Acta Cybernetica **26** (2024) 671–711.

Towards Abstraction-based Probabilistic Program Analysis[*]

Dániel Szekeres[ab] and István Majzik[ac]

Abstract

Probabilistic programs that can represent both probabilistic and non-deterministic choices are useful for creating reliability models of complex safety-critical systems that interact with humans or external systems. Such models are often quite complex, so their analysis can be hindered by state-space explosion. One common approach to deal with this problem is the application of abstraction techniques. We present improvements for an abstraction-refinement scheme for the analysis of probabilistic programs, aiming to improve the scalability of the scheme by adapting modern techniques from qualitative software model checking, and make the analysis result more reliable using better convergence checks. We implemented and evaluated the improvements in our Theta model checking framework.

Keywords: probabilistic systems, stochastic games, abstraction, reliability analysis

1 Introduction

Probabilistic programs are models specified using a software source code syntax, extended with special statements that sample values from probability distributions. Their similarity to program source code makes them easy to use to describe complex probabilistic systems, especially for engineers already familiar with programming.

Probabilistic programs are able to describe both probabilistic and proper non-deterministic behavior, which is useful for creating models for reliability analysis of complex safety-critical systems that interact with humans or external systems with unknown behavior. In such models, the probabilistic behavior described using sampling statements comes from the uncertainty inherent in the physical environment

[*]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.

[a]Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

[b]E-mail: szekeres@mit.bme.hu, ORCID: 0000-0002-2912-028X

[c]E-mail: majzik@mit.bme.hu, ORCID: 0000-0002-1184-2882

DOI: 10.14232/actacyb.298287

of the system and the uncertain failures of hardware components of the system. Proper non-determinism comes from the behavior of humans and external systems interacting with the system under analysis, for which we do not have statistical information to base probabilistic modeling on.

There are two main use cases for formulating reliability models as probabilistic programs:

- The expressivity makes it comfortable to use programs for describing the reliability model when the engineers are already familiar with standard programming languages. The advantages of program code compared to other formalisms (like the compositional language of the PRISM tool [23]) are especially apparent when the *process* described by the model is more complex than the *compositional structure* of the model.

- The program-based approach can be advantageous when a modeling formalism for which standard code generation is already available is extended with probabilistic semantics. As an example, our Gamma tool [14] supports modeling by statecharts and offers program code generation from these models. Extending the statecharts with stochastic concepts for dependability modeling leads to the generation of probabilistic programs that fit for analysis.

The analysis of such models is often hindered by *state-space explosion*: even when the program describing the model is relatively short, its actual state space can become intractable large. Abstraction is a widespread approach to counteract this problem: instead of analyzing the original model, an *abstract model* is created by ignoring some information present in the original. As the abstract model is constructed as a conservative approximation of the original, proving that a property is satisfied by the abstract model also proves satisfaction for the original model.

The most successful implementation of the idea of abstraction is a scheme called *Counterexample-Guided Abstraction Refinement (CEGAR)* [10], which automatically finds the appropriate level of abstraction. It starts with a very coarse model, analyzes it, and in case the property is violated, it generates an abstract counterexample. This counterexample is checked to decide whether it appeared only because of ignoring some information (spurious counterexample), or it actually proves violation in the original model (concretizable counterexample). In the spurious case, a refinement of the abstraction is computed based on the counterexample, and the CEGAR loop starts over.

In this work, we focus on the game-based framework proposed in [25] and extended in [18], and present improvements to it. We chose this approach because of its relative success and its ability to provide both upper and lower bounds for the analysed numeric property, giving a metric for measuring how precise the current abstraction is with respect to the property we are trying to check. [13] extended this scheme to domains used in classic abstract interpretation while also switching to an abstract interpretation framework instead of the CEGAR loop. We stayed with the original CEGAR-like refinement loop scheme. Most of the enhancements we implemented are based on our experiences in qualitative software model checking [28, 15].

The game-based abstraction scheme is applicable to any analysis question that can be formulated as computation of expected total rewards on a Markov Decision Process, which is the underlying low-level formalism we use for the semantics probabilistic programs in this work. Our proposed changes do not constrain the generality of this scheme any further, but in this paper, we will only focus on checking probabilistic reachability properties, i.e. computing the probability of reaching a specific set of target states.

Our contributions are the following:

1. We improved the scalability of the scheme by adapting modern techniques from qualitative software model checking:
    a) Adapted a version of *Large Block Encoding* to the probabilistic case.
    b) Adapted the usage of *inexact transition functions* and formally proved that the abstraction scheme remains sound when employing them.
    c) Extended the possible abstract domains with the *explicit value domain* (also known as visible variables domain).
2. We tackled the problem of convergence checking for the abstract model using *bounded value iteration*, and gave an example where using standard value iteration can lead to problems with the refinement loop.
3. We performed *numerical measurements* to answer research questions related to how our improvements and extensions affect performance.

These enhancements make the probabilistic analysis of a larger set of complex safety-critical systems possible. As these techniques are heuristic, they do not promise to enhance performance on *every* reliability model used in practice, but they give new options, which perform better on a set of practical models, some of which are not analysable without them.

The application of bounded value iteration is a semi-exception to this: the overall analysis result will always be more reliable, than with standard value iteration, while the potential performance gain from performing better refinements matters only for some (non-empty, as we show in our measurements) subset of analysis cases, similarly to the other enhancements.

We implemented enhancements these as a probabilistic extension of our open-source Theta model checking framework[1]. The implementation only supports probabilistic reachability properties for now, and for this reason, the benchmarks we performed are also constrained to this analysis task.

Section 2 introduces the necessary mathematical and formal modeling background along with the game-based abstraction refinement scheme. Section 3 describes the main idea of bounded value iteration and shows why a reliable convergence check is especially important in the abstraction refinement scheme. Section

[1]https://github.com/ftsrg/theta

4 describes the technique of Large Block Encoding in general and how we incorporated it for probabilistic program analysis. Section 5 describes the Explicit Value abstract domain and elaborates on our adaptation method. Section 6 introduces the general idea of inexact abstract transition functions, and proposes a way to apply them in the game-based scheme. Section 7 describes our numerical experiments and the related research questions, along with plots of the results and answers to the research questions. Conclusions and proposals for future work can be found at the end of the paper. The appendix contains proofs for the soundness theorems stated in Section 6.

1.1 Related work

Here, we present the general relations between our contributions and previous works. The formalisms used in this work are presented in Section 2. Details regarding the background algorithms can be found in their respective sections (Sections 3, 4, 5 and 6).

Several different abstraction approaches have been proposed for probabilistic systems [12, 21, 27, 17, 9, 25, 29]. Our work builds on the game-based abstraction refinement algorithm presented in [25, 18].

Cartesian abstraction and explicit abstraction have already been applied successfully to non-probabilistic model checking in [4] and [7, 15] respectively. We build on these results and adapt the corresponding algorithms to the analysis of probabilistic systems. We incorporate ideas from [19] to prove the soundness of using these domains (the proofs can be found in the appendix). [13] extended the same abstraction refinement algorithm to numerical abstract domains used in abstract interpretation, similarly to our extensions. While they switched to an abstract interpretation approach, we stayed with the original abstraction refinement loop.

Large block encoding has been used in non-probabilistic model checking for example in [6]. We use a simpler version of it to make sure that the resulting model satisfies the constraints required by the abstraction refinement algorithm of [18].

It has been shown in [3], that employing the classic stopping criterion of value iteration can lead to arbitrarily wrong results. We build on this result to show that this stopping criterion can even lead to performance degradation when the MDP is analysed using abstraction refinement. [3, 24] proposed a bounded version of value iteration to solve the problem of convergence checking, which [20] extended to stochastic games. We incorporated this algorithm in our abstraction refinement implementation, and evaluated its impact on the overall performance of the analysis.

2 General Background and Formalisms

This section introduces the necessary mathematical and formal modeling background and describes the abstraction framework that the paper builds upon.

2.1 Probabilistic Programs

Probabilistic programs are syntactically similar to standard structured program code with basic features, like ANSI C or lisp, but are extended with features making them suitable for the description of random processes and probabilistic models.

The most important such feature is *sampling*, which draws a sample from a given probability distribution - syntactically similar to calling pseudorandom number generators, but with different semantics. When analyzing a probabilistic program, a sampling statement is treated as a random variable with the given distribution, and can be either discrete or continuous with any support, even an infinite one. Probabilistic program interpreters and analyzers can then simulate sample trajectories, compute moments of expressions over program variables, etc.

In our current work, we use a probabilistic extension of ANSI C to specify probabilistic programs. The extension means that there are some predefined functions with special semantics: these functions are interpreted as proper probabilistic sampling when analyzed.

Only discrete distributions with finite support are in the scope of this work. Handling infinite and continuous distributions are possible future extensions planned to be implemented in the probabilistic module of our Theta framework.

Some modern probabilistic programming constructs, like higher-order probabilistic functions and observe statements are out of scope for this paper, our current work focuses on a basic set of features that cover the most important aspects.

Example. The code of a simple running example we will use throughout this paper can be seen in Listing 1. The program has two integer variables, x and y, both initially set to 0. When we show different kinds of state spaces for this program, we will restrict their domain to the set $\{0, 1, 2, 3\}$, instead of the complete set of integers.

```
main() {
    int x = 0, y = 0;
    while(y <= x) {
        //probabilistic assignment with a biased coin-flip
        x = x + coin(0.4);
        if(x > 2) {
            //havoc: assignment to a non-deterministic value
            y = *;
        }
    }
    assert(!(x < 3));
}
```

Listing 1: Probabilistic C code of a simple probabilistic program

These variables are repeatedly changed in a loop while the value of y is less than or equal to the value of x. In this loop, we first "flip a biased coin" by calling a sampling function, and increment x by the result (1 with probability 0.4, 0 with the remaining probability 0.6). If x is greater than 2, we set y to a non-deterministic value from its domain. In software model checking, this action is called *havoc*, and we denote it as $y = *$ here.

After exiting the loop, we *assert* that x is less than 3. Assert calls lead to an error state if the expression they contain evaluates to false. When analysing this probabilistic program, we will be interested in the probability of violating this assertion.

2.2 The Probabilistic Control Flow Automaton formalism

The control flow automaton (CFA) formalism is a widespread formal model used for the qualitative analysis of software source code. Probabilistic programs have similar structure to classical source code, but some functions are equipped with special probabilistic semantics. To enable the analysis of such programs, we use an extension of the CFA formalism: the *Probabilistic Control Flow Automaton (PCFA)*.

The definition we give for this formalism is similar to the definition of probabilistic programs in [18], but we would like to keep the notion of the *program itself*, and the *analysis formalism* used for model checking separate, as is standard in classical software model checking. Treating it as an extension of the classical CFA formalism also makes the relationship between techniques used in this paper and techniques of classical software model checking clearer.

Definition 1 (Probabilistic Control Flow Automaton)**.** *A* Probabilistic control flow automaton (PCFA) *is a tuple* $P = ((L, E), l_{init}, Var, v_{init}, S, Stmt)$*, where:*

- (L, E) *is a finite directed control-flow graph; the nodes of the graph are called the* locations *of the PCFA, and the edges represent successor relationships between them*
- $l_{init} \in L$ *is the initial location*
- Var *is a finite set of* variables
- $v_{init} \in U_{Var}$ *is the* initial valuation *(described below)*
- S *is a set of* statements *(described below)*
- $Stmt : E \to S$ *is a function mapping each edge to an associated statement*

The definition uses a set of *variables* to represent the data state of the program. Each variable v has a type, or equivalently, a countable *domain* denoted by $Dom(v)$, which is the set of values it can take. The set of *valuations* U_{Var} for the variable set Var consists of functions mapping every variable $v \in Var$ to an element of their domain $Dom(v)$.

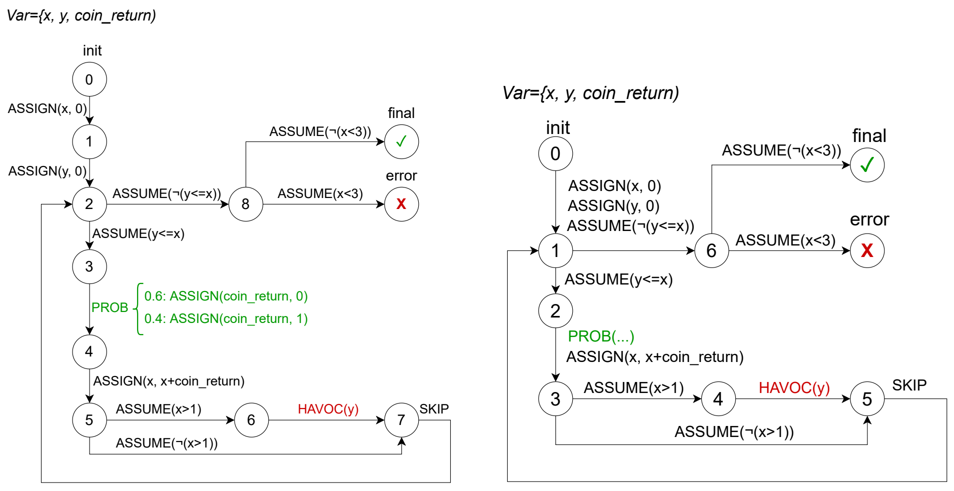
Although this definition allows only a single initial valuation, non-deterministic variable initialization can be modeled by using `HAVOC` statements at the beginning of the program.

A *statement* $s : U_{Var} \rightharpoonup 2^{\mathbb{D}(U_{Var})}$ is a probabilistic valuation transformer: given the current valuation, it returns a set of distributions of resulting valuations after applying the statement. It is a partial function, which is how we handle conditional statements (`ASSUME`s, see below): a statement is *enabled* by a valuation v, if it is defined at v. The result is a *set of valuations*, to make the representation of non-deterministic statements possible. In this paper, this is only used in the case of `HAVOC` statements, which result from calling functions with unknown results (e.g. asking for user input).

We use the following types of statements:

- `ASSIGN`$(v \in Var, e : U_{Var} \to Dom(v))$: The result is a singleton set of a dirac distribution of a single resulting valuation. This valuation maps the variable v to the result of applying the function e to the original valuation, and maps all other variables to their value in the original valuation. Used for assignment statements, where the right-hand side can be any (type-compatible) expression.
- `ASSUME`$(e : U_{Var} \to \{\top, \bot\})$: The partial function is defined exactly for those valuations that the function e maps to $\top$. For these valuations, it is basically an identity function, the returned set is a singleton set of a dirac distribution of the original valuation. Used for conditions in the program, like `if` statements and loop condition checks.
- `HAVOC`$(v \in Var)$: The resulting valuation set is $\{\, dirac(U_{orig}^{[v \longrightarrow v_i]}) \mid v_i \in Dom(v) \,\}$, where $U_{orig}^{[v \longrightarrow v_i]}$ is a valuation that maps v to v_i and any other variable to its value in the original valuation. This statement is used for assigning a totally non-deterministic value to a variable from its domain. Combined with an assume statement, it can be used to assign non-deterministic values to a variable from a constrained set of its full domain.
- `PROB`$(d : \mathbb{D}(S_{det}))$: This is the only statement that produces a distribution that is not a dirac distribution. It is parameterized by a distribution over other *deterministic* statements – S_{det} is the set of statements that map only to singleton sets. This is needed as we defined the result of applying a statement as a set of distributions, not distributions of sets. The resulting set has a single distribution in it. This distribution over valuation sets can be derived from the parameter distribution by computing the resulting valuation set from each statement with non-zero probability in the parameter distribution, and assigning the same probability to the resulting valuation as the statement. If multiple substatements lead to the same valuation, their probabilities are summed.
- `SKIP`: Enabled in any valuation, and does not change the valuation. Used for simplifying control flow structures like loops.

In most cases the code under analysis contains `assert` calls, which check the truth of a Boolean expression. If it evaluates to true, the program continues normally. If the expression is false, an error is raised. When transforming the code to a (P)CFA, `assert` calls are transformed to two `ASSUME` edges, one of them leading to the next location in the normal flow of the program, and the other one entering an



(a) Single-block encoding (b) Large-block encoding

Figure 1: PCFA of the running example

error location. Checking the probability of failing an assertion thus can be reduced to computing the probability of entering an error location.

We will describe the formal semantics of PCFA models denotationally by deriving a Markov Decision Process from them later, but for intuitive understanding, we describe the operational semantics of a PCFA can be described semi-formally here:

- The model starts in the location l_{init} with the valuation v_{init}.
- In each step, an outgoing edge e is selected non-deterministically from the outgoing edges of the current location such that $Stmt(e)$ is enabled by the current valuation v_{curr}.
- An element of the set $Stmt(e)(v_{curr})$ is selected non-deterministically, which is a distribution over the possible next valuations.
- A valuation v_{next} is sampled from this distribution. The location of the model in the next step is the location at the end of the selected edge, and the valuation in the next step is v_{next}.
- If the current location has no outgoing edges, the model stops.

Precise formal denotational semantics can be found for example in [5].

Example. The PCFA of our running example can be seen in Figure 1a. Although we did not need it in the code, an auxiliary variable *coin_return* is added to keep track of the value returned from the *coin* function call. This is needed because we need to separate the sampling call and the assignment to two edges.

Deciding whether the body of the loop should be executed or skipped is done using two edges with opposing `ASSUME` statements. The `PROB` statement corresponds to the call to the special sampling function *coin* in the code. This statement sets the coin_return variable to either 1 with probability 0.4 or to 0 with probability 0.6. After this, an `ASSIGN` increments the value of the variable x by the value of coin_return.

The if statement of the code is converted to two edges with opposing `ASSUME`s, similarly to the while loop. Non-deterministic assignment is done through an edge with a `HAVOC` statement. The body of the loop ends with a `SKIP` statement leading back to the head of the loop. The assert call in the code is converted to two `ASSUME`s, one leading to the (non-erroneous) final state, the other leading to the error state.

Figure 1b shows another possible PCFA for the same program, where large block encoding is used to create a more compact model: a single transition here applies multiple statements in succession. This will be explained in detail in Section 4.

2.3 Markov Decision Processes

Definition 2 (Markov Decision Process)**.** *A* Markov Decision Process (MDP) *is a tuple* $M = (S, s_{init}, A, Av, \delta)$ *where*

- S *is a finite set of* states
- $s_{init} \in S$ *is the initial state*
- A *is a finite set of actions*
- $Av : S \to 2^A$ *is a function mapping states to the set of* available actions *in the state*
- $\delta : S \times A \rightharpoonup \mathbb{D}(S)$ *is the transition function of the MDP. It is a partial function, which must be defined for every* (s, a) *where* $a \in Av(s)$*. For such a pair, it gives the distribution of successor states after taking the action* a *in state* s*.*

The transition notation $s \xrightarrow{a} \mu$ is used for $\delta(s, a) = \mu$. We will also use the notation $\delta(s, a, s') = \delta(s, a)(s')$ for the probability of transitioning from s to s' when taking the action a.

A lot of analysis tasks on MDPs use the notion of reward functions on the MDP. A *state reward function* on an MDP is a function $\mathbf{r} : S \to \mathbb{R}_{\geqslant 0}$ that assigns rewards to each state in the MDP.

A *strategy* on an MDP is a function $S^* \to \mathbb{D}(A)$ that assigns a distribution over actions to all possible state sequences. A *memoryless* strategy depends only on the last state. A *deterministic* strategy uses only dirac distributions.

In this paper, we will focus on *probabilistic reachability properties*: computing the probability of reaching a given set of *target states*. Such properties can be formulated using rewards: the reward function assigns 1 to the target states, and 0 to all others. The target states must be made absorbing. The expected value of the accumulated reward until absorption with a fixed strategy gives the probability of reaching one of the target states. Computing optimal strategies thus can give the maximal or minimal probability of hitting an error if the targets are the error states of the system.

2.4 MDP semantics of probabilistic programs

The denotational semantics of probabilistic programs can be defined by deriving an MDP from the probabilistic control flow automaton of the program.

Definition 3 (MDP of a PCFA)**.** *The MDP describing the semantics of the PCFA* $P = ((L, E), l_{init}, Var, v_{init}, S, Stmt)$ *is* $M = (S, s_{init}, A, Av, \delta)$*, where*

- $\mathcal{S} = L \times U_{Var}$
- $s_{init} = (l_{init}, v_{init})$
- $\forall e = (l, l') \in E, v \in U_{var} : (v \textit{ enables } Stmt(e)) \iff (\forall d \in Stmt(e)(v) : \exists! a \in A : a \in Av((l, v)) \wedge \delta((l, v), a) = join(l', d))$,
  where the distribution $join(l, d) \in \mathbb{D}(L \times U_{var})$ *is defined by* $join(l, d)((l, v)) = d(v)$*, and for any other location, it is 0. This condition defines* A, Av, δ*.*

A state can be identified by selecting the location of the program and the value of each variable. Therefore, the state-space of the derived MDP is the Cartesian product of the set of program locations and valuations of the variable set. The initial state of the MDP is derived from the initial location and initial valuation of the PCFA.

The edges of the PCFA are transformed to actions of the MDP. For each state $s = (l, v_0, ..., v_{|Var|})$ of the MDP, we take the edges whose statement is enabled by the valuation of the state. For each such edge, we compute the possible distributions resulting from applying the statement of the edge. Let l' denote the ending location of the edge. For each computed distribution, we assign a new action to s leading to that distribution, with l' added as the location part of the states.

Example. The MDP obtained from the semantics of our running example can be seen in Figure 2. For the sake of readability, we used the PCFA with large-block encoding (Figure 1b) to reduce the number of states. The value of each variable is unknown at the beginning, so the initial data state can be anything. To make the figure easier to read, the possible initial states are merged and the unknown values are indicated by question marks. As a statement cannot lead to both probabilistic and (proper) non-deterministic choice in the next state, we have either only a single action optionally with multiple states in the next state distribution or multiple actions, each leading to dirac distributions in each state. Green nodes highlight the states where a probabilistic decision is performed, the red node highlights the non-deterministic decision (multiple possible actions).

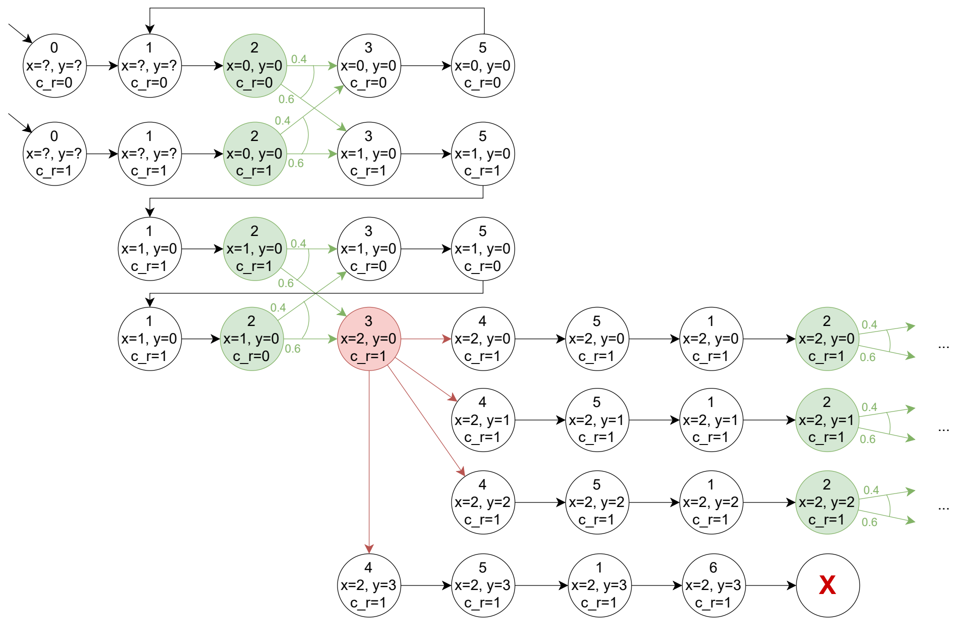


Figure 2: MDP of the running example corresponding to the PCFA in Figure 1b. c_r stands for the auxiliary variable coin_return.

2.5 Stochastic Games

Stochastic games are an extension of MDPs where instead of a single agent, two different players make decisions. This allows representing two separate kinds of non-determinism, which are resolved according to two separate strategies with different objectives. The relevance of Stochastic Games to this work is that they are used as abstract models for abstraction-based analysis of probabilistic programs.

Definition 4. *A* Turn-Based Stochastic Two-Player Game *is a tuple* $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$, *where:*

- *S is a finite set of* states, *partitioned into the sets of Player 1 (S_1) and Player 2 (S_2) states*
- *$s_{init} \in V$ is the initial state*
- *A is the set of* actions
- *$Av : S \to 2^A$ is a function mapping each state to the set of available actions.*
- *$\delta : S \times A \rightharpoonup \mathbb{D}(S)$ is the transition function of the game. It is a partial function, which must be defined for every (s, a) where $a \in Av(s)$. For such a pair, it gives the distribution of successor states after taking the action a in state s.*

Definition 5 (Play). *A* play ω *in a two-player simple stochastic game* $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$ *is a (possibly infinite) sequence of state-action pairs* $\omega = (s_1^1, a_1^1)(s_1^2, a_2^1)(s_2^1, a_2^1)(s_2^2, a_2^2)\dots$ *such that*

- $s_1^1 = s_{init}$
- $\forall s_j^i : a_j^i \in Av(s_j^i)$
- $\forall s_2^i : \delta(s_1^i, a_1^i, s_2^i) > 0$
- $\forall s_1^i : \delta(s_2^{i-1}, a_2^{i-1}, s_1^i) > 0$

In most applications, the goal of one player is to *minimize* the (expected) reward gathered during a play according to a specified reward function, while the other player aims to *maximize* it. A *reward function* assigns rewards to states or transitions. When the game enters a state or takes a transition, the reward associated with that state or transition is collected.

Probabilistic reachability properties can be formulated as a total reward computation problem the same way as for MDPs: the reward function assigns 1 to the target states, and 0 to all others, and the target states must be made absorbing.

Throughout this work, the shortened name *Stochastic Game (SG)* will refer to turn-based stochastic two-player games.

A special case is when the goals of the two players coincide, which is equivalent to an MDP. Here, we will consider only those cases when the players have opposite goals, as the cooperative case can be analysed as an MDP.

Definition 6 (Strategy). *A Player 1 (resp. Player 2) strategy in a stochastic game* $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$ *is a function* $Strat : S^*S_1 \to \mathbb{D}(A)$ *(resp.* $S^*S_2 \to \mathbb{D}(A)$*) from the set of finite plays ending in a Player 1 (resp. Player 2) state to the set of distributions over Player 2 (resp. Player 1) states, such that* $\forall p \in S^*S_1 (resp. \forall p \in S^*S_2) : Strat(p)(a) > 0.0 \implies a \in Av(end(p))$.

For probabilistic reachability properties, there always exists a pair of memoryless deterministic Player 1 and 2 strategies that are optimal among all strategies [11]. This means that we can focus only on this finite subset of strategies, making synthesis of optimal strategies much easier than in the general case. We only need to determine a single optimal action to take for each state, not a distribution over actions for every possible state sequence.

If a fixed strategy is considered for each player, we can equip the set of plays on the game with a probability measure derived from the strategies and the probabilistic transition function of the game. Informally, the probability of a play is product of the probability of each transition taken in the play. The probability of a transition is the probability that the player takes an action that can lead to the given transition according to the fixed strategy multiplied by the probability that the action results in the given transition. A more precise formal specification of the probability measure can be found e.g. in [2, Chapter 10].

The reachability probability of a set of target states is the probability measure of the set of all plays that contain at least one target state.

From now on we will assume that the game is played by a maximizer and minimizer (referring to their goals with respect to the reward function), and refer to their state sets as S_{min} and S_{max} respectively.

Definition 7 (Value Function for reachability properties)**.** *The* value function $V : S \to \mathbb{R}$ *for a game* $G = (S = S_{max} \uplus S_{min}, s_{init}, A, Av, \delta)$ *with respect to a set of target states* $T \subset S$ *is the least solution to the Bellman equations:*

$$V(s) = \begin{cases} \max_{a \in Av(s)} V(s, a) & \textit{if } s \in S_{max} \\ \min_{a \in Av(s)} V(s, a) & \textit{if } s \in S_{min} \\ 1 & \textit{if } s \in T \\ 0 & \textit{if } T \textit{ is unreachable from } s \end{cases}$$

The importance of the value function comes from the fact that a strategy is optimal if and only if it always chooses an optimal action with respect to the value function.

The value of a state is equal to the expected accumulated reward if the corresponding optimal strategies are used. If the reward function is derived from a reachability property, the value of a state is equal to the probability that a target state is reached from that state if the players follow the optimal strategies, which is exactly what we aim to calculate for the initial state.

2.6 Abstraction-based analysis

The aim of abstraction is to make the analysis of a model or program tractable by analyzing an abstract model with a smaller state-space than the original model under analysis. This is done by discarding information which is deemed not useful for proving or disproving the target property.

The resulting state-space is called the *abstract state-space*, while the original one is called the *concrete state-space.* As a new state space is created, the transition relation of the system is also lifted to this state space, resulting in the *abstract transition function.*

Discarding information must be performed in a conservative way: the abstract model must be built in such a way that the property being checked is satisfied by the abstract model *only if* it is satisfied by the original system.

The reverse need not be true. If the target property is disproved in the abstraction, we can try to *concretize* the proof of non-satisfaction, mapping the abstract counterexample back to the concrete state-space, thereby proving that the original system does not satisfy the target property either. Obviously, this can only be done if the original system does not satisfy it.

Different approaches for retaining only a fraction of the original information is captured by different *abstract domains.* Abstract domains define what kind of information can be contained in an abstract state and specify the correspondence between concrete and abstract states.

The notion of *abstraction precision* is used to differentiate between retaining different parts of the originally available information.

Abstract domains can have a convenient property called *disjointness*, which means that for a given precision, exactly one abstract state corresponds to each concrete state. In other words, the abstract states considered as sets of concrete states are disjoint for a fixed precision. This makes soundness proofs often much simpler than the general case.

2.7 Predicate Abstraction

Predicate abstraction can be applied to systems described symbolically using a set of state variables. It creates an abstract state-space by defining a set of *predicates*: Boolean expressions over these variables. The precision of the abstraction is the set of predicates.

As the concrete states assign exactly one value to each program variable, each predicate can be evaluated unambiguously for a concrete state. The abstract state corresponding to the concrete state is the Boolean vector resulting from evaluating all of the predicates in the precision against the variable valuation of the concrete state.

Although we can always evaluate the predicates for the concrete states, abstraction is often used exactly because exploration of the concrete state-space is intractable. Instead of exploring the full concrete state-space and mapping the concrete states, abstraction-based analysis techniques explore only the abstract state-space.

Exploration of the abstract state space needs a method for determining which actions are available in a given abstract state, and generating the possible next states after applying one of these actions. This is generally done by solving some form of a SAT (satisfiability) problem. As software involve not only Boolean variables, but other types like integers as well, their operations also have to be encoded into the SAT problem. One option for this is to use bitwise encoding of all types and use standard SAT solvers. This was the approach used in [18].

However, modern model checkers mostly employ *Satisfiability Modulo Theories (SMT)* solvers to handle non-Boolean variables [1]. In this case, the logic is augmented with theories that can handle the operations of variable types. In our implementation, we rely on SMT encoding instead of bitwise SAT.

Example. If in the current state we know that the predicate $x < 3$ is true and $y = 0$ is false, and we apply the statement $ASSIGN(x, x+1)$, the SMT problem for the next state computation is:

$$\underline{x_0 < 3 \wedge \neg(y_0 = 0)} \wedge \underline{x_1 = x_0 + 1} \wedge \underline{(x_1 < 3 \iff a_1) \wedge (y_0 = 0 \iff a_2)}$$

The first underlined part corresponds to the current state, the second underlined part to the statement applied, and the third part is a representation of the next state.

The Boolean literals a_1 and a_2 are called *activation literals.* All satisfying models of the above expression are computed, and the next abstract states are derived from the values of a_1 and a_2 in each model: the truth value of each predicate is set to the value of the corresponding activation literal.

When the model under analysis is a control flow automaton, abstraction is commonly applied to the data variables, and the location is also added to the abstract states (i.e. it is always precisely tracked).

2.8 Game-based Abstraction Refinement

The game-based abstraction refinement framework for probabilistic systems has been proposed in [25], and its application to probabilistic programs described in [18].

The main idea of this abstraction scheme is to use a stochastic game as the abstract model of an MDP, which allows introducing a second kind of non-deterministic choice which is resolved using a different objective than the one in the original model. This way we can introduce an "abstraction player" (referred to as Player A from now on), whose responsibility will be intuitively to resolve the choice of which concrete state we are in when we know only the abstract state (which represents a set of possible concrete states). The other player will be referred to as Player C, as its responsibility is resolving the original non-determinism already present in the *concrete* model.

By setting the objective of Player A to minimizing the probability of reaching the target state set, we can compute a lower bound for the original probability. By maximizing it instead, we get an upper bound. By computing both bounds, we get not only a conservative approximation of the error probability, but also a measure of how precise the current abstraction is with respect to the property of interest.

Refinement is performed based on the difference between the minimizing and the maximizing strategies of Player A. A state is refinable if the minimizing and the maximizing strategies choose a different action in it. The new precision is computed such that it prevents making this choice.

Two different strategies were proposed in [19] for choosing the state to base the refinement on. We can choose the coarsest refinable state, for which the difference between the upper and the lower abstraction values is the highest (hoping that this will have the largest effect on the precision of values), or choose the refinable state nearest to the initial state (hoping that this will make the abstraction precise enough early in the state space exploration process).

After a refinable state has been chosen, a new predicate is computed to eliminate the choice of the abstraction player in this state using weakest precondition computation.

When analysing a PCFA, it is possible to use a different precision for abstracting the data state in each location. This is called *local* precision, while using a single one for the whole model is called *global* precision. When local precision is used, the new predicate that was computed is added only to the location corresponding to the chosen state to refine. However, propagating the newly chosen predicate to other locations using weakest precondition computation can lead to a much lower number of refinement steps needed. [19] proposed two different strategies for this propagation, one based on the explored abstract state space, the other based directly on the CFA model.

2.8.1 Computing the abstraction game

Here we will briefly describe how the abstraction game is built in the case of predicate abstraction applied to probabilistic programs. An example step of building the game can be seen in Figure 3 for each statement type. A more detailed description and the SMT formulae used can be found in [19].

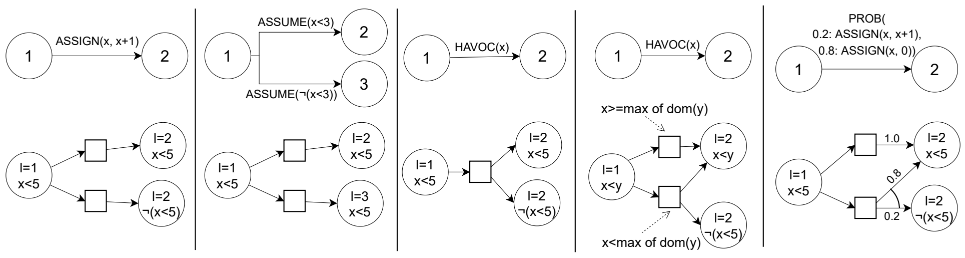


Figure 3: Example PCFA components and the resulting game parts. Circles represent Player A nodes, rectangles represent Player C nodes.

The construction method assumes the following constraints to be true for the model, which are always satisfied by PCFA models created using single-block encoding, and are also satisfied by our large-block encoding implementation:

- If multiple outgoing edges are present in a location, then only one of them can be enabled in any valuation. Because of this, the only source of non-determinism in the model can be `HAVOC` statements.
- If there is a `PROB` or `HAVOC` statement present on an outgoing edge of a location, then there must be no other outgoing edges.

The game used as abstract model alternates between Player A and Player C nodes. Player A nodes correspond to abstract states. For each Player A node (starting with the node corresponding to the initial abstract state), we need to compute groups of next states resulting from applying a statement available in the current state.

A-nodes are labeled by abstract states, while C-nodes get their identity from the set of A-node distributions it can choose from. When building the abstraction game, we take an A-node that has not been expanded yet, take a statement available in it, and compute the result of the statement.

`ASSIGN` and `ASSUME` statements When the statement to apply is an `ASSIGN` or an `ASSUME` statement, only the abstraction player has power over the resulting state.

The result of an assignment and the satisfaction of the assumption depends on the current values of the variables, but we might not know all variables that affect the result exactly with the currently used precision. Thus the abstraction player can select these values arbitrarily as long as they do not contradict what we know from the currently tracked predicates. As there is no concrete non-determinism involved with these statement types, Player C has no power in this state.

`PROB` statements Similarly to non-probabilistic assignments and assumes, only Player A has any power over choosing the result of a `PROB` statement (as we disallowed non-deterministic statements inside probabilistic statements in the definition). In this case however, we have to compute a set of *distributions* over abstract states from which the abstraction player can choose, not stand-alone states. Because of this, the successor states must be computed *in groups*, all elements of a possible distribution at a time. For the example in Figure 3, we need the following query:

$$x_0 < 5 \wedge (x_1^{(1)} = x_0 + 1 \wedge (x_1^{(1)} < 5 \iff a_1^{(1)})) \wedge (x_1^{(2)} = 0 \wedge (x_1^{(2)} < 5 \iff a_1^{(2)}))$$

We are interested in all satisfying models that differ in at least one of $a_1^{(1)}$ and $a_1^{(2)}$, because these models encode different next state distributions. The satisfying models in this example are $\{a_1^{(1)}, a_1^{(2)}\}$, representing the dirac distribution $\{1.0 : (l = 2, x < 5)\}$, and $\{a_1^{(1)}, \neg a_1^{(2)}\}$ representing the distribution $\{0.2 : (l = 2, x < 5), 0.2 : (l = 2, \neg(x < 5))\}$.

We cannot simply evaluate the possible successors for the statements independently: the results of applying the statements depend on each other through the unknown variable values. Computing the possible results of the statements, and then composing them into a set of distributions by taking a Cartesian product would lead to "hallucinating" some successor distributions over the abstract states that cannot result from applying the `PROB` statement to any concrete state. This would be conceptually similar to the inexactness that Cartesian abstraction introduces, described in Section 6.

General `HAVOC` statements Handling `HAVOC`s is the most problematic part, as in the general case, both players can have power over the resulting abstract state. To compute the set of possible next states and group them appropriately, we have to introduce a term in the SMT query for each possible new value of the variable modified by the `HAVOC`, similarly to what we do for each substatement of a `PROB`. This is intractable if the domain of a variable is large.

Simplifiable `HAVOC` statements However, if we can split the information tracked by the current precision so that the predicates one partition depends *only* on the variable modified by the `HAVOC`, and the ones in the other partition *do not* depend on that variable at all, then the computation can be simplified: in this case, only Player C has any actual power over next state selection: it can decide how the first partition changes, while the second partition must stay the same as it was. Thus we know that only one group has to be computed, making a standard flat list of successor states sufficient. For this, a standard SMT query for the successors is enough.

3 Incorporating Bounded Value Iteration

Value iteration (VI) computes an optimal strategy by iteratively approximating the value function of the game. The resulting strategy is derived from the value function using the fact that any strategy that always chooses an optimal action with respect to the value function is optimal.

The algorithm iteratively refines a lower bound approximation to the value function using the Bellman equations. The initial approximation is set to 1 for every target state, and 0 everywhere else.

The approximate value function computed during value iteration converges to the real value function only asymptotically. Because of this, unlike for strategy iteration, it is not guaranteed that the algorithm terminates in a finite number of steps. However, the value function often becomes good enough to compute the optimal strategy in much less time than it would take strategy iteration to converge. Value iteration is often preferred in practice for this reason.

The problem of determining when the value function is good enough still remains. A very basic, but often used approach is to stop value iteration when the difference between two consecutive value function approximations becomes smaller than a predefined threshold. This stopping criterion cannot actually give any guarantees about the optimality of the resulting strategy: as long as the change of the value function is not exactly zero in an iteration, the chosen transition can change in any state, potentially leading to a vastly different mean reward [3].

Bounded Value Iteration (BVI) [24, 3] is a modified version of value iteration originally proposed for Markov Decision Processes. It has been extended to Stochastic Games in [20]. BVI computes both a lower and an upper approximation for the value function. The reason for this is that the difference between the upper and the lower bound can be used as stopping criterion. As the two approximations are constructed in such a way that the real value function is known to be between them, the error is no longer totally unknown, we have an upper bound for it, and can thus be controlled. This leads to a guaranteed ε-optimal strategy as a result. Making the upper approximation converge on stochastic games is non-trivial - for a detailed explanation of the algorithm, see [20].

Having more control over the optimality of strategies computed using the approximate value function has two advantages in the game-based abstraction scheme: apart from the general advantage of making the end result more precise, the refinement step is also enhanced by having a more reliable value function.

We constructed a simple example, where the problem of ineffective refinement can be easily seen to highlight the importance of more reliable convergence checks when computing values in the game-based abstraction refinement scheme.

Based on the model given in [3] as an example for when the standard VI convergence check can lead to a wrong result, we can construct an example for any ε, where a seemingly coarse abstract state is actually perfectly precise: the values of a state with maximizing and minimizing abstraction choices coincide, the difference between lower and upper abstraction comes only from the imprecision of value computation.

Example. An example where the unreliable result of standard value iteration can be problematic for refinement can be seen in Figure 4. The only non-determinism in the abstract state space is an abstraction choice in s_0. As there is no concrete non-determinism, the Player C nodes between the Player A nodes have been omitted from the figure. The target states are c_n and s_1.

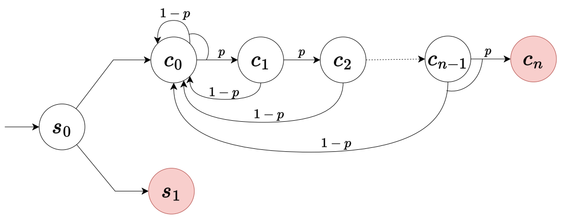


Figure 4: Example model for refinement problem with standard value iteration

The upper part of the model is the chain from [3]. The parameters n and p can be chosen appropriately for any ε such that if standard value iteration is stopped when the latest change was smaller than ε, then the final approximation of the probability of reaching a target from s_0 is less than $\frac{5}{8}$, while the real probability is 1.

The lower part of the model is simply an instant target state, so both the approximate computed probability and the real probability of reaching a target by choosing the lower action are 1.

If the approximate values computed with standard VI are used to compute the refinement, the abstract state s_0 seems to be quite coarse: the computed value difference between the maximizing and the minimizing action is greater than $\frac{3}{8}$. In reality, though, s_0 is perfectly precise with respect to the property of interest, as both actions eventually lead to a target state with probability 1. This can lead to sub-optimal refinement if the refinable state selection strategy is set to *coarsest*: if the given example is only a part of the state space, an actually coarsely abstracted part would be much more important to refine.

If qualitative pre-analysis is used to determine almost sure reachability precisely before running VI, the example model can be modified to "leak" some probability from the chain before reaching the target.

We have also implemented the topological version of VI and BVI [22]: this approach splits the game into the strongly connected components of its edge relation graph, and computes the value function for each component in reverse topological order. This change can often speed up the convergence of value iteration for structurally problematic models, as it enforces propagation of already converged values instead of propagating non-converged values through the whole state-space.

4 Adapting Large-block Encoding

When creating a (P)CFA from the source code of a computer program, the most obvious approach is to create an edge for every atomic statement in the code, as seen in the example in Figure 1a. A disadvantage of this conversion is that it introduces a high number of locations, leading to a wastefully large state-space. This version is called *single-block encoding (SBE)*.

Another possibility is *large-block encoding (LBE)*. There are several possible versions of this, but the general idea is that an edge is associated with multiple statements. By allowing a single edge to hold multiple statements, we can reduce the number of edges and locations, making the graph of the (P)CFA much smaller. This can both speed up the analysis, and make the results easier to interpret. We can introduce composite statements to be able to do this while staying with our original definition of PCFA.

There are multiple variants of composite statements with different semantics, but our current work uses only *sequence statements (SEQ)* representing a sequential composition of other statements (the statements given in an ordered list are executed one after another according to their order in the list).

A SEQ statement is enabled by a valuation if and only if its first substatement is enabled by the valuation, and each substatement is enabled by the result of transforming the valuation through all substatements before it.

An example PCFA with large-block encoding can be seen in Figure 1b.

The idea of large block encoding can be implemented in several different ways. For example, the authors in [6] used it for standard qualitative software model checking and merged both sequential and parallel (non-deterministic and if-else) edge combinations into single edges.

However, in the probabilistic abstraction case, doing this would prove problematic: using a non-deterministic statement to combine guarded choices would make the edge look like a concrete non-deterministic choice, while it is actually an abstraction choice, or it could merge the choices of the two players into a single edge making building the game much more complex depending on the implementation. Another difference from the qualitative case is that the game construction algorithm of [18] needs all locations of the PCFA to be either choice, non-deterministic or probabilistic, clearly separating ASSUME statements from HAVOCs and PROB.

Accordingly, we decided to implement only sequential LBE for the probabilistic case, and defer the exploration of possibilities for merging parallel edges to future work, and we split the LBE at PROB and HAVOC statements, making those have their own edges separated from the sequences between them. This sequential LBE is implemented as a simple preprocessing step on the PCFA: $l_1 \xrightarrow{stmt_1} l_2 \xrightarrow{stmt_2} l_3$ patterns, where l_2 has no other edges are merged to $l_1 \xrightarrow{SEQ\{stmt_1, stmt_2\}} l_3$. These statements can be converted to SMT expressions by converting each substatement into an expression with the appropriate indexing, and taking the conjunction of these expressions.

5 Adapting Explicit Abstraction

Explicit abstraction (also called explicit-value abstraction and visible variables abstraction) [7] chooses for each variable if it is visible or not. This means that for each variable, its value is either tracked exactly, or it is not tracked at all. The precision of the abstraction is the list of tracked variables. Each abstract state assigns an exact value to each variable tracked according to the precision used.

To use the explicit domain, we needed to adapt the game construction algorithm of [18] to it. Constructing the abstraction game was originally formulated only for predicate abstraction, so we needed to apply the main ideas of the grouped successor computation to the explicit domain. Classical software model checking, which explicit abstraction was previously used for, did not need *grouped* successor computation in the abstract model construction, as the abstraction-induced non-determinism did not need to be handled separately from the original (concrete) non-determinism. Because of this, implementing the game-based abstraction scheme with explicit abstraction involved determining how the flat successor computation must be modified for the grouped computation needs.

One advantage of the explicit domain is that instead of always relying on expensive SMT solver calls to compute the successor states, we can often simply evaluate the program statements after all variables currently tracked have been substituted with their values in the current abstract state. SMT calls are needed only if an important variable is missing, and even then, the SMT query often contains much fewer variables than without substituting the known ones.

For deterministic (`ASSIGN` and `ASSUME`) and `HAVOC` statements, explicit abstraction can be used the same way as in the qualitative case, as we compute only a flat list of next states, and assign the choice between them either to Player A (for deterministic statements) or Player C (for `HAVOC` statement). In the case of `HAVOC`, this is correct because the simplification described earlier is always applicable when using explicit abstraction: a `HAVOC` statement always affects only one variable, and no matter what it was, Player C can set it to any value. For `PROB` statements, we simply use the same approach as in predicate abstraction: the state is converted to a Boolean expression as a conjunction of equalities for the known variable values, and then we can solve the same SMT problem as in the predicate case. When LBE is used, `SEQ` statements are handled the same ways as deterministic statements, as they can contain only `ASSIGN`s and `ASSUME`s in our version.

Refinement is also similar to the predicate case, first discovering new predicates using weakest precondition computation for splitting the state selected for refinement and propagating if needed. The states are converted to Boolean expressions for this computation by taking a conjunction of equalities for the known variable values. Then, instead of adding the newly discovered predicates to the precision, we add all variables contained in them - just like in the qualitative version of explicit abstraction.

6 Inexact Abstract Transition Functions for Game-based Abstraction

The exact computation of the abstract transition relation is often computationally very expensive, sometimes infeasible. Because of this, inexact transition functions are often used in standard software model checking that conservatively overapproximate the original relation. Two examples of this are Cartesian abstraction in the case of predicate abstraction and limiting the number of next states enumerated for a single successor state set computation in the case of explicit abstraction.

Here, we will first describe these and their application in the game-based abstraction scheme for probabilistic programs, then state two theorems related to the correctness of using them. The proofs of these theorems are relegated to the appendix.

The approximation introduced by these techniques can be considered a second layer of abstraction. Instead of abstracting the state-space itself, it abstracts the transition function, as it discards some information about the original relation.

Inexact transition functions need more flexible abstract domains than the ones described before, which allow on-demand omission of information during exploration. These domains do not have the disjointness property: for a given precision, multiple abstract states can correspond to the same concrete state, as the precision is not strictly respected by all abstract states. Thus, the correctness of the abstraction scheme must be reevaluated when using inexact transition functions.

We first describe the two inexact abstract transition functions that we implemented, Cartesian abstraction [4] for predicate abstraction and limited enumeration [15] for explicit abstraction, along with how we adapted them to the game-based abstraction refinement scheme. After that, we elaborate on the correctness of their usage.

6.1 Cartesian abstraction

In Section 2, we described the standard version of predicate abstraction, where each predicate in the current precision must be stated to either hold or not in each abstract state. A more general version of predicate abstraction uses three-valued logic for the predicates: true, false and unknown/any.

The generalized version makes it possible to use (conservative) approximations of the exact abstract transition relation which leave predicates unknown instead of branching in both directions during exploration.

One such approximate computation of the abstract transition relation is *Cartesian abstraction*. Instead of evaluating the SMT problem with the whole Boolean vector of the next state, it computes for each predicate whether it or its negation is implied by applying a statement to the current abstract state. If none of these is implied, the predicate is set to unknown in the subsequent abstract state. If the predicate itself is implied, it is set to true, if the negation is implied, the predicate is set to false. Implying both means that the conjunction of the current state expression and the statement expression is unsatisfiable, leading to an empty set of next states (as the statement is not enabled by the current state).

This is an over-approximation of the exact abstract transition function. Consider the case for example when two predicates must be the opposite of each other in the subsequent abstract state, but we do not know which one of them is true. Because of this, both are set to unknown. This abstract state contains all of the possible concrete states, but also contain states where both of the predicates are true or both are false.

Our proposed adaptation of Cartesian abstraction is applying it only partially in the game construction. As Cartesian abstraction modifies the method which we use for the next state computation, we needed to decide how to use that in the game construction.

For deterministic statements and simplified `HAVOC` statement computation, Cartesian abstraction can be used the same way as in the qualitative case, as we compute only a flat list of next states, and assign the choice between them either to Player A (for deterministic statements) or Player C (for `HAVOC` statement when simplification is applicable).

For `PROB` statements and general `HAVOC` statements, we cannot break the grouped SMT problem into atomic implications without introducing a high number of spurious distribution choices (it would still be a conservative abstraction, but a very imprecise one). Because of this, we propose to use Cartesian abstraction in the probabilistic case only partially, and use the precise computation for `PROB` and non-simplifiable `HAVOC` statements.

As most probabilistic programs have only a small amount of `PROB` statements, and `HAVOC` statements are simplifiable in most cases, Cartesian abstraction can still be much more efficient than the precise abstract transition relation.

Another possibility would be to compute an implication for each predicate for all next states in one SMT problem. In this case, instead of computing if the negated or ponated version is implied, we would need to check if any *combination* of ponated and negated versions of the given predicate in each element of the next state group is implied. This is tractable for `PROB`s with a small number of substatements (like `coin()`), but the number of SMT problems for each predicate blows up exponentially with the number of substatements, so we decided against this option.

Investigating other ways to apply the idea of Cartesian abstraction to the next state computations used in the abstraction game construction is part of our future plans, as the grouped computation parts could also benefit from the idea, but care has to be taken to not make the abstraction either unsound or too coarse.

6.2 Explicit abstraction with limited enumeration

The version of explicit abstraction described earlier has the disjointness property. However, there is a more general version of explicit abstraction, which is much more useful: in an abstract state, a variable which is tracked according to the precision can also be set to $\top$, meaning that the value is unknown in that state. This is useful for example for `HAVOC` statements, where the abstract state space could be infinite, or at least intractably large.

In this generalized domain, the abstract state space exploration can set the value of a variable to unknown when there would be too many possible values to evaluate. It is often possible to encounter an `ASSUME` statement or `ASSIGN` statement somewhere after the `HAVOC`, which constrains the possible values of the variable to a set of manageable size again.

The technique of limited enumeration for explicit abstraction means that we set a limit for how many possible successor states we are willing to compute in a single successor set computation. If the number of states would exceed this limit, we merge them into a single state by selecting all the variables that have different values in the possible next states, and set their value to $\top$, even though they should be tracked according to the precision. Unlike in the predicate case, explicit abstraction can be used only for a very small subset of programs without this generalization.

Using explicit abstraction with limited enumeration in the game-based abstraction refinement scheme is quite straightforward.

Similarly to the unlimited explicit case, when computing `ASSIGN`, `ASSUME` and `HAVOC` statements, the successor computation is done the same way as in the qualitative case, no adaptation is needed, as a non-grouped list of successors is sufficient for the construction. In the case of `HAVOC`, although Player C has full power in theory, limited enumeration will lead to Player A having full power over the value of the variable in practice if its domain is larger than the enumeration limit.

`PROB` statements are computed similarly to the unlimited explicit case, but because of the enumeration limit, resulting states might have to be merged. If the states that would be merged are in the same distribution, we can merge them in the distribution by replacing them with a single next state with original probabilities summed. If the enumeration that must be limited is in the different distribution choices, then we can instead merge them into a single distribution, where the variable is set to $\top$ in the next state for the corresponding substatement which caused the enumeration exceeding the limit.

6.3 Correctness

Here, we analyze the correctness of applying these techniques in the game-based scheme.

For an abstraction-based analysis scheme to result in a correct verdict about the analyzed property, the construction used for the abstract model must be *sound*, meaning that whenever the property is provable for the abstract model, it is true in the original model. In the context of game-based abstract analysis of stochastic models, this means that analyzing the game with the abstraction player aiming to minimize/maximize the objective results in valid lower and upper bounds respectively.

The original soundness proof in [25] assumed that the abstract domain used for the abstraction has the disjointness property. However, this does not allow the usage of generalized predicate and explicit abstraction, which Cartesian abstracion and limited enumeration depend on. [13] proved that the approach can be extended to non-disjoint domains. However, the context in that work was abstract interpretation with widening, and they used a slightly different construction for the abstraction game. Because of this, a new proof of soundness is necessary for the adaptation of Cartesian abstraction and explicit abstraction with limited enumeration described above.

Here, we only state the soundness theorems formally, and the proofs themselves are relegated to the appendix.

Theorem 1 (Soundness of Cartesian abstraction)**.** *Given a PCFA P with a set of target locations L_T, optimal strategies on the game abstraction of its MDP semantics using Cartesian abstraction to compute the abstract transition relation gives lower and upper bounds on the reachability probability of L_T. The optimal strategies are understood w.r.t. the reward function that assigns* 1 *to states with a location in L_T and* 0 *to all other states.*

Theorem 2 (Soundness of limited enumeration)**.** *Given a PCFA P with a set of target locations L_T, optimal strategies on the game abstraction of its MDP semantics using explicit abstraction with limited enumeration to compute the abstract transition relation gives lower and upper bounds on the reachability probability of L_T. The optimal strategies are understood w.r.t. the reward function that assigns* 1 *to states with a location in L_T and* 0 *to all other states.*

These correctness proofs make it possible to use Cartesian abstraction and explicit abstraction with limited enumeration in the game-based abstraction refinement scheme, guaranteeing that the maximizing and minimizing strategies lead to upper and lower approximations of the concrete value function respectively, which result in a correct final verdict. Keep in mind, that these results only state *soundness*, not *completeness*: using inexact transition functions without any other modifications to the overall scheme does not lead to a complete decision procedure, the verification might get stuck in the refinement loop, as the information that would be needed for more precise abstraction might be discarded by the transition function itself, which we do not refine.

7 Implementation and Numerical Experiments

We performed numerical experiments to evaluate our modifications: using large block encoding (LBE) in the PCFA model, using the explicit abstract domain, applying inexact abstract transition functions (Cartesian abstraction and limited enumeration), and enhancing the reliability of strategy computation using bounded value iteration (BVI). We used models from [18] which are standard benchmark model in this area. Some of the models are direct implementations of probabilistic protocols, while others are standard stochastic analysis benchmark models converted from PRISM models to probabilistic C code.

These models are scalable using scaling parameters, which makes it possible to analyze how the analysis approach scales with model size. Multiple properties to check are given for each model.

We used only a subset of the benchmark models, because the C frontend of Theta does not support multiple compilation units yet - this feature is work in progress, and further measurements are planned to be performed in the future. Applying the GCC preprocessor to create a single compilation unit was enough for some models originally given as multiple source files, but not all of them.

Our aim with the numerical experiments was to answer the following research questions:

RQ1. How does using large block encoding affect the running time of the analysis?

RQ2. How does using BVI instead of standard VI affect the running time? Is the impact on the running time sufficiently small to be an acceptable cost for higher reliability?

RQ3. How does the running time with the explicit domain differ from the predicate domain? What is the effect of using limited enumeration?

RQ4. How does applying Cartesian abstraction affect the running time?

RQ5. How prevalent is the problem of the refinement loop getting stuck with inexact transition functions on practical models?

The measurements were performed with our prototype implementation in the Theta model-checking framework. The following aspects of the analysis are configurable: abstract domain (including abstract next state computation method), refinable state selection strategy, precision locality (local: each PCFA location has its own precision, global: a single global precision is used), predicate propagation strategy (only applicable with local precision), stochastic game solver (VI, topological VI, BVI, topological BVI), SBE/LBE. Local precision (and because of this, refinement propagation) is not implemented for explicit abstraction yet. For enumeration limits in the explicit domain, we used ∞, 1, and 2.

We measured the performance of all possible configurations on all input models. Analyzing the effect of precision locality, refinement propagation strategies and refinable state selection strategies was not among the goals of these experiments. Therefore, we merged those configurations in the analysis results that differed only in these parameters by taking the smallest running time among them to reduce clutter in the plots. This results in an analysis where these parameters are always assumed to be chosen optimally.

The benchmarks were executed using *BenchExec* [8], the official tool used for the Software Verification Competition (SV-COMP) [2], providing reliable measurements

[2]`https://sv-comp.sosy-lab.org/2022/`

on resources. The experiment was executed on virtual machines running Ubuntu 20.04.2 LTS with Java OpenJDK 11.0.13. There was a memory limit of 15GB and a CPU core limit of 6 cores set. A timeout of 6 minutes was set for each input-configuration pair.

Figures 5, 6 and 7 show our measurement results with different groupings, aiming to make the effect of different features observable on each plot. The horizontal axis shows the value of the scaling parameter, while the vertical axis shows running time of the analysis. Each subplot has its own y scale in seconds, as the running time can differ greatly depending on inputs and configuration, so having a common scale would make data points with shorter running times incomparable.

Data points with 360s running time are timeouts, as that was the time limit.

The results provided the following answers for the research questions:

RQ1: Effect of LBE In general, LBE increased performance of the analysis. 240 input-configuration pairs succeeded before timing out with LBE, and only 154 input-configuration pairs without it. The difference can be easily seen in Figures 5 and 6 by comparing the blue (SBE) and red (LBE) data points, and in Figure 7 by comparing the number of not timed out data points in the two facet rows. LBE was beneficial regardless of the other configuration parameters. On the herman_P3 input, it even meant a difference between timing out or not.

RQ2: Effect of BVI The running time of BVI was close to the running time of standard VI, when the best refinable state selection strategy was chosen. This can

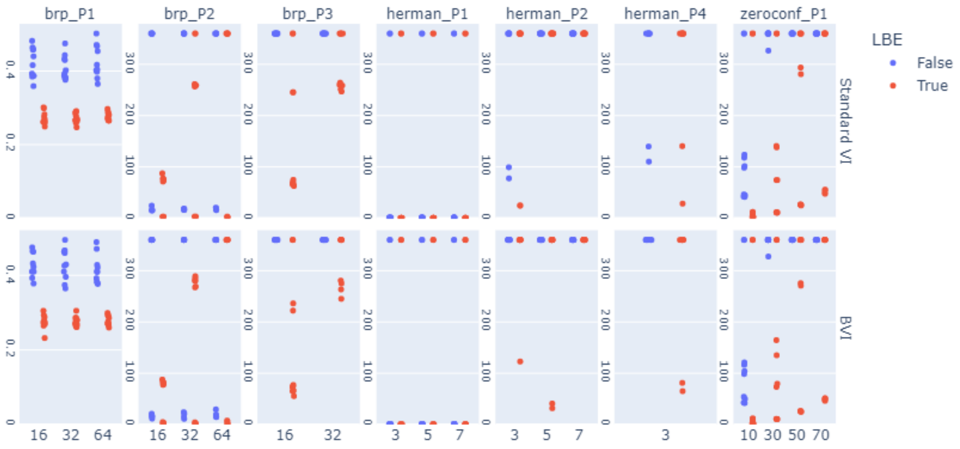


Figure 5: Strip plot showing the measured running times in seconds (note the different scaling in case of brp_P1) grouped by the input (model and analyzed property together) and the application of BVI. Color shows whether LBE was on (red) or off (blue)

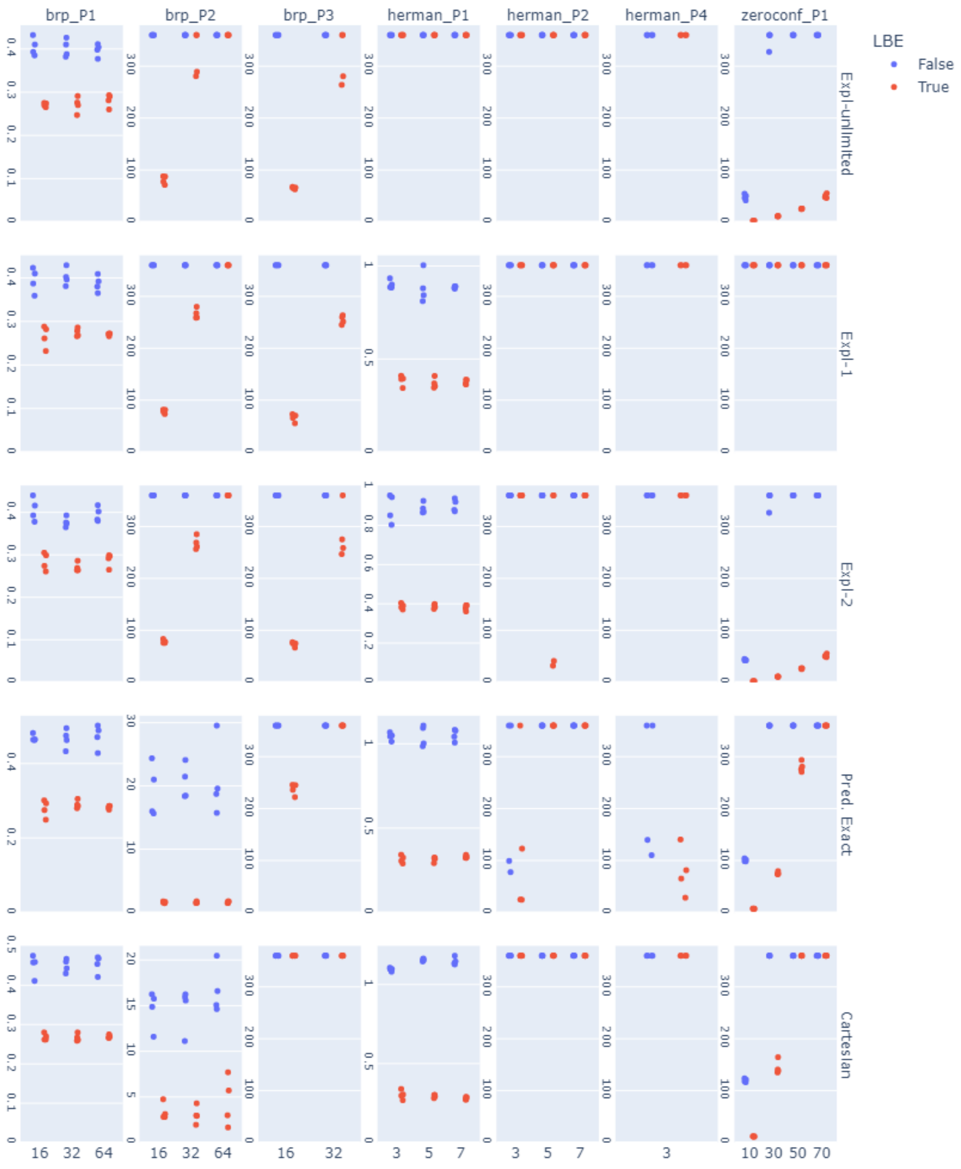


Figure 6: Strip plot showing the measured running times in seconds grouped by the input (model and analyzed property together) and the domain used (including the next state computation method). Color shows whether LBE was on (red) or off (blue)

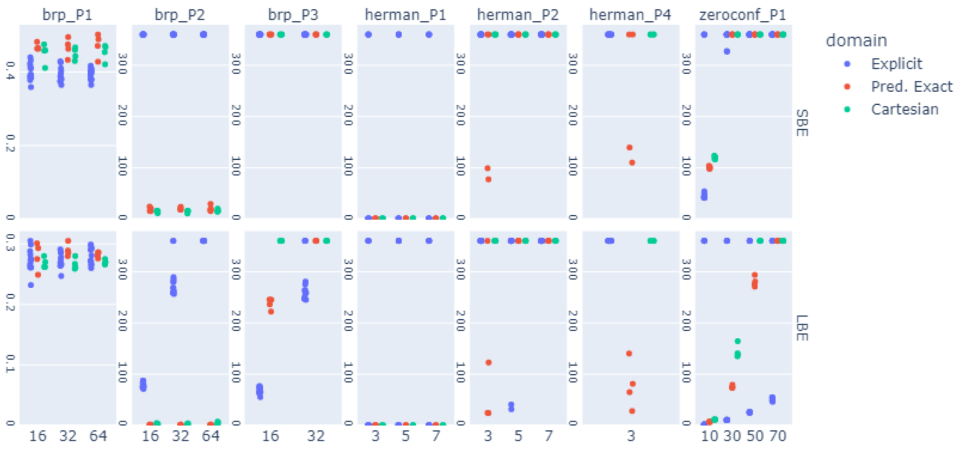


Figure 7: Strip plot showing the measured running times in seconds grouped by the input (model and analyzed property together) and whether LBE was used or not. Color shows which domain was used.

be seen most clearly in Figure 5, by comparing the two facet rows. We observed though that with other refinable state selection choices, BVI can be unacceptably slow compared to standard VI.

RQ3: Efficiency of the explicit domain Explicit abstraction performed better than predicate abstraction on some models. Neither is clearly better than the other, but having both option increases the number of models we can analyze. The unlimited version can often be problematic because of the infinite number of possible next states, and limit 1 often gets stuck, but explicit abstraction with enumeration limit 2 was able to analyze the most models without timing out among all the domain configurations. The different domain configurations can be compared on Figure 6 by comparing the facet rows (but the different y-axis scales might make exact comparison a bit harder), and in Figure 7 by comparing the differently colored data points. In the latter figure, we decided to not distinguish between the limits in the color to not clutter the plot with too many different colors.

RQ4: Effect of Cartesian abstraction Whenever Cartesian abstraction did not get stuck, it was often faster than exact predicate abstraction, but not significantly. In some cases, it was even slower and, unfortunately, the refinement loop did become stuck in a lot of cases. Because of this, these measurements did not lead to any significant answer for this research question. Cartesian abstraction can be compared to the other options in Figures 6 and 7.

RQ5: Refinement getting stuck Inexact next state computations were very prone to getting stuck in the refinement loop, because they dropped the information that would have been needed for refining based on the selected state.

Explicit abstraction with limit 2 is an exception when combined with LBE: it performed well on a lot of models, and it was able to solve the most analysis tasks overall among all domains. More precisely, if we count the number of solved inputs for each domain disregarding all other configuration parameters, Explicit abstraction with limit 2 was able to solve the highest number of tasks without timing out. With this limit, Boolean variables are always tracked exactly, while other variables are abstracted when multiple values are possible. This limit is also able to not merge probabilistic choices with two branches, while limit 1 will merge them, basically getting rid of probabilistic choices in the abstract model.

The problem of getting stuck might be mitigated by different refinable state selection approaches, or by on-demand refinement of the post operator itself. Further research is needed in this area.

8 Conclusions and Future Work

In this work, we set out to (1) improve the scalability of game-based abstraction of probabilistic programs by adapting modern techniques from qualitative software model checking, and to (2) tackle the problem of convergence checking for the abstract model.

Regarding (1) we can conclude the following based on our numerical evaluation:

- Introducing the explicit abstraction and limited enumeration options made the analysis of more models possible.
- Using large block encoding generally resulted in faster analysis.
- Cartesian abstraction and limited enumeration often get stuck in the refinement loop, so more sophisticated techniques are needed to make them usable.

Regarding (2), we can say that applying BVI does not incur a significant increase in running time, making it worth to use the more reliable version instead of standard VI. Performance of BVI was highly dependent on the refinable state selection strategy. We plan to investigate this phenomenon in detail in the future. We also plan to implement optimistic value iteration [16] and sound value iteration [26] adapted to stochastic games as abstract model solvers to compare them with BVI in the game-abstraction context.

Building on the conclusions we have drawn from these experiments, we plan to make inexact post operators more stable by introducing partial switching to exact computation as a refinement option.

References

[1] Armando, A., Mantovani, J., and Platania, L. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009. DOI: 10.1007/11691617_9.

[2] Baier, C. and Katoen, J.-P. *Principles of model checking*. MIT Press, 2008.

[3] Baier, C., Klein, J., Leuschner, L., Parker, D., and Wunderlich, S. Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In *International Conference on Computer Aided Verification*, pages 160–180. Springer, 2017. DOI: 10.1007/978-3-319-63387-9_8.

[4] Ball, T., Podelski, A., and Rajamani, S. K. Boolean and Cartesian abstraction for model checking C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283. Springer, 2001. DOI: 10.1007/s10009-002-0095-0.

[5] Barthe, G., Katoen, J.-P., and Silva, A. *Foundations of Probabilistic Programming*. Cambridge University Press, 2020. DOI: 10.1017/9781108770750.

[6] Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M. E., University, S. F., and Sebastiani, R. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design*, pages 25–32, 2009. DOI: 10.1109/FMCAD.2009.5351147.

[7] Beyer, D. and Löwe, S. Explicit-state software model checking based on CEGAR and interpolation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 146–162. Springer, 2013. DOI: 10.1007/978-3-642-37057-1_11.

[8] Beyer, D., Löwe, S., and Wendler, P. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019. DOI: 10.1007/s10009-017-0469-y.

[9] Chadha, R. and Viswanathan, M. A counterexample-guided abstraction-refinement framework for Markov decision processes. *ACM Transactions on Computational Logic (TOCL)*, 12(1):1–49, 2010. DOI: 10.1145/1838552.1838553.

[10] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003. DOI: 10.1145/876638.876643.

[11] Condon, A. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992. DOI: 10.1016/0890-5401(92)90048-K.

[12] De Alfaro, L. and Roy, P. Magnifying-lens abstraction for Markov decision processes. In *International Conference on Computer Aided Verification*, pages 325–338. Springer, 2007. DOI: 10.1007/978-3-540-73368-3_38.

[13] Esparza, J. and Gaiser, A. Probabilistic abstractions with arbitrary domains. In *International Static Analysis Symposium*, pages 334–350. Springer, 2011. DOI: 10.1007/978-3-642-23702-7_25.

[14] Graics, B., Molnár, V., Vörös, A., Majzik, I., and Varró, D. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling*, 19(6):1483–1517, 2020. DOI: 10.1007/s10270-020-00806-5.

[15] Hajdu, Á. and Micskei, Z. Efficient strategies for CEGAR-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, 2020. DOI: 10.1007/s10817-019-09535-x.

[16] Hartmanns, A. and Kaminski, B. L. Optimistic Value Iteration. In *International Conference on Computer Aided Verification*, pages 488–511. Springer, 2020. DOI: 10.1007/978-3-030-53291-8_26.

[17] Hermanns, H., Wachter, B., and Zhang, L. Probabilistic CEGAR. In *International Conference on Computer Aided Verification*, pages 162–175. Springer, 2008. DOI: 10.1007/978-3-540-70545-1_16.

[18] Kattenbelt, M., Kwiatkowska, M., Norman, G., and Parker, D. Abstraction refinement for probabilistic software. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 182–197. Springer, 2009. DOI: 10.1007/978-3-540-93900-9_17.

[19] Kattenbelt, M. A. *Automated quantitative software verification.* PhD thesis, University of Oxford, 2010.

[20] Kelmendi, E., Krämer, J., Křetínskỳ, J., and Weininger, M. Value iteration for simple stochastic games: Stopping criterion and learning algorithm. In *International Conference on Computer Aided Verification*, pages 623–642. Springer, 2018. DOI: 10.1007/978-3-319-96145-3_36.

[21] Komuravelli, A., Păsăreanu, C. S., and Clarke, E. M. Assume-guarantee abstraction refinement for probabilistic systems. In *International Conference on Computer Aided Verification*, pages 310–326. Springer, 2012. DOI: 10.1007/978-3-642-31424-7_25.

[22] Křetínskỳ, J., Ramneantu, E., Slivinskiy, A., and Weininger, M. Comparison of algorithms for simple stochastic games. *Information and Computation*, page 104885, 2022. DOI: 10.1016/j.ic.2022.104885.

[23] Kwiatkowska, M., Norman, G., and Parker, D. PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G. and Qadeer, S., editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, Volume 6806 of *LNCS*, pages 585–591. Springer, 2011. DOI: 10.1007/978-3-642-22110-1_47.

[24] McMahan, H. B., Likhachev, M., and Gordon, G. J. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 569–576, 2005. DOI: 10.1145/1102351.1102423.

[25] Parker, D., Norman, G., and Kwiatkowska, M. Game-based abstraction for Markov decision processes. In *Third International Conference on the Quantitative Evaluation of Systems-(QEST'06)*, pages 157–166. IEEE, 2006. DOI: 10.1109/QEST.2006.19.

[26] Quatmann, T. and Katoen, J.-P. Sound value iteration. In Chockler, H. and Weissenbacher, G., editors, *Computer Aided Verification*, pages 643–661, Cham, 2018. Springer International Publishing. DOI: 10.1007/978-3-319-96145-3_37.

[27] Song, L., Zhang, L., Hermanns, H., and Godskesen, J. C. Incremental bisimulation abstraction refinement. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):1–23, 2014. DOI: 10.1145/2627352.

[28] Tóth, T., Hajdu, Á., Vörös, A., Micskei, Z., and Majzik, I. Theta: A framework for abstraction refinement-based model checking. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 176–179. IEEE, 2017. DOI: 10.23919/FMCAD.2017.8102257.

[29] Wachter, B. and Zhang, L. Best probabilistic transformers. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 362–379. Springer, 2010. DOI: 10.1007/978-3-642-11319-2_26.

A Proofs of Soundness

A.1 Notions used in the proofs

Here we show that both Cartesian abstraction with the generalized predicate domain and the usage of the generalized explicit abstraction domain with limited enumeration of values lead to sound abstractions. We will use the abstraction game simulation approach that was proposed in [18] for the proofs.

As abstract states in the same model are no longer disjoint when considered as sets of concrete states, we will use the notion of *covering* when analyzing the resulting abstraction game and the construction algorithm: an abstract state S_1 covers another abstract state S_2, when the set of concrete states corresponding to

S_2 is a subset of the set corresponding to S_1. Covering between A-nodes can be defined using the abstract states of the A-nodes.

By describing abstract states via SMT formulae, cover checking can be formulated using the language of logic: a state A covers the state B if and only if $Expr(B) \implies Expr(A)$, where $Expr(\cdot)$ denotes the SMT formula describing the state.

In the case of (P)CFA analysis, the location is also retained by the abstraction. Covering in this case also requires that the two states have the same location. We will omit mentioning this from now on, but it is always an implicit requirement of covering.

An example game with covering can be seen in Figure 8. It can be seen on this example that whenever an A-node covers another one, the set of C-nodes that can be chosen from it is a superset of the C-nodes that can be chosen from the covered node. This property is ensured by the construction method. This leads to the more abstract A-node having more choices, and thereby its associated abstract state having less strict lower and upper value bounds, as expected.

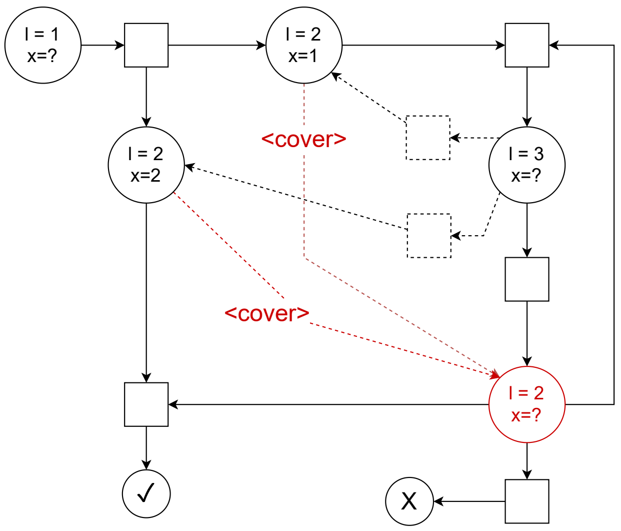


Figure 8: Example of an abstraction game on a domain without the disjointness property (generalized explicit domain).

This intuitively leads to conjecturing that the abstraction is sound, which we will formally prove in the next subsection.

Covering can also be used between states of different abstractions of the same MDP.

We will need to extend the notion of covering to C-nodes as well to simplify the wording of the proofs. In order to do this we first need to define the process of *lifting a relation to distributions*.

Definition 8 (Lifting a relation to distributions). *For a relation $\mathcal{R} \in S_1 \times S_2$, the* lifting of the relation to distributions *is a relation $D(\mathcal{R}) \in \mathbb{D}(S_1) \times \mathbb{D}(S_2)$, defined by:*

$$(d_1, d_2) \in D(\mathcal{R}) \iff \exists \delta : S_1 \times S_2 \to [0, 1], \text{ such that:}$$

$$\forall s_1 \in S_1 : d_1(s_1) = \sum_{s_2 \in S_2} \delta(s_1, s_2)$$

$$\forall s_2 \in S_2 : d_2(s_2) = \sum_{s_1 \in S_1} \delta(s_1, s_2)$$

$$(s_1, s_2) \notin \mathcal{R} \implies \delta(s_1, s_2) = 0$$

Unlike for A-nodes which are identified by an associated abstract state, the identity of C-nodes comes from their available choices, which are distributions over A-nodes. Therefore, covering is also determined by the available choices. Let us take the covering relation over A-nodes $R = \{(\hat{s}, s) \mid \hat{s} \text{ covers } s\}$. We say that a C-node $\hat{c}$ covers another C-node c if for all A-node distributions d that can be chosen in c, there is an A-node distribution $\hat{d}$ available in $\hat{c}$ such that $(\hat{d}, d) \in D(R)$.

The covering C-node thus does not have to have a superset of the exact choices of the covered C-node: A-nodes can be replaced by others that cover them. This means that for example, if the covering C-node has a single choice $(0.5 : s_1, 0.5 : s_2)$, and there is an A-node s_3 covering both s_1 and s_2, another C-node with the choice $(1 : s_3)$ covers it.

To prove the soundness of our inexact abstract transition functions, will make use of the notion of *strong probabilistic game simulation* proposed in [19]. Its aim was to define a relation that can be used to decide if a game is *more abstract* than another one.

The definition is asymmetric for the players, as it was defined specifically for game-based abstractions, where Player 1 is the abstraction player, with all of its actions leading to dirac distributions.

When defining the simulation relation for stochastic games, the simulating game is allowed to use *combined actions* to simulate actions of the other game, not only existing pure actions. Intuitively, this is similar to using probabilistic strategies when a player plays the game: instead of choosing a specific action, a distribution over the available actions can be chosen.

Although we adopt the full definition of strong probabilistic game simulation from [19] to be able to directly reference the proofs therein, we do not need the full power of this notion. Specifically, we will not need combined transitions to prove the soundness of Cartesian predicate abstraction and explicit abstraction with limited enumeration.

Combined actions will introduce *"virtual states"* for the players: the result of using a combined action will be treated similarly to existing nodes of the game, although it is not originally in the set.

Definition 9 (Virtual state). *Given a game $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$, a* virtual state *is a formal sum $\sum_{i=1}^{n} p_i s_i$, where $s_i \in S$, $\sum_{i=1}^{n} p_i = 1$, $0 < p \leqslant 1$ and*

$i \neq j \implies s_i \neq s_j$. All component states must belong to the same player, and the resulting virtual state also belongs to that player.

The set of available actions in a virtual state is given by: $Av(\sum_{i=1}^{n} p_i s_i) = \times_{i=1}^{n} Av(s_i)$.

The transition function of the game is extended to virtual states as:

$$\delta(\sum_{i=1}^{n} p_i s_i\,,\,(a_1, \ldots, a_n)) = \sum_{i=1}^{n} p_i \delta(s_i, a_i).$$

The resulting sum is not only a formal one: it is a sum over functions and addition is understood pointwise. The constraints on the weights p_i ensure that the result is a probability distribution over states.

Intuitively, a virtual state is a probabilistic superposition of states corresponding to the same player. The available (virtual) actions in the virtual state are combinations of real actions that can be considered one-step strategies: the player does not know which state they are actually in from the superposition, but it can decide which action it would choose in each of the component states.

Definition 10 (Combined actions)**.** *Given a stochastic game $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$, the set of* combined transitions *available in a state $s \in S$ is $Cmb(s) = \{\sum_i \alpha_i a_i | a_i \in Av(s), 0 \leqslant \alpha_i \leqslant 1, \sum_i \alpha_i = 1\}$. We also extend the transition function δ to accept inputs with combined transitions. For any $s \in S, a = \sum_i \alpha_i a_i \in Cmb(s)$, $\delta(s, a) = \sum_j p_j s_j$.*

Combined actions are also available in virtual states as a similar combination over the virtual actions.

Definition 11 (Strong Probabilistic Game Simulation)**.**
Let $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$ be a stochastic game where all actions of Player 1 lead to dirac distributions on S_2 and let $\mathcal{R} \subset \mathcal{S}_\infty \times \mathcal{S}_\infty$ be a relation on S_1. $\tilde{\delta} : S_1 \times A \rightharpoonup S_2$ will denote a deterministic version of the transition function constrained to Player 1, meaning that $\tilde{\delta}(s, a) = s' \iff \delta(s, a) = dirac(s')$.

$\mathcal{R}$ is a strong probabilistic game simulation *on G if for all $(\hat{s}, s) \in \mathcal{R}$ all of the following conditions hold:*

- $\forall a \in Av(s) : \exists \hat{a} \in Cmb(\hat{s}) : s_2 = \tilde{\delta}(s, a) \wedge \forall a_2 \in Av(s_2) : \hat{s}_2 = \tilde{\delta}(\hat{s}, \hat{a}) \wedge \exists \hat{a}_2 \in Cmb(\hat{s}_2) : (\delta(\hat{s}_2, \hat{a}_2), \delta(s_2, a_2)) \in D(R)$
- $\forall a \in Av(s) : \exists \hat{a} \in Cmb(\hat{s}) : \hat{s}_2 = \tilde{\delta}(\hat{s}, \hat{a}) \wedge \forall \hat{a}_2 \in Av(\hat{s}_2) : s_2 = \tilde{\delta}(s, a) \wedge \exists a_2 \in Cmb(s_2) : (\delta(\hat{s}_2, \hat{a}_2), \delta(s_2, a_2)) \in D(R)$

A relation between the Player 1 states of two different games is a strong probabilistic game simulation if it is a strong probabilistic game simulation on the disjoint union of the two games, and the pair of the two initial states is in the relation.

The two required properties of the relation intuitively mean that the abstraction has a choice both to limit the choice of the concrete non-determinism at least as much as in the simulated state (underapproximation of the choice set), and to make the choice at least as free as in the simulated state (overapproximation of the choice set).

Being able to convert and MDP to an abstraction game without ignoring any information will also help with the proof. The *game embedding* of an MDP captures this notion.

Definition 12 (Game embedding of an MDP)**.** *The* game embedding *of an MDP $M = (S, s_{init}, A, Av, \delta)$ is the abstraction game $\rho(M) = (\hat{S} = S_a \uplus S_c, \hat{s}_{init}, \hat{A}, \hat{Av}, \hat{\delta})$, where*

- *$S_a = S$ (there is exactly one A-node for each state of the MDP)*
- *There is a single C-node in S_c for each A-node $s \in S$, which will be denoted by $c(s)$*
- *$\forall \hat{s} \in S_a : \exists! a \in \hat{A} : \hat{a} \in \hat{Av}(\hat{s}) \wedge \hat{\delta}(\hat{s}, \hat{a}) = dirac(c(s))$*
- *$\forall s \in S, a \in Av(s), d \in \delta(s,a) : \exists! \hat{a} \in \hat{A} : \hat{a} \in \hat{Av}(c(s)) \wedge \hat{\delta}(c(s), \hat{a}) = \delta(s,a)$ (and $\hat{A}$ is the smallest set satisfying this constraint)*

Game embeddings make it possible to use strong stochastic game simulation to define when a game correctly abstracts an MDP. As Player A has at most one choice in each A-node of a game embedding, there is exactly one Player A strategy on this game. This means that the lower and upper bounds for probabilistic reachability properties coincide when using this game as an abstraction, giving exact results. Figure 9 shows a simple MDP and its game embedding as an example.

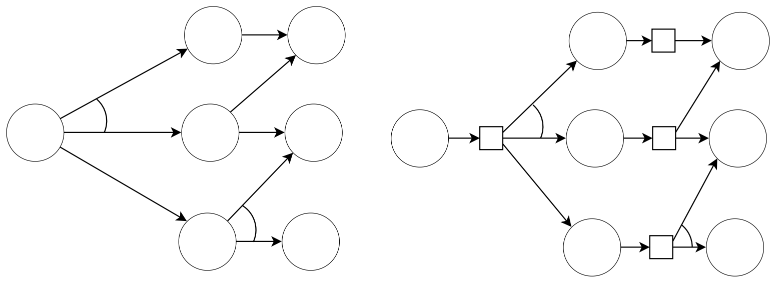


Figure 9: An MDP (left) and its game embedding (right)

Theorem 1. *Given two stochastic games $\hat{G}$ and G such that there is a strong probabilistic game simulation relation R between $\hat{G}$ and G and two sets of target states $\hat{T}$, T on them respectively such that $(\hat{s}, s) \in R \implies (\hat{s} \in \hat{T} \iff s \in T)$, we have that*

$$Prob^{-,-}(\hat{G}, \hat{T}) \leqslant Prob^{-,-}(G, T) \leqslant Prob^{+,-}(G, T) \leqslant Prob^{+,-}(\hat{G}, \hat{T})$$

$$Prob^{-,+}(\hat{G}, \hat{T}) \leqslant Prob^{-,+}(G, T) \leqslant Prob^{+,+}(G, T) \leqslant Prob^{+,+}(\hat{G}, \hat{T}),$$

where $Prob^{a,b}(G,T)$ denotes the probability of eventually reaching a state in the target set T when playing the game G with optimal strategies, (a,b) denoting the optimization objectives of the two players (+: maximize, -: minimize).

Proof. See Theorem 6.18. in [19]. □

This means that computing the reachability probabilities with optimal strategies in a more abstract game provide upper and lower bounds on the reachability probabilities of a less abstract game.

The original version of the theorem in [19] is a bit stronger as it allows the usage of abstract reward functions. For us this version will be enough, as we only care about the analysis of PCFA for now, where the target states of a reachability property are always based on the location, and the location is always exactly known in the abstraction.

A.2 Soundness of Cartesian abstraction

Theorem 1 (Soundness of Cartesian abstraction)**.** *Given a PCFA P with a set of target locations L_T, optimal strategies on the game abstraction of its MDP semantics using Cartesian abstraction to compute the abstract transition relation gives lower and upper bounds on the reachability probability of L_T. The optimal strategies are understood w.r.t. the reward function that assigns 1 to states with a location in L_T and 0 to all other states.*

Proof. It is shown in [19] that strong probabilistic game simulation is transitive, meaning that given a strong probabilistic game simulation $\hat{R}$ between the games $\hat{G}$ and G' and another one R' between G' and G, then $R = R' \circ \hat{R}$ is a strong probabilistic game simulation between $\hat{G}$ and G. It is also proved that there is a strong probabilistic game simulation between an abstraction game of the MDP semantics computed using exact predicate abstraction and the game embedding of the MDP semantics.

These two facts mean that it suffices to show only that there is a strong probabilistic game simulation between the abstraction game computed using Cartesian abstraction and the one using the exact abstract transition function.

Let $\hat{G} = (\hat{S} = \hat{S}_a \uplus \hat{S}_c, s_{init}, \hat{A}, \hat{Av}, \hat{\delta})$ be the abstraction game computed using Cartesian abstraction and let $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$ be the one computed using the exact transition function. We will show that $R = \{(\hat{s}, s) \mid \hat{s} \in \hat{S}_1, s \in S_1, \hat{s} \text{ covers } s\}$ is a strong probabilistic game simulation relation between $\hat{G}$ and G.

Recall that to prove that R is a strong probabilistic game simulation, we need to show that two properties hold for all $(\hat{s}, s) \in R$:

- $\forall a \in Av(s) : \exists \hat{a} \in Cmb(\hat{s}) : s_2 = \tilde{\delta}(s,a) \wedge \forall a_2 \in Av(s_2) : \hat{s}_2 = \tilde{\delta}(\hat{s},\hat{a}) \wedge \exists \hat{a}_2 \in Cmb(\hat{s}_2) : (\delta(\hat{s}_2,\hat{a}_2), \delta(s_2,a_2)) \in D(R)$
- $\forall a \in Av(s) : \exists \hat{a} \in Cmb(\hat{s}) : \hat{s}_2 = \tilde{\delta}(\hat{s},\hat{a}) \wedge \forall \hat{a}_2 \in Av(\hat{s}_2) : s_2 = \tilde{\delta}(s,a) \wedge \exists a_2 \in Cmb(s_2) : (\delta(\hat{s}_2,\hat{a}_2), \delta(s_2,a_2)) \in D(R)$

The difference between Cartesian abstraction and the precise abstract transition function is that a Cartesian abstraction might set a subset of the predicates to unknown. Let s be an abstract state in the stricter predicate domain, where all predicates in the precision are assigned to be either true or false, and let $\mathit{next}(s)$ denote the exact next states available from s. Let $\hat{s}$ denote an abstract state in the generalized predicate abstraction domain with the same precision such that $\hat{s}$ covers s, and let $\mathit{next}(\hat{s})$ denote the set of next states available from $\hat{s}$ using Cartesian abstraction.

From the covering relation we know that predicate assignments in s and $\hat{s}$ never contradict each other: if a predicate is not unknown in $\hat{s}$, it has the same truth value in $\hat{s}$ as in s. This also leads to the fact that $\forall s' \in \mathit{next}(s) : \exists \hat{s}' \in \mathit{next}(\hat{s}) : \hat{s}'$ covers s'.

To see why this is true, let us examine the interaction between next state computation and the covering relation. In the exact case, we compute all satisfying models for the SMT formula $\mathit{expr}(s) \wedge \mathit{expr}(\mathit{stmt}) \wedge \mathit{activation}(s')$, where $\mathit{activation}(\cdot)$ stands for the activation literal representation of the next state. From the logic-based formulation of covering, we know that $\mathit{expr}(s) \implies \mathit{expr}(\hat{s})$, from which

$$(\mathit{expr}(s) \wedge \mathit{expr}(\mathit{stmt}) \wedge \mathit{activation}(s')) \implies (\mathit{expr}(\hat{s}) \wedge \mathit{expr}(\mathit{stmt}) \wedge \mathit{activation}(s'))$$

follows. This means that any model that satisfied the SMT formulation of the step from the covered state also satisfies it from the covering state, so if we used exact computations, at least the same next states are available from the covering state.

Now we have to show that the approximate computation of Cartesian abstraction retains this property, modulo covering, as it suffices if a covering state is available instead of exactly the states available from the covered state. If there exists a satisfying model for $\mathit{expr}(\hat{s}) \wedge \mathit{expr}(\mathit{stmt}) \wedge \mathit{activation}(s')$, where the activation literal of a predicate is set to true, than $\mathit{expr}(\hat{s}) \wedge \mathit{expr}(\mathit{stmt})$ cannot imply the negation of the predicate, as that would lead to a contradiction. A similar statement is true for a false activation literal and the ponated version of the predicate. Because of this, for each resulting state of the exact computation, there is always a state in the result of the Cartesian computation that is consistent with it: each predicate is either the same as in the exact one, or unknown. This state covers the exact state.

Now we have to prove that an extension of this holds for the case when we use Cartesian abstraction with grouped transition functions (when the abstract state is in the generalized domain, but the transition is computed exactly, as explained above), as we want to show that the resulting C-nodes also cover the original ones.

The same reasoning can be used here as above for the flat list of states by simply replacing the single $\mathit{expr}(\mathit{stmt}) \wedge \mathit{activation}(s')$ by $\bigwedge_i (\mathit{expr}(\mathit{stmt}_i) \wedge \mathit{activation}(s'_i))$ with i ranging over the indices of the substatements. By doing this we can see that the same groups (leading to the same distributions) are obtained from the covering state, as well as some additional choices. As these groups are all Player A choices, we only give the abstraction player more choices, but the original choices are still available. Here we do not even have to take the "modulo covering" into account, as the exact computations can result only in states that do not contain unknown predicates.

This means that for all A-nodes s of G, if an A-node $\hat{s}$ of $\hat{G}$ covers it, than for every C-node directly available from s, there is a C-node directly available from $\hat{s}$ covering it. This leads to the satisfaction of both properties required for the simulation: under and overapproximation of the choice sets modulo covering are both satisfied by actually having the same choice sets with some states in the distributions replaced by another state covering the original one. □

A.3 Soundness of limited enumeration

Theorem 2 (Soundness of limited enumeration). *Given a PCFA P with a set of target locations L_T, optimal strategies on the game abstraction of its MDP semantics using explicit abstraction with limited enumeration to compute the abstract transition relation gives lower and upper bounds on the reachability probability of L_T. The optimal strategies are understood w.r.t. the reward function that assigns* 1 *to states with a location in L_T and* 0 *to all other states.*

Proof. Although explicit abstraction was not mentioned in [19], it can be regarded as a special case of predicate abstraction where all predicates are equalities, and we have one equality for each possible assignment of a tracked variable. This means that the proof of predicate abstraction being sound also extends to exact explicit abstraction.

Now, similarly to the Cartesian abstraction case, we only have to show that there is a strong probabilistic game simulation between the game constructed using limited enumeration and the one constructed using precise explicit abstraction (without limiting enumeration).

The same reasoning is used here as above: we can show that the covering relation $R = \{(\hat{s}, s) \mid \hat{s} \in \hat{S}_1, s \in S_1, \hat{s} \text{ covers } s\}$ is a strong probabilistic game simulation relation between the game $\hat{G}$ computed with limited enumeration and G compute using the exact abstract transition relation.

For the generalized explicit domain, a state $\hat{s}$ covering another state s means that for all tracked variables if its value is known in $\hat{s}$, it must be the same as it is in s. Now we proceed similarly to the Cartesian abstraction case.

We can see that

$$(expr(s) \wedge expr(stmt) \wedge expr(s')) \implies (expr(\hat{s}) \wedge expr(stmt) \wedge expr(s')).$$

For explicit abstraction, we can use the expression form of the next state instead of a formulation based in activation literals. This shows that if we used the exact next state computation from a covering state, the set of available next states would contain the set of states available from the covered state.

Now we have to show that using limited enumeration instead of this does not cause any problems. With limited enumeration, whenever we encounter a situation where the number of possible next values for a variable grows over a limit, instead of enumerating all possible next states, we merge these into a single one with the variable set to unknown. This merged state will then cover all of the originally available ones where the value of the variable is exactly known.

This means that for each state available from s computed using the exact next state relation, there is always a state available from s' computed using limited enumeration that covers it.

From this, the theorem can be proven using the same reasoning as in the previous proof.

$\square$