros Evangelidis, Gricel Vázquez, and Simos Gerasimou. 2026. Accelerating Policy Synthesis in Large-Scale MDPs via Hierarchical Adaptive Refinement. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE196 (July 2026), 23 pages. https://doi.org/10.1145/3808203

1 Introduction

Modern software-intensive systems range from cloud microservices with frequent deployments [39, 50] to self-adaptive systems and autonomous robots navigating dynamic warehouses [28, 57, 58]. These systems must reason rigorously about uncertainty [38, 56]. Markov decision processes (MDPs) capture both nondeterminism and stochasticity, making them an expressive mathematical framework for sequential decision-making under uncertainty [51]. In software engineering (SE), MDPs support the continuous verification of self-adaptive systems [14, 15, 52], dynamic QoS management and optimisation of service-based applications [16], variability-aware model checking of software product lines [19], and robotic software stacks for task and motion planning [2].

However, despite the widespread adoption of MDPs to support decision-making in these complex, software-intensive systems, their practical application is often hindered by the so-called state-space explosion problem [21]. This explosion in the number of states and transitions renders standard policy synthesis algorithms, which determine optimal actions, computationally prohibitive due to

*Alexandros Evangelidis is the corresponding author.

Authors' Contact Information: Alexandros Evangelidis, alexandros.evangelidis@york.ac.uk, University of York, York, United Kingdom; Gricel Vázquez, gricel.vazquez@york.ac.uk, University of York, York, United Kingdom; Simos Gerasimou, simos.gerasimou@york.ac.uk, Cyprus University of Technology, Limassol, Cyprus and University of York, York, United Kingdom.

ACM 2994-970X/2026/7-ARTFSE196
https://doi.org/10.1145/3808203

excessive time and memory requirements [5, 52]. In probabilistic model checking, policy (strategy) synthesis is a verification task that not only checks a quantitative specification but also returns an executable controller (policy) that optimises it [47].

For engineering teams that rely on MDP-based analysis to ensure safety requirements (e.g., "the robot never becomes stuck") or the achievement of performance targets (e.g., "the task must be completed within T time steps"), these scalability and efficiency limitations can significantly undermine the adoption of verification techniques for real-world software systems [47]. Large models can stall verification pipelines and delay releases. For example, even adjusting a single parameter (such as shelf layout) might force re-verification of the entire MDP, escalating development costs and turnaround time [13, 41].

Despite significant research, the scalability of MDP solution methods remains a fundamental challenge [4]. Recent benchmarks [11] report that PRISM [46] and Storm [36] still time out on large instances, whether using symbolic [1], explicit [34], or sampling-based [35] backends.

To address this challenge, we introduce **SHARP**, an approach for **S**calable **H**ierarchical **A**daptive **R**efinement for **P**olicy synthesis. The core of our approach is based upon a "refine-where-needed" principle, which focuses computational effort only on the most critical regions of the state space. SHARP begins with a coarse partitioning of the MDP and iteratively improves it by subdividing and solving state space partitions, yielding a refined and more detailed analysis. Specifically, SHARP hierarchically partitions the state space, solves sub-MDPs with boundary values, and refines blocks whose spread exceeds a depth-adapted threshold, since low-spread blocks are already well-resolved.

Partition-refinement has a long history in probabilistic verification: CEGAR [20, 37, 43] refines abstractions to eliminate spurious counterexamples, bisimulation-based methods [27] aggregate states with similar dynamics into abstract states, and parametric abstraction-refinement [41] analyses hierarchical MDPs. These techniques are primarily *verification-driven*: they are designed to establish quantitative bounds and may employ state aggregation as long as the bound is preserved. Policy synthesis is more demanding, because it must commit to an action choice at each concrete state; aggregating states that require different actions can of course degrade the resulting controller [5] (though not necessarily a fatal degradation). In a partitioned solve, boundary values between partitions evolve across iterations. Consequently, the locally optimal action can change, potentially degrading the composed global policy unless boundary drift is controlled (see Section 4.5). SHARP addresses these challenges because it operates on the concrete state space (without merging states) and refines blocks based on value spread and not on counterexample elimination. We also show that, provided the boundary drift is controlled, the composed global policy remains near-optimal.

Our contributions can be summarised as follows. We propose SHARP, a scalable policy-synthesis method based upon hierarchical adaptive refinement, with block-based selection, boundary-aware sub-MDPs, and bottom-up propagation. Furthermore, we provide formal guarantees on convergence, termination, and error under standard assumptions. Moreover, we implement the approach as part of the PRISM model checker, with configurable refinement strategies. Finally, we evaluate SHARP on a diverse set of MDPs, including models with up to 1M states, and show speedups of up to 2× over PRISM's explicit-state engine on large structured instances.

The rest of the paper is organised as follows. Sections 2–3 provide background and a motivating example. Section 4 presents the SHARP approach with its theoretical guarantees, followed by the implementation (Section 5) and evaluation (Section 6). Sections 7–8 discuss related work and conclude the paper.

2 Preliminaries

2.1 Probabilistic Modelling

Markov decision process (MDP). An MDP is a tuple $\mathcal{M} = (S, \bar{s}, A, \delta, c, AP, L)$, where S is a finite set of states; $\bar{s} \in S$ is an initial state; A is a finite set of actions; $\delta : (S \times A) \to Dist(S)$ is a probabilistic transition function mapping each (s, a) pair to a probability distribution over S; $c : S \times A \to \mathbb{R}_{\geq 0}$ is a (per-action) cost function; AP is a set of atomic propositions; and $L : S \to 2^{AP}$ is a labelling function. Intuitively, states are configurations, actions are controllable choices, and δ gives probabilistic transitions, while costs encode penalties.

MDPs support nondeterministic actions as each state $s \in S$ can have multiple enabled actions for which $\delta(s, a)$ is defined, given by $A(s)$. An MDP *policy* is a possible resolution of nondeterminism. Resolving this nondeterminism allows us to probabilistically reason about MDPs and synthesise *memoryless deterministic policies*, i.e., policies where the action chosen in a state depends solely on the current state. Formally, a (deterministic memoryless) policy of an MDP is a function $\sigma : S \to A$ mapping each state $s \in S$ to an action in $A(s)$.

Stochastic shortest path (SSP) MDP. Let $\mathcal{M}$ be the MDP defined above and $G \subseteq S$ be a set of goal states. We say $\mathcal{M}$ is an SSP instance [8] if: (i) goals are absorbing and carry zero cost, i.e., for all $g \in G$ and $a \in A(g)$ we have $\delta(g, a)(g) = 1$ and $c(g, a) = 0$; (ii) there exists a *proper* policy that reaches G with probability 1 from the states of interest; and (iii) every improper policy (one that reaches G with probability < 1 from some state) has infinite expected total cost from at least one state. We use non-negative per-action costs $c(s, a) \geq 0$, accumulated *until the first visit to* G. From an SSP point of view, the task is to reach the goal while accounting for the cost incurred before the first visit to G. We use PCTL for the probabilistic reachability objectives studied in the paper, and reward operators for the cost objectives, following the standard definitions [10, 33].

2.2 Probabilistic Model Checking

Probabilistic model checking (PMC) [45] computes quantitative reachability/cost objectives on stochastic models (e.g., MDPs), using tools such as PRISM [46] and Storm [36].

Quantitative reachability. Let $\mathcal{M} = (S, \bar{s}, A, \delta, c, AP, L)$ be the MDP defined above and $G \subseteq S$ a set of target states. We denote by $\mathsf{F}\, G$ the event that G is eventually reached. For any state $s \in S$ and policy σ, let $\Pr_s^\sigma(\cdot)$ and $\mathbb{E}_s^\sigma[\cdot]$ denote probability and expectation over infinite paths of the Markov chain induced by σ from s. The maximum reachability probability is $\mathsf{P}_{\max}(s, \mathsf{F}\, G) = \sup_\sigma \Pr_s^\sigma(\mathsf{F}\, G)$. The minimum expected cumulative cost to reach G is $\mathsf{R}_{\min}(s, \mathsf{F}G) = \inf_\sigma \mathbb{E}_s^\sigma\left[\sum_{k<k_G} c(\omega_k, a_k)\right]$, where $a_k = \sigma(\omega_k)$ and $k_G = \min\{j \mid \omega_j \in G\}$ is the first index at which a state in G is reached.

In other words, $\mathsf{P}_{\max}$ asks for the maximum probability of reaching the goal over all policies, while $\mathsf{R}_{\min}$ asks for the minimum expected cumulative cost to do so. In the rest of the paper, we focus on $\mathsf{P}_{\max}$ and $\mathsf{R}_{\min}$, which match our use cases and experiments.

Value Iteration (VI). We use VI [17, 51] as the underlying dynamic-programming procedure to compute both reachability probabilities and expected costs. The method repeatedly applies the Bellman operator to update the current value estimates. For example, after k iterations, $V^{(k)}(s)$ and $W^{(k)}(s)$ denote the current values for state s in the probability and cost cases, respectively. For each state s, these converge to $\mathsf{P}_{\max}(s, \mathsf{F}\, G)$ and $\mathsf{R}_{\min}(s, \mathsf{F}\, G)$, respectively. Gauss-Seidel (GS) follows the same principle, but it uses the newest available value estimates within an iteration, leading to faster convergence in practice [5].

3 Motivating Example: Warehouse Robotics

Consider an autonomous robot operating in a warehouse with multiple shelves (W1–W3 in Figure 1), each possibly blocking certain aisles. The robot should collect items from these shelves and deliver

them to a packing station (Goal). This environment is typically modelled as an MDP, where each state captures the robot's position and which aisles are blocked, while actions enable the robot to move up, down, left, or right. Since the robot does not always move precisely as intended, each action leads to multiple possible next states with known probabilities (e.g., it might fail to move or slip to an adjacent cell). This probabilistic behaviour, combined with large grids of shelves, yields MDPs with millions of states, rendering policy synthesis prohibitively expensive.

This challenging scenario frequently arises in large-scale, realistic warehouses [29, 57]. In these situations, policy synthesis techniques, such as VI, must repeatedly update every state, resulting in severe runtime and memory overhead. For example, open areas around $(0, 0)$ may be relatively safe and uninteresting; thus, they can often be treated coarsely with little loss of accuracy. Likewise, crowded corridors near shelves (W1–W3) can be prone to jams or delays; thus, a fine-grained analysis is required to ensure the robot's strategy remains robust. Despite these semantic differences in the underlying system model, state-of-the-art PMC tools use VI and its variants [46] and treat these regions the same, potentially wasting resources on trivial warehouse sections.

Fig. 1. A warehouse layout with shelves W1–W3.

The key idea is not to treat all parts of the warehouse in the same way. SHARP refines uncertain or congested regions near W1–W3, while keeping simpler regions at a coarser level.

4 SHARP

SHARP is a hierarchical partition-based approach for MDP policy synthesis that adaptively refines MDP partitions only where needed. Its high-level workflow is shown in Figure 2 and comprises five stages (S1–S5). SHARP takes as inputs an MDP model, a reachability objective *either* $\mathsf{P}_{\max}$ (maximise probability of reaching G) *or* $\mathsf{R}_{\min}$ (minimise expected cumulative cost to G), a partition strategy Π, and four parameters: D (maximum depth), θ (refinement threshold), ε (solver tolerance), and η_{thr} (boundary-change threshold). The first two (D, θ) control the hierarchical decomposition, while the latter two (ε, η_{thr}) control the numerical accuracy and determine when changes in boundary values are significant enough to trigger re-solving, respectively.

The first SHARP stage (S1) involves executing an initial partitioning of the MDP into a set of *blocks*, producing one sub-MDP per block. SHARP uses this partitioning to construct the hierarchy tree (S2), whose root node corresponds to the entire set of states, and the children correspond to the individual blocks. Each partition, at this stage, is considered a leaf node in the tree and is solved *locally* using VI (S3). This local solution uses fixed boundary conditions, derived from the global value vector V, to properly account for any transitions leaving the partition. This enables obtaining refined value estimates and a partial policy for each partition.

Once these refined estimates and partial policies are available, SHARP recursively propagates them up to their parent nodes in the tree (S4). Then, the iterative process is triggered (S3–S4) until values stabilise globally, re-solving leaves only when their boundary values change significantly.

Then, SHARP checks whether further refinement is needed (S5). If this holds (S5.1), it refines the sub-MDP by subdividing selected partitions, extends the partition tree, and repeats the iterative process. Finally, if no further refinement is required and the global solution has converged, SHARP extracts the synthesised policy σ (S5.2).

During the execution of Algorithm 1, each invocation of SolveLocally produces both a value function and a local policy (the action achieving the Bellman optimum at each state). We then compose these local policies into a single global policy σ over all states (Section 4.4).

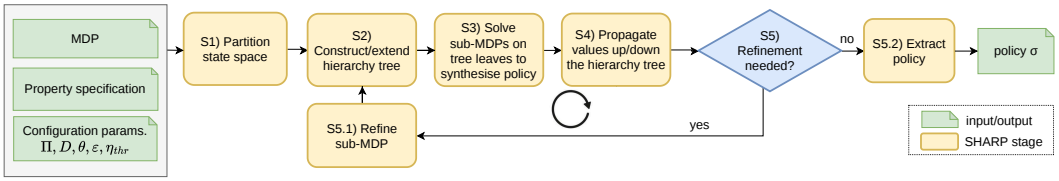

Fig. 2. SHARP overview showing its various stages for hierarchical partitioning and adaptive refinement.

4.1 Partitioning and Tree Hierarchy Construction

Foundational approaches to solving large MDPs include partitioning the state space to construct abstractions or smaller subproblems that can be solved more efficiently [23]. While in our work we do not perform state aggregation, we leverage the principle of partitioning to structure the policy synthesis process hierarchically. SHARP is built upon two main ideas. First, the state space is partitioned into *blocks*. Second, these blocks are organised hierarchically by recursively subdividing them, resulting in a nested structure represented by a *hierarchy tree*.

Definition 1 (Partitioning). *Given an MDP $\mathcal{M}$, a* partitioning *of its state space S is a collection $\mathcal{B} = \{B_1, B_2, \ldots, B_k\}$ of subsets of S, called* **blocks**, *such that: (i) Each block B_i is non-empty ($B_i \neq \emptyset$ for all $i = 1, \ldots, k$); (ii) The blocks are pairwise disjoint ($B_i \cap B_j = \emptyset$ for all $i \neq j$); and (iii) the union of all blocks covers the entire state space ($\bigcup_{i=1}^{k} B_i = S$).*

Remark 1 (Symmetry). *The construction for probabilities extends verbatim to minimisation by replacing* max *with* min *in the Bellman operator. Our experiments focus on* $\mathsf{P}_{\max}$ *and* $\mathsf{R}_{\min}$.

Definition 2 (Hierarchy tree). *A rooted tree $\mathcal{T} = (\mathcal{N}, E)$ is a* hierarchy tree *for $\mathcal{M}$ if (i) the root r corresponds to the entire state space S; (ii) every node $v \in \mathcal{N}$ corresponds to a block $B_v \subseteq S$; and (iii) the blocks corresponding to the children of v form a partitioning of B_v. We denote the distance of v from the root with v.*depth.

Each node $v \in \mathcal{N}$ stores pointers to its parent and children, and also maintains its current local values (v.values) and synthesised local policy ($v.\sigma$). To detect when re-solving is needed, leaf nodes additionally track the following metadata: (i) a snapshot of the boundary values from the previous solve (v.prevBoundary); (ii) a flag indicating if the node has been solved at least once (v.everSolved); and (iii) the residual error from the last local VI run (v.localResidual).

Partition Strategy Π. We parameterise SHARP with a partition strategy Π that provides: (i) Π.InitialPartition(S); and (ii) Π.Refine(B, d) at depth d. Typical instantiations include uniform grid partitioning for MDPs with spatial state spaces and graph-based partitioning (e.g., counter-based, Strongly Connected Component (SCC) layering) for general MDPs.

Algorithm 1 summarises the SHARP workflow introduced above. It takes the inputs described in Section 4 and returns a value vector V for the MDP states S and a memoryless policy σ. We first initialise the global value vector to 0 for all non-goal states. The values of the goal states are then fixed according to the type of the objective. Specifically, we set $V(s)$=1 for probability objectives and $V(s)$=0 for reward objectives. These assignments provide the boundary conditions that are used when the initial blocks are solved locally.

Example 1. Consider again the warehouse robotics model from Section 3. Our goal is to synthesise an optimal policy for the reachability property $\mathsf{P}_{\max}[\,\mathsf{F}\,\text{“goal”}\,]$.

Hierarchy initialisation: We begin with a single root node that contains all states of the MDP, that is the 36 grid positions of the 6×6 grid. Next, we apply a coarse geometric partition to the grid: a

Algorithm 1 SHARP: Scalable Hierarchical Adaptive Refinement for Policy Synthesis

procedure SHARP($\mathcal{M}, \phi, G, \Pi, D, \theta, \varepsilon, \eta_{\text{thr}}$)
  Init: Create root on S. Initialise V to 0 on all states and fix goals: $V(s)$=1 for $s \in G$ under P, and $V(s)$=0 for $s \in G$ under R.
  Init pass: $\mathcal{P}_0 \leftarrow \Pi$.InitialPartition($S$); create children for all $B \in \mathcal{P}_0$
  Solve/propagate: SolveLocally(*child*,ϕ, ε) for all children; PropagateValues(root)
  repeat
    $V^{\text{old}} \leftarrow V$ ▷ Store for convergence check
    for all leaves v with BoundaryChanged(v, η_{thr}) **do**
      SolveLocally(v, ϕ, ε)
    end for
    PropagateValues(root)
    $L \leftarrow \{\ell$ leaf $\mid \text{depth}(\ell) < D \wedge$ ShouldRefine$(\ell, \phi, \theta)\}$
    for all $\ell \in L$ **do**
      RefinePartition($\ell, \Pi, \phi, \varepsilon$)
    end for
    PropagateValues(root)
  until no leaf satisfies BoundaryChanged$(\cdot, \eta_{\text{thr}})$, $L = \emptyset$, and $\|V - V^{\text{old}}\|_\infty \leq \varepsilon$
  return (V, σ)
end procedure
procedure SolveLocally(v, ϕ, ε) ▷ Section 4.3
  Build the induced sub-MDP on B_v: add one-step outside successors as boundary states and fix their values from the global V
  Fix internal goals (V=1 for P, V=0 for R); preserve per-action costs $c(s, a)$
  Run VI (min/max per ϕ) to tolerance ε; store (V_v, σ_v); snapshot boundaries: b_v^{prev}
end procedure
procedure RefinePartition($v, \Pi, \phi, \varepsilon$) ▷ Section 4.2
  Split B_v via Π; create children; SolveLocally(*child*,ϕ, ε) each
end procedure
procedure PropagateValues(v) ▷ Section 4.4
  Post-order: recurse on all children of v (if any), then copy each state's value and policy from its unique owning child
end procedure
function BoundaryChanged(v, η_{thr})
  return if normalised boundary change exceeds η_{thr} ▷ Section 4.3
end function
function ShouldRefine(ℓ, ϕ, θ)
  $\theta_d \leftarrow$ AdaptThreshold$(\theta, \text{depth}(\ell), \phi)$
  return $\tau(\ell) > \theta_d$ ▷ Section 4.2
end function

uniform 3×3 tiling yields nine non-overlapping blocks (Figure 3). Each block is non-empty, and together they cover the entire warehouse, satisfying the conditions of Definition 1; each becomes a child of the root node, forming a hierarchy tree (Definition 2) with a proper partition of the state space.

After the initial partitioning, SHARP solves each of the nine blocks locally using SolveLocally with boundary values from V, then propagates values up to the root. The algorithm then enters its main loop, where it may refine blocks whose value spread exceeds θ. For instance, block 9 containing the goal area might be refined further if its value spread exceeds θ.

4.2 Adaptive Partition Refinement

In order to decide whether to refine a leaf node or not in the hierarchy tree, we create a refinement criterion that compares the value spread $\Delta(v)$ within each block against a threshold.

Refinement criterion. For a node v with block B_v, we define the value spread as $\Delta(v) = \max_{s \in B_v} V_v(s) - \min_{s \in B_v} V_v(s)$. If this spread is small, then the block is already fairly homogeneous and there is little reason to split it further. If, on the other hand, the spread is large, then the block is mixing states with rather different values, which can affect the resulting policy. For $\mathsf{P}_{\max}$ we simply use $\tau(v) = \Delta(v)$. For $\mathsf{R}_{\min}$, we use either the same absolute spread in the grid-based warehouse experiments or the normalised spread $\Delta(v) \,/\, (|\text{avg}(V_v)| + \beta)$ in the non-grid experiments. A leaf v at depth $d < D$ is refined once $\tau(v)$ exceeds the depth-adapted threshold θ_d.[1] This is the test

[1] AdaptThreshold computes θ_d internally. Concrete settings for each instantiation appear in Section 6.

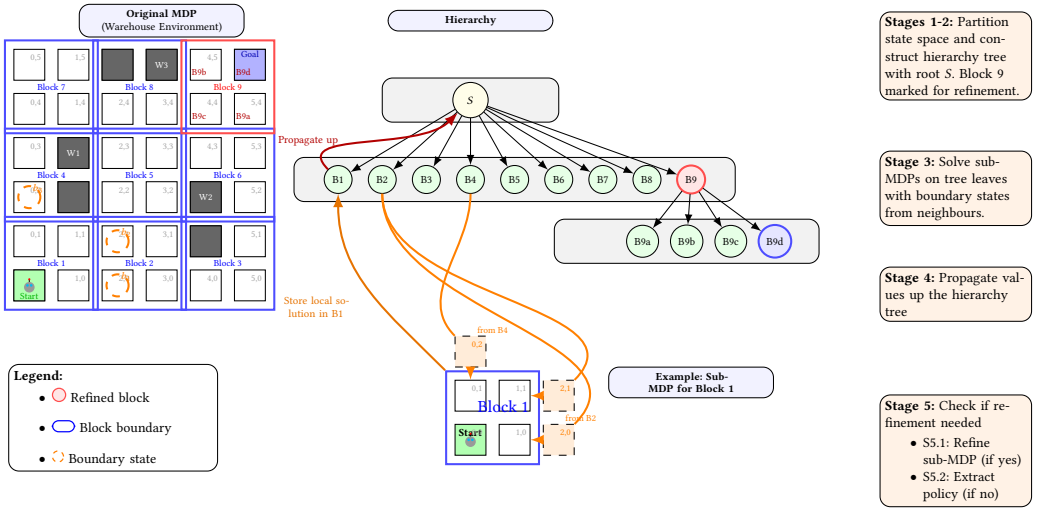

Fig. 3. **Overview of Sub-MDP Creation and Hierarchical Refinement in SHARP.** A 6×6 warehouse with shelves (W1–W3) is partitioned into nine blocks. *Block 9* contains the goal and is refined further. *Block 1* contains the start state, and its dashed boundary states connect to adjacent *Block 2* and *Block 4*. Solved values then propagate back to the parent node.

implemented by ShouldRefine(v, ϕ, θ) in Algorithm 1. When the test succeeds, RefinePartition splits the block into disjoint child blocks, solves them using boundary values from V, and propagates the updated values upward.

Example 2 (continued). Continuing the warehouse scenario of Example 1, suppose that the nine top-level partitions in Figure 3 have already been solved, and that partition 9, which contains the goal cell $(5, 5)$, shows a large value spread Δ. This is not surprising, since states close to the goal have values close to 1.0, whereas states one step farther away are noticeably smaller, and so $\Delta > \theta$. The block is therefore refined. This refinement of partition 9 separates its four cells into the child blocks `B9a`–`B9d`, isolating the goal cell and nearby high-value states from the lower-value states farther away.

4.3 Local Sub-MDP Policy Synthesis

Once a node v is partitioned, either as part of the initial partitioning or due to further refinement, SHARP views each partition as a sub-MDP that includes both the partition's states and boundary states. The next definitions formalise this local-solving step. In particular, we generate the induced sub-MDP of a block B by adding its one-step outside successors as boundary states (Def. 3), making them absorbing, and assigning them fixed values from the current global vector V (Def. 4); local VI then solves the standard Bellman equations inside this closed sub-MDP.

Constructing the sub-MDP. SHARP builds the induced sub-MDP $\mathcal{M}_B$ for block $B \subseteq S$ as specified in Definition 3 below. Boundary states $\mathrm{bd}(B)$ are made absorbing, and their values are fixed from the current global vector V as specified in Definition 4.

Definition 3 (Induced sub-MDP). *Let $\mathcal{M} = (S, \bar{s}, A, \delta, c, AP, L)$ be an MDP and $B \subseteq S$, $B \neq \emptyset$. The boundary is $\mathrm{bd}(B) = \{t \in S \setminus B \mid \exists s \in B, a \in A(s) : \delta(s, a)(t) > 0\}$. The* sub-MDP induced by *B is $\mathcal{M}_B = (B \cup \mathrm{bd}(B), \bar{s}_B, A_B, \delta_B, c_B, AP, L_B)$ with $A_B = A$, $c_B = c$, $L_B = L$, and for $s \in B \cup \mathrm{bd}(B)$,*

$a \in A(s)$:

$$\bar{s}_B = \begin{cases} \bar{s} & \text{if } \bar{s} \in B \\ \text{any in } B & \text{otherwise} \end{cases}, \quad \delta_B(s,a)(t) = \begin{cases} \delta(s,a)(t) & s \in B, t \in B \cup \mathrm{bd}(B) \\ 1 & s \in \mathrm{bd}(B), t = s \\ 0 & \text{otherwise} \end{cases}$$

Accordingly, each state in $\mathrm{bd}(B)$ becomes absorbing in the sub-MDP $\mathcal{M}_B$.

The following definition describes how the local solve fixes its value using $V(t)$ as a fixed boundary condition, i.e., whenever a transition from B reaches a boundary state t, we use the current global estimate $V(t)$.

Definition 4 (Boundary-value assignment). *Let $B \subseteq S$ be a block represented by a node v in the hierarchy tree $\mathcal{T}$, and let $\mathcal{M}_B$ be the sub-MDP induced by B with boundary states* $\mathrm{bd}(B)$*. For every boundary state $t \in \mathrm{bd}(B)$, its* boundary value *is set to $V_B(t) = V(t)$, where V is the global value vector maintained at the root of $\mathcal{T}$. These values are held fixed throughout the local VI step on $\mathcal{M}_B$, thereby incorporating the influence of the surrounding MDP. That is, the Bellman equations are solved only for internal non-goal states in B; boundary values are held fixed and excluded from Bellman updates.*

Boundary Change Detection. During the iterative refinement process, SHARP must detect when a node's boundary values have changed significantly enough to trigger the re-solving process of its local sub-MDP. Let $\mathrm{bd}(B_v)$ denote the boundary states of block B_v, and let $b_v^{\text{prev}} := V^{(\text{last solve})}|_{\mathrm{bd}(B_v)}$ be the boundary snapshot stored after v's previous local solve. We define the BoundaryChanged function as:

$$\textsc{BoundaryChanged}(v, \eta_{\text{thr}}) := \frac{\left\| V|_{\mathrm{bd}(B_v)} - b_v^{\text{prev}} \right\|_\infty}{\max\left\{1, \left\| b_v^{\text{prev}} \right\|_\infty\right\}} > \eta_{\text{thr}}.$$

When this predicate evaluates to false (i.e., the boundary change is not significant enough to trigger re-solving), it implies the absolute mismatch bound $\left\| V|_{\mathrm{bd}(B_v)} - b_v^{\text{prev}} \right\|_\infty \le \eta_{\text{thr}} \cdot \max\{1, \|b_v^{\text{prev}}\|_\infty\}$. We use this absolute quantity in the residual bounds. Newly created leaves are solved immediately upon creation (RefinePartition), establishing b_v^{prev} before the predicate is ever evaluated.

Local VI. The sub-MDP is solved using VI until the local Bellman residual $\|T_B V - V\|_\infty < \varepsilon$, where T_B is the local Bellman operator on $\mathcal{M}_B$ (equivalently, for synchronous VI, $\|V^{(k+1)} - V^{(k)}\|_\infty < \varepsilon$). The action that attains the optimum value is stored as the local policy.

Definition 5 (Local value function). *Given a reachability property $\mathsf{P}_{\max}[\mathsf{F}\, G]$ or $\mathsf{R}_{\min}[\mathsf{F}\, G]$ with global goal set G, fix a block B and its induced sub-MDP $\mathcal{M}_B$ (Def. 3). Let $G_B = G \cap B$ and use* $\mathrm{bd}(B)$ *for the boundary states.*

The local value $V_B : B \cup \mathrm{bd}(B) \to \mathbb{R}$ *is the unique solution of:*

Probability objective *($\mathsf{P}_{\max}$)*

$$V_B(s) = \begin{cases} 1, & s \in G_B, \\ \max\limits_{a \in A_B(s)} \sum\limits_{t \in B \cup \mathrm{bd}(B)} \delta_B(s,a)(t)\, V_B(t), & s \in B \setminus G_B, \\ V(s), & s \in \mathrm{bd}(B). \end{cases}$$

Cost objective ($\mathsf{R}_{\min}$)

$$V_B(s) = \begin{cases} 0, & s \in G_B, \\ \min\limits_{a \in A_B(s)} \Big(c(s,a) + \sum\limits_{t \in B \cup \mathrm{bd}(B)} \delta_B(s,a)(t)\, V_B(t) \Big), & s \in B \setminus G_B, \\ V(s), & s \in \mathrm{bd}(B). \end{cases}$$

Storing and propagating. After local convergence, SHARP stores both the value vector $\{V_B(s) \mid s \in B \cup \mathrm{bd}(B)\}$ and the corresponding partial policy σ in the node associated with block B. If the node is a leaf, the sub-MDP solving is complete. Otherwise, once all child nodes are solved, we propagate their values upward to update this parent node's estimates as detailed in Section 4.4.

Example 3 (continued). Suppose we refine B9 into four single-state child blocks: `B9a`, `B9b`, `B9c`, and `B9d`, containing states $(5,4)$, $(4,5)$, $(4,4)$, and the goal $(5,5)$, respectively. The goal child `B9d` has its value fixed at 1 and, for this reason, requires no solving. In practice, the algorithm would likely stop refinement before reaching single-state blocks, but we use those here in order to show how our approach works. Following Definition 3, to construct the sub-MDP for `B9a` (containing cell $(5,4)$), we include: (i) the state $(5,4)$ with all its original actions and transitions, and (ii) any states directly reachable from $(5,4)$ and lying outside `B9a` as boundary states bd(`B9a`), including the goal $(5,5)$ and adjacent cells. These boundary states are absorbing in the sub-MDP (self-loops with probability 1) and have been assigned fixed values from V (Definition 4). For such a small sub-MDP, local VI converges quickly. Since $(5,4)$ is adjacent to the goal (whose value is 1.0), the converged value for $(5,4)$ will be high. Moreover, the local solve also produces a policy: $\sigma_{\texttt{B9a}}(5,4) = \mathsf{up}$, since moving up toward the goal $(5,5)$ maximises the reachability probability. After solving the three non-goal child sub-MDPs, we propagate their values and policies back to update block B9's estimates.

4.4 Value Propagation Up the Tree Hierarchy

Having described how local sub-MDPs are solved (Section 4.3), we now describe how their solutions are combined to produce a globally consistent policy.

Global Policy Composition. Since leaf nodes partition S, each state s belongs to exactly one leaf v, which stores a local value function V_v and a deterministic memoryless policy $\sigma_v : B_v \to A$ attaining the Bellman optimum (SolveLocally). For each $s \in S$, let $\mathrm{leaf}(s)$ denote the unique leaf with $s \in B_{\mathrm{leaf}(s)}$. We can therefore define the global policy by simply copying from the leaves: $\sigma(s) := \sigma_{\mathrm{leaf}(s)}(s)$ and $V(s) := V_{\mathrm{leaf}(s)}(s)$, which essentially is the action/value that the owning leaf deemed optimal. This composition is well-defined because the partition is disjoint and no state appears in more than one leaf. PropagateValues implements this copying as a bottom-up traversal of $\mathcal{T}$, i.e., for each node v, it recurses into children first, then pulls their values/policies into the parent. Since children partition B_v, SHARP (Algorithm 1) by construction ensures that no two children claim the same state, meaning that the root ends up with a globally consistent policy σ. We formally show in Section 4.5 that the policy thus composed is near-optimal when boundary mismatch is kept small.

Example 4 (continued). Following the PropagateValues procedure of Algorithm 1, with the local solutions for `B9a`, `B9b`, and `B9c` now computed, we merge them back into block B9 by assigning each sub-block's value (and policy) to the corresponding cell. For instance, if `B9a` determined that $(5,4)$ has a probability ≈ 1.0 to reach $(5,5)$, we set $V_9(5,4) \approx 1.0$ and copy the policy $\sigma_9(5,4) = \mathsf{up}$ from $\sigma_{\texttt{B9a}}$. Similarly, `B9b` and `B9c` provide values and policies for their respective cells. At this point, block B9's value estimates are updated based on its children's refined solutions. Although in this

example, we reached single-state blocks for illustration, SHARP, in general, checks in subsequent iterations if further refinement is needed based on the updated value spread.

4.5 Theoretical Guarantees

We show that, under the assumptions stated below, SHARP returns a near-optimal final value vector when each leaf sub-MDP is solved to tolerance ε and the boundary mismatch at termination remains small. Let η denote the maximum boundary mismatch at termination, i.e., the largest absolute difference between the boundary values used in a leaf's last local solve and the final global values (defined formally below). The *global Bellman residual* then satisfies $\|TV^{\text{fin}} - V^{\text{fin}}\|_\infty \leq \varepsilon + \eta$. Furthermore, under the assumptions stated below, this residual bound allows us to obtain a formal guarantee on the actual error by standard fixed-point arguments [9, 24]. We analyse the two objectives used in our evaluation: minimum expected cost ($\mathsf{R}_{\min}$) and maximum reachability probability ($\mathsf{P}_{\max}$). Our analysis focuses on the main arguments; complete proofs are provided in the supplementary material.

We first state the assumptions used below, which follow the standard SSP and reachability assumptions for MDPs [5]:

(1) $\mathsf{R}_{\min}$**:** The MDP is an SSP instance with the standard conditions, as defined in Section 2: (i) a proper policy exists; (ii) costs are non-negative with zero cost at goals; (iii) goals are absorbing; and (iv) every improper policy has infinite expected cost from some state [8, 9].

(2) $\mathsf{P}_{\max}$**:** *Uniform absorption:* there exists an $\alpha > 0$ such that from every non-absorbing state, under any action, the probability to enter the absorbing set (goal or sink) in one step is at least α [51].

In our case, all $\mathsf{R}_{\min}$ instances satisfy the SSP assumptions. For the $\mathsf{P}_{\max}$ warehouse benchmarks, the relevant assumption is the uniform-absorption condition used in part (b), rather than SSP. More precisely, from every non-absorbing state, every enabled action reaches the absorbing set, that is, either the goal or the sink, in one step with probability at least $\alpha > 0$. In our models, α is given by `P_FAIL`. Furthermore, both the goal and sink states are absorbing, with values fixed to $V(s) = 1$ for goal states and $V(s) = 0$ for sink states.

Boundary changes and re-solving strategy. Let T be the Bellman optimality operator with fixed point V^*. During the execution of the algorithm, each leaf sub-MDP is solved to a specified tolerance ε using the SolveLocally procedure, with fixed boundary values taken from the current global vector at the time of solving. Moreover, note that blocks are not re-solved immediately when boundary values change; instead, we take a snapshot of these boundary values at each solve (as specified in SolveLocally) and only trigger re-solving when BoundaryChanged detects that the change exceeds a threshold at runtime. This yields an *asynchronous fixed-point* schedule [7]; our contribution is the hierarchical partitioning and boundary-change trigger that decides where and when to re-solve. However, this also means that changes below this threshold will remain unresolved, and they continue to accumulate until the algorithm's termination. This maximum accumulated discrepancy is captured by η, measured as $\eta = \max_{\text{leaf } B} \max_{t \in \text{bd}(B)} \left| V^{\text{fin}}(t) - V_B^{\text{used}}(t) \right|$, where V^{fin} is the final global value vector when SHARP terminates and $V_B^{\text{used}}(t)$ denotes the boundary value that block B used at its most recent solve.

On the re-solve trigger. In the implementation, we do not check η directly. Instead, we use the relative test BoundaryChanged$(\cdot, \eta_{\text{thr}})$ defined above. When SHARP terminates, this test is false for every leaf. This gives the absolute mismatch bound $\left\| V|_{\text{bd}(B)} - b^{\text{prev}} \right\|_\infty \leq \eta_{\text{thr}} \cdot \max\{1, \|b^{\text{prev}}\|_\infty\}$, and therefore η is bounded in terms of η_{thr} (scaled by the chosen normalisation), allowing the error bounds in Theorem 4.1 to be stated directly in terms of the input threshold. The next result shows how the residual bound translates into a bound on the actual error.

Theorem 4.1 (Convergence and Error). *When SHARP terminates, its final value vector V^{fin} is a close approximation of the true optimal solution V^*. First, the final solution is nearly stable, in the sense that the global Bellman residual is bounded by the local solver tolerance and the measured boundary mismatch: $\|TV^{\text{fin}} - V^{\text{fin}}\|_\infty \le \varepsilon + \eta$. This residual bound, in turn, yields bounds on the true error $\|V^{\text{fin}} - V^*\|_\infty$:*

(a) ***Minimum expected cost ($\mathsf{R}_{\min}$).*** *For SSP problems, $\|V^{\text{fin}} - V^*\|_\infty \le C\,(\varepsilon + \eta)$, where C depends on the expected number of steps to reach the goal under suitable proper policies. In other words, the error scales with $\varepsilon + \eta$, up to the constant C.*

(b) ***Maximum reachability probability ($\mathsf{P}_{\max}$).*** *For $\mathsf{P}_{\max}$, we additionally assume that from any non-absorbing state, every action available in that state reaches either the goal or the sink in one step with probability at least $\alpha > 0$. Under this assumption, $\|V^{\text{fin}} - V^*\|_\infty \le (\varepsilon + \eta)/\alpha$.*

Intuitively, if each leaf is solved accurately (within ε) and boundary changes that remain at stopping are small (captured by η), then the assembled solution is near-optimal. The theorem above concerns the final value vector. We next relate this guarantee to the global policy obtained by composing the local leaf policies.

Policy Quality Guarantee. Each leaf policy σ_v is optimal with respect to the local value V_v. Boundary values may drift by up to η between the local solve and termination of the algorithm. Since transition probabilities sum to one, a one-step Bellman backup under the composed policy σ can therefore change by at most η when the boundary values change. Combining this with the local solver tolerance ε, the Bellman residual under σ is bounded by $\varepsilon+\eta$. Based upon standard contraction mapping arguments [9, 24] and the global error bound $(\varepsilon+\eta)/\alpha$ from Theorem 4.1, we obtain $\|V^\sigma - V^*\|_\infty \le (2\varepsilon + 2\eta)/\alpha$ for $\mathsf{P}_{\max}$ under uniform absorption. For $\mathsf{R}_{\min}$, an analogous bound can be obtained when the composed policy σ is proper.

SHARP terminates in finitely many iterations, as we now state formally.

Theorem 4.2 (Termination). *Given bounded refinement depth D and fixed thresholds $\varepsilon, \eta_{\text{thr}} > 0$, if in each outer pass every leaf satisfying BoundaryChanged is re-solved via SolveLocally, then SHARP terminates in finite time.*

5 Implementation

SHARP is implemented in Java on top of the probabilistic model checker PRISM [46]. The core of our implementation lies in newly developed algorithms integrated within PRISM's explicit engine, including hierarchical partitioning, adaptive refinement based on value spread, sub-MDP construction with boundary states, and bottom-up value propagation. SHARP can be configured by selecting a partition strategy Π (grids: (N_x, N_y); general MDPs: SCC-based or counter-based), as well as the maximum refinement depth D, refinement threshold θ, solver tolerance ε, and boundary-change threshold η_{thr}. The SHARP implementation, case studies, and experimental results are available at https://github.com/alexEvangelidis/sharp.

6 Evaluation

6.1 Research Questions

We use the research questions below to assess SHARP and extract decision-making insights, helping practitioners effectively embed SHARP in their verification workflows.

RQ1 (Efficiency): How efficient is SHARP compared to PRISM in terms of execution time and memory overhead across various problem instances?

RQ2 (Scalability): How does SHARP scale with problem instances of increasing sizes in terms of states and transitions?

Table 1. MDP warehouse benchmarks. NW: no walls, SW: single wall, MW: multiple walls.

Grid	#States	Walls	#Trans (P_{max})	#Trans (R_{min})	ID (P_{max})	ID (R_{min})
512^2	262K	NW	3.14M	2.10M	W512NW-a	W512NW-b
		SW	3.14M	2.09M	W512SW-a	W512SW-b
		MW	3.14M	2.09M	W512MW-a	W512MW-b
1024^2	1.05M	NW	12.58M	8.38M	W1024NW-a	W1024NW-b
		SW	12.57M	8.38M	W1024SW-a	W1024SW-b
		MW	12.56M	8.37M	W1024MW-a	W1024MW-b

RQ3 (Generalisation): How well does SHARP generalise to different MDP structures beyond spatial models, and what structural properties determine its effectiveness?
RQ4 (Hyperparameter sensitivity): How is SHARP's performance affected by adjusting hyper-parameters: initial partition size, maximum depth, and refinement thresholds?

6.2 Experimental Setup

Case studies. For the evaluation, we use two benchmark groups. The first is a set of warehouse MDPs (Table 1), and the second is a set of non-grid benchmarks drawn from RL [42, 48, 53] and verification [6, 12, 54]. For the warehouse models, we consider 512×512 and 1024×1024 grids with three obstacle layouts, namely NW, SW, and MW. For each layout, we study two variants. In the Pmax case, the objective is to maximise the probability of reaching the goal in the presence of a failure sink, with `P_FAIL`$=5 \times 10^{-4}$ and `P_SUCC` $\in \{0.80, 0.90\}$. In the Rmin case, the objective is to minimise the expected number of steps, assuming 0.8 probability of moving as intended, 0.2 probability of a self-loop, and unit step cost. Additionally, we tested 5 random 1024×1024 MW MDPs with varied wall configurations.
Non-grid benchmarks. To assess SHARP's generalisation beyond spatial models, we evaluated two different MDP families. The first type of model is a *sequential arena* with several configurations of ($K \in \{8, 16\}$ arenas, $\sim 5{\times}10^4$ to $\sim 8{\times}10^5$ states). These represent multi-phase SSP problems in which an agent must complete K arenas in sequence. Moreover, each arena evolves independently (selected by a global counter), with local progress variables. The interaction between the various arenas is limited and happens only at transition points. Our goal is to minimise expected steps to completion: $R_{min}[\,\mathsf{F}\ \texttt{finished}\,]$.

In addition, we evaluate the well-known *protocol* models such as the IEEE 802.11 wireless LAN (`wlan3`, $\sim$97K states) and the IEEE 1394 FireWire root contention protocol (`firewire`, $\sim$4K states). The first model involves three stations competing for medium access via exponential backoff with shared channel variables. The `firewire` model represents the tree identification phase where two nodes compete to become root through randomised leader election. For these models, we check $P_{max}[\,\mathit{true}\ \mathsf{U}\ (\texttt{bc1=3} \vee \texttt{bc2=3})\,]$ and minimise expected time to leader election for `wlan3` and `firewire`, respectively. Additionally, we evaluate the *consensus* shared coin protocol ($\sim$22K states, P_{max}) and the *zeroconf* network configuration protocol ($\sim$1K states, R_{min}). All four protocol models are from the PRISM benchmark suite [46].

Baselines and configurations. We compare SHARP against PRISM's explicit engine using VI and GS with/without pre-computations [46]. As SHARP extends PRISM in Java, comparing against PRISM's engine isolates algorithmic improvements from implementation differences (unlike comparing to the C++-based Storm [36] probabilistic model checker).

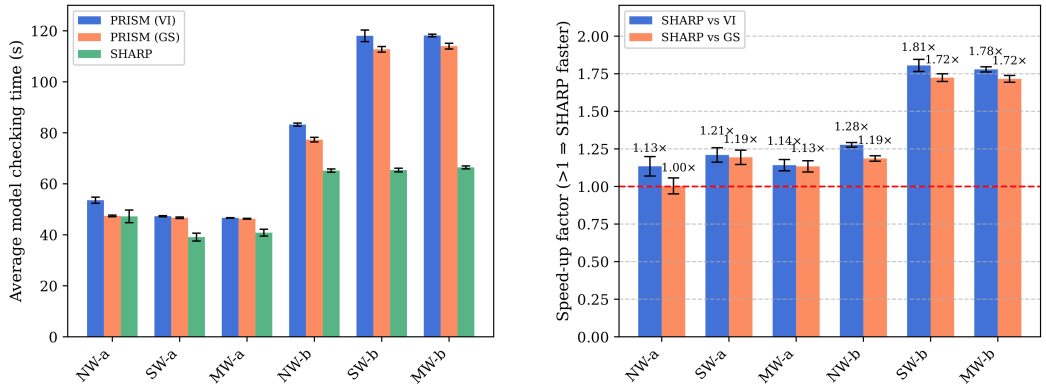

Fig. 4. SHARP vs. PRISM on 1024×1024 warehouse instances: model-checking time (left) and speedup (right).

We explored SHARP configurations with: maximum depth $D \in \{1, 2, 3, 4, 5\}$; initial partition sizes from 8×8 to 25×25; cost thresholds $\theta \in \{200, 300, 400, 500, 600, 800\}$; and probability thresholds $\theta \in \{0.2, 0.3, 0.5, 0.7, 1.0\}$. Unless stated otherwise, all sub-MDPs were solved using VI (convergence threshold $\varepsilon = 10^{-6}$). For depth-adapted refinement thresholds, we used $\theta_d = \theta$ for probability objectives and $\theta_d = \max(0.01, \theta/(10(1+d)))$ for cost objectives. The boundary-change threshold was set to $\eta_{\text{thr}} = 0.001$ for probability objectives and $\eta_{\text{thr}} = 10^{-6}$ for cost objectives. High uncertainty blocks (for probability objectives) were identified as those with average values in $(0.1, 0.9)$. The numerical stability parameter in the refinement score formula was set to $\beta = 10^{-6}$.

Objective-specific refinement score. In our experiments, we used different refinement scores depending on the verification objective. For the warehouse (grid) experiments, we used the absolute score $\tau = \Delta$, together with $\theta \in \{200, 300, 400, 500, 600, 800\}$, measured in expected steps. For the non-grid benchmarks, we used the normalised score $\tau = \Delta/(|\text{avg}(V)| + \beta)$, together with small threshold values (e.g., 0.05). In the results, we report, for each instance, the configuration that achieved the best execution time, and, where relevant, we indicate the corresponding value of θ in the captions.

Experimental environment. We conducted the experiments on a laptop with 64 GB of RAM and a 5 GHz Intel 13th Gen i7-13700H processor running Ubuntu 22.04.5 LTS (64-bit). To develop and evaluate SHARP, we used OpenJDK 23.0.2 and PRISM 4.8.1 [46]. In all experiments, we set a timeout of 30 min and a memory limit of 32 GB. Also, to mitigate any potential effect of randomisation, we run all experiments 30 times and report mean±std where applicable.

6.3 Results & Discussion

RQ1 (Efficiency). We assess SHARP's efficiency against PRISM [46] by measuring the model checking time and memory overhead across the case studies from Table 1. In Table 2, we compare SHARP with PRISM's explicit-engine methods, VI and Gauss-Seidel (GS), both with and without precomputation on the two properties described earlier.

We observe that across the smaller model instances (e.g., W512NW-a/b), PRISM yields a policy in approximately 6 s to 15 s, using 1.6 GB to 1.9 GB of memory. On the other hand, SHARP yields the policy in about 8 s to 13 s and uses 6.4 GB to 8.5 GB. For the W512SW-a/b and W512MW-a/b instances, both PRISM and SHARP finish in less than 20 s, although PRISM is sometimes faster when no precomputation is used. However, in larger problem instances with over 1 million states, such as the W1024* ones, PRISM-VI typically takes 46 s to 118 s and uses at least 5 GB of

Table 2. Comparison of SHARP against PRISM using Value Iteration (VI) and Gauss-Seidel (GS) for policy synthesis. *Note:* θ is the configured refinement threshold (Pmax: probability difference; Rmin: absolute spread in expected steps).

	PRISM						SHARP					
	VI			GS								
ID	T (s) (std)	M (MBs)	R	T (s) (std)	M (MBs)	R	T (s) (std)	M (MBs)	R	D	θ	Blocks X/Y
Property: Pmax = [F "goal"]												
W512NW-a (nopre)	6.27 (±0.27)	1901 (±29)	0.567	6.14 (±0.13)	1899 (±40)	0.567	7.99 (±0.37)	8371 (±207)	0.567	1	0.5	8/8
W512NW-a (pre)	15.69 (±0.32)	1907 (±27)	0.567	15.54 (±0.24)	1905 (±18)	0.567						
W512SW-a (nopre)	5.96 (±0.04)	1889 (±19)	0.567	5.94 (±0.05)	1889 (±14)	0.567	8.29 (±0.43)	8369 (±438)	0.567	1	0.5	8/8
W512SW-a (pre)	15.72 (±0.13)	1912 (±24)	0.567	15.27 (±0.11)	1900 (±15)	0.567						
W512MW-a (nopre)	6.01 (±0.05)	1928 (±26)	0.567	6.05 (±0.04)	1926 (±55)	0.567	7.61 (±0.31)	8506 (±264)	0.567	1	0.5	8/8
W512MW-a (pre)	15.75 (±0.19)	1935 (±57)	0.567	15.62 (±0.22)	1932 (±53)	0.567						
W1024NW-a (nopre)	53.57 (±1.20)	6213 (±98)	0.278	47.41 (±0.31)	6204 (±156)	0.278	47.23 (±2.48)	19769 (±1660)	0.278	1	0.5	8/8
W1024NW-a (pre)	129.71 (±1.92)	6209 (±95)	0.278	123.41 (±3.21)	6228 (±52)	0.278						
W1024SW-a (nopre)	47.30 (±0.22)	6268 (±128)	0.321	46.68 (±0.30)	6242 (±93)	0.321	39.10 (±1.54)	20029 (±1598)	0.321	1	1.0	8/8
W1024SW-a (pre)	125.55 (±0.73)	6262 (±87)	0.321	122.02 (±1.04)	6300 (±56)	0.321						
W1024MW-a (nopre)	46.63 (±0.10)	6612 (±58)	0.321	46.30 (±0.16)	6597 (±96)	0.321	40.84 (±1.34)	19339 (±729)	0.321	1	0.5	8/8
W1024MW-a (pre)	125.17 (±0.84)	6608 (±104)	0.321	121.75 (±0.85)	6630 (±45)	0.321						
Property: R{"steps"}min=? [F "goal"]												
W512NW-b (nopre)	10.92 (±0.20)	1624 (±48)	1277.49	10.36 (±0.30)	1615 (±41)	1277.50	12.58 (±0.19)	6385 (±163)	1277.50	1	500	8/8
W512NW-b (pre)	10.96 (±0.26)	1609 (±37)	1277.49	10.37 (±0.27)	1604 (±38)	1277.50						
W512SW-b (nopre)	14.93 (±0.15)	1170 (±8)	1277.49	14.28 (±0.14)	1167 (±12)	1277.50	12.49 (±0.19)	6470 (±230)	1277.50	1	200	8/8
W512SW-b (pre)	14.93 (±0.17)	1171 (±11)	1277.49	14.26 (±0.13)	1170 (±10)	1277.50						
W512MW-b (nopre)	15.51 (±0.30)	1741 (±23)	1277.49	14.96 (±0.28)	1735 (±28)	1277.50	12.65 (±0.19)	6453 (±293)	1277.50	1	200	8/8
W512MW-b (pre)	15.42 (±0.22)	1734 (±35)	1277.49	14.70 (±0.29)	1744 (±31)	1277.50						
W1024NW-b (nopre)	83.23 (±0.58)	5358 (±60)	2557.48	77.32 (±0.90)	5360 (±84)	2557.50	65.18 (±0.64)	18179 (±1048)	2557.50	1	200	8/8
W1024NW-b (pre)	83.29 (±0.36)	5344 (±32)	2557.48	77.80 (±1.25)	5363 (±85)	2557.50						
W1024SW-b (nopre)	118.04 (±2.28)	5427 (±64)	2557.48	112.74 (±1.10)	5412 (±33)	2557.50	65.39 (±0.73)	18297 (±1027)	2557.50	1	500	8/8
W1024SW-b (pre)	117.83 (±1.03)	5491 (±172)	2557.48	113.29 (±1.53)	5410 (±29)	2557.50						
W1024MW-b (nopre)	118.17 (±0.50)	5808 (±38)	2557.48	113.99 (±1.13)	5778 (±97)	2557.50	66.43 (±0.58)	16938 (±1337)	2557.50	1	500	8/8
W1024MW-b (pre)	118.52 (±1.05)	5814 (±52)	2557.48	114.37 (±2.00)	5796 (±105)	2557.50						

memory, whereas PRISM-GS does not consistently perform better, except in a few scenarios. Instead, SHARP's hierarchical analysis completes the $\mathsf{P}_{\max}$ instances in approximately 39 s to 48 s and the $\mathsf{R}_{\min}$ instances in approximately 65 s to 67 s (Figure 4 and Table 2).

For example, in W1024SW-a ($\mathsf{P}_{\max}$), PRISM-VI requires around 47.3 s, whereas SHARP takes only about 39.1 s, resulting in a saving of over 8 s. For W1024SW-b ($\mathsf{R}_{\min}$), PRISM-VI/GS need approximately 118 s/113 s, while SHARP requires 65.4 s, achieving a 1.7× speedup.

Note that even in the $\mathsf{P}_{\max}$ cases where the performance gap is smaller than the 8 s difference observed in W1024SW-a, SHARP still offers a noticeable speedup of about 1.2×. For instance, in W1024MW-a, PRISM takes ≈46.3–46.6 s, while SHARP completes in ≈40.8 s.

We note that although SHARP may require 17–20 GB of memory on our million-state instances (Table 2), this remains within typical machine limits. The footprint, it turns out, has two components. Some of the extra memory is inherent to SHARP, since we keep the partition tree and per-leaf data. In our implementation, this part is about 12 bytes per state. Most of the memory we observe, however, comes from PRISM's Java-side representation, especially object headers and the maps used by the collections.

Also, differences in solution quality from PRISM are negligible; the largest gap we measure is 1.3×10^{-4} (see Section 6, RQ4 for a detailed analysis). SHARP's median execution time is lower than PRISM's in 5 out of 6 cases, with the largest gains on $\mathsf{R}_{\min}$ (Figure 5). On our largest warehouse instances, we observe about 1.2–2× speedup, with greater improvements for reward objectives than probability objectives. For time-sensitive tasks, these gains can outweigh the increased memory overhead.

Randomised robustness experiments. Figure 6 assesses all methods on five 1024×1024 models whose wall segments were placed at random positions (RandMW1–RandMW5). The points show

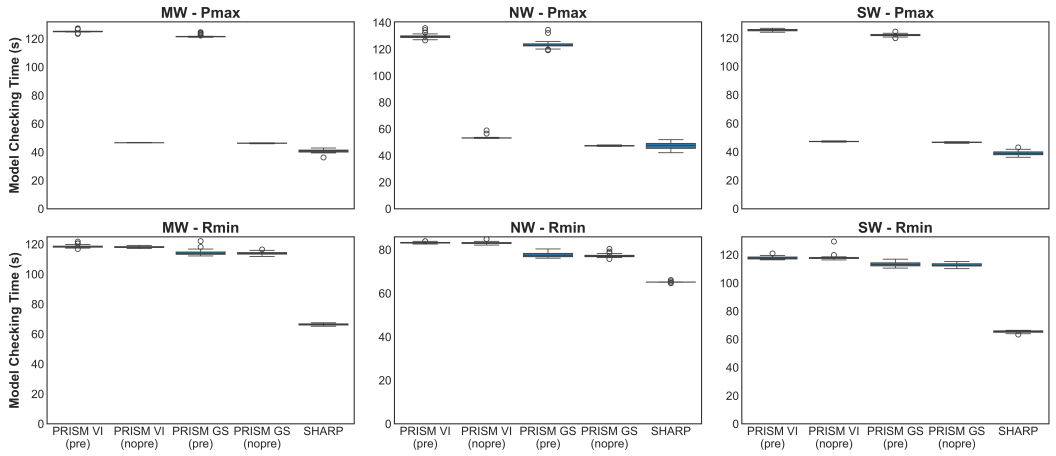

Fig. 5. Execution time comparison: SHARP vs. PRISM (VI/GS) on W1024* models for Pmax and Rmin properties over 30 independent runs.

the mean verification time (y-axis) against the run-to-run standard deviation (x-axis). The lower-left corner represents solutions that are both fast and stable. Across all five randomised problem instances, SHARP forms a tight cluster inside the highlighted "ideal block", with means between 57 s to 66 s and standard deviation below 2 s. PRISM-GS is consistently slower and more variable, while PRISM-VI is slower still, often taking nearly twice as long as SHARP and exhibiting up to three times its standard deviation. The clear separation confirms that SHARP delivers not only faster verification on large grid MDPs with randomly-placed walls, but also far more predictable performance across different random layouts.

RQ2 (Scalability). We evaluate scalability by comparing SHARP against the best-performing PRISM configuration (GS nopre) on the SW warehouse model across grid sizes 128×128 to 1024×1024 (Figure 7). For smaller grids (up to 256 × 256), SHARP's overhead outweighs its benefits. However, at 512×512 grids (2.6×10^5 states), SHARP already performs better on R_{min} (12.49s vs 14.28s), and by 1024 × 1024 (10^6 states) it is faster on both properties: P_{max} in 39.10s vs 46.68s and R_{min} in 65.39s vs 112.74s. Since SHARP maintains additional data structures for hierarchical refinement, its memory footprint is unsurprisingly higher across all grid sizes (Figure 7, right); for instance, 8.2 GB vs 1.1–1.9 GB at 512 × 512. Considering the memory capabilities of modern machines, these overheads are acceptable and well within reasonable limits. Across our grid-scaling experiment (Figure 7), SHARP's runtime grows substantially more slowly with model size than PRISM's; e.g., from 512^2 to 1024^2 (4× states), SHARP increases by about 5× while PRISM increases by about 8×, meaning that SHARP's scaling advantage is valuable for engineers verifying growing models. As we discuss in the guidance section, this scaling advantage is most pronounced on models with spatial or staged-progression structure.

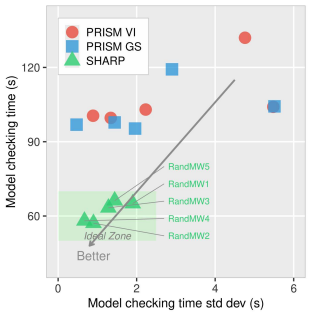

Fig. 6. Performance stability on random 1024 × 1024 MW models.

RQ3 (Generalisation). We used diverse MDP structures to evaluate SHARP's generalisation beyond spatial models. These include sequential arena models, the wlan3 network protocol, the

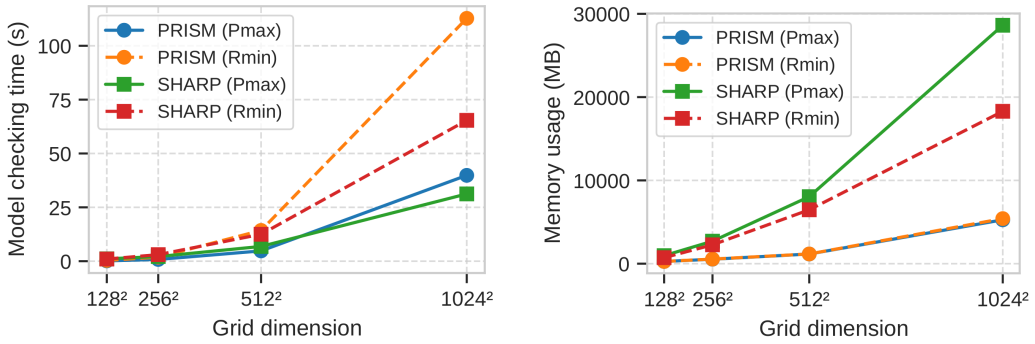

Fig. 7. Scalability comparison: time (left) and memory (right) for PRISM vs. SHARP across grid dimensions.

firewire root contention protocol, the consensus shared coin protocol, and the zeroconf network configuration protocol from the PRISM benchmark suite, using two different partition strategies. For `arenas8`, we used SCC-based partitioning; for `arenas16`, we grouped states by their progress counter, putting consecutive values into the same group (e.g., 0-3, 4-7, etc.). This enabled solving each group as a sub-MDP, only refining groups when needed.

Furthermore, for protocol models, we used SCC-based partitioning for `wlan3`, `consensus`, and `zeroconf`, and counter-based partitioning for `firewire`. SCC-based partitioning groups strongly connected components (clusters of states that can reach each other through transitions) into blocks, while counter-based partitioning groups states by protocol progress variables.

In Table 3, we observe that, on the large sequential arena model (`arenas16` with 788K states), SHARP achieves nearly 2× speedup over PRISM's standard VI, reducing model checking time from 295 s to 153 s. This occurs because there is an alignment between the arena's sequential structure and our hierarchical decomposition approach, where each stage operates independently with minimal inter-block communication.

On the other hand, SHARP did not perform well on protocol models, something which is evident from the model checking time which was 100× and 5× slower compared to PRISM, on `wlan3` and on `firewire`, respectively, and 5.6×–7.4× slower on `consensus` and `zeroconf`. This occurs because these protocol models have many inter-block dependencies due to the shared state variables and the requirement for synchronisation. Hence, the overhead of maintaining the hierarchy and coordinating between blocks outweighs any decomposition benefits, making running standard VI on the flat MDP a superior approach.

These findings demonstrate that SHARP generalises to diverse types of MDP models. Further, the results show that model structure is as critical as size for hierarchical approaches.

RQ4 (Hyperparameter sensitivity). We evaluate SHARP's sensitivity to key hyperparameters (maximum refinement depth D, cost threshold θ, and initial partition sizes) using the 1024×1024 SW instance (R_{min}). In Table 4, we show the performance impact for various configurations (A–E). Moreover, in Figure 9 we plot each configuration's time and memory usage, and colour the points according to the refinement depth D, where darker points indicate a lower D. Results show that initial partition size dominates performance, more than refinement depth or threshold. The $D = 1$ configurations (group A, 8×8 partitions) achieve the best performance with no refinements needed. While deeper refinements (D=2–5) increase resource usage, the initial partition size remains the critical factor: B1–B2 (8×8) outperform B3–B4 (16×16) even though B1 performs more refinements (5 vs 1); C2 (20×20) requires 23% more time than C1 (12×12) with identical refinements; and E2

Table 3. SHARP Performance Across Different MDP Structures. All experiments use $D = 3$ and $\theta = 0.05$.

Model	States	Type	PRISM	SHARP	Speedup	Partition
arenas8 (R_{min})	53K	Sequential	2.29s	2.94s	0.78×	SCC
wlan3 (P_{max})	97K	Protocol	0.047s	5.28s	0.01×	SCC
firewire (R_{min})	4K	Protocol	0.038s	0.187s	0.20×	Counter
arenas16 (R_{min})	788K	Sequential	295.53s	152.75s	1.93×	Counter
consensus (P_{max})	22K	Protocol	0.735s	5.414s	0.14×	SCC
zeroconf (R_{min})	1.1K	Protocol	0.040s	0.224s	0.18×	SCC

Table 4. SHARP configuration parameters and performance on 1024×1024 SW (R_{min}).

Config	**Time (s)**	**Mem (MB)**	**Leaves**	**Refinements**
A1 (D1, 200, 8×8)	65.6±0.8	18329±947	64	0
A2 (D1, 500, 8×8)	65.4±0.7	18297±1027	64	0
A3 (D1, 300, 8×8)	65.5±0.6	18283±1130	64	0
A4 (D1, 600, 8×8)	65.5±0.6	18178±962	64	0
B1 (D2, 200, 8×8)	67.4±0.5	19258±1101	66	5
B2 (D2, 500, 8×8)	65.8±0.7	18422±980	64	1
B3 (D2, 400, 16×16)	71.5±1.0	25848±523	256	1
B4 (D2, 800, 16×16)	71.2±1.0	25659±685	256	1
C1 (D3, 300, 12×12)	68.3±0.6	23951±1259	147	5
C2 (D3, 600, 20×20)	83.6±1.4	26972±562	403	5
D1 (D4, 400, 16×16)	73.2±1.4	26280±580	256	5
D2 (D4, 600, 16×16)	75.9±1.4	26023±610	256	5
E1 (D5, 500, 10×10)	69.3±1.0	21499±1284	100	7
E2 (D5, 800, 25×25)	103.9±3.7	28387±395	625	7

(25×25) shows the worst overall performance. This pattern confirms that smaller initial partitions (8×8 to 12×12) yield better performance regardless of refinement depth, as the overhead from managing numerous partitions outweighs refinement benefits.

Policy Quality. In order to evaluate the quality of the policies produced by SHARP, we measure the absolute gap $|V_{SHARP} - V^{\text{ref}}_{\text{PRISM}}|$ between the value computed by SHARP and the reference value obtained from PRISM. All 14 configurations listed in Table 4 achieve the same gap of 1.3×10^{-4}. This is an important observation, as it confirms that for the tested configurations and instances, the partitioning and refinement hyperparameters affect only time and memory, not the quality of the computed policy. For example, configuration A1 (D=1, no refinement) and configuration B1 (D=2, refinement enabled) achieve the same value gap despite their different refinement settings. We observe analogous behaviour for the 512×512 SW instance with P_{max}, where the maximum deviation across our configurations is 7.9×10^{-7}.

In contrast to the partitioning/refinement parameters discussed above, the boundary-change threshold η_{thr} is, one can argue, the most correctness-critical parameter. This stems from the fact that it essentially determines whether boundary values are propagated across the hierarchy or not. To investigate this, we sweep η_{thr} from 10^{-6} to 1.0 across 17 values for each refinement depth $D \in \{1, 2, 5\}$, fixing θ=200 and 8×8 partitions. Rather unexpectedly, the result is a sharp phase transition. All $\eta_{\text{thr}} \leq 0.9976$ yield the same gap (1.3×10^{-4}) regardless of depth, while all $\eta_{\text{thr}} \geq 0.998$

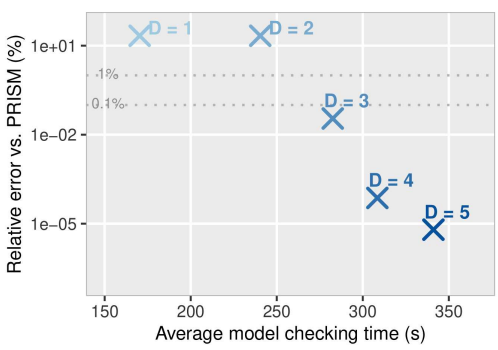

Fig. 8. Policy quality vs. model-checking time across refinement depths (D) under capped local VI on 1024×1024 MW (P_{max}, 3×3 partition).

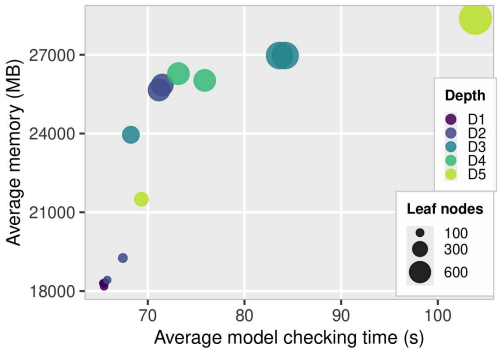

Fig. 9. Scatter plot of SHARP hyperparameter configurations (Table 4) on the 1024 × 1024 SW instance (R_{min}).

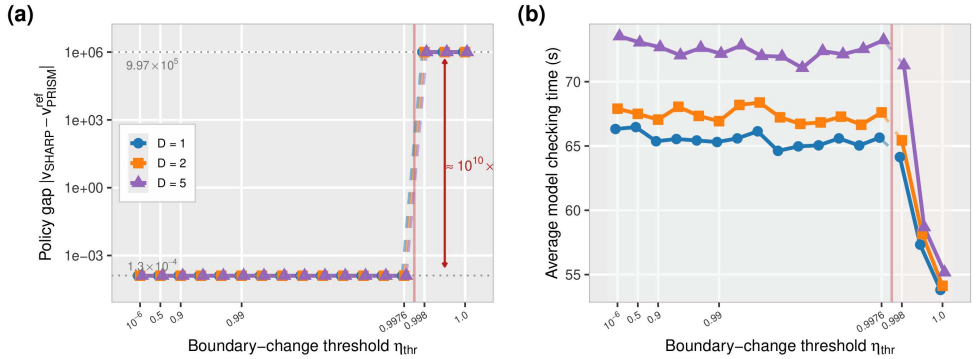

Fig. 10. Effect of η_{thr} on 1024×1024 SW (R_{min}, θ=200, 8×8): (a) policy-quality gap and (b) model-checking time.

produce a gap of 9.97×10^5, reflecting the large cost assigned to states whose local sub-MDPs can no longer reach the goal through stale boundary values. The catch is of course that at η_{thr}=1.0, boundary-triggered re-solving is effectively suppressed altogether and the algorithm terminates after only 2 global iterations. Even deep refinement (D=5) cannot compensate, because refinement alone does not substitute for boundary feedback. This observation is of course consistent with Theorem 4.1: when boundary feedback is suppressed, η grows large and as a result the error guarantee degrades accordingly.

Hence, SHARP users should begin with $D = 1$ or $D = 2$ and 8×8 partitions, as these configurations perform well in diverse problems, while initial partitions larger than 12×12 typically degrade performance. Deeper refinements (higher D) are justified mainly when observing large spreads in critical regions. The boundary-change threshold η_{thr} should be set conservatively (e.g., 10^{-6}) to ensure that boundary feedback is not suppressed (Figure 10).

We should note that the preceding 8×8 experiments use partitions well-matched to the grid structure, leaving little room for refinement to improve quality. To isolate the effect of refinement, we deliberately misalign the partition by using a 3×3 grid on the 1024×1024 MW instance (P_{max}), where vertical walls fall inside blocks, and cap local VI at 300 iterations (Figure 8). Without refinement

(D=1, D=2), the policy gap stays at 21.6%; at D=3, it drops to 0.04%. Deeper refinement (D=4, D=5) yields diminishing returns, though, improving the gap only to $\approx 10^{-4}$% and $\approx 10^{-5}$% respectively. Refinement is most valuable under a fixed compute budget with misaligned partitions; when the partition already matches the problem structure, the gains are marginal.

Guidance on using SHARP. SHARP is most effective on models with limited cross-boundary dependencies (spatial or staged-progression MDPs). In practice, this means that transitions are mostly local within a block and the interaction across blocks is relatively sparse. However, on tightly coupled protocol models flat methods such as standard VI remain preferable. We emphasise that SHARP still produces correct results on such models; the limitation is purely one of efficiency and does not affect solution quality. A natural extension would be to detect when the hierarchical decomposition is slower than flat VI and fall back automatically. We also observed that initial partition size matters more than refinement depth or threshold: small grids (8×8–12×12) with $D \leq 3$ yielded the best results. For new models, we suggest setting $\theta \in [0.2, 0.5]$ for probability objectives and θ scaled to the expected reward range for cost objectives (e.g., 200–500 when expected costs are in the thousands). Finally, η_{thr} should be set conservatively (e.g., 10^{-6}) so that significant boundary changes trigger re-solving of affected sub-MDPs (Figure 10).

6.4 Threats to Validity

We mitigate **construct validity threats** that arise due to the lack of global context in hierarchically decomposed sub-MDPs by grounding SHARP in the well-established VI algorithm for probabilities and rewards. Furthermore, SHARP propagates values up the tree and supplies boundary values downward, while refinement is triggered whenever the value spread criterion is violated. This allows us to recover accuracy in those parts of the hierarchy where it is most needed.

We limit **internal validity threats** by integrating SHARP into PRISM's explicit-state engine, reusing its core while supplying our algorithms for hierarchical partitioning, adaptive refinement, and local sub-MDP solving. By building on a widely tested code base, we minimise the risk that implementation bugs skew our measurements. Setting η_{thr} to a suitably small value, for example 10^{-6}, bounds the boundary mismatch. However, the approach is not necessarily beneficial in all cases. For example, on tightly coupled MDPs the overhead may outweigh the gains (Table 3), while setting η_{thr} too permissively may suppress boundary feedback altogether (Figure 10).

We mitigate **external validity threats** by evaluating SHARP on large benchmarks that correspond to realistic warehouse robot scenarios of varying complexity and model sizes [29], and by comparing it against the VI and GS algorithms of PRISM [46]. Assessing SHARP against other state-of-the-art tools such as Storm [36] is not straightforward, for the reason that Storm is implemented in C++ and employs different data structures and solver backends. Therefore, a fairer evaluation would require porting SHARP's core algorithm (Algorithm 1) to Storm's framework. However, SHARP's partition-refinement approach is solver-agnostic at the leaf level, meaning that any local solver could, in principle, replace VI. As a result, a Storm-based backend constitutes a natural direction for future work.

Additionally, SHARP currently supports only single-objective properties ($\mathsf{P}_{\max}$ and $\mathsf{R}_{\min}$); multi-objective and constrained synthesis remain outside its scope. Lastly, more experiments on additional non-grid MDPs, and on additional application domains (e.g., self-adaptive systems [52], software product lines [19]) are needed to establish SHARP's applicability and scalability in domains beyond those in our evaluation.

7 Related Work

Probabilistic models in Software Engineering. There are numerous applications of probabilistic models in an SE context; for example [18] propose a framework using MDPs to dynamically optimise

mobile phone software, while [44] apply a similar approach to optimise service migration decisions in the cloud for mobile users. Similarly, [16] uses probabilistic modelling for QoS management in service-based systems. Moreover, [26, 49] use probabilistic model checking to derive dependable auto-scaling policies for cloud deployments. SHARP can, in principle, accelerate the policy synthesis step underlying each of these applications when the state space becomes prohibitively large.

Compositionality and abstraction–refinement for verification. CEGAR [20] pioneered iterative refinement for non-probabilistic systems; probabilistic variants refine MDP abstractions with counterexamples [37, 43]. As discussed in Section 1, for hierarchical MDPs, [41] computes coarse probability bounds via parametric analysis and refines subroutines as needed. Watanabe [55] presents a compositional model checking framework for MDPs that uses the formal language of string diagrams to decompose models and compute optimal expected rewards. Unlike these methods, SHARP partitions the concrete state space automatically and refines based upon value spread rather than counterexample elimination or formal compositional structure.

Search-based methods. Search-based approaches have also been used for related probabilistic synthesis problems. EvoChecker [30, 32] uses evolutionary algorithms to synthesise probabilistic model configurations that satisfy QoS requirements, whereas EvoPoli [31] considers constrained multi-objective MDP policy synthesis and approximates Pareto-optimal policy sets. These methods target approximate search over configurations or policies. In contrast, SHARP assumes a fixed MDP and is concerned with accelerating exact single-objective synthesis by means of hierarchical decomposition and adaptive refinement.

Hierarchical RL. Early HRL frameworks such as Feudal RL [22], MAXQ [25], and Options [53] decompose an MDP into subtasks or macro-actions, which significantly improves computational tractability. Recent abstractions (e.g., AVI [40]) extend these ideas to continuous spaces. Similarly, Monte Carlo-based approaches [3] treat abstracted MDPs as POMDPs in order to scale tree search. These methods target the reinforcement learning setting, where the transition model is unknown and must be explored, whereas SHARP operates in the model-based setting where the full MDP is given. Note, however, that unlike SHARP, HRL methods usually learn from samples and they provide no optimality guarantees.

8 Conclusion

We presented SHARP, a hierarchical adaptive refinement approach for accelerating policy synthesis in large MDPs. We implemented SHARP as an open-source extension to PRISM and evaluated it on scenarios with up to 1M states. The results show that SHARP can improve efficiency over PRISM-based policy synthesis using VI and GS, while preserving a high level of accuracy. We also provided formal guarantees showing that the composed policy remains near-optimal when the boundary mismatch is controlled. The approach is most beneficial for MDPs with spatial or staged-progression structure. On tightly coupled models, standard flat solvers may still be preferable from an efficiency point of view.

There are several directions in which this work can be developed further. For example, it would be useful to extend SHARP to multi-objective and constrained synthesis, and also to investigate how the approach behaves in multi-agent settings. In addition, more experimentation is needed in larger model instances in order to assess the scalability limits of the approach more thoroughly. Finally, it would be interesting to examine whether partition sizes and refinement thresholds can be selected automatically based upon the structure of the model, rather than being chosen manually.

Data Availability

The SHARP implementation, benchmark models, experimental data, and supplementary proofs are available at https://github.com/alexEvangelidis/sharp.

Acknowledgments

This research was supported by the Horizon Europe projects AI4Work (101135990) and SOPRANO (101120990).

References

[1] Luca de Alfaro, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. 2000. Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker Representation. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000 (TACAS '00)*. Springer-Verlag, Berlin, Heidelberg, 395–410.

[2] Luke Antonyshyn, Jefferson Silveira, Sidney Givigi, and Joshua Marshall. 2023. Multiple Mobile Robot Task and Motion Planning: A Survey. *ACM Comput. Surv.* 55, 10, Article 213 (Feb. 2023), 35 pages. doi:10.1145/3564696

[3] Aijun Bai, Siddharth Srivastava, and Stuart Russell. 2016. Markovian state and action abstractions for MDPS via hierarchical MCTS. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (New York, New York, USA) *(IJCAI'16)*. AAAI Press, 3029–3037.

[4] Christel Baier, Holger Hermanns, and Joost-Pieter Katoen. 2019. The 10,000 facets of MDP model checking. *Computing and Software Science: State of the Art and Perspectives* (2019), 420–451.

[5] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

[6] Severin Bals, Alexandros Evangelidis, Jan Křetínský, and Jakob Waibel. 2024. MULTIGAIN 2.0: MDP controller synthesis for multiple mean-payoff, LTL and steady-state constraints. In *Proceedings of the 27th ACM International Conference on Hybrid Systems: Computation and Control* (Hong Kong SAR, China) *(HSCC '24)*. Association for Computing Machinery, New York, NY, USA, Article 24, 7 pages. doi:10.1145/3641513.3650135

[7] Dimitri P. Bertsekas. 1983. Distributed asynchronous computation of fixed points. *Math. Program.* 27, 1 (Sept. 1983), 107–120. doi:10.1007/BF02591967

[8] Dimitri P. Bertsekas and John N. Tsitsiklis. 1991. An Analysis of Stochastic Shortest Path Problems. *Mathematics of Operations Research* 16, 3 (1991), 580–595. http://www.jstor.org/stable/3690040

[9] Dimitri P. Bertsekas and John N. Tsitsiklis. 1996. *Neuro-dynamic programming*. Optimization and neural computation series, Vol. 3. Athena Scientific. I–XIII, 1–491 pages.

[10] Andrea Bianco and Luca De Alfaro. 1995. Model checking of probabilistic and nondeterministic systems. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 499–513.

[11] Carlos E. Budde, Arnd Hartmanns, Michaela Klauck, Jan Křetínský, David Parker, Tim Quatmann, Andrea Turrini, and Zhen Zhang. 2021. On Correctness, Precision, and Performance in Quantitative Verification. In *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 216–241.

[12] Mingyu Cai, Hao Peng, Zhijun Li, and Zhen Kan. 2021. Learning-Based Probabilistic LTL Motion Planning With Environment and Motion Uncertainties. *IEEE Trans. Automat. Control* 66, 5 (2021), 2386–2392. doi:10.1109/TAC.2020.3006967

[13] Radu Calinescu, Milan Češka, Simos Gerasimou, Marta Kwiatkowska, and Nicola Paoletti. 2018. Efficient synthesis of robust models for stochastic systems. *Journal of Systems and Software* 143 (2018), 140–158.

[14] Radu Calinescu, Simos Gerasimou, Kenneth Johnson, and Colin Paterson. 2018. Using runtime quantitative verification to provide assurance evidence for self-adaptive software: advances, applications and research challenges. In *Software Engineering for Self-Adaptive Systems III. Assurances: International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*. Springer, 223–248.

[15] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. 2012. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM* 55, 9 (2012), 69–77.

[16] Radu Calinescu, Lars Grunske, Marta Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. 2011. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Trans. Softw. Eng.* 37, 3 (May 2011), 387–409. doi:10.1109/TSE.2010.92

[17] Krishnendu Chatterjee and Thomas A. Henzinger. 2008. *Value Iteration*. Springer Berlin Heidelberg, Berlin, Heidelberg, 107–138. doi:10.1007/978-3-540-69850-0_7

[18] Tang Lung Cheung, Kari Okamoto, Frank Maker, Xin Liu, and Venkatesh Akella. 2009. Markov decision process (MDP) framework for optimizing software on mobile phones. In *Proceedings of the Seventh ACM International Conference on Embedded Software* (Grenoble, France) *(EMSOFT '09)*. Association for Computing Machinery, New York, NY, USA, 11–20. doi:10.1145/1629335.1629338

[19] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. 2016. Family-Based Modeling and Analysis for Probabilistic Systems — Featuring ProFeat. In *Proceedings of the 19th International Conference on Fundamental*

Approaches to Software Engineering - Volume 9633. Springer-Verlag, Berlin, Heidelberg, 287–304. doi:10.1007/978-3-662-49665-7_17

[20] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, E. Allen Emerson and Aravinda Prasad Sistla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–169.

[21] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2012. *Model Checking and the State Explosion Problem.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1–30. doi:10.1007/978-3-642-35746-6_1

[22] Peter Dayan and Geoffrey E Hinton. 1992. Feudal Reinforcement Learning. In *Advances in Neural Information Processing Systems*, S. Hanson, J. Cowan, and C. Giles (Eds.), Vol. 5. Morgan-Kaufmann. https://proceedings.neurips.cc/paper_files/paper/1992/file/d14220ee66aeec73c49038385428ec4c-Paper.pdf

[23] Thomas Dean and Robert Givan. 1997. Model minimization in Markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence* (Providence, Rhode Island) *(AAAI'97/IAAI'97)*. AAAI Press, 106–111.

[24] Eric V. Denardo. 1967. Contraction Mappings in the Theory Underlying Dynamic Programming. *SIAM Rev.* 9, 2 (April 1967), 165–177. doi:10.1137/1009030

[25] Thomas G. Dietterich. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *J. Artif. Int. Res.* 13, 1 (Nov. 2000), 227–303.

[26] Alexandros Evangelidis, David Parker, and Rami Bahsoon. 2018. Performance modelling and verification of cloud-based auto-scaling policies. *Future Generation Computer Systems* 87 (2018), 629–638. doi:10.1016/j.future.2017.12.047

[27] Norm Ferns, Prakash Panangaden, and Doina Precup. 2004. Metrics for finite Markov decision processes. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence* (Banff, Canada) *(UAI '04)*. AUAI Press, Arlington, Virginia, USA, 162–169.

[28] Michael Fisher, Rafael C Cardoso, Emily C Collins, Christopher Dadswell, Louise A Dennis, Clare Dixon, Marie Farrell, Angelo Ferrando, Xiaowei Huang, Mike Jump, et al. 2021. An overview of verification and validation challenges for inspection robots. *Robotics* 10, 2 (2021), 67.

[29] Giuseppe Fragapane, René de Koster, Fabio Sgarbossa, and Jan Ola Strandhagen. 2021. Planning and control of autonomous mobile robots for intralogistics: Literature review and research agenda. *European Journal of Operational Research* 294, 2 (2021), 405–426. doi:10.1016/j.ejor.2021.01.019

[30] Simos Gerasimou, Radu Calinescu, and Giordano Tamburrelli. 2018. Synthesis of probabilistic models for quality-of-service software engineering. *Automated Software Engg.* 25, 4 (Dec. 2018), 785–831.

[31] Simos Gerasimou, Javier Cámara, Radu Calinescu, Naif Alasmari, Faisal Alhwikem, and Xinwei Fang. 2022. Evolutionary-guided synthesis of verified pareto-optimal MDP policies. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 842–853. doi:10.1109/ASE51524.2021.9678727

[32] Simos Gerasimou, Giordano Tamburrelli, and Radu Calinescu. 2015. Search-Based Synthesis of Probabilistic Models for Quality-of-Service Software Engineering (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 319–330. doi:10.1109/ASE.2015.22

[33] Hans Hansson and Bengt Jonsson. 1994. A logic for reasoning about time and reliability. *Form. Asp. Comput.* 6, 5 (Sept. 1994), 512–535. doi:10.1007/BF01211866

[34] Arnd Hartmanns and Holger Hermanns. 2015. Explicit Model Checking of Very Large MDP Using Partitioning and Secondary Storage. In *Automated Technology for Verification and Analysis*, Bernd Finkbeiner, Geguang Pu, and Lijun Zhang (Eds.). Springer International Publishing, Cham, 131–147.

[35] David Henriques, Joao G. Martins, Paolo Zuliani, Andre Platzer, and Edmund M. Clarke. 2012. Statistical Model Checking for Markov Decision Processes. In *Proceedings of the 2012 Ninth International Conference on Quantitative Evaluation of Systems (QEST '12)*. IEEE Computer Society, USA, 84–93. doi:10.1109/QEST.2012.19

[36] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The probabilistic model checker Storm. *Int. J. Softw. Tools Technol. Transf.* 24, 4 (Aug. 2022), 589–610. doi:10.1007/s10009-021-00633-z

[37] Holger Hermanns, Björn Wachter, and Lijun Zhang. 2008. Probabilistic cegar. In *International Conference on Computer Aided Verification*. Springer, 162–175.

[38] Sara M Hezavehi, Danny Weyns, Paris Avgeriou, Radu Calinescu, Raffaela Mirandola, and Diego Perez-Palacin. 2021. Uncertainty in self-adaptive systems: A research community perspective. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 15, 4 (2021), 1–36.

[39] Siti Nuraishah Agos Jawaddi, Muhammad Hamizan Johari, and Azlan Ismail. 2022. A review of microservices autoscaling with formal verification perspective. *Software: Practice and Experience* 52, 11 (2022), 2476–2495.

[40] Kishor Jothimurugan, Osbert Bastani, and Rajeev Alur. 2021. Abstract Value Iteration for Hierarchical Reinforcement Learning. In *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 130)*, Arindam Banerjee and Kenji Fukumizu (Eds.). PMLR, 1162–1170. https:

//proceedings.mlr.press/v130/jothimurugan21a.html
[41] Sebastian Junges and Matthijs TJ Spaan. 2022. Abstraction-refinement for hierarchical probabilistic models. In *International Conference on Computer Aided Verification*. Springer, 102–123.
[42] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement learning: a survey. *J. Artif. Int. Res.* 4, 1 (May 1996), 237–285.
[43] Mark Kattenbelt, Marta Kwiatkowska, Gethin Norman, and David Parker. 2010. A game-based abstraction-refinement framework for Markov decision processes. *Formal Methods in System Design* 36 (2010), 246–280.
[44] Adlen Ksentini, Tarik Taleb, and Min Chen. 2014. A Markov Decision Process-based service migration procedure for follow me cloud. In *2014 IEEE International Conference on Communications (ICC)*. 1350–1354. doi:10.1109/ICC.2014.6883509
[45] Marta Kwiatkowska, Gethin Norman, and David Parker. 2007. *Stochastic Model Checking*. Springer Berlin Heidelberg, Berlin, Heidelberg, 220–270. doi:10.1007/978-3-540-72522-0_6
[46] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11) (LNCS, Vol. 6806)*, G. Gopalakrishnan and S. Qadeer (Eds.). Springer, 585–591.
[47] Marta Kwiatkowska, Gethin Norman, and David Parker. 2022. Probabilistic Model Checking and Autonomy. *Annual Review of Control, Robotics, and Autonomous Systems* 5, Volume 5, 2022 (2022), 385–410. doi:10.1146/annurev-control-042820-010947
[48] Jan Leike, Miljan Martic, Victoria Krakovna, Pedro A. Ortega, Tom Everitt, Andrew Lefrancq, Laurent Orseau, and Shane Legg. 2017. AI Safety Gridworlds. *CoRR* abs/1711.09883 (2017). arXiv:1711.09883 http://arxiv.org/abs/1711.09883
[49] Athanasios Naskos, Emmanouela Stachtiari, Anastasios Gounaris, Panagiotis Katsaros, Dimitrios Tsoumakos, Ioannis Konstantinou, and Spyros Sioutas. 2015. Dependable horizontal scaling based on probabilistic model checking. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing* (Shenzhen, China) *(CCGRID '15)*. IEEE Press, 31–40. doi:10.1109/CCGrid.2015.91
[50] Victor Casamayor Pujol, Praveen Kumar Donta, Andrea Morichetta, Ilir Murturi, and Schahram Dustdar. 2023. Edge intelligence—research opportunities for distributed computing continuum systems. *IEEE Internet Computing* 27, 4 (2023), 53–74.
[51] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st ed.). John Wiley & Sons, Inc., USA.
[52] Guoxin Su, Taolue Chen, Yuan Feng, David S. Rosenblum, and P. S. Thiagarajan. 2016. An Iterative Decision-Making Scheme for Markov Decision Processes and Its Application to Self-adaptive Systems. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering - Volume 9633*. Springer-Verlag, Berlin, Heidelberg, 269–286. doi:10.1007/978-3-662-49665-7_16
[53] Richard S. Sutton, Doina Precup, and Satinder Singh. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112, 1 (1999), 181–211. doi:10.1016/S0004-3702(99)00052-1
[54] Alvaro Velasquez, Ismail Alkhouri, Andre Beckus, Ashutosh Trivedi, and George Atia. 2022. Controller Synthesis for Omega-Regular and Steady-State Specifications. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems* (Virtual Event, New Zealand) *(AAMAS '22)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1310–1318.
[55] Kazuki Watanabe, Clovis Eberhart, Kazuyuki Asada, and Ichiro Hasuo. 2023. Compositional Probabilistic Model Checking with String Diagrams of MDPs. In *Computer Aided Verification - 35th International Conference, CAV 2023, Proceedings (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics))*, Constantin Enea and Akash Lal (Eds.). Springer Science and Business Media Deutschland GmbH, Germany, 40–61. doi:10.1007/978-3-031-37709-9_3
[56] Danny Weyns, Radu Calinescu, Raffaela Mirandola, Kenji Tei, Maribel Acosta, Nelly Bencomo, Amel Bennaceur, Nicolas Boltz, Tomas Bures, Javier Camara, et al. 2023. Towards a research agenda for understanding and managing uncertainty in self-adaptive systems. *ACM SIGSOFT Software Engineering Notes* 48, 4 (2023), 20–36.
[57] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. 2007. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. In *Proceedings of the 19th National Conference on Innovative Applications of Artificial Intelligence - Volume 2* (Vancouver, British Columbia, Canada) *(IAAI'07)*. AAAI Press, 1752–1759.
[58] Xingyu Zhao, Valentin Robu, David Flynn, Fateme Dinmohammadi, Michael Fisher, and Matt Webster. 2019. Probabilistic model checking of robots deployed in extreme environments. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 8066–8074.