Journal of Artificial Intelligence Research 68 (2020) 247-310 Submitted 07/2019; published 05/2020

Bridging the Gap Between Probabilistic Model Checking and Probabilistic Planning: Survey, Compilations, and Empirical Comparison

Michaela Klauck KLAUCK@DEPEND.UNI-SAARLAND.DE
Marcel Steinmetz STEINMETZ@CS.UNI-SAARLAND.DE
Jörg Hoffmann HOFFMANN@CS.UNI-SAARLAND.DE
Holger Hermanns HERMANNS@DEPEND.UNI-SAARLAND.DE
Saarland University,
Saarland Informatics Campus,
Saarbrücken, Germany

Abstract

Markov decision processes are of major interest in the planning community as well as in the model checking community. But in spite of the similarity in the considered formal models, the development of new techniques and methods happened largely independently in both communities. This work is intended as a beginning to unite the two research branches. We consider goal-reachability analysis as a common basis between both communities. The core of this paper is the translation from Jani, an overarching input language for quantitative model checkers, into the probabilistic planning domain definition language (PPDDL), and vice versa from PPDDL into Jani. These translations allow the creation of an overarching benchmark collection, including existing case studies from the model checking community, as well as benchmarks from the international probabilistic planning competitions (IPPC). We use this benchmark set as a basis for an extensive empirical comparison of various approaches from the model checking community, variants of value iteration, and MDP heuristic search algorithms developed by the AI planning community. On a per benchmark domain basis, techniques from one community can achieve state-of-the-art performance in benchmarks of the other community. Across all benchmark domains of one community, the performance comparison is however in favor of the solvers and algorithms of that particular community. Reasons are the design of the benchmarks, as well as tool-related limitations. Our translation methods and benchmark collection foster cross-fertilization between both communities, pointing out specific opportunities for widening the scope of solvers to different kinds of models, as well as for exchanging and adopting algorithms across communities.

1. Introduction

Running systems (be it purely software based, or with a physical component) in the real-world adds uncertainty to the system's execution from all kinds of sources. This uncertainty can for instance come from the environment itself (e.g., external events), the possibility of failures (e.g., a robot hand might fail to grasp), or even the intrinsic probabilistic nature of actions (e.g., tossing a coin). Reasoning over models that exhibit stochastic behavior has thus become an important topic in the model checking and planning communities.

Markov decision processes (MDPs) are the common formalism to encode stochastic models under the assumption of full observability of the model's states and action effects.

An MDP provides a formal description of the states and the probabilistic transition function between the states. Given an MDP, in planning one is usually interested in finding a *way* to reach some predefined goal starting from an initial state. Unlike in the classical case, solutions are no longer simple sequences of actions, but *policies*, i.e., whole functions detailing what action to do in which situation. To differentiate between policies, various different quality criteria are considered. Here, we are interested in *MaxProb* (Kolobov, Mausam, Weld, & Geffner, 2011; Teichteil-Königsbuch, 2012; Trevizan, Teichteil-Knigsbuch, & Thiébaux, 2017), which is finding the policy whose execution leads to the satisfaction of the goal with maximal possible probability.

Traditionally, model checking is concerned with proving a system to be failure free (given a formal model and specification of the system). However, under the presence of stochastic behavior this statement might be too strong, or simply not possible to achieve. Instead one looks for a quantification of the likelihood of system failures. Desired properties of a system are formally expressed in temporal logics, such as LTL (Pnueli, 1977) or PCTL (Hansson & Jonsson, 1994), the latter being specifically designed to reason over stochastic behavior. A key feature of temporal logics is the ability to consider sequences of events for defining correct or erroneous system behavior. In spite of the complexity of these logics, in practice, computing the (minimal or maximal) probability that the model violates the property typically boils down to reachability analysis (Baier, de Alfaro, Forejt, & Kwiatkowska, 2018).

Despite this strong link between *MaxProb* analysis in probabilistic planning and reachability analysis in probabilistic model checking, the research in general, and the development of new algorithms more specifically, has happened largely independently in each community. As an effect, the available model checkers and probabilistic planners follow fundamentally different approaches for MDP goal-reachability analysis. The former tools pay the support of complex temporal properties by a limitation of the space of possible analysis algorithms. Concretely, most model checkers fall back to different variants of value iteration (VI). As VI requires the explicit construction of the MDP's state space in memory, to make it work in practice, a lot of effort was spent on storing the required information as compactly as possible, through the use of *symbolic* data structures (Miner & Parker, 2004; Kwiatkowska, Norman, & Parker, 2004). In contrast, probabilistic planners focus on a different class of algorithms, particularly tailored for reachability analysis: MDP heuristic search (Barto, Bradtke, & Singh, 1995; Hansen & Zilberstein, 2001; Bonet & Geffner, 2003b). Heuristic search algorithms use their current knowledge of the value function, in our case the goal-probability estimations computed so far, in order to disregard parts of the MDP state space that provably cannot be part of an optimal solution. To make the heuristic search's initial value function estimates more accurate, and thus to improve the overall efficiency, one can additionally exploit *heuristics*, functions that provide per-state goal-probability over-approximations.

Traditionally, research on probabilistic heuristic search algorithms focused on a particular class of MDPs: stochastic-shortest path problems (SSP) (Bertsekas & Tsitsiklis, 1996; Mausam & Kolobov, 2012). SSPs make specific assumptions that can be exploited by the heuristic search algorithms for the sake of simplicity and efficiency. These assumptions are however more restrictive than what is usually considered by the probabilistic model checking community (also most of their publicly available benchmarks do not fall into this category). Crucially, and seemingly contrary to our intention of using goal-probability analysis as a

common ground, the SSP heuristic search algorithms cannot handle *MaxProb* analysis right away. Fortunately, the restrictions of SSP MDPs can be bypassed via the FRET (find, revise, and eliminate traps) framework (Kolobov et al., 2011). In a nutshell, FRET runs a series of heuristic search executions, identifying and eliminating *traps* after every iteration, i.e., end-components in an accordingly defined sub-graph of the overall state space, so to ensure progress in the next iteration. Recently, Trevizan (2017) introduced I-Dual, a heuristic search method capable of directly dealing with *MaxProb* MDPs, and thus making the FRET outer-loop obsolete. I-Dual's key to the support for more general MDPs is the use of linear programming instead of following the asynchronous value iteration scheme, which underlies most of the SSP heuristic search algorithms.

In this paper, we begin to bring together the work of both communities using goal reachability analysis as a common basis. We will be presenting translations between Jani (Budde, Dehnert, Hahn, Hartmanns, Junges, & Turrini, 2017), an input language for quantitative model checkers, and PPDDL (probabilistic planning domain definition language) (Younes & Littman, 2004), an input language for probabilistic planners. Jani and PPDDL already come with a wide range of benchmarks. We use our compilations to combine them into one comprehensive collection. This common benchmark set serves as basis for an extensive empirical comparison of state-of-the-art quantitative model checkers and optimal PPDDL planners.

Jani (Budde et al., 2017) is a powerful language that can express models of stochastic, distributed, and concurrent systems. In full generality, Jani models are networks of stochastic timed automata. But the core formalism are MDPs. Jani was designed with a particular focus on extensibility and machine readability in mind. Extensibility simplifies the integration of new features, not part of the Jani core already (an example is the support of arrays). Machine readability makes it easier to add Jani support to existing tools. Jani is targeted at establishing a common input language for probabilistic model checkers.

On the planning side, PDDL (McDermott, 2000) is the de-facto standard input language for classical planners. PPDDL (Younes & Littman, 2004) extends PDDL by the possibility to define probabilistic action effects.[1]

Jani offers many modeling features that are not supported by PPDDL directly. To foster comparability in our experiments, we restrict our attention to Jani models that allow for a largely *structure-preserving* translation into PPDDL, i.e., avoiding the introduction of auxiliary state variables and actions if possible. Concretely, the restrictions pertain to (1) the properties to be checked, in particular we consider *MaxProb* because we restrict the experiments to this property type, (2) the state variables, (3) the initial state description, and (4) automata synchronization. Regarding (1), we restrict our attention to temporal formulas that can be represented directly in PPDDL, without the necessity of complex, often exponentially size ω-automaton constructions (Camacho, Chen, Sanner, & McIlraith, 2017; Brafman, Giacomo, & Patrizi, 2018). Regarding (2), to be able to use the available PPDDL planners, we limit the consideration to finite-state models. Regarding (3), we assume Jani models with a single initial state. Jani allows to succinctly represent multi-

1. The RDDL language (Sanner, 2010) has been introduced to cope with PPDDL's inability to efficiently model exogenous events (such as the arrival rate of cars in a traffic control system). Yet RDDL goal-probability benchmarks and RDDL planners supporting goal-probability analysis do, to the best of our knowledge, not exist. So we do not deal with RDDL in our work here.

ple initial states via set constraints. Although theoretically possible, the translation into PPDDL may come with an exponential blow-up. Regarding (4), Jani supports action-label based synchronization between multiple automata. Representing this in PPDDL would require the introduction of additional state variables and actions, potentially biasing the performance comparison between model checkers and planners. All features but the last one are rarely – if at all – used by the existing Jani benchmarks. Thus, while all these assumptions constitute theoretical limitations of our method, their effect on the practical use of our translation are benign.

In our experiments, we compare three state-of-the-art probabilistic model checkers: the Modest Toolset (Hartmanns & Hermanns, 2014), Prism (Kwiatkowska, Norman, & Parker, 2011; Brázdil, Chatterjee, Chmelik, Forejt, Kretínský, Kwiatkowska, Parker, & Ujma, 2014), and Storm (Dehnert, Junges, Katoen, & Volk, 2017), and two optimal PPDDL planners: Trevizan et al.'s (2017) extension of mGPT (Bonet & Geffner, 2005), and Probabilistic Fast Downward (PFD) (Steinmetz, Hoffmann, & Buffet, 2016). Each tool is run in various configurations, thus covering a broad range of different VI and heuristic search algorithms.

As usual, performance heavily depends on the domain. There are both, cases where heuristic search achieves state-of-the-art performance in model checking benchmarks, and cases where model checking VI variants achieve state-of-the-art performance in planning benchmarks. Across domains, benchmarks from one community tend to favor the tools and algorithms from that community. Reasons for this are benchmark design as well as tool specific issues. Many model checking benchmarks consist of multiple largely independent components that generate the same transition behavior. Model checkers are extremely efficient at exploiting such redundancies via symbolic representations. In contrast, planning benchmarks often don't contain such structure, and if they do then that structure is much less visible in the input file. The latter relates to a weakness of current model checking tools based on BDD techniques: even small changes to the input, e.g., the ordering of state variable definitions, can improve or degrade performance tremendously. The optimal probabilistic planners considered here, on the other hand, are geared towards models whose states and transitions can be easily expressed with propositional logic. Model checking benchmarks often use numeric variables and operations, which translated into PPDDL generate considerable overhead.

Our combined benchmark collection and experiments identify these issues, pinpointing weaknesses that may be addressed by broadening the tools' scope. Moreover, the exchange of ideas and algorithms across communities is motivated by the cases where a tool/algorithm from one community exhibits state-of-the-art performance on benchmarks from the other community. The latter pertains concretely to VI enhancements from model checking that could be useful in planning, and to heuristic search algorithms from planning that could be useful in model checking. In own work motivated by our observations here, we developed a preliminary adaptation of MDP heuristic search with FRET in the Modest Toolset. Further, our compilations already contributed to the *2019 Comparison of Tools for the Analysis of Quantitative Formal Models*, *QComp 2019* (Hahn, Hartmanns, Hensel, Klauck, Klein, Kretínský, Parker, Quatmann, Ruijters, & Steinmetz, 2019) and the *Quantitative Verification Benchmark Set* (Hartmanns, Klauck, Parker, Quatmann, & Ruijters, 2019).

The connection between planning and model checking is, in general, well known, and prior work has explored a variety of its aspects. Edelkamp (2003) introduces a compilation from the Promela language (Holzmann, 2004) into classical planning. Brázdil et al. (2014) applied and extended the heuristic search algorithm BRTDP (McMahan, Likhachev, & Gordon, 2005) in probabilistic model checking, showing promising results in their preliminary experiments. Baumgartner et al. (2018) have adapted the I-Dual algorithm (Trevizan et al., 2017) for the purpose of probabilistic automata synthesis, outperforming existing methods in several domains. Against the background of these works, our contribution lies in a systematic exploration of the state of the art, joined across both communities, in goal/reachability probability analysis.

The paper is organized as follows. We first introduce the core background of MDP goal-reachability analysis in Section 2.1. In Section 2.2, we provide a broad overview of the most relevant algorithms for that problem today, from both the model checking and the planning communities. After this literature review, we describe the input languages and spell out our compilations in Sections 3 and 4. We then provide a detailed empirical comparison of model checkers and planners over a large benchmark set. The benchmark collection is described in Section 5. The tools and the experiment setup are discussed in Section 6. The evaluation is split up in a comparison of the basic approaches (Section 7), an ablation study considering the various optimizations developed by both communities (Section 8), and an evaluation of the input reading and preprocessing (Section 9). We discuss related work in Section 10 before concluding the paper in Section 11.

2. Probabilistic Reachability Analysis

We consider goal-reachability probability analysis in probabilistic transition systems as a common basis to probabilistic model-checking and probabilistic planning. In this section, we describe the principle definitions of this analysis in terms of Markov decision processes (MDPs). We give a brief overview of the most common and central algorithms to solve MDP reachability problems today.

2.1 MDPs

An MDP is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{T}, \mathcal{A}, s_0, \mathcal{S}_* \rangle$ consisting of a finite set of *states* $\mathcal{S}$, a finite set of *actions* $\mathcal{A}$, the *transition probability function* $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$, a single *initial state* $s_0 \in \mathcal{S}$, and a set of *goal states* $\mathcal{S}_* \subseteq \mathcal{S}$. An action $a \in \mathcal{A}$ is *applicable* in a state $s \in \mathcal{S}$ if there exists $t \in \mathcal{S}$ such that $\mathcal{T}(s, a, t) > 0$. We denote by $\mathcal{A}(s) \subseteq \mathcal{A}$ the set of all actions that are applicable in s. For every state $s \in \mathcal{S}$ and applicable action $a \in \mathcal{A}(s)$, $\mathcal{T}(s, a)$ must induce a probability distribution over the states $\mathcal{S}$, i.e., it must hold that $\sum_{t \in \mathcal{S}} \mathcal{T}(s, a, t) = 1$. A state s is called *terminal* if $\mathcal{A}(s) = \emptyset$. The set of all terminal and non-goal states are denoted $\mathcal{S}_\perp$.

A (*deterministic and stationary*) *policy* is a partial function $\pi : \mathcal{S} \mapsto \mathcal{A}$, deciding which action to perform in a given state. For every state s, if $\pi(s)$ is defined, then it must hold that $\pi(s) \in \mathcal{A}(s)$. π is *closed for a state* $s \in \mathcal{S}$, if $\pi(t)$ is defined for every state t that is reachable from s by following π and $t \notin (\mathcal{S}_\perp \cup \mathcal{S}_*)$. π is *closed* if π is closed for s_0.

In this paper, we are particularly interested in the *MaxProb* objective, i.e., finding a policy that reaches from s_0 states in $\mathcal{S}_*$ with maximal possible probability. We denote

by $V^\pi : \mathcal{S} \mapsto [0,1]$ the goal-reachability probabilities induced by policy π. If π is closed for a state s, V^π constitutes the point-wise smallest function satisfying the *Bellman equation* (Puterman, 1994):

$$V^\pi(s) = \begin{cases} 0 & \text{if } s \in \mathcal{S}_\perp \\ 1 & \text{if } s \in \mathcal{S}_* \\ \max_{a \in \mathcal{A}(s)} \sum_{t \in \mathcal{S}} \mathcal{T}(s,a,t) \cdot V^\pi(t) & \text{otherwise} \end{cases} \tag{1}$$

The *maximal goal-reachability probability* for a state s is given by

$$V^*(s) = \max_{\pi : \pi \text{ closed for } s} V^\pi(s) \tag{2}$$

Below we will mainly focus on finding V^* itself. An optimal policy can be obtained directly from V^* by choosing, for every state $s \notin (\mathcal{S}_\perp \cup \mathcal{S}_*)$, the action $a \in \mathcal{A}(s)$ that maximizes the expression in Equation (1) corresponding to $V^*(s)$.

The definition above assumes that the MDP is given in an explicit form, i.e., explicitly providing the set of all states and transitions. In practice, this would however be very inconvenient, or due to the enormous size of the models often even impossible. Instead, MDPs are described implicitly, factorizing states into state variables, and defining actions based on conditions and effects on those variables. We will discuss two particular MDP modeling languages for that purpose in Section 3.

2.2 Algorithmic Approaches

Interesting models from a practical perspective often come with huge probabilistic transition systems, comprising millions or even billions of states. By exploiting particular structures of the application, such models can often still be represented very succinctly in terms of implicit encodings of the probabilistic transition systems. However, to give an answer to the optimization objective in consideration, the actual MDP solvers usually still need to reason over the underlying explicit transition structure after all. The size discrepancy between explicit and implicit representation is commonly known as the *state explosion problem*. To deal with implicitly defined models as efficiently as possible, the model checking and planning communities invented different kinds of approaches, to a large extend independent from each other. In model checking, the support of complex temporal properties often requires to consider, and to actually reconstruct, the entire transition system from the compact input description. A considerable portion of work was therefore spent on developing compact representation methods that efficiently support operations required for the analysis of the given properties. On the other hand, the focus on reachability properties, or variants thereof, has led the planning community to follow a different direction. A central idea underlying many of the algorithms invented there is the incorporation of additional information, automatically extracted from the compact model description. This information is exploited in order to take into consideration only a small part of the whole transition system, deemed to be relevant to answer the desired objective. So far, there are only a few works that started to bridge the gap between both communities (e.g., Teichteil-Königsbuch, 2012; Sprauel, Kolobov, & Teichteil-Königsbuch, 2014; Brázdil et al., 2014; Baumgartner et al., 2018).

For the sake of simplicity, we will specifically consider MDP *MaxProb* analysis in what follows. However, all of the presented algorithms support more general reward objectives as well. The presented approaches can roughly be split into two categories: *value iteration*, and *heuristic search*. We will next discuss different variants of each category. There also exist algorithms to solve MDP reachability problems that do not fit into those categories, notably, *policy iteration* (Puterman, 1994) and various linear program (LP) encodings (d'Epenoux, 1963). These are however less frequently used in practice, and their discussion is thus omitted, respectively shortened or discussed with the related work in Section 10, for brevity's sake.

2.2.1 Value Iteration

One of the most fundamental algorithms to the analysis of quantitative properties in MDPs is *value iteration* (VI) (Puterman, 1994). Given the input description of the model, VI follows roughly two steps: (S1) a forward pass, starting from the initial state, and building up the reachable part of the state space; and subsequently (S2) the actual value iteration part, updating a solution vector until the values of all reachable states have converged. Due to numeric instability issues, one usually uses an ϵ-convergence condition to check termination, where the value propagation part is stopped as soon as no state value changes by more than ϵ. Some probabilistic model checkers feature an option that allows to compute *exact* results (Kwiatkowska et al., 2011; Dehnert et al., 2017).

Various variants of VI have been proposed in literature, differing in how the required per-state and per-transition information is stored, and how the value updates are exactly performed. In its basic form, VI maintains in (S2) a solution vector V, storing for each state its current value estimation, i.e., in our case the current goal-reachability probability estimate. This solution vector is initialized to $V(s_*) = 1$ for the goal states $s_* \in \mathcal{S}_*$, and $V(s) = 0$ for all other states. To update the values in V, VI iterates over all reachable non-goal states in turn and applies for each state s the *Bellman backup operator*:

$$V(s) := \max_{a \in \mathcal{A}(s)} \sum_{t \in \mathcal{S}} \mathcal{T}(s, a, t) \cdot V(t) \tag{3}$$

This iteration of updates continues until the values of all states have reached ϵ-convergence. We will next detail different improvements to this basic VI variant.

Precomputation and State Reduction The bulk of work done in VI is usually spent on the value propagation phase. To avoid the consideration of all states in (S2), one can precompute states that either cannot reach any goal state at all, i.e., *dead-end* states, or can reach goal states with absolute certainty (Forejt, Kwiatkowska, Norman, & Parker, 2011). This precomputation step is done in between of (S1) and (S2) through a graph-based analysis, propagating Boolean flags through the probabilistic transition system built in (S1). Having identified such states, their values in V are initialized accordingly, and once set, the values will no longer change when performing the value updates. In other words, those states can then be ignored in (S2) completely. Additionally to saving some numeric computations, numerical rounding errors on these states' values are avoided, hence improving the overall precision of the final result.

Related to the idea of precomputing dead-end states, Steinmetz et al (2016) used goal-distance estimators, *heuristics functions*, from classical planning to already identify some of the dead-ends during the construction of the probabilistic transition system. Identifying dead-ends within (S1) may allow to construct, and thus consider, only a part of the actual reachable state space at the first place. Namely, during the forward pass, successor states are only generated for states that are not identified as dead-ends by the heuristic. The entries in V for all identified dead-ends are initialized and fixed to 0. Moreover, those states will be ignored in the value iteration phase later on.

Topological Value Iteration The order in which the state values are updated does not affect the final result. Moreover, $V(s)$ may only change if, since its last value update, the value of at least one of the successors of s has changed. *Topological value iteration* (TVI) (Ciesinski, Baier, Größer, & Klein, 2008; Dai, Mausam, Weld, & Goldsmith, 2011) exploits those observations by performing updates according to the topological order of the state space. For that, TVI first finds all strongly connected components (SCCs) in the state space. Afterwards, each SCC is handled separately, considering child SCCs before their parents, and updating the values only for the states part of the considered SCC. By following this order, it is guaranteed that the value of a state can no longer change after the corresponding SCC has been processed.

Symbolic Value Iteration Clearly, the requirement of having to construct and to visit the entire probabilistic transition system puts a major restriction onto the size of the model that can be dealt with using VI. Even though the dead-end reduction method presented above can work around this issue to some extent, the reduced state space often still remains too large to be built and represented *explicitly*. Motivated by the success in qualitative model checking, a VI variant has been proposed that adopts the idea of using *symbolic* data structures to compactly represent and efficiently reason over large transition systems (Kwiatkowska et al., 2011). In a nutshell, the value iteration part can be considered an iteration of matrix-vector multiplications, interleaved with maximum aggregations to resolve the non-deterministic choices of the MDP. In this process, the vector constitutes the current state-value estimations, as before. The MDP's transition function is interpreted as a matrix with number of rows equal to the number of probabilistic transitions, and number of columns equal to the number of states. Each row gives for every state the corresponding transition probability into that state. A single iteration of value updates then corresponds to the multiplication of the transition matrix with the current solution vector, followed by computing for every state the maximum over the entries in the resulting vector corresponding to the states' outgoing transitions. In *symbolic value iteration* the transition matrix and the solution vector are represented as *multi-terminal binary decision diagrams* (MTBDDs) (Fujita, McGeer, & Yang, 1997), an extension of the widely-used BDD data structure (Akers, 1959). MTBDDs natively support the two demanded operations. Through the ability to exploit regularities in the encoded functions, often present in, e.g., probabilistic transition systems derived from higher level modeling languages, MTBDDs can store the required information very compactly. Since both, the matrix-vector multiplication and the max aggregation operations have polynomial time complexity in the size of the MTBDD data structure itself – not the actual encoded function – symbolic value iteration hence works particularly well in cases where this representation is more compact.

Hybrid Value Iteration The numeric operations on the MTBDD data structures come with an additional overhead, which pays off only if the size of the symbolic encodings is significantly smaller than the actual probabilistic transition system and state-value vector. To benefit from both, a compact representation as well as faster numerical operations on explicit data structures, mixtures between the two have been considered (Kwiatkowska et al., 2011; Dehnert et al., 2017). *Hybrid value iteration* uses the MTBDD data structure to store the transition matrix symbolically, while the state-value vector is represented *explicitly* (e.g., as an array). The iteration of value updates follows the same procedure as in the full symbolic case. To compute the multiplication of the symbolically represented matrix with the explicitly represented vector, hybrid value iteration interleaves individual matrix entry lookups (done through a single MTBDD traversal) with numerical computations. PRISM (Kwiatkowska et al., 2011) furthermore features an optimized version of this algorithm, caching some sub-matrices of the transition matrix explicitly, to reduce the number of (redundant) accesses to the MTBDD data structure.

2.2.2 Heuristic Search

Heuristic search algorithms are based on the observation that for computing the maximal goal-reachability probability for just the initial state, $V^*(s_0)$, it is not necessarily required to consider and to compute V^* for all states reachable from s_0. As a simple example, assume that s_0 has two applicable actions: a_1 leading to a goal state with probability $p > 0.5$ (to another state with the remaining probability $1 - p$), and a_2 leading to a terminal non-goal state $s_\perp$ with the same probability p and to some state t with the remaining probability. Since we want to maximize the probability of reaching the goal, a_2 can never be the action selected in the max part of Equation (3) for s_0. In particular, we do not need to know the value of t to prove that a_2 cannot be chosen for s_0 by any optimal policy. At the same time, t might be rooting a subgraph of the state space. This entire subgraph could however be ignored, if its consideration is only required for the computation of $V^*(t)$.

This property can be exploited by interweaving state space exploration, part (S1) of VI, and value computations, (S2). Moreover, to further narrow down the focus to relevant states, one can make use of external information in terms of *heuristic functions*, i.e., state-value estimations. In the example above, assume that $s_\perp$ was not a terminal state. Then it would be no longer possible to exclude a_2 from consideration right away, since now an optimal solution could potentially pass through $s_\perp$. On the other hand, assume that we are also given an optimistic approximation of V^*, i.e., an (efficient to compute) function $h : \mathcal{S} \mapsto \mathbb{R}$ such that $h(s) \geq V^*(s)$ for all states s. Observe that h provides a necessary condition for a probabilistic transition to be part of an optimal solution. In the example, a_2 is provably non-optimal if it holds that $p \cdot h(s_\perp) + (1-p) \cdot h(t) < p$, since a_1 already leads to a goal state with probability of at least p. This check relies entirely on the information provided by h. In particular, if this condition is satisfied, then we could omit a_2 from consideration in s_0 while still not touching any of the successor states of $s_\perp$ or t.

Due to the lack of heuristic functions for the *MaxProb* objective, we will follow earlier works (Steinmetz et al., 2016; Trevizan, Thiébaux, Santana, & Williams, 2016), and entirely rely on heuristic functions from classical planning to obtain an approximative condition whether or not a given state can reach any goal state at all. Dead-ends recognized by the

heuristic will be assigned a goal-reachability probability of 0, all other states of 1. As long as the underlying classical planning heuristic does not wrongly classify states to be dead-ends, the associated *MaxProb* heuristic hence constitutes an optimistic approximation of V^*.

In the following, we outline three algorithms for the purpose of heuristic search in MDPs. First, we will introduce a well-known representative of the heuristic search algorithms, specifically tailored for a subclass of MDPs: *stochastic shortest path* (SSP) problems (Bertsekas & Tsitsiklis, 1996). Since *MaxProb* MDPs do not fall into this class of MDPs, additional steps, known as *FRET*, must be taken in order to apply those algorithms to the MDPs considered in this paper. We will finally present an approach conceptually slightly different from the SSP heuristic search algorithms, in particular not requiring the FRET outer loop.

LRTDP *Real-time dynamic programming (RTDP)* (Barto et al., 1995) is one of the first heuristic search algorithms for MDPs introduced in literature. It constitutes an anytime algorithm that maintains and constantly keeps updating a *current best solution*, i.e., a partial function providing the current state value estimates. When seeing a state for the first time, its corresponding value entry is initialized by consulting a heuristic function. To update the estimates, RTDP succeedingly runs *trials*, sample executions of the MDP, starting from the initial state and ending once a terminal state is reached. To determine which successor state to follow after state s, RTDP considers an action $a \in \mathcal{A}(s)$ *greedy* with respect to the current value function, i.e., one that maximizes Equation (3) for s, and then randomly selects a state according to the probability distribution $\mathcal{T}(s, a)$. The value updates are preformed along the trial execution. It is well-known that the value function will eventually converge to V^*, provided that (1) the MDP falls into the class of SSP problems (Bertsekas & Tsitsiklis, 1996), and (2) an optimistic heuristic was used for the value initialization. While (1) is a common assumption in many heuristic search algorithms, *MaxProb* MDPs unfortunately lie outside this category. We show in the next paragraph how to nevertheless use those algorithms for *MaxProb* analysis.

Labeled RTDP (LRTDP) (Bonet & Geffner, 2003b) extends RTDP by a mechanism to mark, after each trial, those states as *solved* whose values have provably reached ϵ-consistency. The trials are terminated already at states marked as solved, fostering convergence. Moreover, in contrast to RTDP which does not come with a termination condition, LRTDP terminates the value update procedure as soon as the initial state is marked as solved.

Find, Revise and Eliminate Traps (FRET) Heuristic search algorithms in the AI literature were invented in the context of SSPs problems (e.g., Hansen & Zilberstein, 2001; Bonet & Geffner, 2003a; Bonet, 2006), but were later on applied successfully also to non-SSP problems like the *MaxProb* analysis we are interested in here (Kolobov et al., 2011; Steinmetz et al., 2016). SSP heuristic search algorithms can be applied to *MaxProb* analysis by wrapping them into a loop of iterations, known as the *FRET* framework: *find, revise and eliminate traps* (Kolobov et al., 2011). In between calls to the heuristic search algorithm, FRET analyses and accordingly changes the value function V computed in the last heuristic search iteration, guaranteeing progress in the next heuristic search iteration. This analysis comprises of finding and eliminating *traps*, i.e., sets of states without outgoing probabilistic transitions in the *greedy graph* associated with V. In the original proposal (Kolobov et al.,

2011), the greedy graph was built by starting from the initial state and following all the probabilistic transitions greedy under V, i.e., considering for a state the transitions of all actions maximizing Equation (3). We denote this version by FRET-V. Later, Steinmetz et al. (2016) proposed a variant where for every state only the transitions of a single action, greedy under V, were considered. This optimization may help particularly in cases where the heuristic function only provides little information. The latter variant is denoted FRET-π. In both cases FRET terminates as soon as the greedy graph does not contain traps anymore.

I-Dual All the algorithms presented so far constitute different attempts to improve VI. An alternative to VI to solve MDPs is given by a linear programming (LP) encoding of Equation (3) (d'Epenoux, 1963). The LP has one variable for every state s reachable from s_0, representing the value of $V^*(s)$. For every probabilistic transition, there is a constraint forcing the left hand side of Equation (3) of the source state of this transition to be at least as large as the corresponding part in the right hand side. In case of *MaxProb* the variables of all goal states are enforced to be 1. The LP's objective is to minimize the value of s_0. Although this LP encoding provides a much more direct way of computing V^*, it is usually more expensive to use than the iterative algorithms.

I-Dual (Trevizan et al., 2016) operates on the dual of this LP encoding. Instead of constructing the whole LP at once, I-Dual iteratively builds and solves larger fractions of the LP. By carefully choosing the refinements, $V^*(s_0)$ can sometimes be computed without actually considering the whole LP, and thus without actually visiting the entire state space. The LP is constructed for a subset of states only. After every iteration, the LP is extended, adding new states to this set. For this refinement I-Dual maintains a set of *fringe* states, i.e., states occurring in the LP but whose outgoing transitions have not been considered so far. By default, fringe states are treated as if they were goal states. To extend the LP, I-Dual selects those fringe states that are visited by the last LP solution. The dual LP encoding makes this selection effective, the corresponding states can be read off directly from the LP solution. Once the states have been identified, I-Dual generates their outgoing transitions, and modifies the LP accordingly. When generating the new transitions, states might be encountered that do not appear in the LP built so far. These states become fringe states in the next iteration. The algorithm terminates as soon as the last LP solution did not visit any fringe state. At this point $V^*(s_0)$ has been found. To direct the LP solution towards an optimal solution, and to hence reduce the number of fringe states considered for expansion, I-Dual deploys heuristic functions to distinct between fringe states.

3. MDP Modeling Languages

For the purpose of compactly and naturally describing probabilistic transition systems, various modeling languages have been proposed in both the model checking and planning communities, e.g., Jani (Budde et al., 2017), Prism's language (Kwiatkowska, Norman, & Parker, 2012), Modest (Hahn, Hartmanns, Hermanns, & Katoen, 2013), PPDDL (Younes & Littman, 2004) and RDDL (Sanner, 2010). Each one focuses on different modeling aspects, making it easier to model specific applications with one language than with another. All commonly share the property to be able to succinctly express probabilistic transition systems, being often even exponentially more compact than the represented system.

In the following, we will particularly consider two modeling languages: PPDDL (Younes & Littman, 2004) as an input language of probabilistic planners, being more suited for the kinds of models considered in this paper than the more recently introduced planning modeling language RDDL; and Jani (Budde et al., 2017). While PRISM is by far the most prominent model checker for probabilistic systems, the Jani language is distinguished as the overarching notation for a variety of different quantitative model checkers (PRISM included). It crucially enables cross-comparison of results obtained by different tools (Hahn et al., 2019), and its language features are of considerable interest to the AI community. For example, it has recently been shown by Hoffmann, Hermanns, Klauck, Steinmetz, Karpas & Magazzeni (2020) that Jani is much more general and capable than PDDL even for describing probabilistic planning tasks, which shows that the language of the model checking community has benefits when used on the planning side. With our compilations between PPDDL and Jani we pave the way for future connections and research in this direction.

3.1 Probabilistic Planning Domain Definition Language (PPDDL)

PPDDL (Younes & Littman, 2004) is a syntactic extension of PDDL (McDermott, 2000), the standard input language of all classical planners today. PPDDL 1.0 is based on PDDL 2.1 level 2 (Fox & Long, 2011) and adds the possibility to define probabilistic action effects, i.e., probability distributions over multiple possible outcomes. PDDL has Lisp-like syntax and, at its core, is centered around first-order logic over finite sets of objects. The description of a planning model is divided into a *domain* and a *problem* specification.

The PDDL domain defines the general behavior of a whole family of individual planning models. It consists of a hierarchical definition of the *types* of the objects within this planning domain, *predicates* with arbitrary but finite arity, and *action schemas*. Both predicates and action schemas may be parameterized in an ordered list of typed variables. Action schemas additionally give an *action name*, a *precondition* – an arbitrary first-order formula, without free variables over the defined predicates and action schema parameters – and an *effect* – a conjunctive list of instantiated and possibly negated predicates. As an example, Figure 1 shows parts of the probabilistic version of the well-known Blocksworld domain. Consider the top part. It defines 5 different predicates with their obvious meaning. The action schema `pick-up` is parameterized in two variables: `?b1` the block to be picked up, and another block `?b2`. The precondition requires that no other block is currently held, the block referenced by `?b1` is located on top of `?b2`, and no other block is stacked on top of `?b1`. The action successfully moves `?b1` from `?b2` into the robot's hand with a probability of $\frac{3}{4}$. With a probability of $\frac{1}{4}$, `?b1` is however placed onto the table instead.

A PDDL problem serves as an instantiation of a PDDL domain. It specifies a finite set of (typed) objects. Plugging in the objects into the predicate and action schema parameters yields the set of *grounded facts*, *facts* for short, and set of *grounded actions*, *actions* for short, of the planning model. Moreover, the problem specifies the planning model's initial state and goal states. The initial state is described by a conjunctive list of facts true in it. All facts that are not given are assumed to be false initially. The goal states are represented through an arbitrary first-order formula over the facts, without free variables. The Blocksworld instance shown at the bottom of Figure 1 defines three block objects. Initially all blocks

are standing on the table. The goal is to have a single stack of `b3`, `b2`, and `b1` from top to bottom in that order.

```
(define (domain blocks-domain)
  (:predicates (holding ?b - block) (emptyhand) (on-table ?b - block) (on ?b1 ?b2
    - block) (clear ?b - block))
  (:action pick-up
    :parameters (?b1 ?b2 - block)
    :precondition (and (emptyhand) (clear ?b1) (on ?b1 ?b2))
    :effect (and
      (probabilistic
        3/4 (and (holding ?b1) (clear ?b2) (not (emptyhand)) (not (on ?b1 ?b2)))
        1/4 (and (clear ?b2) (on-table ?b1) (not (on ?b1 ?b2))))))
  (:action put-on-block
    :parameters (?b1 ?b2 - block)
    :precondition (and (holding ?b1) (clear ?b1) (clear ?b2) (not (= ?b1 ?b2)))
    :effect (and (probabilistic 3/4 (and (on ?b1 ?b2) (emptyhand) (clear ?b1) (not
     (holding ?b1)) (not (clear ?b2)))
         1/4 (and (on-table ?b1) (emptyhand) (clear ?b1) (not (holding ?b1)))))))
```

```
(define (problem blocksworld)
  (:domain blocks-domain)(:objects b1 b2 b3 - block)
  (:init (emptyhand) (on-table b1) (on-table b2) (on-table b3) (clear b1) (clear
    b2) (clear b3))
  (:goal and((emptyhand) (on b2 b1) (on b3 b2) (clear b3) (on-table b1))))
```

Figure 1: Snippets of a simple PPDDL Blocksworld example, domain specification at the top, problem file below.

Since its initial proposal by McDermott (2000), PDDL was constantly extended by additional features (Fox & Long, 2011; Edelkamp & Hoffmann, 2004; Gerevini, Haslum, Long, Saetti, & Dimopoulos, 2009), including the specification of numerical fluents and corresponding action preconditions and effects (PDDL level 2), temporal aspects (PDDL level 3), and many more. However, due to the limited support of those features in the probabilistic planners considered in this paper, we will restrict ourselves to the PDDL level 1 subset, outlined above.

As already hinted before, the first step of dealing with a pair of PPDDL domain and problem is *grounding*, i.e., instantiating the PPDDL domain with the information provided by the PPDDL problem. Instead of enumerating all possible instantiations of the predicates and action schemas for the objects defined in the PPDDL problem, one often performs some kind of reasoning to identify (and generate in the first place) only those grounded facts and actions that may actually become relevant in the resulting planning model. In the Blocksworld example above, a block can be never stacked on itself. Thus modern planning systems won't even generate facts of the form `(on b b)`, or actions `pick-up b b`. To produce and to represent the grounded planning model, different planners follow different approaches. Of particular interest for this paper, specifically for the translation presented

later on, is Probabilistic Fast Downward's (Steinmetz et al., 2016) grounding procedure. Probabilistic Fast Downward delivers the grounded planning model in terms of a *probabilistic multi-valued planning task* (PMPT), an extension of MPTs (Helmert, 2006) by probabilistic action effects:

Definition 1. *A PMPT is a tuple* $\langle \mathcal{V}, s_0, \mathcal{G}, \mathcal{A}, \mathcal{X} \rangle$ *of*

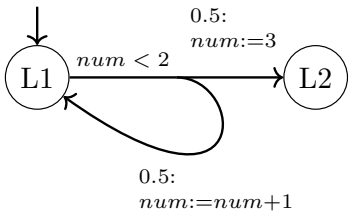
- *A finite set of* state variables $\mathcal{V}$. *Each variable* $v \in \mathcal{V}$ *has a finite domain* $\mathcal{D}_v$, *encapsulating individual* PPDDL *facts.*
- *The* initial state s_0, *a complete assignment to the variables* $\mathcal{V}$.
- *A conjunctive* goal $\mathcal{G}$, *interpreted as a partial variable assignment to* $\mathcal{V}$.
- *A finite set of (probabilistic)* actions $\mathcal{A}$. *Each action* $a \in \mathcal{A}$ *is associated with a* precondition pre_a, *a partial assignment to* $\mathcal{V}$, *and a discrete probability distribution over* outcomes out_a. *Each outcome* $o \in out_a$ *is associated with a probability* $p_o \in [0, 1]$ *and a set of* effects eff_o. *It must hold that* $\sum_{o \in out_a} p_o = 1$. *Effects are tuples* $\langle cond, v, d \rangle$ *of* effect condition $cond$, *again a partial variable assignment to* $\mathcal{V}$, *the* affected variable $v \in \mathcal{V}$, *and the* new value $d \in \mathcal{D}_v$ *for* v. *An effect is called* conditional *if cond is not empty. All effects of an outcome will be executed atomically, applying only those effects whose conditions are satisfied.*
- *A finite set of* axioms $\mathcal{X}$. *Axioms are not of interest here, and we thus skip the details.*

One possible PMPT representation of the Blocksworld instance in Figure 1 can be defined as follows. All (relevant) ground facts are collectively represented via seven variables: $empty$ with $\mathcal{D}_{empty} = \{1, 0\}$ representing the facts `(emptyhand)` and its negation; two variables b_i and $clear_i$ for every block where $\mathcal{D}_{b_i} = \{on_j \mid j \in \{1, 2, 3\} \setminus \{i\}\} \cup \{table, held\}$ representing the facts `(on bi bj)`, `(on-table bi)`, and `(holding bi)`, and $\mathcal{D}_{clear_i} = \{1, 0\}$. The initial state is given by $s_0(empty) = 1$, and for all blocks $s_0(b_i) = table$ and $s_0(clear_i) = 1$. The goal is defined in the same manner. The two depicted action schemas are grounded into 12 actions (each block can be picked up from and put onto every other block apart from the respective block itself). The action `pick-up b1 b2` has precondition $pre = \{empty{=}1, clear_1{=}1, b_1{=}on_2\}$, and two outcomes o_1 and o_2 where $p_{o_1} = \frac{3}{4}$ and $eff_{o_1} = \{\langle \emptyset, empty, 0 \rangle, \langle \emptyset, clear_2, 1 \rangle, \langle \emptyset, b_1, held \rangle\}$, and o_2 is defined similarly.

3.2 JANI

The Jani-model format (Budde et al., 2017), from now on referred to by Jani, allows to express models of distributed and concurrent systems in the form of networks of automata decorated with variables, clocks and probabilities. It was conceived to foster verification tool interoperation and comparability. Jani allows to express properties to be checked based on the probabilistic computation tree logic (PCTL) (Hansson & Jonsson, 1994), in principle supporting the representation of properties beyond reachability. The Jani language follows a Json-based format, which makes it particularly easy to parse and to extend it by new features. This simplicity quickly led to many quantitative model checkers offering direct support of Jani input, e.g., (Hahn, Li, Schewe, Turrini, & Zhang, 2014; Hartmanns &

```
"variables": [
  { "name": "num",
    "type":
    { "kind": "bounded", "base": "int",
      "lower-bound": 0, "upper-bound": 3
    } } ],
"restrict-initial": {
  "exp": "=", "left": "num", "right": 0 },
"automata": [
  { "name": "aut1",
    "locations":
      [ {"name": "L1"}, {"name": "L2"} ],
    "initial-locations": ["L1"],
    "edges": [ {
      "location": "L1",
      "guard": { "exp":
        { "op": "<", "left": "num", "right": "2"
     }
      },
      "destinations": [
        { "probability": { "exp": 0.5 },
          "location": "L1",
          "assignments": [
            { "ref": "num", "value": { "exp":
              { "op": "+", "left": "num", "right
    ": 1 }
            } } ] },
        { "probability": { "exp": 0.5 },
          "location": "L2",
          "assignments": [
            { "ref": "num", "value": 3 } ] } ]
    } ] } ]
```



$(aut1)$

Figure 2: A Jani excerpt (left) encoding the global variable *num* of type integer with finite range $[0, 3]$ and the example automaton shown on the right.

Hermanns, 2014; Dehnert et al., 2017). Moreover, automatic translations from and into different other modeling languages, e.g., the Prism language, are readily available.

A Jani model consists of three main components: 1) a list of *global variables*; 2) an *automata network*, i.e., a set of individual automaton specifications; and 3) the *property* to be checked. Jani allows the specification of variables of many different types, including discrete, bounded integer and real, continuous, and clock variables, enabling the representation of many different kinds of probabilistic systems. Expressions over these variables can be composed of all standard arithmetic operations, as well as conjunction and disjunction. An automaton is specified through a set of *local variables*, a set of *locations*, and a set of

directed *edges*. Each edge defines a single source location, a *guard*, i.e., a condition that must be satisfied to use the edge, and either a single destination or a discrete probability distribution over multiple *destinations*. Besides the target location, each destination additionally carries a list of assignments to global and local variables that apply atomically whenever taking the edge leads to the destination. To give a formal semantics to the overall automata network, Jani requires the definition of the *system composition*. In the simplest case, in each step of the overall system, exactly one applicable automaton edge is executed non-deterministically. However, Jani also supports the specification of *synchronization vectors*, allowing the parallel or synchronized execution of edges of multiple automata based on action labels. The property description may follow different schemes, depending on the exact type of the quantitative measure to consider. Most relevant for us is the maximum probability property `Pmax`, which is accompanied by the specification of the (temporal) expression to be checked. Jani allows to define one or multiple initial states by putting constraints on the global and local variables. Moreover, each automaton assigns a single or multiple initial locations.

Figure 2 shows an example snippet of a Jani encoded automaton, called *aut1*, and a single global variable *num*. *num* is a bounded integer variable with initial value 0. The automaton has two *locations L1* and *L2*. From the initial location *L1*, there is a single edge with two possible destinations, each weighted by a probability. Taking this edge requires the automaton's current location to be at *L1*, and the variable *num* to hold a value smaller than 2. If the edge is executed and the first destination applies, the automaton's location is changed to *L1*, and the variable *num* is assigned to 3. The second destination introduces a self-loop, leading back to *L1*, and increasing the value of *num* by 1.

4. Translations Between the Modeling Languages

This section presents translations between the modeling languages Jani and PPDDL in both directions. The translations serve two important contributions towards the main motivation of this paper, closing up the gap between the probabilistic model checking and probabilistic planning communities: (1) the exchange of benchmarks between both communities; and (2) the comparison of the different techniques developed, both based on a common testbed. Point (2), in particular, might further facilitate the exchange of algorithmic ideas between the communities. And point (1) connects to the statement presented in (Hoffmann et al., 2020) that Jani is a very general language which has several benefits over classical planning languages when describing planning domains. To simplify the translation part, and to not skew our empirical comparison by modeling artifacts introduced by the translation, we focus specifically on models that allow for a largely structure-preserving compilation between the two languages. We will specify the restrictions we make along with the technical discussion within this section. The translators are made available as an online appendix.

4.1 From Jani to PPDDL

This section describes our procedure translating Jani into PPDDL. We first list all the requirements on the Jani input model that we make. We then show step-by-step how the

different Jani components are translated into PPDDL. We close the section with a small discussion of this translation.

4.1.1 Restrictions to Jani

To match PPDDL planning, we make the following assumptions:

- **Initial state**: Jani supports the definition of a set of initial states encoded as arbitrary Boolean expressions over state variables. In contrast, the `init` block in PPDDL's problem description always defines a single state. In principle, it would be possible to encode in PPDDL multiple initial states by introducing for each one a separate PPDDL action schema, only applicable once at the beginning, and whose application results in the corresponding state. In general, however, Jani's initial state formula may represent exponentially many states. As it is not possible to represent such sets of states compactly in PPDDL, we restrict our attention to Jani models that define a unique initial state by enforcing a single value for every variable, and one initial location for every automaton. Evaluating multiple initial states at once in a Jani model is very rare, i.e., in practice, the impact of this restriction is benign.

- **Variables and constants**: We consider the finite-state model fragment of Jani. Although PDDL level 2.1, and thus also PPDDL, in principle support numeric state variables and durative actions, the available optimal PPDDL planners don't. We hence exclude Jani models from consideration that contain continuous or clock variables. As will be detailed below, we compile integer arithmetic into PPDDL predicates. For that compilation to work, we additionally require all Jani integer variables to be associated with a lower and upper bound, cf. *num* in Figure 2. In practice, the variables of most of Jani models are bounded anyhow. In other words, this restriction is mainly a theoretical one.

- **Automaton edge synchronization**: Our translation assumes that synchronization and communication of multiple automata are done via global variables only. Jani additionally allows to define handshake synchronization between automata using action labels. In a nutshell, every automaton edge may be associated with an action label. Multiple edges of the same automaton may share the same label. Jani provides a special block `"syncs"`, which enforces the synchronous execution of automata edges through the specification of action label vectors. For instance, in a model with three automata, `"syncs": [ "synchronise": [ "l", "l", null ] ]` forces the first two automata to execute `l`-labeled edges in parallel. The third automaton does not take part of this synchronization, so cannot execute any edge during this step.

  Jani's handshake synchronization could be modeled straightforwardly in PPDDL by creating a single action schema for every possible combination of edges that may execute in parallel. In the above example, this would mean to generate an action schema for every pair of `l`-labeled edges in the first and second automaton. In general, however, this requires the consideration of an exponential number of action schemas in the number of automata. Alternatively, one could use similar constructions as in earlier works (Edelkamp, 2003), via the introduction of additional small protocols.

This on the other hand requires to add many auxiliary state variables and actions, and hence results in a considerable change of the underlying model. Our intention being to stick to the fragment of Jani that allows for a structure-preserving translation into PPDDL– without notable changes to the model itself – we therefore do not consider Jani models using handshake synchronization here.

- **Properties**: Properties in Jani constitute an extension of PCTL (Forejt et al., 2011) with operators to query for minimal and maximal probabilities, as well as for expected reward computation. To support the evaluation of formulas on multiple initial states, Jani furthermore provides a *filter* operation. The filter operation assumes three arguments: the formula representing a set of states to take into consideration (e.g., the initial states, but could also be a different set of states), the PCTL property to be checked for these states, and an aggregation function matching the type of the property (e.g., conjunction, minimum, maximum). In Jani models that satisfy our aforementioned restrictions, more specifically in models with a single initial state, the filter operation becomes unnecessary. If present, we will thus simply ignore the filter operation in our PPDDL translation, and continue directly with the compilation of the referenced PCTL property.

  The focus of this paper is on the analysis of maximal goal-reachability probability. In terms of Jani's PCTL properties, this leads to two restrictions:

  Firstly, we restrict Jani's quantitative property operands to only `Pmax`, i.e., targeting at the computation of the maximal probability that a PCTL formula is satisfied. This is the most relevant case in practice anyhow. In effect, we rule out quantitative properties that request for minimal probabilities (`Pmin`), as well as minimal and maximal expected rewards (`Emin` and `Emax`). However, all three property types can in principle be encoded in PPDDL as well, using similar compilation steps as we will show for the maximal probability case. In particular, minimal probability analysis works out of the box with our current translation, yet requires the PPDDL planner to support minimal probability analysis. The compilation of reward properties is slightly more difficult, since the reward function in Jani is specified based on state properties, while the rewards in PPDDL are defined on a per-action basis.

  Our second restriction applies to the PCTL formulas themselves. We specifically consider probabilistic co-safety properties. In particular, we consider only simple PCTL properties of the form $F\phi$ (eventually ϕ), represented in Jani as $\top U \phi$ (true until ϕ), where ϕ does not use any temporal operator. The compilation of arbitrary PCTL reachability properties into PPDDL would in theory be possible, following, e.g., earlier works on compiling LTL properties into PDDL and PPDDL (Edelkamp, 2006; Baier, Bacchus, & McIlraith, 2009; Camacho et al., 2017; Brafman et al., 2018).

  These compilations however come with a doubly exponential explosion: in a first step, the temporal formula is translated into some ω-automaton whose worst-case size might be exponential in the size of the formula; to identify states that satisfy the formula, this automaton is in turn multiplied with the actual MDP state space. Hence, we avoid such complications and can focus on the translation of Jani's automata networks into PPDDL. The impact of this restriction is low for the case study presented later

because we concentrate on *MaxProb* analysis of well established benchmarks in both communities which all use properties which match the restriction.

4.1.2 Translation into PPDDL

This section describes how we encode Jani's variables, automata, and properties into PPDDL. We assume Jani models as described in Section 4.1.1.

Variables and Algebraic Operations The available PPDDL planning tools do not support PPDDL's numeric fluent feature. To be able to actually use the models translated from Jani, we therefore compile bounded integer variables and operations on them into predicate logic, akin to previous works that encoded finite-range integer variables into PDDL (e.g., Nakhost, Hoffmann, & Müller, 2012). All variable types in Jani directly translate into PPDDL types, with additional types `var` for variables, and `loc` for automata locations. Values associated with a type in Jani are encoded as PPDDL objects of that type. The generation of objects of integer type is restricted to the necessary numbers, determined from the bounds of all integer variables. Jani variables are also encoded as PPDDL objects, variable-value assignments are represented by the PDDL predicate `(value var val)`. For global variables, the PPDDL objects are named like the variables in Jani. To be able to uniquely refer to variables defined locally within an automaton, their PPDDL object names additionally include the name of the automaton. We list all Jani constants and variables as PPDDL constants, so we can use them in action descriptions. The initial variable assignments, as specified by Jani's `restrict-initial` blocks, translate directly into PPDDL's `:init` part. Figure 3 (a) exemplifies the translation of the global variable `num` from Figure 2. The PPDDL objects for the integer values $1, 2$ and 3 are defined as PPDDL constants, as they are required for the encoding of `aut1`'s edge later on.

Note that our translation does not enforce at a syntactic level that the values assigned to bounded variables indeed lie within the specified ranges. Jani considers edges setting bounded variables to values outside their domains as modeling flaws. Hence, assuming a properly designed Jani model as input, the translation of edge guards suffices to ensure that the variable assignments remain well-founded at any point in time.

Arithmetic and Boolean operations on integer variables are handled through additional predicates, enumerating and explicitly listing the operations' outcomes in the PPDDL initial state. Figure 3 (b) shows two examples, considering integers within $[0, 3]$. To compactly encode nested expressions, we split these into the recursive application of arithmetic operations. The outcome of each operation is stored in an auxiliary PPDDL parameter. For instance, the expression $(x_1 + x_2) * y$ will be encoded in PPDDL as the conjunction `(and (sum ?x1 ?x2 ?aux1) (product ?aux1 ?y ?aux2))`. The result of this expression can then be accessed through `?aux2`.

Automata networks We keep track of the current locations of the automata by introducing a separate predicate `(at_X ?l - loc)` for every automaton, where `X` gives the automaton's name. The different automata locations are mapped to PPDDL constants. To uniquely identify an automaton's locations across all automata, we additionally include in the object's name of every location an identifier of the automaton to which it belongs. The initial location of every automaton is added to PPDDL's `:init` block.

Domain
```
(:types
  var val - object
  int - val)
(:constants num - var
            n1 n2 n3 - int)
(:predicates
  (value ?x - var ?val - val))
```
Problem
```
(:objects n0 - int)
(:init (value num n0))
```

(a)

Domain
```
(:predicates
  (sum ?i ?j ?k - int)
  (less ?i ?j - int))
```
Problem
```
(:init
  (sum n0 n0 n0) (sum n0 n1 n1)
    ...
  (sum n1 n0 n1) (sum n1 n1 n2)
    ...
  (sum n2 n0 n2) (sum n2 n1 n3)
  (sum n3 n0 n3)
  (less n0 n1) (less n0 n2) ...
  (less n1 n2) (less n1 n3)
  (less n2 n3))
```

(b)

Domain
```
(:action aut1_l1
 :parameters (?i ?j - int)
 :precondition(and (at_aut1
    aut1_L1)
   (value num ?i) (less ?i n2)
   (sum ?i n1 ?j))
 :effect (and (probabilistic
   0.5 (and (not (value num ?i))
            (value num ?j))
   0.5 (and (not (at_aut1
    aut1_L1))
            (at_aut1 aut1_L2)
            (not (value num ?i))
            (value num n3)))))
```

(c)

Domain
```
(:predicates (property_satisfied
    ))
(:action achieve_property_dnf1
 :parameters ()
 :precondition (and (at_A A_G))
 :effect (and (
    property_satisfied)))
(:action achieve_property_dnf2
 :parameters (?i ?j - int)
 :precondition (and
   (value x1 ?i) (value x2 ?j)
   (sum ?i ?j n2))
 :effect (and (
    property_satisfied)))
```
Problem
```
(:goal (and (property_satisfied)
    ))
```

(d)

Figure 3: PPDDL snippets corresponding to the translation of (a) the Jani variable `num` from Figure 2; (b) bounded-integer "+" and "<" operations; (c) `aut1`'s edge in Figure 2; (d) the Jani property expressing reachability of a state where either automaton `A` is at location `G`, or where it holds that $x_1 + x_2 = 2$.

The edges of the automata are represented by PPDDL actions. As we do not consider the parallel execution of edges of multiple automata, every edge can be translated into an independent action. The name of the PPDDL action is built from the Jani action name as specified by the edge, the automaton's name, the name of the source location, and possibly the name of the destination location (if there is only one). The edge's guard and source

location are encoded as the precondition, the edge's destinations, i.e., variable assignments and the automaton's new locations, as the effect. To access in PPDDL the values of the variables referred in the edge description, as well as to represent and to access the results of (nested) algebraic expressions, we introduce action parameters accordingly. Figure 3 (c) shows the PPDDL action translation of the edge of the automaton `aut1` given in Figure 2.

Jani permits disjunctive edge guards. Those can, in principle, be written into PPDDL directly. However, due to the way probabilistic PDDL support is implemented in Probabilistic Fast Downward (Steinmetz et al., 2016), one of the main planning tools that we will use later for comparison, disjunctive action preconditions are not supported. Therefore, we compile away such guards using standard techniques for disjunctive normal form (DNF) transformation (Gazen & Knoblock, 1997), followed by splitting the action into one copy per disjunct.

Properties We introduce an auxiliary 0-ary predicate called `property_satisfied`. The property to be checked is translated into a goal, through an action whose precondition is the property expression and whose (deterministic) effect is `(property_satisfied)`. This predicate also becomes the PPDDL goal. Similar to the translation of Jani's edges, the goal achieving action obtains parameters to access current variable values and to represent arithmetic operations. Since Jani's properties may contain disjunctions, in the translation into PPDDL, we again do a DNF transformation and create one action per disjunct. Figure 3 (d) gives an example.

4.1.3 Discussion

The size of the compiled PPDDL encoding relates linearly to that of the input Jani model, except for (a) our encoding of finite-range arithmetic operations and (b) DNF transformation of edge guards/action preconditions. Both can be expected to be uncritical in practice. Regarding (a), our encoding is exponential in the arity of arithmetic operations, which is harmless as that arity typically is 2. Regarding (b), the exponential blow-up in DNF transformation would be relevant only on highly complex transition guards.

That said, a potentially problematic aspect is the size of the *grounded* encoding resulting from our PPDDL, as the number of ground actions is exponential in the number of action parameters required to capture all variable values relevant to the action. Any one action may contain, in principle, arbitrarily many arithmetic expressions. Again though, in practice, Jani edges – individual steps in the execution of a concurrent system – involve few arithmetic operations. Moreover, the parameter-value combinations for ground actions are constrained by the static predicates giving the semantics of the arithmetic operations. Today's planning systems exploit this property to not even generate obviously unreachable ground actions in the first place. We will come back to this issue in our experimental evaluation, Section 5.2.

4.2 From PPDDL to Jani

Since PPDDL heavily relies on first-order logic operations, particularly the parameterized description of predicates and action schemas, which is not supported by Jani, PPDDL cannot be translated into Jani right away. Prior to the translation into Jani, we therefore first create a fully-grounded version of the PPDDL input model. We use Probabilistic

Fast Downward (PFD) for this purpose (Steinmetz et al., 2016). PFD describes the grounded model as a PMPT, cf. Definition 1. Following the structure of the previous section, we next list the assumptions we make on the PPDDL input models, detail how PMPTs are translated into Jani, and finally summarize the compilation in a short discussion paragraph.

Restrictions to PPDDL and PMPTs Our translation into Jani does not support axioms. Nevertheless, this assumption does not bring any limitation in practice. All the PPDDL benchmarks considered later on do not rely on this feature. Our translation itself does not assume any other restrictions to PPDDL, i.e., with the exception of axioms, we support all PPDDL features supported by PFD's grounding procedure as well. The most notable limitation caused by the latter is the support of numeric fluents. Being based on the classical planning system Fast Downward (Helmert, 2006), PFD also inherits the inability to handle numeric variables.

Translating PMPTs into Jani PMPTs without conditional effects can straightforwardly be compiled into Jani. The variables $\mathcal{V}$ translate into global bounded-integer variables. For every PMPT variable $v \in \mathcal{V}$ the range of the corresponding Jani variable is set to $[0, |\mathcal{D}_v|]$, identifying every value $d \in \mathcal{D}_v$ by a unique integer of this range. The initial variable assignments of s_0 are listed accordingly in Jani's `restrict-initial` block. The goal is translated as the (temporal) property of eventually reaching the conjunction $\mathcal{G}$. To represent the actions of a PMPT in Jani, we introduce one automaton `main` with a single location. For every action $a \in \mathcal{A}$ a separate self-loop edge is added to this automaton. The precondition of a becomes the edge's guard, the outcomes become the destinations. Without conditional effects, the effects of an outcome can be listed directly as the assignments of the corresponding edge destination.

The variable assignments associated with any destination of an edge in Jani are always applied whenever the edge is taken and the destination occurs. To handle conditional effects, one could use known methods to compile them away (Nebel, 2000), which would then allow to use the translation method as described above. Such compilations however involve considerable changes to the model. Our intention being to obtain a largely-structure preserving translation, we thus follow a different approach by making use of Jani's edge synchronization feature.

We model a single PMPT action a with conditional effects as a two step execution in Jani. The first step determines which probabilistic outcome $o \in \mathit{out}_a$ of a should be executed. The second step atomically evaluates the conditions and, if satisfied, applies the variable assignments of the effects of o. In order to determine in Jani which outcome should be executed, we add, for every $o \in \mathit{out}_a$, a new location to `main`. Those locations are connected to `main`'s original location via a single edge, leading to the location of outcome o with probability p_o. This edge represents the choice of the application of a. The precondition of a becomes the guard of this edge.

To atomically evaluate the conditions of all effects of outcome o, every effect is modeled as a separate automaton, consisting of exactly one location. We enforce the parallel execution of all those automata via the synchronization over the action label $l_{a,o}$, chosen to be unique for all pairs of $a \in \mathcal{A}$ and $o \in \mathit{out}_a$. For every conditional effect $\langle \mathit{cond}, v, d \rangle \in \mathit{eff}_o$, the corresponding automaton has two (self-loop) edges: one with guard cond, deterministi-

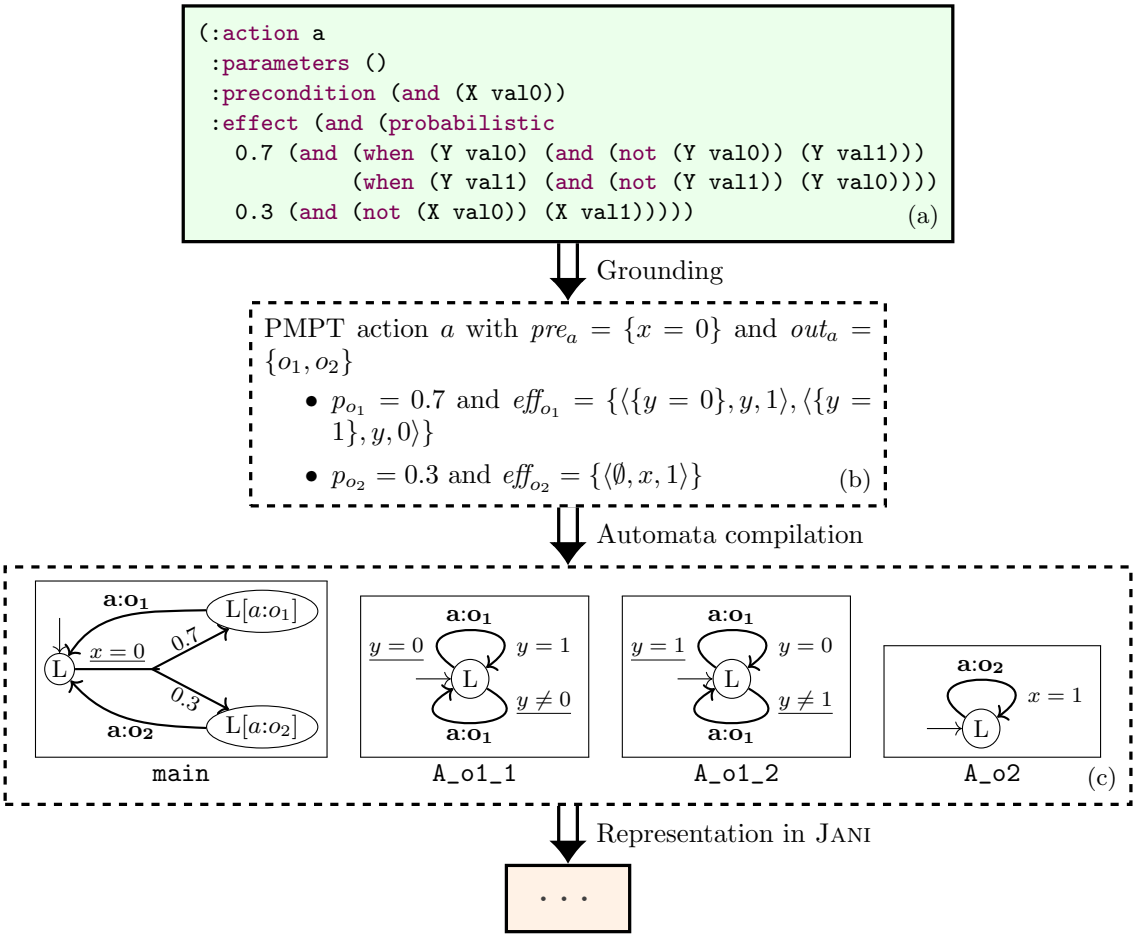


Figure 4: Translation chain from PPDDL to Jani. The example particularly focuses on the compilation of conditional effects into an automata network. The automata network can then directly be encoded in Jani. Jani code is omitted for the sake of brevity. In (c), the action labels used for edge synchronization are highlighted in boldface.

cally changing the value of v to d; and one with guard $\neg cond$ without affecting the value of any variable. Both edges are labeled with $l_{a,o}$. Finally, to make sure that the edges in the automata for the effects of o can be executed, i.e., that the effects of o apply only if a was actually applied and o has triggered, we connect the location in `main` of every outcome of a to `main`'s original location via separate deterministic edges. The edge associated with o is labeled with $l_{a,o}$ and added to the corresponding synchronization vector.

Figure 4 shows an example of this compilation. For simplicity, the PPDDL snippet at the top only includes the description of the action schema `a`. We assume that during grounding, the predicates `X` and `Y` are translated into variables x and y, e.g., $x = 0$ representing the ground fact `(X val0)`, $x = 1$ the fact `(X val1)`, and so on. The PMPT action

a for `a` is given in Figure 4 (b). a has two probabilistic outcomes, one of them contains conditional effects. Hence, for every one of a outcomes' effects, an automaton is added to the overall network, cf. Figure 4 (c). In the construction of `main`, two additional locations are created to distinguish which outcome of a to apply. Both are connected to `main`'s initial location via a single probabilistic edge. This edge requires pre_a in its guard. After taking this edge, the current location of `main` changes to either L[a:o_1] or L[a:o_2], according to a's outcome probability distribution. Depending on `main`'s new location, the overall system is forced to execute either **a:o$_1$** or **a:o$_2$**. Consider **a:o$_1$**: Due to the edge synchronization between `main`, `A_o1_1`, and `A_o1_2`, all three automata must execute an edge labeled with **a:o$_1$** simultaneously. In `main`, there is only one such edge, setting the current location back to where it was initially. Regarding `A_o1_1` and `A_o1_2`, the guard of only one of their two **a:o$_1$**-labeled edges can be satisfied at any point in time. Note that their guards are evaluated atomically before any variable is assigned to another value. If $y = 0$ holds in the current state, then `A_o1_1` must execute the upper self-loop, while `A_o1_2` must execute the self-loop at the bottom. Only after evaluating the edges' guards and determining which edges to execute, the corresponding variable assignments are applied. In the case $y = 0$ `main`'s location changes to L, and simultaneously, the value of y is changed to 1. The self-loop in `A_o1_2` constitutes a no-op, changing neither the current location of `A_o1_2`, nor the value of any variable. This edge is only required for the synchronization on **a:o$_1$** over `main`, `A_o1_1`, and `A_o1_2` to be well-defined. The case for $y = 1$ is symmetric.

Discussion The size of the automata network, and hence the size of its Jani encoding, is linear in the size of the PMPT. However, the PMPT can be exponentially large in the size of the PPDDL input in the worst case. In other words, our translation into Jani might actually add an exponential blowup in the size of the PPDDL input. That said, Jani was not designed for the description of first-order sentences or parameterized functions, making the grounding part a necessary step in the translation from PPDDL to Jani. In practice, grounding is usually not a bottleneck for the overall runtime of a benchmark and the size explosion is moderate. PPDDL input processing is often slower than the preprocessing in tools using other input languages but the analysis time afterwards can often be reduced significantly by preparing the input well during grounding. We will have a closer look at this issue in our empirical evaluation of input reading and processing, Section 9.

The representation of conditional effects in Jani requires some modifications to the model. Thus we could not completely satisfy our goal of obtaining a structure-preserving translation (from PMPT to Jani). Nevertheless, we claim that those changes are kept as minimal as possible as described in detail above. The overhead introduced by our compilation is entirely due of the combination of probabilistic action outcomes and conditional effects. Since all automata except `main` only have a single location, which cannot change through the entirety of the system execution, their current locations actually do not have to be included in the state description. The introduction of additional locations to `main` may indeed lead to a larger state space compared to the state space of the PPDDL model. A comparison for the benchmarks used, is done in the empirical evaluation in Section 5.2. However note that all additionally introduced states will only have a single outgoing transition, and that the compilation does not affect the branching behavior for any state, compared to the PPDDL state space. Jani's edge synchronization feature allows a more natural

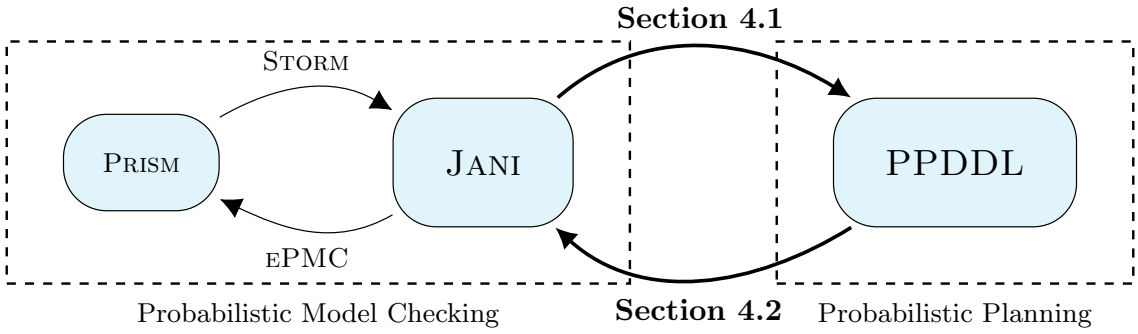


Figure 5: Benchmark translation chain. For translating models between Prism and Jani, we use existing tools (Hahn et al., 2014; Dehnert et al., 2017). The translation between Jani and PPDDL is discussed in Section 4.

representation of conditional effects. In particular, we avoid the introduction of additional state features, required, e.g., by other techniques to compile away condition effects (Nebel, 2000) to be able to sequence the execution of conditional effects.

5. Towards A Joint Benchmark Set: Translations and Empirical Study

As the development of different approaches to MDP reachability analysis was largely separated between the model checking and planning communities, it is not very surprising that the available planners do not offer implementations of model checking techniques, and vice versa that the available model checkers do not offer implementations of the planning techniques. Thus, to be able to actually compare empirically the different algorithms presented in Section 2.1, we must consider model checkers as well as planners. However, there is no single input language accepted by all these tools. Therefore, for this comparison to be possible we need to have a benchmark selection that offers encodings in several modeling languages for every benchmark instance. We use the methods presented in the previous section, along with some existing translation methods, to create such a collection from previously established benchmarks. The resulting benchmark collection is available as an online appendix, and provides all models in PPDDL (Younes & Littman, 2004), Jani (Budde et al., 2017) and Prism's (Kwiatkowska et al., 2011) language.

This section is structured as follows. We first describe how we created the benchmark collection exactly. We provide information on which tools were used for the translation, where the individual benchmark domains and instances originated from, and the criteria that we used to select a benchmark into our collection. We conclude the section with statistics about the translation process itself, shedding some more light on the issues that were already touched in the discussions of Section 4.

Prism Benchmark Suite		
●	Consensus	18
●	ContractSigning	9
●	CouponCollector	54
●	Dice	28
●	DiningCryptographers	16
●	DiningPhilosophers	12
●	FairExchange	17
●	Firewire	33
●	IsraeliJalfon	24
●	MutualPnueliZuck	17
●	PinCracking	26
●	Rabin	8

IPPC benchmarks		
▲	Blocksworld	15
▲	Boxworld	15
▲	Drive	15
▲	Elevators	15
▲	ExplodingBlocksworld	15
▲	Random	15
▲	RectangleTireworld	12
▲	Tireworld	15
▲	TriangleTireworld	40
▲	Zenotravel	15

Table 1: Benchmark collection: list of considered domains, their color codes, and the number of instances.

5.1 Benchmark Collection

Our collection consists of benchmarks from (i) the Prism Benchmark Suite (Kwiatkowska et al., 2012) and (ii) the last international probabilistic planning competitions using PDDL as input, (IPPC) 2004 – 2008 (Younes, Littman, Weissman, & Asmuth, 2005). By default, the models of source (i) are described in the Prism language, those of (ii) in PPDDL. To provide all models as PPDDL, Jani, as well as Prism input files, thereby covering the input format for most of the state-of-the-art tools in probabilistic model checking and probabilistic planning, we applied the translation chain depicted in Figure 5. We used Storm's (Dehnert et al., 2017) functionality to convert Prism models into Jani, which in turn are translated into PPDDL via the compilation presented in Section 4.1. We obtained Jani encodings of the IPPC benchmarks using the translation of Section 4.2. The Prism files are generated via a conversion from Jani using ePMC (Hahn et al., 2014).

We selected a subset of all benchmarks from (i) and (ii) according to the requirements of the methods translating between Jani and PPDDL. The list of considered domains is shown in Table 1. To allow to distinguish between the domains later on, we will use different color codes. For the ease of reference, those are listed in this table as well.

Specifically, we considered from (i) all models where the translated Jani files satisfied the requirements of Section 4.1.1. The only exception is DiningCryptographers, where edge synchronization was introduced during the translation from Prism. We instead used a slightly modified version of this domain as part of the Jani *Model Library*[2], where we

2. `https://github.com/ahartmanns/jani-models`

removed the edge synchronization feature and slightly simplified the property description. The Prism files are generated via a translation from the modified Jani files. This selection resulted in 12 different domains and 262 instances.

From (ii), we considered all models that PFD (Steinmetz et al., 2016) could handle. Particularly, 3 domains included action schemas with complex conditional effects, currently not supported by PFD's grounding procedure. The largest 3 instances of RectangleTireworld could not have been grounded due to exceeding memory limits. For the remaining domains, we used the most recent competition version. For TriangleTireworld, we also included instances larger than those contained in the standard competition set, scaling the triangle-side length to up to 74, cf. (Little & Thiebaux, 2007). After this selection, we are left with 10 different IPPC domains and 172 instances. All in all, our collection consists of 22 domains with a total of 434 instances.

5.2 Empirical Study

All translations guarantee to produce equivalent models in terms of the overall system behavior, i.e., when translating the MDP $\mathcal{M}^L$ described in language L into language L', it is guaranteed that every solution π^L to $\mathcal{M}^L$ has a correspondence $\pi^{L'}$ in $\mathcal{M}^{L'}$ with $V^{\pi^L}(s_0^L) = V^{\pi^{L'}}(s_0^{L'})$, and vice versa in the other direction. However, it is in general not guaranteed that the MDPs $\mathcal{M}^L$ and $\mathcal{M}^{L'}$ are exactly the same, i.e., have exactly same states and transitions (subject to labeling). In the translation from L into L', representing in L' certain language features of L may require the introduction of additional state variables or actions. An example for this is the Jani to PPDDL goal compilation. Changes to the MDP themselves may however bias the comparison of different tools or techniques when they are run on different encodings of the same benchmark instance. Here we investigate this issue empirically for the translations done as part of creating our benchmark collection. We compare the size of the input models for the different modeling languages. Moreover, we report statistics for PFD's grounding procedure, showing in how far it affects the translations between Jani and PPDDL.

Model Encoding Size Figure 6 (a) reports average file sizes per domain for the three modeling languages. Overall Prism offers in many cases significantly more compact representations than Jani. The largest differences can be observed in the domains of the Prism benchmark suite. This can be partially explained because the Prism files there are handwritten, exploiting various modeling features to obtain compact, while human-readable, descriptions, whereas the Jani files are all generated by software. In the IPPC benchmarks, the model encodings in Jani are still larger than those in Prism but the difference is much less noticeable. Also keep in mind that Jani was particularly designed with a focus on the simplification of automatically reading and parsing the files. The Json-based format however comes with a considerable overhead in terms of file syntax.

Regarding the translation from Jani into PPDDL, consider in Figure 6 (a) the domains from the Prism benchmark suite. The encodings in both languages are about equally large in the majority of the domains. There are, however, a few cases where the PPDDL files are several orders of magnitude larger than the Jani files. In all but one case, this discrepancy is due to the enumeration of numeric operations in the the PPDDL problem description. As indicated by Figure 6 (b), in MutualPnueliZuck, the DNF transformation of Jani's edge

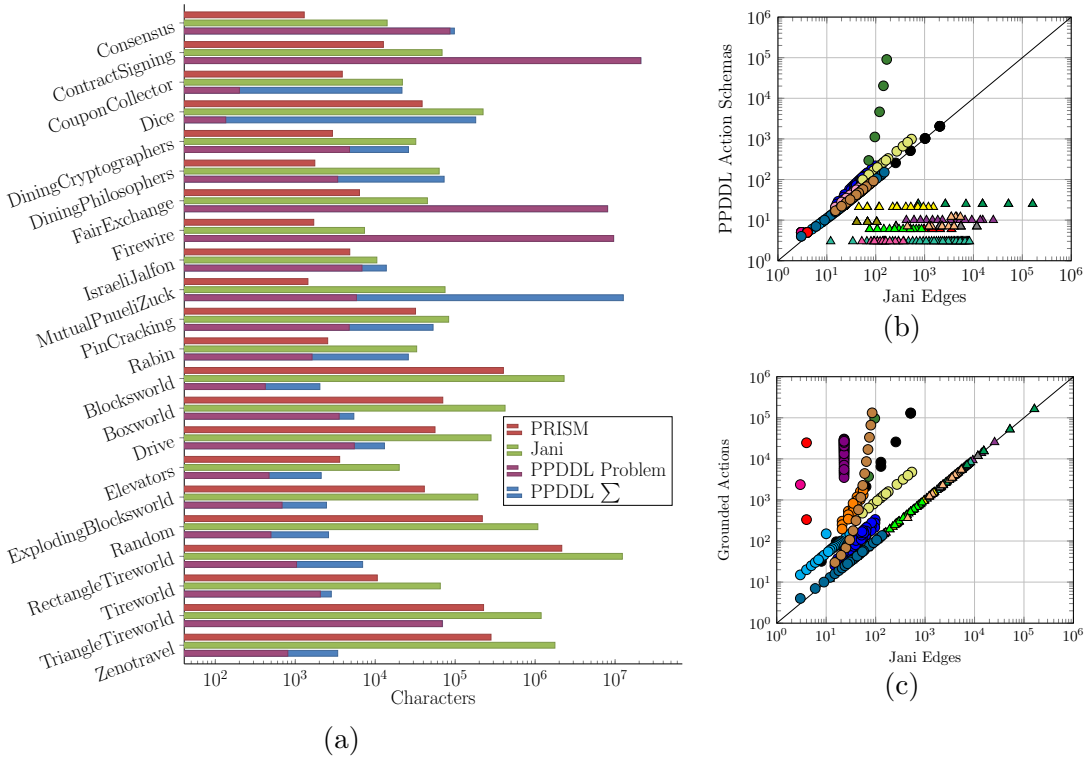


Figure 6: (a) Per-domain average file sizes, counting the number of non-whitespace characters. "PPDDL $\sum$" gives the average sum of the domain and problem file sizes, "PPDDL Problem" shows the average problem file size for reference. (b) Per-instance comparison of the number of edges in Jani vs. number of action schemas in PPDDL. (c) Same as (b) but considering the number of PFD's grounded actions instead.

guards caused an exponential overhead during PPDDL's action schema generation. In all other domains, this compilation step was not an issue. The number of generated action schemas corresponds almost directly to the number of edges.

For translations in the other direction, from PPDDL into Jani, recall that Jani's edges are created based on grounded actions. In the worst case, the grounding of PPDDL's action schemas is exponential in the number of parameters and the number of objects defined in the PPDDL problem. As shown in Figure 6 (b), this indeed tends to happen in practice. The overall structure of this plot is a result of the way the IPPC benchmarks are obtained. Different instances of a single IPPC domain are usually generated by providing different PPDDL problem files, e.g., varying the number of objects between different instances, but using the same PPDDL domain file throughout. The number of action schemas is hence constant per individual domain. In contrast, the exponential relation between number of PPDDL objects and grounded actions leads to significant differences in number of Jani edges. The explosion in the size of the grounded model is also reflected in the overall model sizes, Figure 6 (a). In all IPPC domains, the Jani files are orders of magnitude larger

than the PPDDL ones. Being translated from Jani, the same applies to the comparison to Prism as well.

Finally, consider Figure 6 (c), which compares the number of edges in Jani to the number of actions in PFD's grounded model. The selected IPPC domains contain conditional effects of a rather simple form, allowing a trivial simplification to actions without conditional effects. Consequently, the introduction of additional edges and automata was not required for the translation from the grounded PMPT models into Jani. On the other hand, Figure 6 (c) reveals another issue related to handling numeric expressions in the translations from Jani to PPDDL. In all but a few domains, the number of grounded actions differs significantly from the number of Jani edges. The super-exponential growth of the number of grounded actions originates from action schemas with many parameters, introduced as part of handling nested arithmetic expressions in Jani's edge definitions. Extreme examples are Dice, ContractSigning, FairExchange and Rabin. In the latter domain not a single instance could have been grounded by PFD because of exhausting memory. Unfortunately, with the lack of a direct support of numeric variables and operations by the state-of-the-art probabilistic planning systems, the compilation into propositional logic is unavoidable.

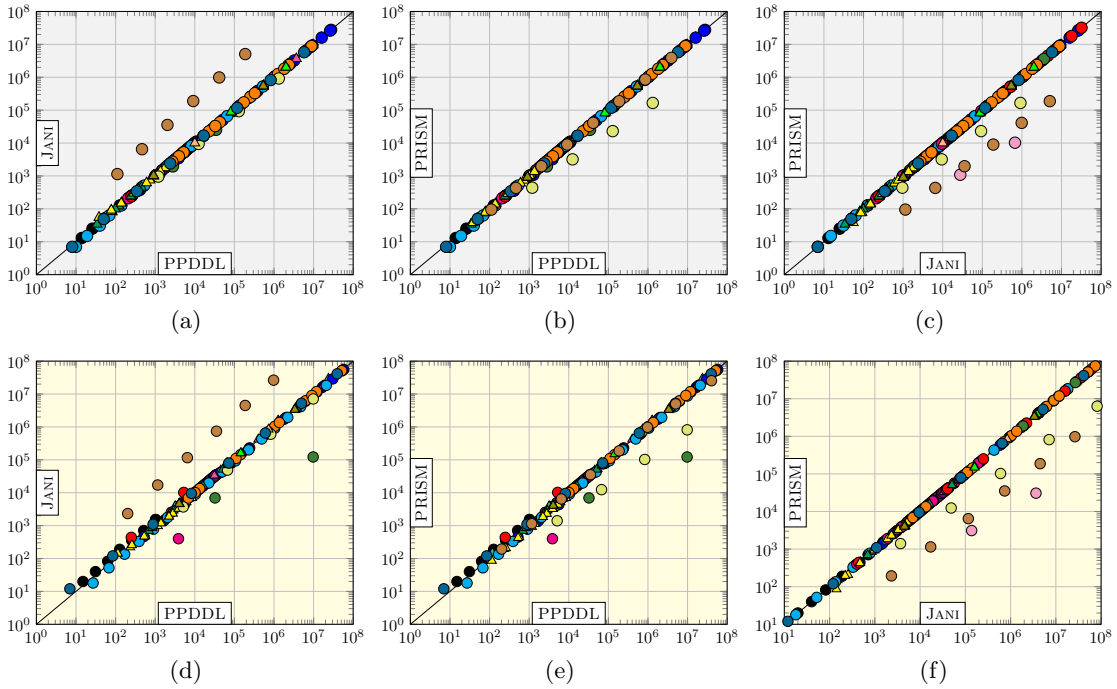


Figure 7: MDP state space size comparison between the PPDDL, Jani and Prism encodings of all benchmark instances: (a) – (c) compare the number of reachable states individually per instance; (d) – (f) compare the number of reachable transitions. PFD was used to generate the data for PPDDL, Storm was used to get the values for Jani and Prism.

MDP State Space Size To compare the different encodings of the same model at semantic level, we roll out and construct the represented MDPs explicitly. Differences between

the resulting MDPs are measured in terms of numbers of states and transitions reachable from the initial state. Figure 7 compares the three modeling languages pairwise on a per-instance basis. As expected, one cannot observe much of a difference between the different encodings of almost every instance. Notable exceptions are DiningCryptographers, DiningPhilosophers and Rabin. The Jani representations result in larger state spaces, in terms of both of our measurements, compared to the respective PPDDL (if grounding was possible within time and memory limits) and Prism encodings. Note that both, the Jani and the Prism data, were generated using the same tool, Storm, and the Jani encoding was translated from the Prism models using Storm's conversion functionality. The differences in the state space sizes originate from simple optimizations in the tool and not from differences in the model description. This means that currently for some inputs, like Prism, the state space is not explored further when a goal has been reached. This is not (always) the case for Jani inputs, but will be fixed in a new updated of Storm. In MutualPnueliZuck, the models in PPDDL have significantly more transitions than those in Jani and Prism. The reason is the DNF transformation of Jani's edge guards, which in this domain led to a blow-up in number of PPDDL action schemas with the additional effect of introducing many (redundant) transitions to the state space.

6. Experiment Setup and Tool Overview

This section gives an overview of the setup of our empirical evaluation of probabilistic model checking and probabilistic planning tools. We provide information for all the different planners and model checkers that we used for this comparison. We used the benchmark set from Section 5 as basis. All experiment runs were executed on a cluster of Intel Xeon E5-2660 machines, running at 2.2 GHz. We enforced a time limit of 30 minutes and a memory limit of 4 GB. All considered tool configurations use the ϵ-consistency property to check for termination. We chose the same value $\epsilon = 10^{-6}$ throughout.

Table 2 gives an overview of which tools have been considered in our experiments, and which of the algorithms of Section 2.2 are supported. For every tool, we selected the most recent version. The table reports version information where available. We next give reasons for this particular selection, introduce each tool individually, and point out particular configuration aspects.

6.1 Probabilistic Model Checkers

We try to cover the whole range of probabilistic model checkers, while keeping the experiments and evaluation feasible. Our selection fell on one of the oldest and best-known, while still very competitive, probabilistic model checkers: Prism (Kwiatkowska et al., 2011); a more recently introduced selection of model checking tools: the Modest Toolset (Hartmanns & Hermanns, 2014); and one of the currently leading tools for all kinds of quantitative model checking tasks: Storm (Dehnert et al., 2017).

PRISM Prism (Kwiatkowska et al., 2011) is a tool for formally modeling and analyzing all sorts of systems that exhibit random or probabilistic behavior. It has a large benchmark set, including communication, security and multimedia protocols, as well as randomized distributed algorithms and biological systems. Prism can handle discrete- and continuous-

	Input	Value Iteration				Heuristic Search		#
		Explicit	Symbolic	Hybrid	Prec.	Variants	Heuristics	
Modest *3.0.108* (Hartmanns & Hermanns, 2014)	Jani	Mcsta	–	–	⋇	–	–	2
Prism *4.4* (Kwiatkowska et al., 2011)	Prism	explicit	mtbdd	sparse, hybrid	⋇	FRET-π BRTDP	–	9
Storm *1.3.1* (Dehnert et al., 2017)	Jani, Prism	sparse	dd	hybrid	×	–	–	12
mGPT I-Dual (Trevizan et al., 2016)	PPDDL	–	–	–	–	I-Dual	$h^\top$, $h^{\max}$	2
Probabilistic Fast Downward (Steinmetz et al., 2016)	PPDDL	tvi	–	–	†	I-Dual, FRET-$\{V, \pi\}$ × {LRTDP, HDP}	$h^\top$, $h^{\max}$, MSa, MSp, PDB, Pot	36

Table 2: Overview of the tools considered in our empirical study: top half shows the probabilistic model checkers, bottom half the probabilistic planners. For each tool, we list the configurations corresponding to the algorithms discussed in Section 2.2. The "Prec." column shows which VI precomputation optimizations are available: precomputing all reachable states with goal-reachability probability of 0 or 1 (⋇); and just 0 (×); pruning dead-ends identified via classical planning heuristics (†). Entries with "–" indicate that an algorithms of the respective category are not supported. The "#" column shows the total number of considered configurations. $h^\top$ denotes the trivial goal-probability heuristic, returning 1 for every state.

time Markov chains, MDPs and probabilistic (timed) automata. It only supports models described in Prism's own language. Prism allows to analyze properties in terms of various temporal logics, e.g., LTL (Pnueli, 1977), PCTL (Hansson & Jonsson, 1994), and with a particular aspect on continuous time CSL (Baier, Haverkort, Hermanns, & Katoen, 2003). For this analysis, Prism offers multiple model-checking engines, different variants of VI: one based on explicit state space and value function representations (explicit), a variant purely based on MTBDD data structures (mtbdd), and two hybrid variants (sparse and hybrid). In explicit models are typically stored as sparse matrices or variants of, while the sparse engine stores the probabilistic transition matrix solely symbolically, hybrid trades off memory usage for faster numeric operations, keeping also explicit representations of some parts of the probabilistic transition matrix in memory. All value iteration variants feature an option to identify states with goal probability 0, respectively 1, in a preprocessing step. For the symbolic and hybrid configurations this precomputation step is entirely based on symbolic data structures. Recently, Brázdil et al (2014) have extended Prism by a variant of FRET-π using BRTDP (McMahan et al., 2005) as underlying heuristic search algorithm. That configuration has however not yet made it into the official Prism version.

Modest Toolset The Modest Toolset (Hartmanns & Hermanns, 2014) provides support for the modeling and analysis of hybrid, real-time, distributed and stochastic systems. Its modular structure supports a variety of input languages, in particular Jani, and features various analysis backends. Several components are provided to solve various forms of quantitative model checking problems. For our MDP setting, we use Mcsta. It provides the Modest Toolset's variants of value iteration, using explicit data structures to build and store the reachable state space and the state value vector. Mcsta optionally supports a forward-search precomputation step to filter out states with goal probability 0, respectively 1. If an MDP gets too large to fit into main memory, the Modest Toolset alternatively offers a hard disk based value iteration implementation (Hartmanns & Hermanns, 2015). To know which states to write to disk and which to keep in memory, a partitioning function has to be provided externally. Since we don't have such partition functions available for the considered benchmarks, we do not consider this configuration here.

Storm Storm (Dehnert et al., 2017) is one of the most competitive quantitative model checkers. It allows the analysis of the branching-time logics PCTL (Hansson & Jonsson, 1994) and CSL (Baier et al., 2003) on various types of models, such as discrete and continuous time Markov chains, MDPs, as well as Markov automata (Eisentraut, Hermanns, & Zhang, 2010). Storm reads input files in many languages. For our purposes, Storm accepts MDP descriptions in Jani and in the Prism language. For model checking MDPs, Storm comes with several engines, including value-, policy-, and interval iteration, as well as direct encodings based on LPs. Following Section 2.2, we specifically consider the engines based on value iteration: a TVI implementation based on explicit representations of the state space and value function using sparse matrices (sparse), a symbolic VI variant (dd), and a hybrid variant (hybrid). Note that in contrast to Prism's hybrid engine, Storm's variant does not cache any part of the probabilistic transition matrix in an explicit form. Prism's sparse engine stores the probabilistic transition matrix symbolically, while Storm-sparse uses sparse matrix datastructures for that purpose. In difference to the Modest Toolset and Prism, Storm only supports the precomputation of states with 0 goal-reachability probability. This option is available in all three VI variants. Additionally to the different engines, Storm is very modular in the use of libraries for internal data representation, such as the symbolic data structure, and internal computations, like methods to solve linear systems of equations or linear programs. We focused on the evaluation of the different algorithmic approaches of the considered engines and stick to the default settings for all other parameters.

6.2 Probabilistic Planners

The choice of optimality guaranteeing probabilistic planners for *MaxProb* analysis is rather limited. The original FRET implementation (Kolobov et al., 2011) is not available anymore. We consider the I-Dual implementation of (Trevizan et al., 2016, 2017) and Probabilistic Fast Downward (Steinmetz et al., 2016). Both planners are designed for reachability analysis specifically, as can be expressed in PPDDL, and thus support only a fraction of the properties handled by the probabilistic model checkers above.

Some of the planners' configurations make use of random number generators. To reduce the impact of "randomness" on our evaluation, we executed every planner configuration 5

times on every benchmark instance, each time with a different random seed. We consider an instance as solved by one configuration if the instance was solved for at least 3 of the random seeds. All other reported data constitute median values over the random seeds. In general, however, the differences between the runs of the seeds were negligible.

mGPT I-Dual mGPT (Bonet & Geffner, 2005) is a probabilistic planner developed particularly for the analysis of stochastic shortest path problems. It is centered around the heuristic search algorithms LRTDP (Bonet & Geffner, 2003b) and HDP (Bonet & Geffner, 2003a) and includes various probabilistic heuristic functions to improve their convergence behavior. It accepts PPDDL as input. Trevizan et al. (2016, 2017) extended mGPT by the support of *constrained SSPs* and the analysis of the *Min-Cost Max-Prob* (*MCMP*) objective, both handled via an implementation of the I-Dual algorithm. *MCMP* further constraints the space of *MaxProb* solutions to those minimizing a second optimization condition. As we are here merely interested in *MaxProb*, we disabled the second part of mGPT I-Dual's analysis. mGPT I-Dual offers two goal-probability heuristics: the trivial heuristic $h^\top$, and a heuristic that returns 0 for states recognized by the classical planning heuristic $h^{\max}$ (Bonet & Geffner, 1999) as dead-end, and 1 otherwise. mGPT I-Dual relies on Gurobi to solve linear programs. We used Gurobi version 7.5.2 in our experiments.

Probabilistic Fast Downward Probabilistic Fast Downward (PFD) (Steinmetz et al., 2016) is an extension of the widely used classical planning system Fast Downward (Helmert, 2006) to the world of MDPs. Models must be provided as PPDDL input. PFD supports the analysis of various reachability objectives such as expected cost (reward) and, most relevant for us, *MaxProb*. Supported engines for *MaxProb* are: a TVI implementation, both FRET-V and FRET-π in combination with various SSP heuristic search algorithms, and an I-Dual implementation. We instantiated the FRET configurations with LRTDP and HDP. There are two goal-probability heuristics available: the trivial heuristic $h^\top$, and the one that utilizes classical planning heuristics' dead-end detection capabilities. Regarding the latter, being based on Fast Downward, plenty of classical planning heuristics are readily available. Since our only demand on those heuristics is recognizing as many dead-ends as possible, we decided to stick to techniques particularly tailored for that purpose. Namely, we selected the heuristics used by participants of the previous unsolvability competition: $h^{\max}$ (Bonet & Geffner, 1999); two unsolvability merge-and-shrink heuristics (Hoffmann, Kissmann, & Torralba, 2014; Torralba, Hoffmann, & Kissmann, 2016), one providing full information about dead-ends (MSp), but being very expensive to construct, and an approximative version which imposes a size limit on the abstraction size (MSa); the dead-end PDB heuristic (PDB) and the dead-end potential heuristic (Pot) part of Aidos (Seipp, Pommerening, Sievers, & Wehrle, 2016). PFD's TVI engine additionally features an option to use classical planning heuristic for pruning dead-ends during the construction of the state space. PFD uses Cplex as LP-solver. Cplex is installed in version 12.6.3 on our machines.

7. Evaluation I: Comparing the Principal Approaches

In this section, we compare the approaches presented in Section 2.2 based on the implementations provided by the selected tools. Due to the large parameter space of the tools, we

split up the evaluation into several sections. This section starts with the comparison of the baseline algorithms only, ignoring the various available optimization options for the time being. More specifically, all VI configurations considered in this section do not make use of any preprocessing or precomputation step. For the heuristic search configurations, we only consider the trivial heuristic. Note that, as has been shown in previous works (Steinmetz et al., 2016), and as we will confirm in Section 7.2 and 8, heuristic search can be useful even when ran without an informative heuristic. We make this selection so to get an isolated view on the performance of the core algorithms first. Given the baseline results, the section hereafter will show in detail how the different heuristic functions and other algorithm extensions impact the performance. Despite ignoring the more elaborate configurations at this point, the presented results reflect the tool and algorithm performances in general – even in consideration of more elaborate configurations. In particular, all observations made regarding the general relation between the different VI variants themselves, the relation between different heuristic search algorithms themselves, as well as the relation across algorithms of both classes, are not affected by the choice of the parameters that are omitted here. All Storm configurations shown in this section use Prism as input.

We proceed as follows. We first evaluate the different VI implementations and the different heuristic search algorithms in isolation. We conclude this section with a comparison of both families. Different configurations are compared individually per tool as well as across tools. In particular, in the comparison of conceptually different algorithms, we try to use and compare implementations within the same tool whenever possible. We do so to avoid running into pitfalls such as low-level implementational differences between the tools, and hence focus more on differences between the actual approaches.

In all reported results, we take into account the entire tool execution, particularly including parsing the input models. Since the different tools use different input languages, this potentially adds some noise to the comparison. The pragmatic reason why we still include input processing is that some of the tools do not provide information about this process. More importantly, however, we do so because some of the tools perform (more or less expensive) preprocessing steps as part of processing the input, which may simplify the work to be done later on by the actual MDP analysis algorithm. None of the results consider the time and memory required for translating the models from their original descriptions. In the following, we exclude the IPPC Boxworld domain from consideration since not a single instance has been solved by any configuration.

7.1 Value Iteration

Table 3 shows a per-domain summary of the number of instances each VI tool configuration could solve before running out of time or memory. We next discuss the results for the explicit VI, symbolic VI, and hybrid configurations in this order.

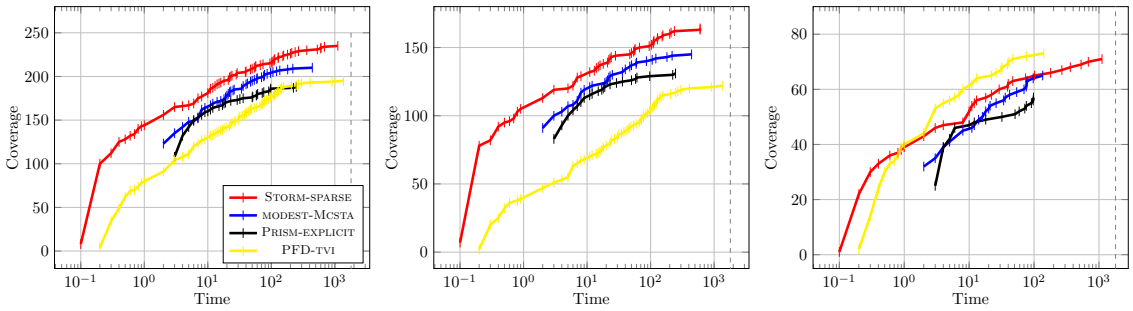
Explicit VI We start with a comparison of the explicit state VI implementations. Both Storm and PFD run topological VI. modest and the Prism run instances of VI that neither require the analysis nor exploit such graph structures. Although, Prism offers a topological VI variant as well, this configuration performed slightly worse than the one shown here due to an overall larger memory footprint.

Domain	#	Modest Mcsta	PFD tvi	Prism explicit	Prism mtbdd	Prism sparse	Storm sparse	Storm dd	Storm hybrid
Consensus	18	12	13	9	10	12	13	12	**15**
ContractSigning	9	**9**	1	**9**	**9**	**9**	**9**	**9**	**9**
CouponCollector	54	19	22	12	12	13	21	42	**43**
Dice	28	17	9	**28**	22	22	**28**	18	18
DiningCryptographers	16	7	9	4	**16**	8	8	**16**	**16**
DiningPhilosophers	12	5	4	3	**9**	4	6	**9**	**9**
FairExchange	17	9	2	6	**17**	9	11	9	9
Firewire	33	**33**	**33**	**33**	**33**	**33**	**33**	**33**	**33**
IsraeliJalfon	24	18	18	16	15	18	18	**24**	**24**
MutualPnueliZuck	17	4	2	3	8	4	4	**9**	**9**
PinCracking	26	8	9	6	20	7	8	**26**	20
Rabin	8	4	0	2	**8**	2	5	**8**	**8**
$\sum$ Prism	262	145	122	131	179	141	164	**215**	213
Blocksworld	15	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
Drive	15	**15**	**15**	**15**	6	6	**15**	8	8
Elevators	15	**15**	**15**	**15**	5	5	**15**	5	5
ExplodingBlocksworld	15	**4**	**4**	2	0	0	**4**	2	2
Random	15	**1**	**1**	**1**	0	0	**1**	0	0
RectangleTireworld	12	8	**12**	10	10	10	**12**	**12**	**12**
Tireworld	15	10	**14**	5	13	7	12	13	12
TriangleTireworld	40	5	5	4	3	3	5	**7**	**7**
Zenotravel	15	**3**	**3**	1	2	**3**	**3**	**3**	**3**
$\sum$ IPPC	157	65	**73**	57	43	38	71	54	53
$\sum\sum$	419	210	195	188	222	179	235	**269**	266

Table 3: VI coverage table: number of instances solved within the limits. All configurations come without any precomputation method. Best results are highlighted in **bold**.

Among the explicit VI configurations in Table 3, Storm's sparse engine clearly stands out. Besides solving the most instances overall, it provides highest per-domain coverage values in almost all domains. PFD-tvi is the only explicit VI configuration that can achieve higher coverage for some domains (some from each of the two benchmark sets). This indicates the potential benefits of the topological VI variant. Unfortunately, not all tools provide detailed statistics about their solving process. In particular, we cannot exclude completely the impact of other implementation related differences between the tools.

In spite of the seeming dominance of topological VI, PFD-tvi still lags behind all model checkers' VI configurations in multiple Prism domains. Prism-explicit is not really competitive with either Modest-Mcsta or Storm-sparse. Even with Prism-explicit's slight advantage over PFD-tvi in the Prism benchmark part, it is the weakest VI configuration in the overall comparison.

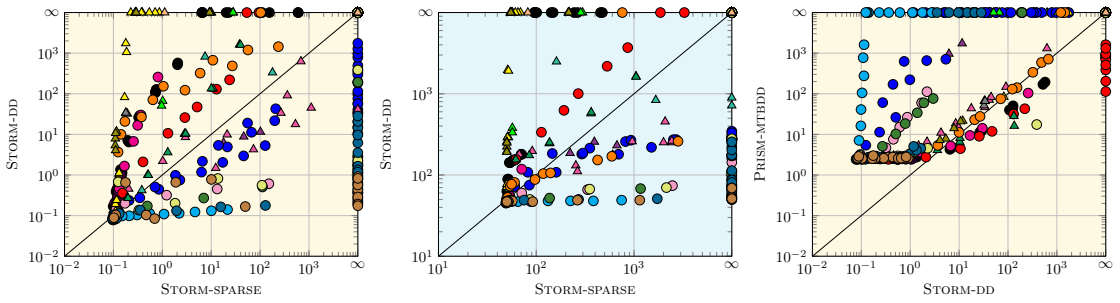


(All) (Prism) (IPPC)

Figure 8: Coverage as a function of overall time required to solve the instances for explicit VI configurations. The three plots differ in the considered domains.

The same observations are reflected in runtime, Figure 8. Additionally, the figure makes clear by how much Storm-sparse is ahead of its competitors. More importantly, however, the different plots show quite nicely that the relative performance of the tools varies greatly in between the two benchmark sets. While PFD-tvi cannot keep up with the performance of the other VI configurations in the Prism set, the picture is the complete opposite for the IPPC benchmarks. Note that we counted here the tools' overall runtime, including input reading and processing. As we will discuss later in more details, the heavy use of numeric variables and operations in some Prism domains causes significant troubles to PFD's grounding procedure. This reason especially pertains to ContractSigning, Dice, and FairExchange, where differences in terms of the coverage results were particularly large compared to the model checker's VI configurations. On the other hand, PFD is able to handle more efficiently the native PPDDL benchmarks (where also grounding isn't that big of a problem). Yet here, the planner and model checkers (specifically Storm-sparse) differ to a much smaller extent. This is to be expected given that the Jani and Prism translations are based on PFD's internal, fully grounded, MDP representation. The lead of PFD-tvi and Storm-sparse in the IPPC part provides another evidence of the effectiveness of enhancing VI via state space graph properties.

Symbolic VI As far as the Prism part of the benchmarks is concerned, none of the explicit VI configuration can really compete with the symbolic VI implementations. As shown in Table 3, over all Prism domains, Prism-mtbdd is able to solve 48 more instances than Prism's explicit variant, Storm-dd even increases coverage by 51 instances compared to Storm-sparse. Only in Dice both symbolic configurations turned out to be worse than their explicit VI counterparts. In FairExchange, Storm-dd solves fewer instances than Storm-sparse, but Prism-mtbdd vastly outperforms Prism-explicit. Vice versa for IsraeliJalfon where Prism-mtbdd solved one instance less than Prism-explicit, but Storm-dd could solve significantly more instances than Storm-sparse. The overall picture looks different in the IPPC domains. The symbolic variants achieved better coverage results only in Tireworld and TriangleTireworld. In the majority of the remaining domains they however led to a significant decrease in coverage. Comparing the two pure symbolic configurations, Storm-dd solves considerably more instances than Prism-mtbdd in al-



(a) (b) (c)

Figure 9: Per-instance comparison of (a, c) runtime in seconds and (b) peak memory usage in MB. Entries marked with "∞" represent instances that were not solved within the limits.

most every domain, especially in CouponCollector. The only two domains where Prism's symbolic engine can beat Storm-dd are FairExchange and Dice.

To shed some more light on these results, Figure 9 provides pairwise comparisons of runtime and memory usage statistics. Figure 9 (a) and (b) compare Storm's explicit and symbolic configurations. The plots for Prism look similar. The following statements thus apply to the Prism configurations as well.

The crucial factor to the performance of the symbolic variants is the compactness of the state space representation. In our experiments, the model checkers' symbolic engines were extremely efficient at exploiting well-structured automata networks where many automata have the same internal behavior, and dependencies and effects on global variables are limited or follow regular patterns. Many of the models of the Prism benchmark set share this property. Taking a look at Figure 9 (b), Storm-dd has a significant advantage in terms of memory usage over Storm-sparse in DiningCryptographers, DiningPhilosophers, IsraeliJalfon, and MutualPnueliZuck, and Rabin. In DiningCryptographers (Lynch, Saias, & Segala, 1994) and DiningPhilosophers (Lehmann & Rabin, 1981), each cryptographer and philosopher is modeled as a separate automaton, all of them have the same local variables and relative transition behavior. The instances are scaled by just scaling the number of automata. MutualPnueliZuck and Rabin are models of two famous mutual exclusion protocols between multiple processes (Pnueli & Zuck, 1986; Rabin, 1982). The individual processes are represented as separate automata, all again with the same internal description. The instances only differ in the number of processes considered. Most of the other Prism domains follow a similar pattern.

While an advantage in representation size indeed carries over to runtime in many cases, see Figure 9 (a), this is by no means guaranteed. For example, yet memory could be reduced noticeably for Consensus, Storm-dd is still significantly slower than Storm-sparse due to more expensive numeric operations on the symbolic data structures. On the other hand, symbolic data structures do not necessarily yield more compact representations than explicit encodings either. In fact, they may actually result in an exponential blow up. This worst-case size explosion can be observed in Dice and FairExchange, as well as several IPPC

domains. Having a symbolic representation much larger than the explicit representation further amplifies the overhead of numeric operations.

Concretely, contrasting other Prism benchmarks, the Dice instances each contains just a single automaton. These automata implicitly encode complex decision trees whose structures are much less suited for symbolic representations. Similarly, most of the IPPC benchmarks come without regularly structured and independent components, too. For example, in the Blocksworld domains, the components (the blocks) are highly interconnected: moving one block has potential preconditions and effects on every other block. In particular, there is no "component-local" information that could be exploited by the symbolic engines. Another example is Random, where the instances are randomly generated, hence eliminating any exploitable structural property already by construction. The benchmark design is however not the only reason why the symbolic engines performed poorly in the IPPC domains. Some of the IPPC domains actually have similar characteristics as sketched above. But as opposed to the native Prism benchmarks, the input files here do not provide information about these structures directly (e.g., in terms of local automata variables, splitting the definition of the transition relation into transitions of multiple automata, or even the order in which the variables and automata are defined). It is well known that the performance of symbolic engines crucially depends on such input file features (e.g., Maisonneuve, 2009).

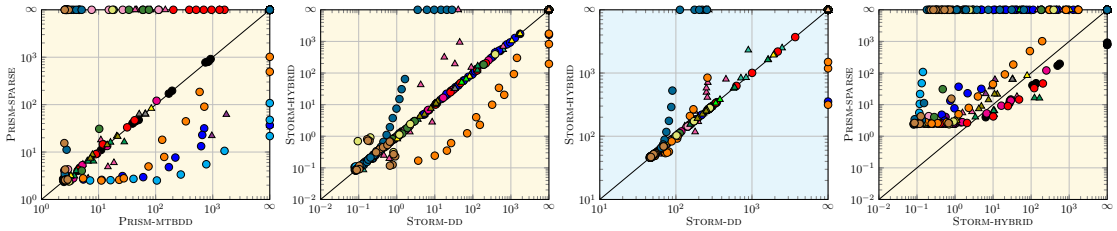
Consider the Tireworld domains. Viewing the car and the spare tires as separate components, the spare tire components are completely independent of each other and solely interact with the car. Moreover, the spare tires all share the same behavior: they are available until the car decides to change the tire at the respective location. As the results in Table 3 and those in Figure 9 already indicate, this simple domain structure can be effectively exploited by the symbolic engines. However, it turns out that theses results don't actually reflect the symbolic engines' full potential. By manually changing the model files that come out of our translation, simply regrouping variables into separate automata (one automaton for every spare tire, and one automaton for the car), Storm-dd is able to scale in TriangleTireworld to instances with triangle length of even up to 108. In comparison, the biggest instance part of our benchmark set has triangle-length 81. The highest previously reported result was 74 (Steinmetz et al., 2016).

In general, however, it is unclear which specific variable orders, or other concrete properties of the input file, have a beneficial effect on the symbolic model representation. Previous works in that direction solely focused on greedy methods (Maisonneuve, 2009; Klein, Baier, Chrszon, Daum, Dubslaff, Klüppelholz, Märcker, & Müller, 2018). We also experimented with Prism's and Storm's possibility to automatically find good variable orderings. These approaches did however not lead to any notable difference to the results presented here.

Consider finally Figure 9 (c) which compares directly the two symbolic configurations based on runtime. Prism-mtbdd is able to clearly outperform Storm-dd in FairExchange, where Storm-dd was not able to find a compact representation. On the other hand, there are many domains where Storm-dd is several orders of magnitude faster than Prism-mtbdd. Overall, and as already indicated by the coverage results, Storm-dd is the more efficient configuration in this comparison.

Hybrid VI By mixing explicit and symbolic methods, both Prism and Storm are able to improve coverage in several domains, cf. Table 3. The benefits of the hybrid variants are

particularly visible in Consensus, a domain known to require many numeric computations until reaching convergence. This together with a compact symbolic representation of the state space, cf. Figure 9 (b), makes the hybrid VI configurations to the best performing choices for this domain among all VI methods. Storm-hybrid can maintain the good overall performance of the pure symbolic dd configuration. The higher memory demand of the hybrid variant results in worse coverage values in only PinCracking (off by 6 instances) and Tireworld (1). In contrast, the performance of Prism-mtbdd does not really carry over to Prism-sparse, losing in coverage in multiple Prism domains, as well as in Tireworld.



(a) (b) (c) (d)

Figure 10: Per instance comparisons of (a,b,d) overall runtime in seconds and (c) peak memory usage in MB. Entries marked with "∞" represent instances that were not solved within the limits.

Figure 10 reports runtime and memory usage results. Comparing runtime of Prism-mtbdd and Prism-sparse, the latter is almost consistently faster than the former. However, Prism-sparse pays the benefits in runtime by memory. Particularly, Prism-sparse performs considerably worse according to coverage due to running out of memory in significantly more instances. Storm's dd and hybrid engines perform overall very similarly in terms of runtime and memory, cf. Figure 10 (a) and (b). For the Consensus instances, Storm-hybrid is able to reduce runtime by more than an order of magnitude compared to dd. As already indicated by the coverage results, the larger memory requirements of the hybrid variant is visible especially in PinCracking and Tireworld. In both domains, the exponentially scaling nature of the instances is reflected directly in the explicitly stored parts of Storm-hybrid, whereas dd efficiently exploits its symbolic data structures showing in a polynomially scaling performance over the instances. As visible in Figure 10 (d), both hybrid configurations Prism-sparse and Storm-hybrid, provide almost the same behavior on the commonly solved instances, with a slight advantage on sides of Storm-hybrid. Storm-hybrid is however more memory effective, what also led to the difference in coverage.

7.2 Heuristic Search

Table 4 provides coverage results for the different heuristic search configurations. We also included PFD-tvi and Storm's engines, the overall best performing VI configurations, for the comparison as part of the next section. The choice of the heuristic search algorithm underlying PFD's different FRET configurations had no effect on the results. For simplicity, we consider only the configurations using LRTDP. All of the following observations however

		mGPT	Prism	PFD				Storm		
Domain	#	i-dual	fret-π	i-dual	fret-v	fret-π	tvi	sparse	dd	hybrid
Consensus	18	4	1	6	2	2	13	13	12	**15**
ContractSigning	9	1	4	1	1	1	1	**9**	**9**	**9**
CouponCollector	54	10	6	10	18	18	22	21	42	**43**
Dice	28	12	**28**	9	9	9	9	**28**	18	18
DiningCryptographers	16	4	5	9	8	15	9	8	**16**	**16**
DiningPhilosophers	12	**12**	8	9	4	9	4	6	9	9
FairExchange	17	2	1	2	2	2	2	**11**	9	9
Firewire	33	18	**33**	**33**	**33**	**33**	**33**	**33**	**33**	**33**
IsraeliJalfon	24	5	13	11	18	20	18	18	**24**	**24**
MutualPnueliZuck	17	2	8	2	2	2	2	4	**9**	**9**
PinCracking	26	6	6	6	8	8	9	8	**26**	20
Rabin	8	0	**8**	0	0	0	0	5	**8**	**8**
∑ Prism	262	76	121	98	105	119	122	164	**215**	213
Blocksworld	15	4	4	4	4	4	4	4	4	4
Drive	15	9	13	**15**	**15**	**15**	**15**	**15**	8	8
Elevators	15	10	10	12	**15**	**15**	**15**	**15**	5	5
ExplodingBlocksworld	15	3	2	2	4	**5**	4	4	2	2
Random	15	2	0	3	1	**6**	1	1	0	0
RectangleTireworld	12	12	11	12	12	12	12	12	12	12
Tireworld	15	13	14	**15**	12	**15**	14	12	13	12
TriangleTireworld	40	15	4	5	5	**40**	5	5	7	7
Zenotravel	15	2	0	2	**3**	**3**	**3**	**3**	**3**	**3**
∑ IPPC	157	70	58	70	71	**115**	73	71	54	53
∑∑	419	146	179	168	176	234	195	235	**269**	266

Table 4: Heuristic search coverage table: number of instances solved within the limits. Best results over all shown configurations are highlighted in **bold**, best results among the heuristic search configurations are underlined.

apply to the HDP variants as well. All shown heuristic search configurations deploy the trivial heuristic $h^{\top}$.

FRET We start with comparing the two FRET-π variants. At the first glance, Prism-FRET-π and PFD-fret-π show quite complementary strength and weaknesses in terms of per-domain coverage results. Taking a closer look, however, Prism-FRET-π is dominated by PFD-fret-π in all cases where the latter does not suffer from PFD's input reading. More specifically, PFD-fret-π outperforms Prism-FRET-π in all IPPC domains. In the Prism part, PFD-fret-π falls behind only in ContractSigning, Dice, MutualPnueliZuck, and Rabin. In all four domains, as we shall see in Section 9, PFD's grounding causes as significant overhead. For the sake of clarity, and since all other heuristic search configura-

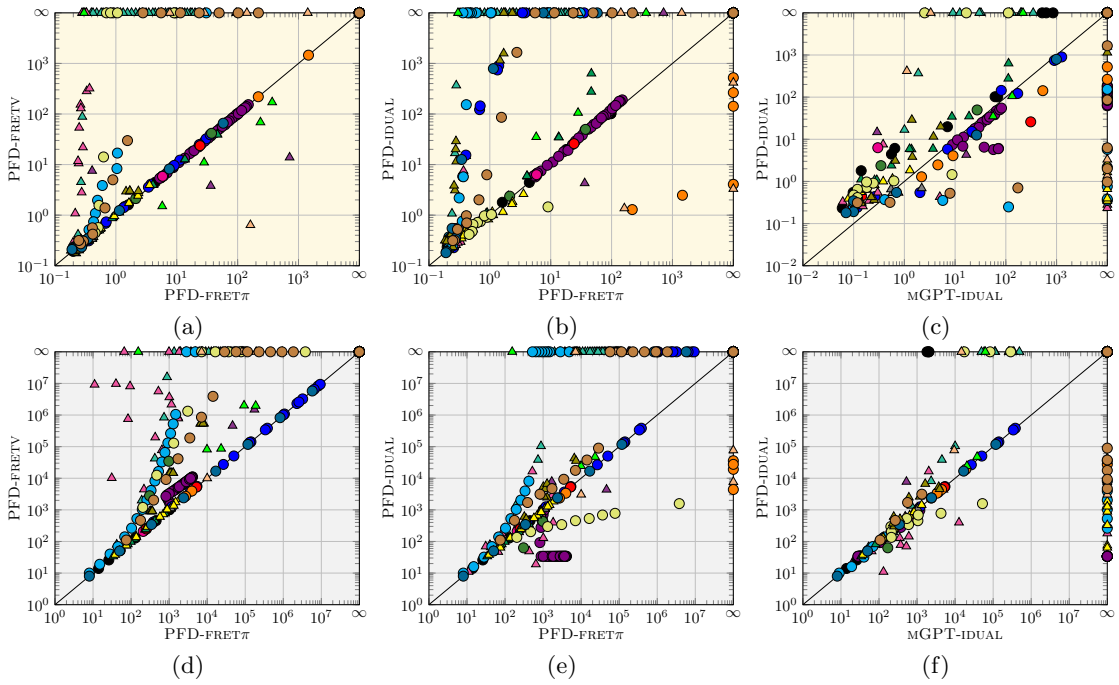


Figure 11: Per instance comparison of (a,b,c) runtime in seconds and (d,e,f) total number of different states touched before termination. "∞" entries mark instances that were not solved by the respective configuration.

tions suffer from the same issue anyway, we will use PFD's implementation as the base for comparisons to FRET-π in the following.

Consider PFD's FRET-V and FRET-π configurations. In terms of coverage, FRET-π clearly wins this comparison, solving more instances in 3 Prism domains and in 4 IPPC domains, and solving the same number of instances in the remaining domains. This result must of course be considered in the context of the heuristic being used: the trivial heuristic. FRET-V is known to perform poorly when run with uninformed heuristics. Nevertheless, our findings for the trivial heuristic carry over and directly apply to the other tested heuristics. We will stick to $h^\top$ in this comparison for clarity's sake and for consistency to the other discussions in this section.

Figure 11 compares the different configurations pairwise, based on the overall tool runtime and numbers of states touched before (regular) termination. As shown by Figure 11 (d), FRET-π's focus on the analysis of individual policies resulted in significant reductions of the numbers of considered states in many domains. This advantage usually translates into runtime, see Figure 11 (a). Largest differences in terms of all of our measures can be observed in DiningCryptographers, DiningPhilosophers, IsraeliJalfon, and the Tireworld domains. In all these cases, after the first call to LRTDP, the current goal-probability value of almost all non-terminal states was still 1. Consequently, FRET-V considered almost the entire reachable part of state space in its first trap analysis call. Yet it is not possible to reach the goal with absolute certainty in most Tireworld instances, lowering the stored value in

LRTDP for only a fraction of the reachable states still sufficed to make the greedy policy optimal. This allows FRET-π to terminate without visiting the whole state space. In the other three mentioned domains, the initial policy found by LRTDP was actually already optimal and did not contain any trap. Therefore, FRET-π could terminate already after just analyzing this single policy graph (much smaller than the complete state space). On the other hand, there are also some domains where FRET-π is not able to find the optimal solution while considering fewer states than FRET-V. In these cases, LRTDP usually had to visit at least once every reachable state in order to turn the greedy policy into an optimal one. This happens for example in PinCracking, where chances of reaching the goal was so small that LRTDP was forced to bring down the goal probability estimate of almost every reachable state to some extent. In some domains, such as CouponCollector or Dice, the models don't contain many non-deterministic choices that could be exploited by heuristic search. In such cases, LRTDP already visits all or at least large parts of the reachable states, canceling out the potential benefits of the FRET-π variant.

I-Dual Comparing PFD-I-DUAL and PFD-FRET-V, both perform very similar overall, though having strengths in different domains. For example, FRET-V solves more instances in CouponCollector, IsraeliJalfon, and Elevators. In contrast, I-DUAL has the clear lead in Consensus, DiningPhilosophers, and Tireworld. In Consensus, I-DUAL is actually able to beat even FRET-π, showing the best performance among the heuristic search configurations of Table 4 in this domain. In most of the domains, I-DUAL is however not competitive with FRET-π, solving considerably fewer instances. It should be noted that the I-Dual algorithm was originally designed for MDP multi-objective analysis. The flexibility of the LP encoding allows very easily to consider and compare multiple policies, different LP solutions, by adding additional LP constraints, or changing the objective function. For *MaxProb* analysis, this functionality is not really required. In fact, as the comparison of FRET-V and FRET-π already showed, it turns out that for *MaxProb* it is much more effective to focus on proving one particular policy to be optimal.

Figure 11 (b) and (e) compare PFD's I-DUAL and FRET-π engines based on runtime and states touched before converging to the solution. To limit the overhead associated with solving multiple linear programs, PFD-I-DUAL and MGPT-I-DUAL exploit that differences between the linear programs of two consecutive iterations are small: Warm starting the LP solver with the previous solution tremendously improves the overall solving process. Despite this optimization, the dependency on LPs still resulted in a severe runtime overhead. PFD-FRET-π usually finds a solution orders of magnitudes faster than PFD I-DUAL, even in cases where both configurations consider the same number of states (such as for example CouponCollector, PinCracking, and RectangleTireworld). A notable exception is Consensus. This domains exhibits a complex transition behavior, which shows in strong cyclic dependencies between the states' values. This causes LRTDP's value estimates to only converge slowly to the optimal value function, requiring many trials until termination. In contrast, PFD-I-DUAL suffers much less from this issue, the LP solver being able to deal efficiently with such variable dependencies.

In terms of the number of visited states, Figure 11 (b), the results are mixed with advantages on sides of both approaches. In most cases, differences must however be contributed to tiebreaking choices within the algorithms. The biggest gap between the two can be

observed in IsraeliJalfon, DiningPhilosophers, Firewire, and TriangleTireworld. All these domains have in common that the goal is reachable from the initial state with absolute certainty. Although not guaranteed by this property in general, LRTDP could stop after doing just single a trial, leaving the exploration of the initial greedy policy as termination check. Moreover, these policies didn't contain any trap, i.e., FRET-π actually only needed this single LRTDP call to derive the maximal goal-probability. Hence, the values in Figure 11 (b) plotted for those domains give exactly the number of states that are visited by the selected policy. PFD-I-Dual visiting more (or less) states then simply shows the existence of another optimal policy visiting more (or fewer) states. With a different initialization of the greedy policy, or different tiebreaking choices within the LP solutions, both could have followed the same policy as well.

To conclude this paragraph, we compare mGPT-i-dual and PFD-i-dual. Even though they have the same underlying algorithm, both configurations show complementary performances in many of the domains. In terms of coverage, this is most clearly visible in Firewire, IsraeliJalfon, and TriangleTireworld. The difference can principally stem from two reasons: low-level implementation details, and tie breaking in the execution of I-Dual. Both configurations being based on different tools, they naturally differ in the way PPDDL files are parsed, the state space is represented, and also in which LP solver is used. Regarding the second reason, I-Dual follows concrete LP solutions which are not necessarily unique in general. Therefore, multiple executions may differ depending on which solution was given by the LP solver. Within the individual tools, variance between different runs of the same configuration was however small. Figure 11 (c) and (f) compare the two implementations on a per instance basis using as measurements runtime and numbers of states touched before reaching the optimal solution. In terms of the state results, both configurations perform quite similarly on the commonly solved instances. This suggests that tie breaking within the LP solutions plays only a minor role in the cross tool comparison, too. On the other hand, runtime varies highly depending on the domain. Extreme examples are DiningCryptographers and IsraeliJalfon where PFD-i-dual is orders of magnitude faster than mGPT-i-dual. By contrast, in DiningPhilosophers and Random the exact opposite is true. These results must be attributed to the implementation specific differences between both planners. Across all domains, PFD offers a slightly better performing I-Dual implementation, yet, the overall advantage coming entirely from the Prism.

7.3 Cross Comparison

Both, explicit VI and the different heuristic search algorithms, use similar data structures to store and access state space information. Naturally, we start with a comparison of those two classes. We use PFD for this comparison, as it offers implementations of all the relevant approaches (although not necessarily the most efficient ones). We again use LRTDP as underlying algorithm of the FRET configurations and restrict consideration to the $h^\top$ heuristic. Consider Figure 12, which compares the three heuristic search approaches with PFD-tvi based on runtime and numbers of states touched before a solution was found. A per domain comparison of coverage results is included in Table 4. The following observations are reflected directly in this table.

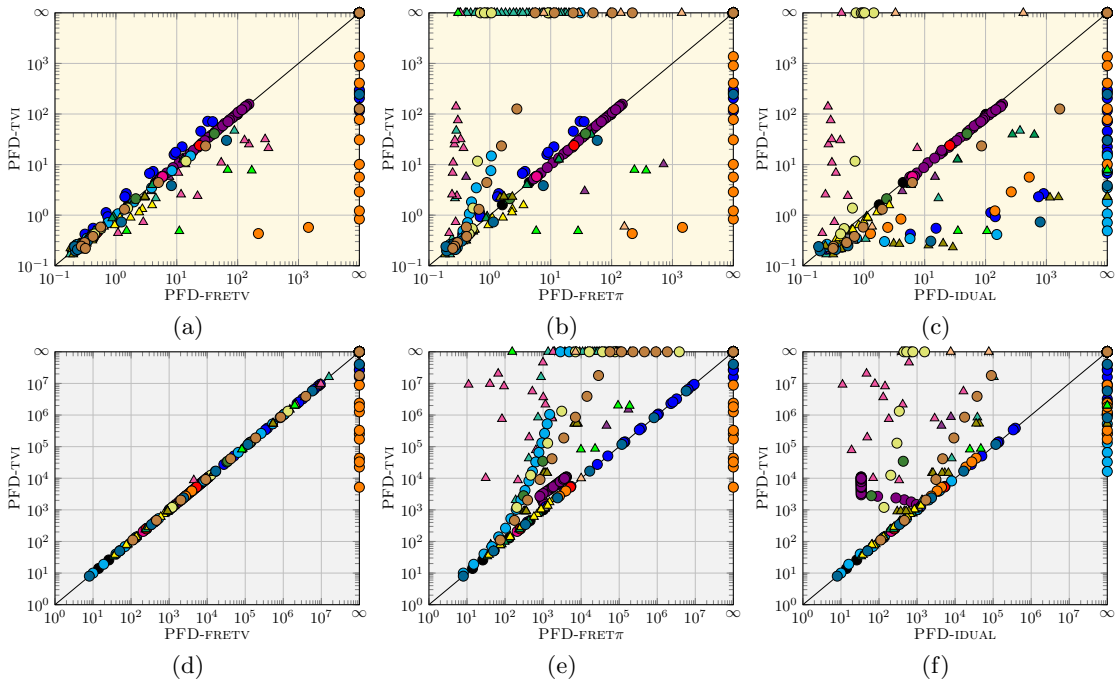


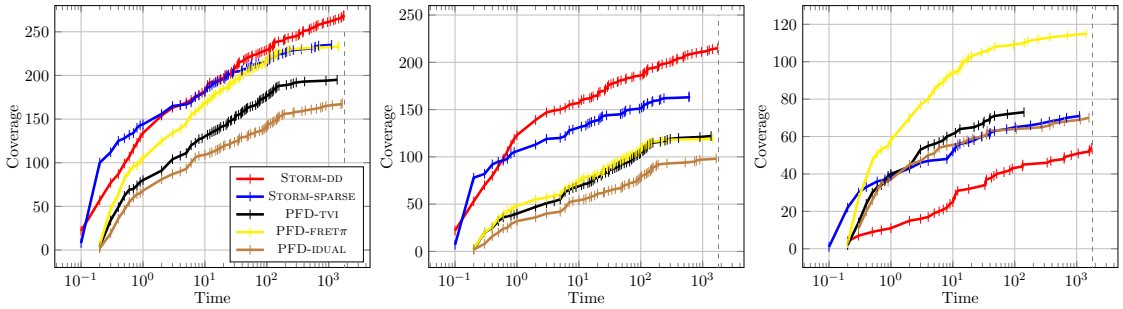
Figure 12: Per instance comparison of (a,b,c) runtime in seconds and (d,e,f) total number of different states touched before termination. "∞" entries mark instances that were not solved by the respective configuration.

Since initially the trivial heuristic does not lead to any restriction in the space of greedy policies, FRET-V expands the entire state space in the first FRET iteration. Hence, as shown in Figure 12 (d), FRET-V does not lead to any reduction compared to value iteration in terms of considered states. On the other hand, there is an overhead associated with FRET, which in some cases shows in runtime results. In Consensus, LRTDP was the main bottleneck, which required a substantial number of iterations of value updates until finally reaching a fix-point. Due to the consideration of $h^\top$, we somehow consider FRET-V's worst possible performance. Nevertheless, plugging in the other heuristics gives almost identical results to those shown in Figure 12 (a) and (d).

Let's turn to FRET-π, Figure 12 (b) and (e). As was shown already in the comparison with FRET-V, FRET-π can effectively restrict focus onto a subset of the reachable states in many domains. Most of the time, advantages in terms of considered states translate directly into runtime reductions. In cases where FRET-π could not reduce the considered states noticeably, runtime was usually no worse than TVI. There are however three exceptions: Consensus, ExplodingBlocksworld, and Zenotravel. The state spaces of these domains have lots of cycles that, due to LRTDP's and HDP's path-based value updates, require them to perform many such updates until eventually converging.

Consider next Figure 12 (c) and (f). Similar to FRET-π, I-DUAL is often able to find optimal solutions by touching only a fraction the reachable states. In terms of runtime however, I-DUAL does not look too good in comparison to TVI. I-DUAL is faster in only

DiningPhilosophers and Tireworld, where the difference between numbers of considered states was also particularly large. In many other domains, I-DUAL was several orders of magnitude slower than TVI.



(All) (PRISM) (IPPC)

Figure 13: Coverage as a function of overall time required to solve the instances for a selection of VI and heuristic search configurations. The three plots differ in the considered domains.

We close this section with the comparison of heuristic search and symbolic VI variants. This comparison particularly makes visible a separation between planning and model checking in our benchmark selection. Figure 13 compares performance considering coverage as a function of time. Table 4 breaks down these results to individual domains. Symbolic VI, here represented by STORM-DD, vastly outperforms the heuristic search configurations in all PRISM domains but DiningPhilosophers. In MutualPnueliZuck and Rabin, only PRISM's FRET-π implementation is able keep up with DD, the other heuristic search configurations suffer from previously mentioned issues. The lead of STORM-DD is because of the following. Firstly, for historical reasons, the PRISM benchmarks were particularly chosen and designed so that symbolic engines work well. This has become apparent already in the comparison between explicit and symbolic VI configurations. Secondly, many PRISM benchmarks heavily use numeric variables and operations (especially ContractSigning, FairExchange, MutualPnueliZuck, and Rabin). As we will show later on, translated into PPDDL, these however caused major troubles in PFD's grounding procedure, negatively affecting the performance of all PFD configurations. Yet, PRISM's heuristic search variant does not suffer from this issue, our previous analysis has shown that this implementation isn't necessarily the most efficient one. DiningPhilosophers, in contrast, does not rely on numeric operations. Moreover, heuristic search approaches work particularly well in this domain, e.g. LRTDP's first trial already finds the optimal policy. On the other hand, as the symbolic approaches were not really competitive to the explicit VI configurations in the IPPC domains, it is thus not surprising that PFD-FRET-π outperforms STORM-DD in almost every IPPC domain.

8. Evaluation II: Ablation Study

In the previous section, we have compared the different MDP goal probability analysis algorithms at principal level, ignoring the various optimization options and other configuration parameters. Here we catch up on this, showing whether and in how far the different

parameters impact the performance. We start with a comparison of the selected heuristic functions from PFD. For the sake of brevity, we will then pick the best-performing heuristic from this comparison to 1.) evaluate the dead-end state reduction method in VI, and 2.) to analyze what difference it makes to use dead-end detection heuristics for heuristic search. Along with 1.), we include in the analysis the precomputation option to identify 0- and 1-goal-probability states prior to performing value updates. We close this section with the evaluation of the remaining, mostly tool specific, options.

8.1 Heuristic Functions

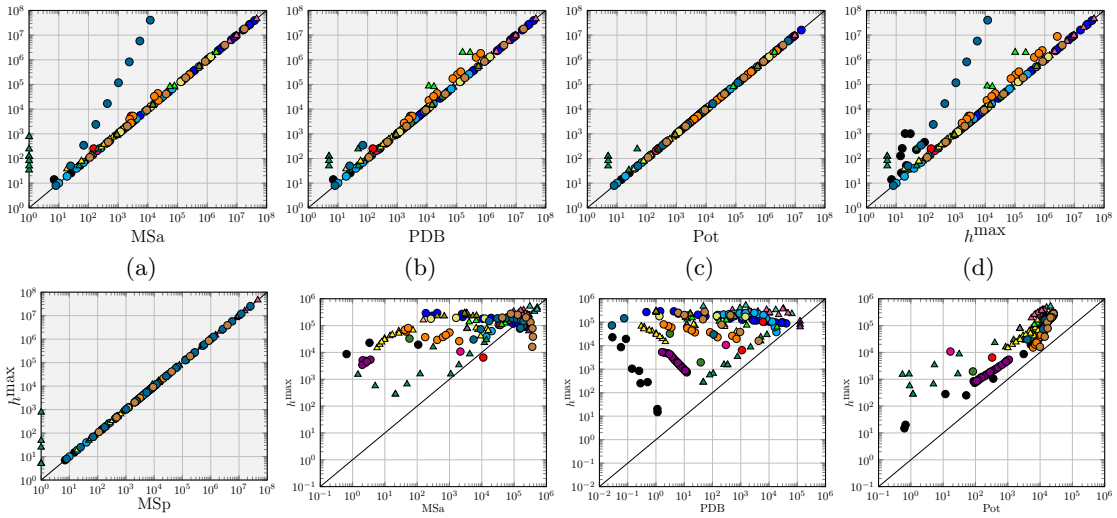


Figure 14: Comparison of the heuristics selected for PFD. Plots (a) – (d) compare total number of states with the number of states visited in exploration with dead-end pruning. (e) compares visited states between $h^{\max}$ and MSp. (f) – (i) compare the different heuristics based on the number of states evaluated per second.

We compare the different classical planning heuristics introduced in Section 6.2 independent of the actual MDP analysis algorithm. The observations made in this section are invariant under the used algorithm, i.e., they are directly reflected in the results of the actual PFD configuration runs. To evaluate a heuristic, we ran a forward exploration of the MDP state space using the heuristic for dead-end detection. If the heuristic recognizes a state as dead-end, we abort the traversal at this state and continue with the next state to be expanded. In other words, we treat recognized dead-ends as if they were terminal states. Similar to the other experiments, we set the time limit to 30 minutes and the memory limit to 4 GB. We evaluate the performance of a heuristic based on two measures: the number of states visited in the corresponding forward exploration compared to the number of states visited in the forward exploration without pruning dead-ends; the overhead of the heuristic functions is captured in terms of numbers of states for which the heuristic was computed

per second (the time includes the initialization of the heuristics' data structures). Figure 14 shows the results. MSp is not shown in this figure. Its results were very similar to MSa.

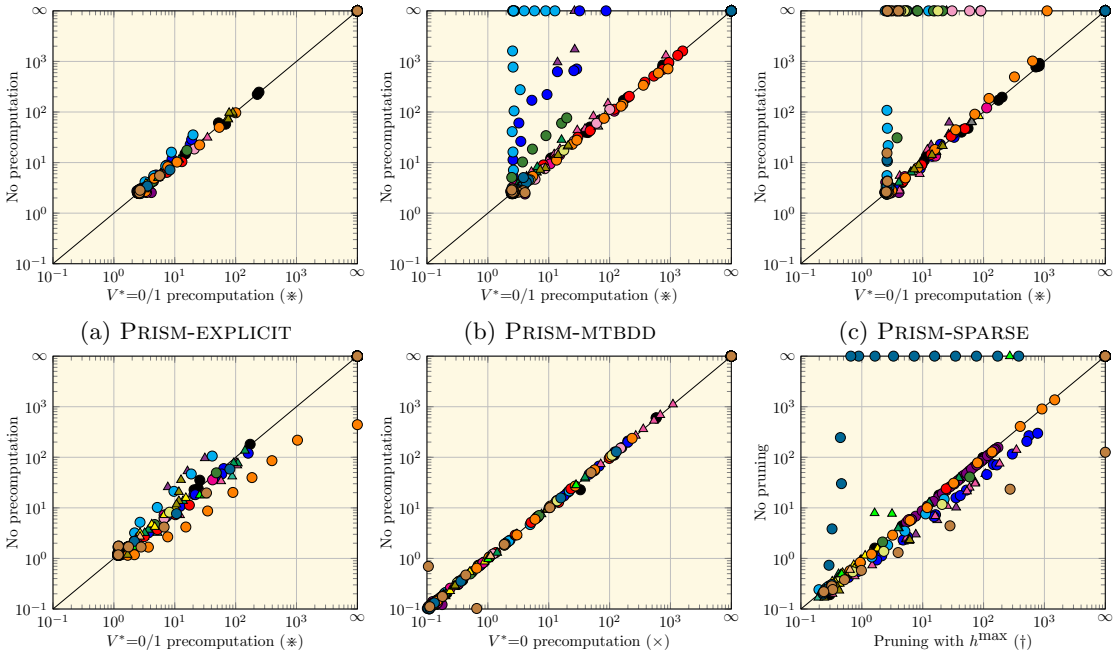
Consider Figure 14 (a) – (d). In most of the domains, none of the heuristics resulted in notable state space size reductions. The reason lies primarily in the benchmark selection. There are several domains which do not contain any dead-end at all, e.g., CouponCollector, DiningCryptographers, IsraeliJalfon, and Elevators. In other domains, a large fraction of the dead-ends, if not all, are actually terminal states, e.g., FairExchange, Tireworld, and TriangleTireworld. The domains where dead-end pruning lead to significant reductions in the number of visited states are Dice, PinCracking, RectangleTireworld, and Exploding-Blocksworld. Overall, Aidos' (Seipp et al., 2016) PDB and potential heuristics are the weakest dead-end detectors in this comparison. The effect of dead-end pruning using Pot is visible in only a single domain: RectangleTireworld. $h^{\max}$ was the only heuristic which achieved noticeable state reductions in all four aforementioned domains. In fact, as indicated by Figure 14 (e), in all domains but RectangleTireworld, $h^{\max}$ provided optimal dead-end information in the sense that it visited the smallest possible number of states considered using any one of the dead-end detection heuristics.

In addition to benefits in terms of quality of the estimations, $h^{\max}$ is also more efficient to compute than the other considered heuristics, see Figure 14 (f) – (i). The estimates of MSa and PDB are based on *abstractions* of the determinized state space, computed as part of the initialization of the heuristics. This initialization almost always caused a prohibitive runtime overhead. $h^{\max}$ does not need such expensive precomputation steps, but may require more time per heuristic evaluation. As depicted in Figure 14 (f) and (g), when counting both time spent on heuristic evaluation and the heuristic initialization time, $h^{\max}$ vastly outperforms MSa and PDB. $h^{\max}$ was also more efficient to compute than Pot, Figure 14 (i). This is also not very surprising, as the latter heuristic requires to solve a linear program in each heuristic evaluation.

8.2 Preprocessing Options for VI

Figure 15 compares runtime between the configurations that perform and do not perform any preprocessing step prior to VI. The performance of Storm's VI configurations was not affected by enabling the precomputation step. The Storm-sparse plot in Figure 15 is representative for all three of Storm's VI variants.

The precomputation of states with 0/1-goal-reachability probability had only a minor to no impact on the explicit VI configurations. Differences can only be observed for modest-Mcsta. In Consensus, modest's graph-analysis added a significant overhead while the precomputation brought almost no reduction in the number of VI iterations required until convergence. In IsraeliJalfon and Zenotravel, Mcsta with the precomputation option was slightly faster. The instances of both domains have goal-reachability probability of 1, i.e., in both domains Mcsta-$*$ can terminate directly after the state-space graph analysis without even performing any VI operation at all. In contrast to explicit VI, the hybrid and symbolic variants benefit more from the precomputation of 0- and 1-goal-reachability probability states. The precomputation configurations of Prism-mtbdd and Prism-sparse significantly outperform their non-precomputation counterparts in several domains. Due to more expensive numerical operations on the symbolic data structures, the impact of



(a) Prism-explicit (b) Prism-mtbdd (c) Prism-sparse

(d) modest-Mcsta (e) Storm-sparse (f) PFD-tvi

Figure 15: Runtime (in seconds) comparison of VI configurations with versus without preprocessing optimizations. "∞" represents instances not solved within the limits.

the reduction of the number of such operations is particularly visible for Prism-mtbdd. Runtime could be substantially reduced in CouponCollector, IsraeliJalfon, and Zenotravel. The runtime advantage also translated into coverage results for CouponCollector (+2 solved instances) and IsraeliJalfon (+9). Prism-sparse does not primarily benefit from the reduction of numerical operations, but rather from precomputation of the 1-goal-probability states itself. The precomputation step in Prism-sparse is solely based on a symbolic analysis of the MDP state space. Runtime and coverage results could be significantly improved in DiningCryptographers (+8 instances solved), DiningPhilosophers (+5), IsraeliJalfon (+6), and Rabin (+6). In all four domains, the goal-reachability probability is 1. Thus, Prism-sparse-⁕ terminates directly after the computation of the 1-goal-probability states, in particular avoiding the initialization of the explicit data structures that would be required for value propagation. On the contrary, Storm's precomputation option only supports the identification of 0-goal-probability states. As exemplified by Figure 15 (e), the performance of none of Storm's VI variants is affected by this preprocessing step. This is in line with the observations made for modest and Prism, whose preprocessing options made only a difference in domains with many 1-goal-probability states.

Finally, consider Figure 15 (f). The previous section has shown that the number of considered states can be reduced via dead-end pruning in only a couple of domains. The heuristic evaluations however always induce a considerable overhead, which is reflected in the runtime plots in Figure 15. The overhead induced by $h^{\max}$ shows particularly dominant in DiningCryptographers, where dead-end pruning does not have any effect (all states being

able to reach the goal). On the other hand, in PinCracking the number of considered states could be reduced tremendously, cf. Figure 14 (d). This also shows in the runtime comparison, PFD-TVI with $h^{\max}$ dead-end-pruning being several orders of magnitude faster than PFD-TVI without such pruning. In terms of coverage, PFD-TVI-$h^{\max}$ solved more instances in PinCracking (+10) and ExplodingBlocksworld (+1), and less instances in DiningCryptographers (−1), compared to PFD-TVI.

8.3 Heuristic Search with Dead-End Detection Heuristics

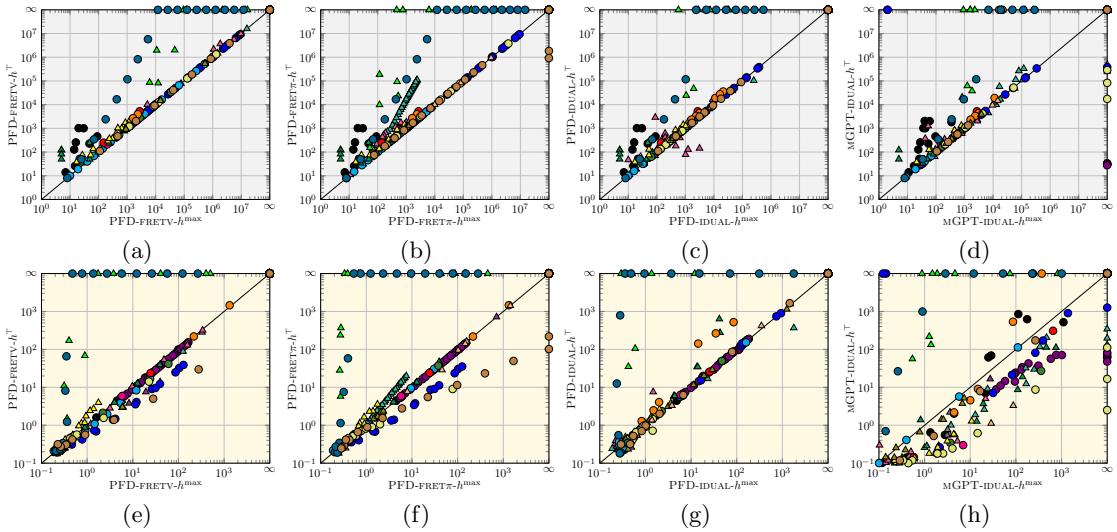


Figure 16: Per-instance comparison of the different heuristic search configurations using the trivial heuristic $h^{\top}$ versus $h^{\max}$ for dead-end detection. Plots (a) – (d) show number of states visited before termination. Plots (e) – (h) compare runtime (in seconds). "∞" indicates instances that could not be solved within the limits.

Figure 16 shows a per-instance comparison of the trivial heuristic $h^{\top}$ and $h^{\max}$ for dead-end detection for the various heuristic search methods. The FRET configurations again deploy LRTDP as underlying heuristic search algorithm. The results for HDP are similar.

In terms of state reduction, Figure 16 (a) – (d), the plots look similar to what we have observed before. Dead-end pruning with $h^{\max}$ again leads to considerable state reductions in PinCracking, ExplodingBlocksworld and RectangleTireworld. What is new here is that the heuristic search algorithms can also benefit from the heuristic even when only terminal states are identified as dead-ends. This particularly shows in Tireworld and TriangleTireworld. In these cases, the heuristic still provides a one-step lookahead for the identification of terminal state. This can be exploited by the heuristic search algorithms to immediately discard actions from consideration whose application may lead to a terminal state recognized by the heuristic. This in turn may result in disregarding other, potentially non-terminal, successor states stochastically reached via the skipped actions, as well.

Regarding runtime, the overhead associated with the heuristic however makes void state reduction advantages in almost all cases, cf. Figure 16 (e) – (h). The $h^{\max}$ configurations are slightly ahead of their $h^{\top}$ counterparts mainly in PinCracking and ExplodingBlocksworld. In both domains, the state reduction achieved by $h^{\max}$ was particularly large. In CouponCollector and DiningCryptographers, $h^{\max}$ doesn't lead to any state reduction, yet its evaluation results in a significant slow down of the respective configurations.

Yet to a smaller extend, similar observations are visible in the coverage results: The results for PFD's $h^{\max}$ configurations differ only in DiningCryptographers (in favor of $h^{\top}$), PinCracking (in favor of $h^{\max}$) , and ExplodingBlocksworld ($h^{\max}$) . The effect of $h^{\max}$ on mGPT-i-dual is similar. $h^{\max}$ is beneficial primarily in Consensus, CouponCollector, PinCracking, and ExplodingBlocksworld, where also dead-end pruning resulted in a noticeable state reductions. As opposed to this, $h^{\max}$ purely constitutes a runtime overhead in DiningPhilosophers, Firewire, and TriangleTireworld, which is reflected in a considerable coverage decrease.

8.4 Caching in Hybrid-VI

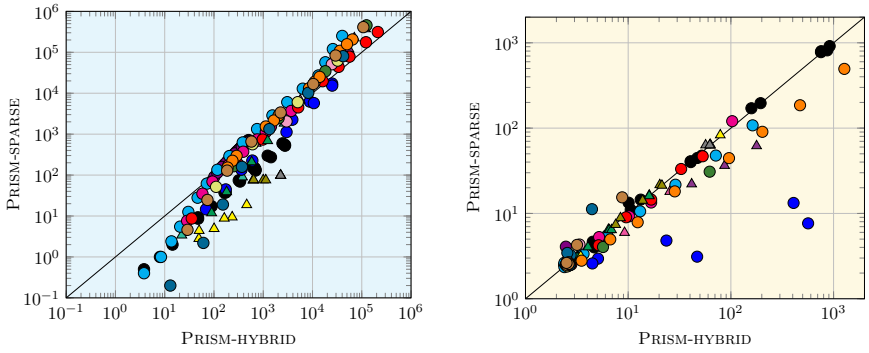


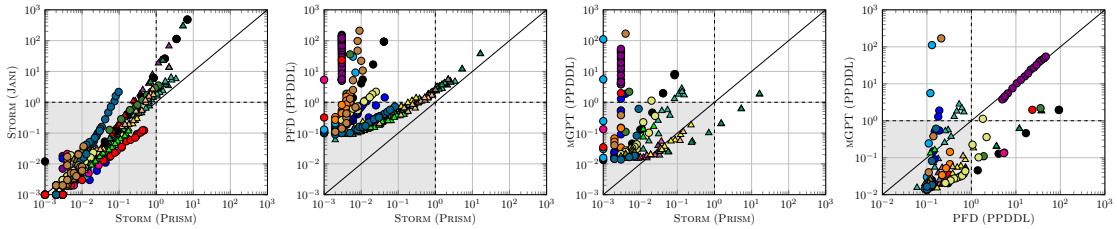
Figure 17: Comparison of Prism's sparse and hybrid engines based on commonly solved instances: (left) peak memory usage in KB, (right) runtime in seconds.

We finally investigate Prism's ability to speed up hybrid VI by explicitly constructing parts of the transition matrix. Figure 17 compares Prism-sparse and Prism-hybrid, both run without the 0/1-goal-probability state precomputation option. Overall, differences between the two configurations were negligible in terms of all our measures (runtime, memory, and coverage). Specifically, consider the left plot (memory usage). The non-symbolic storage of parts of the transition matrix in Prism-hybrid is reflected in memory usage only in a few cases, the symbolic state space occupying the biggest fraction of the used memory. Differences can be observed primarily for instances with (comparatively) small state spaces, where both engines don't require much (in the region below 10 MB) memory anyway. In terms of runtime (right plot), notable differences can be observed primarily in Consensus, IsraeliJalfon, and Tireworld, Prism-sparse actually outperforming Prism-hybrid. In these cases, caching parts of the transition probability matrix does not noticeably reduce

the computational cost of numeric operations, while the extraction from the symbolic data structures still incurs an overhead.

9. Evaluation III: Input Reading and Processing

We close our empirical evaluation with the investigation of a non-algorithmic specific aspect. In the previous sections, we have evaluated the performance of the different algorithms by comparing tools that possibly read models in different input languages. The different modeling languages however require different, more or less, expensive methods to read, parse, and process the input files. This could potentially bias the comparison. In this section, we consider this issue more closely. MODEST and PRISM do not report any statistics about how much time was spent on processing the input files. In the following, we will thus report results only for STORM, PFD, and MGPT. For PFD and MGPT, we consider the time required to ground the PPDDL models, in addition to parsing the input files. JANI and PRISM do not require such operation. The input processing time for STORM consists of reading the model files, and mapping this to STORM's internal model representation. Figure 18 shows the results.



(a) (b) (c) (d)

Figure 18: Per-instance comparison of time (in seconds) spent on processing the input files.

Figure 18 (a) compares the time required to read and parse JANI and PRISM models in STORM. STORM reads most of the instances within a split second, without notable difference between the two modeling languages. There are a few exceptions. Despite JANI's design principle of being easy to parse, there are only two domains where STORM is actually able to read JANI models faster than the corresponding PRISM variants: ContractSigning and FairExchange. In both domains, all instances could however been read within less than one second. When comparing such small numbers, other non-tool related factors may become the decisive factor, making it hard to draw any final conclusion from them. On the other hand, however, there are several domains where processing JANI required significantly more time. This can be observed especially in Dice, RectangleTireworld, and Zenotravel. Since all three domains contain models that take up to (or even more) than 10 seconds to read, with differences between the encodings of several orders of magnitude, the dominance of the PRISM encodings there becomes much more evident. The automatic translation from PRISM into JANI could be a possible explanation. However, since the PRISM files in RectangleTireworld and Zenotravel are constructed from JANI, the PRISM to JANI translation cannot be the only reason. Even though the JANI files are generally larger than the PRISM ones, the size difference in aforementioned domains is not much different

from other domains (cf. Figure 6 in Section 5). It remains open what exactly is causing the performance difference in those domains.

Figures 18 (b) and (c) compare Prism and PPDDL input processing. The planners and Storm yield almost the same results for the IPPC models. Differences can be observed almost only for instances that were processed within one second where, as above, we abstain from drawing any concrete conclusion. The picture however changes completely for the Prism part of our benchmark suite. There both planners' input processing turned out to be consistently slower than Storm. Although many of the sample points still fall into the "≤ 1 seconds" region (the gray areas), there also exist several domains where one can observe a significant deviance. A striking example is Firewire. While the time measured for Storm's input reading remains constant over all instances, the planners exhibit an exponential behavior. The reason for the latter lies in grounding. The Firewire models represent a (discrete) clock via a bounded integer variable that comes with a comparatively large domain. The time period considered, and thus the domain size of this variable, varies between the different Firewire instances. Figures 18 (b) and (c) demonstrate the overhead of encoding the clock variable into a propositional model. Grounding also caused a significant overhead in Dice, DiningCryptographers, and IsraeliJalfon. The explanation provided for Firewire also applies to Dice. IsraeliJalfon is a model of a well-known self-stabilizing algorithm (Israeli & Jalfon, 1990). Different instances scale in the number of processes considered. The PCTL property asks for the probability that the network of processes eventually reaches a stable configuration. To this end, the property is accessing some status variable of every process. In our straightforward compilation into the PPDDL goal action schema, this leads to the introduction of action schema parameters whose number scales polynomially in the number of processes. Grounding this action led to the exponential explosion that can be observed in the comparison plots. The same issue appears in DiningCryptographers.

Figure 18 (d) compares mGPT and PFD. mGPT has an advantage for the instances processed within less than 1 second. Larger differences can be observed in CouponCollector, Dice, IsraeliJalfon, TriangleTireworld, and RectangleTireworld, without clear advantage on either side. The extreme cases are IsraeliJalfon, where mGPT's PPDDL processing is up to 3 orders of magnitude slower than PFD's, and Dice and RectangleTireworld where PFD is almost consistently 2 orders of magnitude slower than mGPT. The differing performance must be attributed to the different grounding algorithms used by both planners.

As we have seen above, the worst-case size explosion of the Jani to PPDDL translation can indeed be observed in a few domains. To examine in how far this affects the runtime of the overall solver configurations, as presented in the previous sections, Figure 19 provides information of the relation of the input processing time to the overall runtime. We chose one representative configuration for each of the three tools. The reported data for a domain and solver configuration are average values over all runtimes and input processing times for those instances which 1) were solved by the respective configuration, and 2) the configuration took longer than 1 second to solve them. We do 2) to reduce bias towards smaller values, assuming that harder instances yield more meaningful results. Note that the data entries of different configurations may include different instances. Since loading Prism models takes usually only a split second, it is not very surprising that input processing takes up only a small fraction of the overall runtime of Storm-dd. Consider the probabilistic planners. The

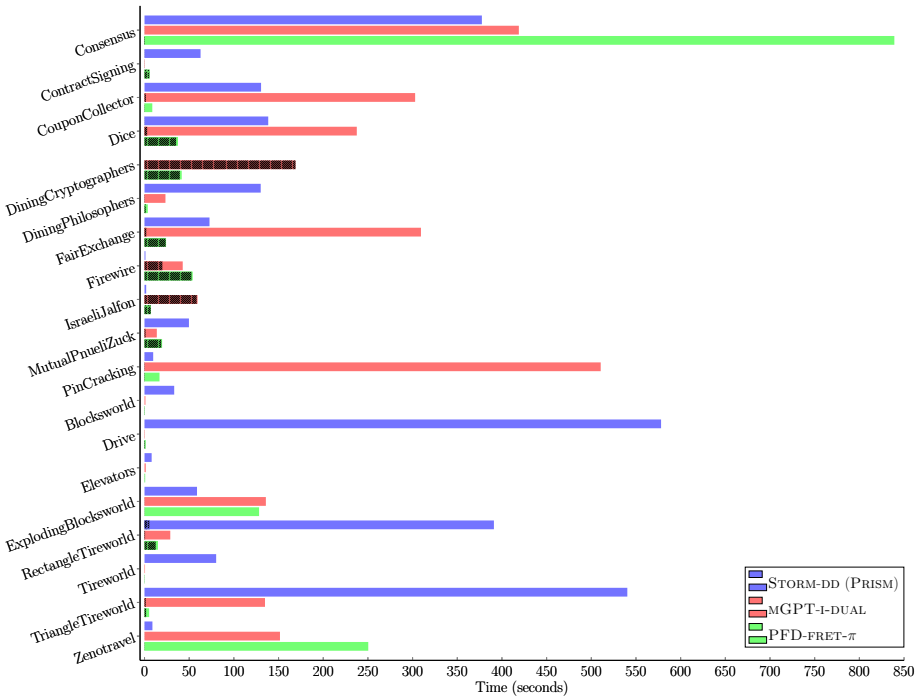


Figure 19: Per-domain average overall tool runtime and input processing time (striped).

overall picture looks again completely different between the IPPC domains and the domains from the Prism benchmark suite. Among the former, mGPT-i-dual's input reading is not noticeable in terms of total runtime at all. For PFD-fret-π, PPDDL processing shows in overall runtime only in RectangleTireworld. On the contrary, in almost half of the Prism domains, mGPT's and PFD's PPDDL grounding takes actually more time than solving the models. In DiningCryptographers and IsraeliJalfon both planners, and in ContractSigning, Dice, FairExchange, Firewire, and MutualPnueliZuck, PFD-fret-π, spend on average even more than 90% of their runtime on just reading the models.

10. Related Work

We presented a compilation from the probabilistic planning domain definition language (Younes & Littman, 2004) into the widely used model checking language Jani (Budde et al., 2017). In a similar fashion Edelkamp (2003) introduced a compilation of the verification of safety properties in protocols modeled in the Promela language (Holzmann, 2004) into classical planning. In difference to our approaches, that compilation is designed to broadly cover Promela, and is not structure-preserving as it requires the introduction of several artificial state variables and actions for each individual Promela automaton transition.

Baier and McIlraith (2006) considered classical planning with goals expressed via a finite state-sequence variant of LTL (Pnueli, 1977). Patrizi, Lipovetzky, De Giacomo, and Geffner (2011) have shown how to use standard classical planning tools in order to analyze general LTL formulas, considering particularly the fraction of LTL concerning infinite

sequences of states. In this paper, we concentrated on reachability analysis (Baier et al., 2018) of *MaxProb* properties (Kolobov et al., 2011; Teichteil-Königsbuch, 2012; Trevizan et al., 2017).

In the context of probabilistic planning, some works exploited temporal logics in order to encode non-Markovian reward functions (Thiébaux, Gretton, Slaney, Price, & Kabanza, 2006; Camacho et al., 2017; Brafman et al., 2018). To be able to still use existing planning algorithms, compilations back to standard formalisms are considered. Teichteil-Königsbuch (2012) has presented a very general class of probabilistic planning models, incorporating formalisms from probabilistic model checking. Due to this generality, however, the models fall out of scope of existing algorithms. Later works (Sprauel et al., 2014) proposed simplifications to that framework in favor of computational complexity, yet the analysis of the resulting models still requires the use of specifically designed methods.

On the algorithmic side in planning MDP heuristic search (Barto et al., 1995; Hansen & Zilberstein, 2001; Bonet & Geffner, 2003b) is a class of algorithms particularly tailored for reachability analysis. In this context Brázdil et al. (2014) applied and extended the heuristic search algorithm BRTDP (McMahan et al., 2005) in probabilistic model checking, showing promising results in their preliminary experiments. Baumgartner et al. (2018) have adapted the I-Dual algorithm (Trevizan et al., 2017) for the purpose of probabilistic automata synthesis, outperforming existing methods in several domains. Other heuristic search algorithms are often only applicable to stochastic-shortest path problems (SSP) (Bertsekas & Tsitsiklis, 1996). To lift the application of SSP heuristic search algorithms to a more general class of MDPs, including particularly *MaxProb* MDPs, Kolobov et al. (2011) proposed the FRET framework, which we presented above and considered in the evaluations.

PROST (Keller & Eyerich, 2012) is a probabilistic planner based on Monte-Carlo tree search. It does not handle goal probability maximization though.

On the model checking side the explicit construction of the MDP's state space is often stored compactly by using *symbolic* data structures (Miner & Parker, 2004; Kwiatkowska et al., 2004). This approach is also implemented in some planners, for example in SPUDD (Hoey, St-Aubin, Hu, & Boutilier, 1999), a stochastic planner using value iteration in combination with decision diagrams on factored MDPs, where dynamic programming is applied without enumerating the whole state space. In addition, there are symbolic extensions of Monte Carlo search (e.g., in Cui & Khardon, 2016). This technique is used for online stochastic planning for problems with large factored state and action space. But these approaches have not been used for goal-probability analysis.

A lot of research in the area of symbolic MDP probabilistic planning has also been done by Sanner et. al. They used symbolic dynamic programming (Sanner & Kersting, 2017; Vianna, de Barros, & Sanner, 2015) for parameterized hybrid MDPs (Kinathil, Soh, & Sanner, 2017a, 2017b) and developed a bounded approximated version thereof (Vianna, Sanner, & de Barros, 2013), where the intractable growth of extended algebraic decision diagrams is avoided by the use of a bounded error compression technique. Symbolic dynamic programming is also used for continuous state and observation POMDPs (partially-observable MDPs) (Zamani, Sanner, & Fang, 2012), where it is shown that exact symbolic dynamic programming solutions for continuous state MDPs can be generalized to continuous state POMDPs with discrete observations. In a comparable way there is also an implementation of symbolic dynamic programming with XADDs on various discrete and continuous MDPs

(Sanner, Delgado, & de Barros, 2012). But these approaches again have not been applied to goal-probability analysis which is the focus of our work.

Furthermore, there are not only symbolic versions of value iteration but also symbolic approaches for policy iteration in the planning community. For example symbolic opportunistic policy iteration for factored-action MDPs, is an approach which lies between VI and modified policy iteration (Raghavan, Khardon, Fern, & Tadepalli, 2013). But, again, these approaches have not been applied to goal-probability analysis.

11. Conclusion

We have established a new connection between probabilistic model-checking and probabilistic planning, both at the level of models, as well as at the level of algorithms. At model level, we developed translations from Jani to PPDDL and back. These compilations made it possible to create a common benchmark set, providing planning systems access to quantitative model checking models, and vice versa, making available the IPPC benchmarks to the model checking community. At algorithm level, we used this benchmark set to empirically compare various techniques developed by both communities, heuristic search methods and value iteration variants. The empirical results demonstrate once more that performance depends, as always, on the domain. The heuristic search algorithms could often find an optimal solution, taking into account only a small fraction of the overall state space. This was possible even with just using the trivial *MaxProb* heuristic. By using more informative heuristics, in terms of dead-end identifiers, the results could be slightly improved in some domains. In many other domains, dead-end identification however turned out to be not very helpful or even detrimental. On the other hand, the model checkers follow a different approach. Instead of reducing the number of states to be considered, they try to represent the entire state space as compactly as possible. The resulting symbolic VI variants provide state-of-the-art performance in many domains.

In the comparison of planning versus model checking techniques, probabilistic heuristic search algorithms turned out to be competitive in several model checking benchmarks, while the model checker's VI variants can also compete with the probabilistic planners in some IPPC domains. In an aggregated view, however, in each communities' own benchmark set, the tools of the respective community outperform the tools of the other community. The reason can be attributed to conceptual differences in the types of models usually considered in each community. On the one hand, historically, the planning models are strongly centered around propositional logic. Although PDDL and PPDDL were extended by the support of numeric fluents, the considered planners do not support these. On the other hand, many of the model checking domains make heavily use of numeric variables and operations. Although these can in principle be compiled into a language fragment supported by the planner, this translation caused a severe overhead in the planner execution. The performance of the model checkers on the other side crucially depends on structural information already provided through the input description. This structural information, if existing at all, is however not readily available for the planning benchmarks.

To foster compilability of model-checking models into probabilistic planning, support for numeric state variables, and for more general goals, would be desirable. Since the planners' grounding operation often constitutes a significant portion of the entire tool execution,

this moreover suggests the development of hybrid methods between grounding and lifted methods in planning, combining the benefits of both sides.

The more promising approach, though, is to instead port probabilistic heuristic search algorithms to model checking tools, to obtain a native realization tackling all the syntactic elements that cannot be easily compiled. In this sense, our research motivates a large area of cross-fertilization, pertaining to the exchange of algorithms. We already went into this direction by implementing FRET-π in the Modest Toolset and comparing it against the classical tools in QComp 2019 (Hahn et al., 2019). For QComp 2020 the implementation has even been extended towards property types beyond *MaxProb*. A major challenge for future work in this area will be the extension of probabilistic heuristic search to the analysis of general temporal formulas. As dead-end detection heuristics were beneficial to heuristic search in only a few domains, more informed *MaxProb* heuristics would be desirable. In this context, the integration of probabilistic heuristic search into probabilistic model checkers would also provide access to the wealth of abstraction techniques developed by the latter community.

Solvers based on symbolic data structures have been used very successfully in optimal classical planning (Torralba, Alcázar, Kissmann, & Edelkamp, 2017). This together with our empirical results suggest the adoption of the model checker's symbolic VI variants into the algorithm portfolio of goal probability analysis in probabilistic planning. A major challenge here is the automatic detection of structural properties of the models that can be efficiently exploited via those data structures. The focus on reachability properties may provide another source of planning model tailored optimizations to the symbolic engines.

The lessons learnt from the study of this paper can be summarized in the following list:

- Performance of model checkers and planners depends as always on the domain structure.
- Many model checking benchmarks consist of multiple largely independent components that generate the same transition behavior, which is exploited by symbolic engines. Planning benchmarks often do not contain such a structure.
- Techniques from one community can achieve state-of-the-art performance in benchmarks of the other community.
- Performance comparison on benchmarks is however in favor of the solvers and algorithms of that particular community. This means:
  - We should broaden tools' scope to make them efficient on both benchmark types.
  - VI enhancements from model checking could be useful in planning and heuristic search algorithms from planning that could be useful in model checking
- More efficient support of numeric variables and operations in planning is needed.
- Hybrid methods between grounding and lifted methods in planning would combine the benefits of both sides.
- Porting probabilistic heuristic search algorithms to model checking tools would be beneficial.

Acknowledgments

Parts of this work were performed while Marcel Steinmetz was employeed by the CISPA Helmholtz Center for Information Security. This work was partially supported by the ERC Advanced Investigators Grant 695614 (POWVER), by DFG grant 389792660 as part of TRR 248 (see `https://perspicuous-computing.science`), and by the German Federal Ministry of Education and Research (BMBF) under grant No. 16KIS0656.

References

Akers, S. B. (1959). On a theory of boolean functions. *Journal of the Society for Industrial and Applied Mathematics*, *7*(4), 487–498.

Baier, C., de Alfaro, L., Forejt, V., & Kwiatkowska, M. (2018). Model checking probabilistic systems. In *Handbook of Model Checking*, pp. 963–999. Springer.

Baier, C., Haverkort, B. R., Hermanns, H., & Katoen, J. (2003). Model-checking algorithms for continuous-time markov chains. *IEEE Trans. Software Eng.*, *29*(6), 524–541.

Baier, J. A., Bacchus, F., & McIlraith, S. A. (2009). A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence*, *173*(5-6), 593–618.

Baier, J. A., & McIlraith, S. A. (2006). Planning with first-order temporally extended goals using heuristic search. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pp. 788–795.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artif. Intell.*, *72*(1-2), 81–138.

Baumgartner, P., Thiébaux, S., & Trevizan, F. (2018). Heuristic Search Planning With Multi-Objective Probabilistic LTL Constraints. In *Proc. of 16th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*.

Bertsekas, D., & Tsitsiklis, J. (1996). *Neurodynamic Programming*. Athena Scientific.

Bonet, B. (2006). Bounded branching and modalities in non-deterministic planning. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, pp. 42–51.

Bonet, B., & Geffner, H. (1999). Planning as heuristic search: New results. In *Recent Advances in AI Planning, 5th European Conference on Planning, ECP'99, Durham, UK, September 8-10, 1999, Proceedings*, pp. 360–372.

Bonet, B., & Geffner, H. (2003a). Faster heuristic search algorithms for planning with uncertainty and full feedback. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pp. 1233–1238.

Bonet, B., & Geffner, H. (2003b). Labeled RTDP: improving the convergence of real-time dynamic programming. In Giunchiglia, E., Muscettola, N., & Nau, D. S. (Eds.), *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy*, pp. 12–21. AAAI.

Bonet, B., & Geffner, H. (2005). mgpt: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research*, *24*, 933–944.

Brafman, R. I., Giacomo, G. D., & Patrizi, F. (2018). LTL_f / LDL_f non-markovian rewards. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence AAAI*.

Brázdil, T., Chatterjee, K., Chmelik, M., Forejt, V., Kretínský, J., Kwiatkowska, M. Z., Parker, D., & Ujma, M. (2014). Verification of markov decision processes using learning algorithms. In *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, pp. 98–114.

Budde, C. E., Dehnert, C., Hahn, E. M., Hartmanns, A., Junges, S., & Turrini, A. (2017). JANI: quantitative model and tool interaction. In Legay, A., & Margaria, T. (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, Vol. 10206 of *Lecture Notes in Computer Science*, pp. 151–168.

Camacho, A., Chen, O., Sanner, S., & McIlraith, S. A. (2017). Decision-making with non-markovian rewards: From LTL to automata-based reward shaping. In *Proceedings of the Multi-disciplinary Conference on Reinforcement Learning and Decision Making (RLDM)*, pp. 279–283.

Ciesinski, F., Baier, C., Größer, M., & Klein, J. (2008). Reduction techniques for model checking markov decision processes. In *Fifth International Conference on the Quantitative Evaluaiton of Systems (QEST 2008), 14-17 September 2008, Saint-Malo, France*, pp. 45–54.

Cui, H., & Khardon, R. (2016). Online symbolic gradient-based optimization for factored action mdps. In Kambhampati, S. (Ed.), *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pp. 3075–3081. IJCAI/AAAI Press.

Dai, P., Mausam, Weld, D. S., & Goldsmith, J. (2011). Topological value iteration algorithms. *Journal of Artificial Intelligence Research*, *42*, 181–209.

Dehnert, C., Junges, S., Katoen, J., & Volk, M. (2017). A storm is coming: A modern probabilistic model checker. In Majumdar, R., & Kuncak, V. (Eds.), *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, Vol. 10427 of *Lecture Notes in Computer Science*, pp. 592–600. Springer.

d'Epenoux, F. (1963). A probabilistic production and inventory problem. *Management Science*, *10*, 98–108.

Edelkamp, S. (2003). Promela planning. In Ball, T., & Rajamani, S. (Eds.), *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN-03)*, pp. 197–212, Portland, OR. Springer-Verlag.

Edelkamp, S. (2006). On the compilation of plan constraints and preferences. In Long, D., & Smith, S. (Eds.), *Proceedings of the 16th International Conference on Auto-*

mated Planning and Scheduling (ICAPS'06), pp. 374–377, Ambleside, UK. Morgan Kaufmann.

Edelkamp, S., & Hoffmann, J. (2004). PDDL2.2: The language for the classical part of the 4th International Planning Competition.. Tech. rep. 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

Eisentraut, C., Hermanns, H., & Zhang, L. (2010). On probabilistic automata in continuous time. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pp. 342–351.

Forejt, V., Kwiatkowska, M. Z., Norman, G., & Parker, D. (2011). Automated verification techniques for probabilistic systems. In *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, pp. 53–113.

Fox, M., & Long, D. (2011). PDDL2.1: an extension to PDDL for expressing temporal planning domains. *CoRR*, *abs/1106.4561*.

Fujita, M., McGeer, P. C., & Yang, J. C. (1997). Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, *10*(2/3), 149–169.

Gazen, B. C., & Knoblock, C. (1997). Combining the expressiveness of UCPOP with the efficiency of Graphplan. In Steel, S., & Alami, R. (Eds.), *Proceedings of the 4th European Conference on Planning (ECP'97)*, pp. 221–233. Springer-Verlag.

Gerevini, A., Haslum, P., Long, D., Saetti, A., & Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, *173*(5-6), 619–668.

Hahn, E. M., Hartmanns, A., Hensel, C., Klauck, M., Klein, J., Kretínský, J., Parker, D., Quatmann, T., Ruijters, E., & Steinmetz, M. (2019). The 2019 comparison of tools for the analysis of quantitative formal models - (qcomp 2019 competition report). In Beyer, D., Huisman, M., Kordon, F., & Steffen, B. (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, Vol. 11429 of *Lecture Notes in Computer Science*, pp. 69–92. Springer.

Hahn, E. M., Hartmanns, A., Hermanns, H., & Katoen, J. (2013). A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods in System Design*, *43*(2), 191–232.

Hahn, E. M., Li, Y., Schewe, S., Turrini, A., & Zhang, L. (2014). iscasmc: A web-based probabilistic model checker. In Jones, C. B., Pihlajasaari, P., & Sun, J. (Eds.), *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, Vol. 8442 of *Lecture Notes in Computer Science*, pp. 312–317. Springer.

Hansen, E. A., & Zilberstein, S. (2001). Lao*: A heuristic search algorithm that finds solutions with loops. *Artif. Intell.*, *129*(1-2), 35–62.

Hansson, H., & Jonsson, B. (1994). A logic for reasoning about time and reliability. *Formal Asp. Comput.*, *6*(5), 512–535.

Hartmanns, A., & Hermanns, H. (2014). The modest toolset: An integrated environment for quantitative modelling and verification. In Ábrahám, E., & Havelund, K. (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, Vol. 8413 of *Lecture Notes in Computer Science*, pp. 593–598. Springer.

Hartmanns, A., & Hermanns, H. (2015). Explicit model checking of very large MDP using partitioning and secondary storage. In Finkbeiner, B., Pu, G., & Zhang, L. (Eds.), *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*, Vol. 9364 of *Lecture Notes in Computer Science*, pp. 131–147. Springer.

Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., & Ruijters, E. (2019). The quantitative verification benchmark set. In Vojnar, T., & Zhang, L. (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, Vol. 11427 of *Lecture Notes in Computer Science*, pp. 344–350. Springer.

Helmert, M. (2006). The Fast Downward planning system. *Journal of Artificial Intelligence Research*, *26*, 191–246.

Hoey, J., St-Aubin, R., Hu, A. J., & Boutilier, C. (1999). SPUDD: stochastic planning using decision diagrams. In Laskey, K. B., & Prade, H. (Eds.), *UAI '99: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Stockholm, Sweden, July 30 - August 1, 1999*, pp. 279–288. Morgan Kaufmann.

Hoffmann, J., Hermanns, H., Klauck, M., Steinmetz, M., Karpas, E., & Magazzeni, D. (2020). Lets learn their language? a case for planning with automata-network languages from model checking.. accepted at AAAI 2020.

Hoffmann, J., Kissmann, P., & Torralba, Á. (2014). "Distance"? Who Cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In Schaub, T. (Ed.), *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI'14)*, Prague, Czech Republic. IOS Press.

Holzmann, G. (2004). *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley.

Israeli, A., & Jalfon, M. (1990). Token management schemes and random walks yield self-stabilizing mutual exclusion. In Dwork, C. (Ed.), *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 22-24, 1990*, pp. 119–131. ACM.

Keller, T., & Eyerich, P. (2012). PROST: probabilistic planning based on UCT. In McCluskey, L., Williams, B. C., Silva, J. R., & Bonet, B. (Eds.), *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*. AAAI.

Kinathil, S., Soh, H., & Sanner, S. (2017a). Analytic decision analysis via symbolic dynamic programming for parameterized hybrid mdps. In Barbulescu, L., Frank, J., Mausam, & Smith, S. F. (Eds.), *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017.*, pp. 181–185. AAAI Press.

Kinathil, S., Soh, H., & Sanner, S. (2017b). Nonlinear optimization and symbolic dynamic programming for parameterized hybrid markov decision processes. In *The Workshops of the The Thirty-First AAAI Conference on Artificial Intelligence, Saturday, February 4-9, 2017, San Francisco, California, USA*, Vol. WS-17 of *AAAI Workshops*. AAAI Press.

Klein, J., Baier, C., Chrszon, P., Daum, M., Dubslaff, C., Klüppelholz, S., Märcker, S., & Müller, D. (2018). Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic büchi automata. *STTT*, *20*(2), 179–194.

Kolobov, A., Mausam, Weld, D. S., & Geffner, H. (2011). Heuristic search for generalized stochastic shortest path mdps. In Bacchus, F., Domshlak, C., Edelkamp, S., & Helmert, M. (Eds.), *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*. AAAI.

Kwiatkowska, M. Z., Norman, G., & Parker, D. (2004). Probabilistic symbolic model checking with PRISM: a hybrid approach. *STTT*, *6*(2), 128–142.

Kwiatkowska, M. Z., Norman, G., & Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G., & Qadeer, S. (Eds.), *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, Vol. 6806 of *Lecture Notes in Computer Science*, pp. 585–591. Springer.

Kwiatkowska, M. Z., Norman, G., & Parker, D. (2012). The PRISM benchmark suite. In *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*, pp. 203–204. IEEE Computer Society.

Lehmann, D., & Rabin, M. (1981). On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In *Proc. 8th Annual ACM Symposium on Principles of Programming Languages (POPL'81)*, pp. 133–138.

Little, I., & Thiebaux, S. (2007). Probabilistic planning vs replanning. In *ICAPS Workshop on the International Planning Competition: Past, Present and Future*.

Lynch, N., Saias, I., & Segala, R. (1994). Proving time bounds for randomized distributed algorithms. In *Proc. 13th Annual ACM Symposium on Principles of Distributed Computing (PODC'94)*, pp. 314–323. ACM Press.

Maisonneuve, V. (2009). Automatic heuristic-based generation of MTBDD variable orderings for PRISM models. Internship report, Oxford University Computing Laboratory.

Mausam, & Kolobov, A. (2012). *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

McDermott, D. V. (2000). The 1998 AI planning systems competition. *AI Magazine*, *21*(2), 35–55.

McMahan, H. B., Likhachev, M., & Gordon, G. J. (2005). Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005*, pp. 569–576.

Miner, A. S., & Parker, D. (2004). Symbolic representations and analysis of large probabilistic systems. In *Validation of Stochastic Systems - A Guide to Current Research*, pp. 296–338.

Nakhost, H., Hoffmann, J., & Müller, M. (2012). Resource-constrained planning: A Monte Carlo random walk approach. In Bonet, B., McCluskey, L., Silva, J. R., & Williams, B. (Eds.), *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pp. 181–189. AAAI Press.

Nebel, B. (2000). On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, *12*, 271–315.

Patrizi, F., Lipovetzky, N., De Giacomo, G., & Geffner, H. (2011). Computing infinite plans for LTL goals using a classical planner. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pp. 2003–2008.

Pnueli, A., & Zuck, L. (1986). Verification of multiprocess probabilistic protocols. *Distributed Computing*, *1*(1), 53–72.

Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pp. 46–57.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st edition). John Wiley & Sons, Inc., New York, NY, USA.

Rabin, M. (1982). N-process mutual exclusion with bounded waiting by $4 \log_2 N$-valued shared variable. *Journal of Computer and System Sciences*, *25*(1), 66–75.

Raghavan, A., Khardon, R., Fern, A., & Tadepalli, P. (2013). Symbolic opportunistic policy iteration for factored-action mdps. In Burges, C. J. C., Bottou, L., Ghahramani, Z., & Weinberger, K. Q. (Eds.), *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, pp. 2499–2507.

Sanner, S. (2010). Relational dynamic influence diagram language (rddl): Language description. Available at `http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf`.

Sanner, S., Delgado, K. V., & de Barros, L. N. (2012). Symbolic dynamic programming for discrete and continuous state mdps. *CoRR*, *abs/1202.3762*.

Sanner, S., & Kersting, K. (2017). Symbolic dynamic programming. In Sammut, C., & Webb, G. I. (Eds.), *Encyclopedia of Machine Learning and Data Mining*, pp. 1220–1228. Springer.

Seipp, J., Pommerening, F., Sievers, S., & Wehrle, M. (2016). Fast Downward Aidos. In *UIPC 2016 planner abstracts*, pp. 28–38.

Sprauel, J., Kolobov, A., & Teichteil-Königsbuch, F. (2014). Saturated path-constrained MDP: planning under uncertainty and deterministic model-checking constraints. In Brodley, C. E., & Stone, P. (Eds.), *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pp. 2367–2373. AAAI Press.

Steinmetz, M., Hoffmann, J., & Buffet, O. (2016). Goal probability analysis in probabilistic planning: Exploring and enhancing the state of the art. *J. Artif. Intell. Res.*, *57*, 229–271.

Teichteil-Königsbuch, F. (2012). Path-constrained markov decision processes: bridging the gap between probabilistic model-checking and decision-theoretic planning. In Raedt, L. D., Bessière, C., Dubois, D., Doherty, P., Frasconi, P., Heintz, F., & Lucas, P. J. F. (Eds.), *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, Vol. 242 of *Frontiers in Artificial Intelligence and Applications*, pp. 744–749. IOS Press.

Thiébaux, S., Gretton, C., Slaney, J. K., Price, D., & Kabanza, F. (2006). Decision-theoretic planning with non-markovian rewards. *J. Artif. Intell. Res.*, *25*, 17–74.

Torralba, Á., Alcázar, V., Kissmann, P., & Edelkamp, S. (2017). Efficient symbolic search for cost-optimal planning. *Artif. Intell.*, *242*, 52–79.

Torralba, Á., Hoffmann, J., & Kissmann, P. (2016). MS-Unsat and SimulationDominance: Merge-and-shrink and dominance pruning for proving unsolvability. In *UIPC 2016 planner abstracts*, pp. 12–15.

Trevizan, F. W., Teichteil-Knigsbuch, F., & Thiébaux, S. (2017). Efficient solutions for stochastic shortest path problems with dead ends. In Elidan, G., Kersting, K., & Ihler, A. T. (Eds.), *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017*. AUAI Press.

Trevizan, F. W., Thiébaux, S., Santana, P. H., & Williams, B. C. (2016). Heuristic search in dual space for constrained stochastic shortest path problems. In Coles, A. J., Coles, A., Edelkamp, S., Magazzeni, D., & Sanner, S. (Eds.), *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016.*, pp. 326–334. AAAI Press.

Vianna, L. G., Sanner, S., & de Barros, L. N. (2013). Bounded approximate symbolic dynamic programming for hybrid mdps. In Nicholson, A., & Smyth, P. (Eds.), *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI 2013, Bellevue, WA, USA, August 11-15, 2013*. AUAI Press.

Vianna, L. G. R., de Barros, L. N., & Sanner, S. (2015). Real-time symbolic dynamic programming. In Bonet, B., & Koenig, S. (Eds.), *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pp. 3402–3408. AAAI Press.

Younes, H. L. S., Littman, M. L., Weissman, D., & Asmuth, J. (2005). The first probabilistic track of the international planning competition. *J. Artif. Intell. Res.*, *24*, 851–887.

Younes, H. L., & Littman, M. L. (2004). Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. Tech. rep., School of Computer Science, Carnegie Mellon University.

Zamani, Z., Sanner, S., & Fang, C. (2012). Symbolic dynamic programming for continuous state and action mdps. In Hoffmann, J., & Selman, B. (Eds.), *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* AAAI Press.