



# Lazy Abstraction for Probabilistic Systems

Dániel Szekeres , István Majzik   
Department of Measurement and Information Systems  
Budapest University of Technology and Economics  
Budapest, Hungary  
szekeresdani@edu.bme.hu, majzik@mit.bme.hu

**Abstract**—Reliability analysis of complex safety-critical systems by probabilistic model checking is often hindered by state space explosion. Abstraction is one way to counteract this problem. In this paper, we adapt an existing lazy abstraction algorithm to the analysis of Markov Decision Process reliability models and prove its soundness.

**Index Terms**—Probabilistic Model Checking, Markov Decision Processes, Lazy Abstraction

## I. INTRODUCTION

It is crucial to guarantee the reliable and safe operation of safety-critical systems such as railway interlocking systems or embedded controllers in vehicles.

Probabilistic model checking is an automatic approach for the analysis of such quantitative properties with formal mathematical guarantees [1]. In this paper, we focus on the most fundamental task of safety-related probabilistic model checking: computing the probability of reaching an error state. If the behavior of the system under analysis includes non-deterministic aspects, we are interested in worst-case analysis: the probability of reaching an error state if the non-determinism is resolved so that it maximizes this probability. A Markov Decision Process (MDP) is able to describe the behavior of the system if discrete-time analysis and a discrete state space is sufficient, and the analysis of other modeling formalisms (like Markov Automata or Probabilistic Timed Automata) can often be reduced to MDP analysis as well.

Practical usage of probabilistic model checking is hindered by the state-space explosion problem: the state-space of the MDP is often intractably large, which can be caused for example by concurrent execution of components that depend on each other. Abstraction is a way to counteract this problem, and several abstraction-based techniques have been adapted to probabilistic systems, like CEGAR [2], [3] and abstract interpretation [4]. Lazy abstraction is another approach that merges abstraction and refinement steps and applies refinement during state-space exploration only for those abstract states where it is necessary. It has seen success in non-probabilistic model checking, but it has yet to find its way into the probabilistic setting. In this paper, we adapt a lazy abstraction algorithm to the analysis of MDP models and prove its soundness.

Project no. 2019-1.3.1-KK-2019-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2019-1.3.1-KK funding scheme.

## II. BACKGROUND AND NOTATIONS

$\mathbb{D}(A)$  is the set of probability distributions over the set  $A$ . For  $d \in \mathbb{D}(A)$ ,  $a \in A$ ,  $d(a)$  denotes the probability measure of  $a$  according to  $d$ .  $f : A \rightharpoonup B$  means that  $f$  is a *partial function* from  $A$  to  $B$ , and  $Supp(f)$  denotes the set of values where  $f$  is defined. For  $d \in \mathbb{D}(A)$ ,  $Supp(d) = \{a \in A | d(a) > 0\}$ .  $\delta_x$  is a dirac distribution:  $\delta_x(x) = 1, \forall y \neq x : \delta_x(y) = 0$ .

*a) Markov Decision Process (MDP):* MDPs are low-level mathematical models able to describe probabilistic and non-deterministic behavior in discrete time. An MDP is a tuple  $M = (S, Act, T, s_0)$ , where  $S$  is the set of states,  $Act$  is the set of actions,  $T : S \times Act \times S \rightarrow [0, 1]$  is the probabilistic transition function satisfying  $\forall s \in S, a \in Act : \sum_{s' \in S} T(s, a, s') \in \{0, 1\}$  and  $s_0 \in S$  is the initial state. An action  $a \in Act$  is *enabled* in  $s \in S$  if  $\sum_{s' \in S} T(s, a, s') = 1$ . If an action  $a \in Act$  is enabled in  $s \in S$ ,  $T(a, s) \in \mathbb{D}(S)$  denotes the next state distribution after taking the action  $a$  in  $s$ , defined as  $T(s, a)(s') = T(s, a, s')$ . The intuitive behavior of an MDP is as follows: we start in  $s_0$ , then in each step, an action  $a$  is chosen non-deterministically from those enabled in the current state  $s$ , and the next state is sampled from  $T(s, a)$ . A *trace* of an MDP is an alternating list of states and actions  $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \dots$  such that  $T(s_{i-1}, a_i, s_i) > 0$  for each step. If we fix a strategy for resolving the non-determinism, the set of traces can be equipped with a probability measure: intuitively, the probability of a trace is the product of the probability of landing in each state of the trace after taking the action specified by the strategy in the previous state. For a detailed formal treatment, see e. g. [1].

Given an MDP  $M$  describing the system behavior and set of error states  $E$ , we want to compute (an upper approximation of) the probability measure of the set of traces that involve a state in  $E$ :  $\mathbb{P}(\{(s_0 s_1 s_2 \dots) \mid \exists i \in \mathbb{Z}^+ : s_i \in E\})$ . The result is the same if we make all error states absorbing – this way, we can restrict the analysis to finite traces.

*b) Symbolic description of MDPs:* Instead of explicitly constructing the state space of the model of a complex real-life system by hand, most such models are defined symbolically using state variables and operations on them.

We assume that the MDP to be analyzed is given by a set of state variables  $\mathcal{V}$  and a set of probabilistic guarded commands  $\mathcal{C}$ . Each  $v \in \mathcal{V}$  has an associated set  $Range(v)$  of values it can take, and an initial value  $v_0 \in Range(v)$ . A *valuation* over  $\mathcal{V}$  is a function  $val : \mathcal{V} \rightarrow \bigcup_{v \in \mathcal{V}} Range(v)$  such that

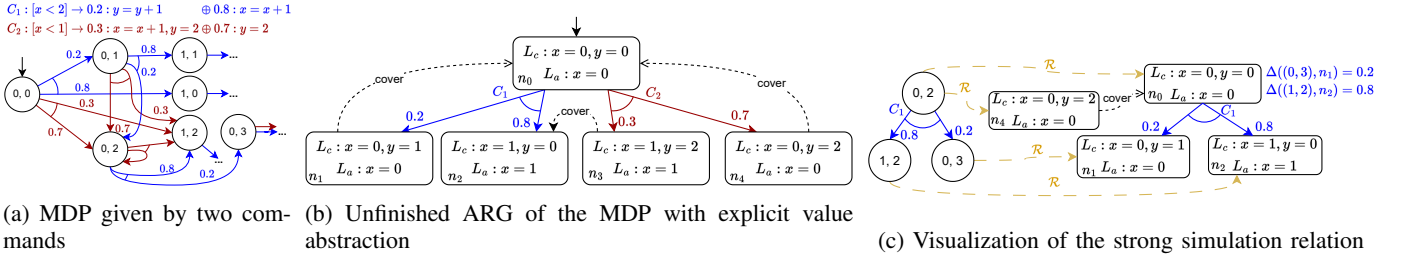


Fig. 1: Example of an MDP given through probabilistic guarded commands, an ARG of this MDP, and how the strong simulation maps from MDP states to ARG nodes

$\forall v \in \mathcal{V} : val(v) \in Range(v)$ . The initial valuation of the model is a valuation  $val_0$  such that  $\forall v \in \mathcal{V} : val_0(v) = v_0$ . The state space of the symbolically described MDP is a subset of  $VAL_{\mathcal{V}}$ , and its initial state distribution is  $\delta_{val_0}$ .

Let  $VAL_{\mathcal{V}}$  denote the set of all valuations over  $\mathcal{V}$ ,  $\mathcal{B}_{\mathcal{V}}$  the set of all Boolean expressions over  $\mathcal{V}$ ,  $\mathcal{E}_{\mathcal{V}}^v$  the set of all expressions over  $\mathcal{V}$  that result in an element of  $Range(v)$ , and  $\mathcal{E}_{\mathcal{V}} = \bigcup_{v \in \mathcal{V}} \mathcal{E}_{\mathcal{V}}^v$ . A *next state assignment* is a function  $n : \mathcal{V} \rightarrow \mathcal{E}_{\mathcal{V}}$ , such that  $\forall v \in \mathcal{V} : n(v) \in \mathcal{E}_{\mathcal{V}}^v$ . Let  $\mathcal{N}_{\mathcal{V}}$  denote the set of all next state assignments for  $\mathcal{V}$ .  $eval(e, val)$  for  $e \in \mathcal{E}_{\mathcal{V}}$  and  $val \in VAL_{\mathcal{V}}$  is the constant resulting from replacing each  $v \in \mathcal{V}$  in  $e$  with  $val(v)$ , then evaluating the (now constant) expression.  $eval(n, val)$  for  $n \in \mathcal{N}_{\mathcal{V}}$  is a valuation  $val'$  such that  $\forall v \in \mathcal{V} : val'(v) = eval(n(v), val)$ .  $eval(d, val)$  for  $d \in \mathbb{D}(\mathcal{N}_{\mathcal{V}})$  is the distribution  $d' \in \mathbb{D}(VAL_{\mathcal{V}})$  such that  $d'(val') = \sum_{n \in Supp(d) | eval(n, val) = val'} d(n)$ .

A command  $c \in \mathcal{C}$  consists of a guard  $g_c \in \mathcal{B}_{\mathcal{V}}$  and a result distribution  $d_c \in \mathbb{D}(\mathcal{N}_{\mathcal{V}})$ .  $c$  is *enabled* by the valuation  $val$  iff  $eval(g_c, val) = True$ . The set of enabled actions in each state  $val$  of the resulting MDP consists of the set of commands enabled by  $val$ , and taking the command  $c$  results in the distribution  $eval(d_c, val)$ . An example MDP can be seen in Figure 1a. Widely used symbolic MDP description formats, like that of PRISM [5] or the JANI [6] interchange format can be mapped to this low-level symbolic description.

*c) Abstraction:* Abstraction is a successful approach to combat state-space explosion by ignoring information that is not needed to prove or disprove the property of interest. An *abstract model* is created which conservatively approximates the original *concrete model*: if the abstract model satisfies the property of interest, then the concrete model does as well.

Abstract states are commonly described symbolically using an *abstract domain*. Given a set of concrete states  $S$ , an abstract domain  $D = (\hat{S}, \preceq, \alpha, \gamma)$  consists of the set of abstract states  $\hat{S}$ , a partial ordering relation  $\preceq \subseteq \hat{S} \times \hat{S}$ , an abstraction function  $\alpha : 2^S \rightarrow \hat{S}$  and a concretization function  $\gamma : \hat{S} \rightarrow 2^S$ .  $\alpha$  and  $\gamma$  form a Galois connection between the posets  $(2^S, \subseteq)$  and  $(\hat{S}, \preceq)$ , meaning  $\forall A \subseteq 2^S, \hat{a} \in \hat{S} : \alpha(A) \preceq \hat{a} \iff A \subseteq \gamma(\hat{a})$ .  $\gamma$  lets us treat abstract states as sets of concrete states; we write “ $s \in \hat{s}$ ” for  $s \in \gamma(\hat{s})$  when  $\gamma$  is clear from the context.  $x \preceq y$  denotes  $(x, y) \in \preceq$ .  $\hat{S}$  has two special elements:  $\top$  and  $\perp$  satisfying  $\gamma(\top) = S, \gamma(\perp) = \{\}$ .

For checking safety properties in the qualitative case, the

conservative direction is to overapproximate the set of reachable states: a transition from  $\hat{s} \in \hat{S}$  to  $\hat{s}' \in \hat{S}$  is possible iff  $\exists s \in \hat{s}, s' \in \hat{s}' : \text{a transition exists from } s \text{ to } s'$ . For the probabilistic setting, this changes to the requirement that the *probability* of reaching an error state in the abstract model must be *at least as high* as in the concrete one, which we will prove for the proposed algorithm.

The lazy abstraction algorithm needs an abstract domain for  $S = VAL_{\mathcal{V}}$ , for which we assume the existence of some operations. The algorithms in this paper are domain-agnostic as long as these can be computed. As an example, the explicit value domain satisfies these requirements.

For  $a \in \mathcal{N}_{\mathcal{V}}$  and  $\hat{s} \in \hat{S}$ ,  $eval(a, \hat{s}) \in \hat{S}$  is the *abstract post operator* computing the result of applying a next state assignment in the abstract state space:  $eval(a, \hat{s}) = \alpha(\{eval(a, s) | s \in \hat{s}\})$ . For  $b \in \mathcal{B}_{\mathcal{V}}$ ,  $eval(b, \hat{s}) \in \{True, False, Unknown\}$  denotes evaluating  $b$  in the abstract state space: *True* if  $b$  evaluates to *True* for all  $s \in \hat{s}$ , *False* if  $b$  evaluates to *False* for all  $s \in \hat{s}$ , otherwise *Unknown*.

We assume the existence of a *block* operation: for an abstract state  $\hat{s} \in \hat{S}$ , a Boolean expression  $b \in \mathcal{B}_{\mathcal{V}}$  and a concrete state  $s \in \hat{s}$  such that  $eval(b, s) = False$ ,  $\hat{s}' = block(\hat{s}, b, s)$  is an abstract state satisfying  $\hat{s}' \preceq \hat{s}, s \in \hat{s}', eval(b, \hat{s}') = False$ . The goal of the operation is to compute a new abstract state by removing at least those states from  $\hat{s}$  which satisfy  $b$  (potentially others as well, when the domain is not granular enough) while making sure to keep  $s$ .

We need to be able to represent the abstract states as Boolean expressions in the sense that for each  $\hat{s} \in \hat{S}$  a  $b_{\hat{s}} \in \mathcal{B}_{\mathcal{V}}$  must exist such that  $\forall s \in S : eval(b_{\hat{s}}, s) = True \iff s \in \hat{s}$ .

A successful approach to implement abstraction-based model checking is using abstraction-refinement methods: starting from a very coarse abstraction and introducing further information as the verification progresses until we can prove satisfaction or violation of the requirement.

*d) Lazy abstraction:* The main idea behind lazy abstraction is on-demand refinement of the abstract states during abstract state space exploration. We selected the lazy abstraction method described in [7] to apply to MDP model checking.

It builds an *Abstract Reachability Graph (ARG)* with each node labeled by both a concrete state and an abstract state: the concrete state is assumed to be able to represent all other states in the abstract state with respect to the enabled actions until this is disproven. The abstract labels are originally very

coarse and are refined when needed. *Covering edges* are used to denote that the traces starting from a node cover all possible traces from another node – paths from the covered need not be explored. When an action is enabled in at least one element of the abstract label of a node, but not in the concrete label, the abstract label is *strengthened*: states where the action is enabled are blocked out of it. This operation can trigger other strengthenings.

If all enabled actions in all non-covered ARG nodes have been explored, the algorithm stops. The ARG overapproximates the concrete state space: for all reachable concrete states, there is a node whose abstract label contains it.

### III. APPLICATION OF THE LAZY ALGORITHM TO MDPs

In this section we adapt the lazy algorithm to analyzing a symbolic MDP given by a variable set  $\mathcal{V}$  and a command set  $\mathcal{C}_0$ . Given an error state formula  $\phi \in \mathcal{E}_{\mathcal{V}}$ , the goal of the analysis is to compute the probability of reaching a state  $s$  where  $eval(\phi, s) = True$ . We show that the abstraction is sound for safety properties in the sense that the error probability in the abstract model overapproximates the concrete probability.

We assume that an abstract domain  $(\hat{S}, \preceq, \alpha, \gamma)$  has been selected. The set of commands is extended with an error command:  $\mathcal{C} = \mathcal{C}_0 \cup \{(\phi, \delta_{id})\}$ , where  $id$  is an assignment that does not change any variable. This ensures that the finished ARG contains a node labeled with an error state exactly if an error state is reachable in the concrete state space.

The lazy abstraction algorithm explores the abstract state space by building an ARG: a tuple  $(N, E_T, E_C, L_c, L_a)$ , where  $N$  is a set of nodes,  $E_T \subseteq N \times \mathcal{C} \times \mathbb{D}(N)$  is a set of transition “edges” from nodes to node distributions labeled with commands,  $E_C \subseteq N \times N$  is a set of directed *covering edges*,  $L_c : N \rightarrow VAL_{\mathcal{V}}$  is the *concrete labeling function*,  $L_a : N \rightarrow \hat{S}$  is the *abstract labeling function*.

The ARG is *well-formed* if it satisfies the following criteria: **a1)** abstract label contains the concrete label:  $\forall n \in N : L_c(n) \in L_a(n)$ , **a2)** concrete label can represent the whole abstract label with respect to the enabled commands:  $\forall n \in N : \forall c \in \mathcal{C} : eval(g_c, L_c(n)) = False \implies eval(g_c, L_a(n)) = False$ , **b1)** commands labeling a transition edge are enabled in the concrete state of the source:  $\forall (n, c, \cdot) \in E_T : eval(g_c, L_c(n)) = True$ , **b2)** there is a probability-preserving bijection  $f$  between nodes at the end of a transition edge and the assignments of the command labeling the edge which respects the result of the assignment (further explained below):  $\forall (n, c, d_e) \in E_T : \exists (f : Supp(d_c) \longleftrightarrow Supp(d_e)) : \forall a \in Supp(d_c) : eval(a, L_c(n)) = L_c(f(a)) \wedge d_c(a) = d_e(f(a)) \wedge eval(a, L_a(n)) \preceq L_a(f(a))$ , **c1)** abstract label of covering node must contain concrete label of covered node:  $\forall (n, n') \in E_C : L_c(n) \in L_a(n')$ , **c2)** covering node must be at least as abstract as the covered node:  $\forall (n, n') \in E_C : L_a(n) \preceq L_a(n')$ . **c3)** covering node cannot be covered:  $\forall (n, n') \in E_C : \neg \exists (n'', n'') \in E_C$ .

An example ARG can be seen in Figure 1b. To make **b2)** easier to understand, let us see how it is satisfied in this example for the  $C_1$  edge of node  $n_0$ . Intuitively, it means that

the distribution at the end of the edge results from applying the command to the source node of the edge. The nodes at the end of this edge and their probabilities are  $n_1$  with 0.2 and  $n_2$  with 0.8. The assignments of this command are  $A_1$  assigning  $y+1$  to  $y$  and  $x$  to  $x$ , and  $A_2$  assigning  $x+1$  to  $x$  and  $y$  to  $y$ . The probability of  $A_1$  is 0.2, that of  $A_2$  is 0.8. The bijection  $f$  that makes **b2)** satisfied maps  $A_1$  to  $n_1$  and  $A_2$  to  $n_2$ . Considering first  $A_1$ , this mapping is probability preserving:  $d_c(A_1) = d_e(n_1) = 0.2$ , it respects the assignments for the concrete labels:  $eval(A_1, L_c(n_0)) = L_c(n_1)$ , and also for the abstract labels:  $eval(A_1, L_a(n_0)) \preceq L_a(n_1)$ . Each of these also holds for  $A_2$ . In the lazy abstraction algorithm, the abstract label of a node can be coarser than the exact result of applying the post operator to its ancestor as long as all other constraints are satisfied.

The algorithm starts with an ARG with a single node  $n_0, L_c(n_0) = val_0, L_a(n_0) = \top$ . We extend this to a well-formed ARG where all nodes are either covered or fully expanded using expansion, covering and strengthening operations. If a node  $n \in N$  is selected for expansion, we check for each  $c \in \mathcal{C}$  whether  $eval(g_c, L_c(n)) = True$ . If so, a new node  $n'_i$  is created for each  $a_i \in Supp(d_c)$  with  $L_c(n'_i) = eval(a_i, L_c(n)), L_a(n'_i) = \top$ , and a transition edge  $(n, c, d_e)$  is created such that  $d_e(n'_i) = d_c(a_i)$  for  $i = 1 \dots |Supp(d_c)|$ . For each newly created node  $n'$ , we check whether  $\exists n_c \neq n' \in N : L_c(n') \in L_a(n_c)$  such that  $n_c$  is not covered, and if so, a covering edge  $(n', n_c)$  is created for such an  $n_c$  and  $L_a(n')$  is strengthened. If this leads to a violation of well-formedness constraint **b2)**, then  $n$  has to be strengthened as well.

If  $eval(g_c, L_c(n)) = False$ , then we compute  $eval(g_c, L_a(n))$ . If *False*, we move on to the next command or node. However, if it is true or *Unknown*, then **a2)** is violated, and  $L_a(n)$  needs to be strengthened.

Whenever the value of  $L_a(n)$  is changed for some node  $n \in N$ , the well-formedness constraints can be violated. If **c1)** is violated, the problematic covering edge is simply removed from  $E_C$ . Other constraints can then be restored by *strengthening* the abstract labels of nodes related to  $n$ :  $L_a(n)$  is replaced by some  $\hat{s}'$  such that  $\hat{s}' \preceq L_a(n)$  and at least one well-formedness violation is eliminated.

If **a2)** is violated by a node  $n$  because of a command  $c$  that is enabled somewhere in  $L_a(n)$  but not in  $L_c(n)$ , a new abstract label can be computed as  $\hat{s}' = block(L_a(n), g_c, L_c(n))$ . Because of the contract of *block*,  $eval(g_c, \hat{s}') = False$ , so this command no longer causes a constraint violation.

If **c2)** is violated by some covering edge  $(n, n') \in E_C$ , but **c1)** still holds for these nodes, then the current  $L_a(n)$  must be replaced with  $\hat{s}'$  such that  $L_c(n) \in \hat{s}'$ ,  $\hat{s}' \preceq L_a(n')$  and  $\hat{s}' \preceq L_a(n)$  (referring to the current  $L_a$ ). By describing  $L_a(n')$  as a Boolean expression  $b_c$ , we can compute an appropriate  $\hat{s}'$  as  $block(L_a(n), \neg b_c, L_c(n))$ .

Now assume that **b2)** is violated by an edge  $(n, c, d)$ . Because of how the ARG is constructed, this means that a bijection  $f$  mentioned in the constraint could be constructed if we ignored the constraints on  $L_a$ , but the  $L_a$  part is violated by

some state. Let us construct such a bijection  $f$  ignoring the  $L_a$  part. There exists an  $a \in \text{Supp}(d_c)$  such that  $\text{eval}(a, L_a(n)) \not\preceq L_a(f(a))$ . By representing  $\text{eval}^{-1}(a, L_a(f(a)))$  as a Boolean expression  $b$  and changing  $L_a(n)$  to  $\text{block}(L_a(n), b, L_c(n))$ , the violation caused by the assignment  $a$  is eliminated. As the abstract label of  $n$  became finer, this might trigger another strengthening, which can propagate back up until the root node. Efficient implementations of the algorithm can use sequence interpolation to compute the strengthening of the whole path up to the root at once [7].

Strengthenings can create new violations triggering another strengthening, but all violations are eliminated after finite steps, and we continue expanding non-covered nodes.

The finished ARG can be treated as an MDP  $(N, \mathcal{C}, T_{ARG}, n_0)$ . Its transition function  $T_{ARG}$  is defined as follows. For a non-covered node  $n$ , a command  $c \in \mathcal{C}$  is enabled if there is an edge  $(n, c, d) \in E_T$  in the ARG, and  $T_{ARG}(n, c) = d$ . For a covered node  $n$ , let  $n'$  denote its covering node (so  $(n, n') \in E_C$ ). A command  $c \in \mathcal{C}$  is enabled in  $n$  iff it is enabled in  $n'$ , and  $T_{ARG}(n, c) = T_{ARG}(n', c)$ . As the covering node cannot be covered,  $T_{ARG}$  is a well-defined function. Intuitively, we merge covered nodes into their covering nodes.

We prove the soundness of this abstraction using the notion of *strong simulation* of MDPs, similarly to how the soundness of another MDP abstraction algorithm was proven in [8]. It was shown in [9], that strong simulation preserves the safety subset of Probabilistic Computation Tree Logic (PCTL) properties [9], [10], meaning that whenever a strong simulation relation exists from an MDP  $M_1$  to another MDP  $M_2$ , then the satisfaction of a safe PCTL property  $\phi$  for  $M_2$  implies the satisfaction of  $\phi$  for  $M_1$ . As probabilistic reachability properties are a subset of safe PCTL, the soundness of our algorithm can be proven by constructing a strong simulation relation  $\mathcal{R} \subseteq S \times N$  from the original MDP to the ARG.

Given two distributions  $d^1 \in \mathbb{D}(S^1)$ ,  $d^2 \in \mathbb{D}(S^2)$  and a relation  $\mathcal{R} \subseteq S^1 \times S^2$ ,  $\Delta : S^1 \times S^2 \rightarrow [0, 1]$  is a *weight function* if:  $\Delta(s^1, s^2) > 0$  implies  $s^1 \mathcal{R} s^2$ ,  $\forall s^1 \in S^1 : \sum_{s^2 \in S^2} \Delta(s^1, s^2) = d^1(s^1)$  and  $\forall s^2 \in S^2 : \sum_{s^1 \in S^1} \Delta(s^1, s^2) = d^2(s^2)$ . The existence of a weight function between  $d^1$  and  $d^2$  for a relation  $\mathcal{R}$  will be denoted  $d^1 \sqsubseteq_{\mathcal{R}} d^2$ . Given two MDPs  $(S^1, \text{Act}, T^1, s_0^1)$  and  $(S^2, \text{Act}, T^2, s_0^2)$ , a relation  $\mathcal{R} \subseteq S^1 \times S^2$  is a *strong simulation relation* if **i)**  $s_0^1 \mathcal{R} s_0^2$ , **ii)**  $\forall (s^1, s^2) \in \mathcal{R} : \forall a \in \text{Act}$  enabled in  $s^1 : a$  is enabled in  $s^2 \wedge T^1(s^1, a) \sqsubseteq_{\mathcal{R}} T^2(s^2, a)$ .

The relation that we use is given by  $\mathcal{R} = \{(s, n) | s \in L_a(n)\}$ .  $s_0 \mathcal{R} n_0$ , as  $L_c(n_0) = s_0$  implies  $s_0 \in L_a(n_0)$ , so the starting states are related, **i)** holds. For all  $(s, n) \in \mathcal{R}$  if  $n$  is not covered, then from  $s \in L_a(n)$ , **a2)** ( $L_c(n)$  represents the whole  $L_a(n)$ ) and the finishedness of the ARG (there is a transition edge labeled with  $c$  for each command enabled in  $L_c(n)$ ) follows that all commands enabled in  $s$  are also enabled in  $n$ . A similar reasoning holds if  $n$  is covered by  $n'$ , with the addition that  $s \in L_a(n')$  because of **c2)**. Let  $s, n$  and  $c$  be any fixed concrete state, ARG node and command respectively from now such that  $(s, n) \in \mathcal{R}$  and

$c \in \mathcal{C}$  is enabled in  $s$ . Now we need to show the existence of weight function  $\Delta$  between  $\text{eval}(d_c, s)$  and  $T_{ARG}(n, c)$ . There is an edge  $(n, c, d) \in E_T$  if  $n$  is not covered, and  $(n', c, d) \in E_T$  if  $n'$  covers  $n$ . Let  $f$  be a bijection mapping the assignments of  $c$  to  $\text{Supp}(d)$  that makes **b2)** true for this edge. Let  $s_i = \text{eval}(a_i, s)$  and  $d_i = d_c(a_i)$  for each assignment  $a_i$  of  $c$ . As  $s \in L_a(n)$ ,  $s_i \in \text{eval}(a_i, L_a(n))$ . If  $n$  is not covered, we have from **b2)** that  $\text{eval}(a_i, L_a(n)) \preceq L_a(f(a_i))$ , so  $s_i \in L_a(f(a_i))$ . If  $(n, n') \in E_C$ ,  $L_a(n) \preceq L_a(n')$  from **c2)** implies  $\text{eval}(a_i, L_a(n)) \preceq \text{eval}(a_i, L_a(n')) \preceq L_a(f(n))$ , so  $s_i \in L_a(f(n))$  in this case as well. This means that  $s_i \mathcal{R} f(a_i)$  for each  $i$ . Now let us set  $\Delta(s_i, f(a_i)) = d_i$  for each  $i$ , and let  $\Delta$  be zero everywhere else. As  $s_i \mathcal{R} f(a_i)$ , the first condition of weight functions is satisfied. The second and third conditions are satisfied as the only non-zero element of the sum in both cases is exactly  $d_i = \text{eval}(c, s)(s_i) = T_{ARG}(n, c)(n_i)$ . This proves the existence of a valid weight function, which completes the proof of  $\mathcal{R}$  being a strong simulation relation. From Thm. 19 in [9], it follows that if a probabilistic reachability property holds in the ARG, it also holds in the original MDP, which completes our proof. Figure 1c visualizes the relation.

As this relation is not a *bisimulation*, a problem with this abstraction is that the computed error probability on the ARG is only an overapproximation, and we have no control over how close it is to the concrete model. To make more precise error probability computation possible, we need to introduce an option to refine the ARG further on demand. We aim to address this issue in our future work.

#### IV. CONCLUSIONS

We have shown that the lazy abstraction scheme of [7] is applicable to symbolic MDPs and gave a formal proof of its soundness. We have shown a shortcoming of the direct adaptation in that it cannot be further refined. We plan to address this issue in the future by developing a variant based on stochastic games. This work is currently in its theoretical phase, implementation of the proposed algorithm is in progress in the THETA model checking framework.

#### REFERENCES

- [1] M. Kwiatkowska, G. Norman, and D. Parker, “Stochastic model checking,” in *SFM'07*, ser. LNCS, vol. 4486. Springer, 2007, pp. 220–270.
- [2] M. Kattenbelt, M. Kwiatkowska *et al.*, “Abstraction refinement for probabilistic software,” in *VMCAI'09*, 2009.
- [3] H. Hermanns, B. Wachter, and L. Zhang, “Probabilistic CEGAR,” in *CAV'08*, ser. LNCS, vol. 5123. Springer, 2008, pp. 162–175.
- [4] P. Cousot and M. Monerau, “Probabilistic abstract interpretation,” in *ESOP'12*, ser. LNCS, vol. 7211. Springer, 2012, pp. 169–193.
- [5] M. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of probabilistic real-time systems,” in *CAV'11*, 2011.
- [6] C. E. Budde, C. Dehnert *et al.*, “Jani: Quantitative model and tool interaction,” in *TACAS'17*, ser. LNCS, vol. 10206, 2017, pp. 151–168.
- [7] T. Tóth and I. Majzik, “Configurable verification of timed automata with discrete variables,” *Acta Informatica* 59, pp. 1–35, 2022.
- [8] B. Wachter, L. Zhang, and H. Hermanns, “Probabilistic model checking modulo theories,” in *QEST'07*. IEEE Computer Society, 2007, pp. 129–140.
- [9] R. Segala and N. A. Lynch, “Probabilistic simulations for probabilistic processes,” in *CONCUR '94*, ser. Lecture Notes in Computer Science, B. Jonsson and J. Parrow, Eds., vol. 836. Springer, 1994, pp. 481–496.
- [10] C. Baier, J. Katoen *et al.*, “Comparative branching-time semantics for Markov chains,” *Inf. Comput.*, vol. 200, no. 2, pp. 149–214, 2005.