Compiling Probabilistic Model Checking into Probabilistic Planning (Technical Report)

Michaela Klauck and **Marcel Steinmetz** and **Jörg Hoffmann** and **Holger Hermanns**
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany

Abstract

It has previously been observed that the verification of safety properties in deterministic model-checking frameworks can be compiled into classical planning. A similar connection exists between goal probability analysis on either side, yet that connection has not been explored. We fill that gap with a translation from Jani, an input language for quantitative model checkers including the Modest toolset, into PPDDL. Our experiments motivate further cross-fertilization between both research areas, specifically the exchange of algorithms. Our study also initiates the creation of new benchmarks for goal probability analysis.

Introduction

Previous works have explored connections between qualitative model-checking and classical planning (Edelkamp 2003). A similar connection exists for probabilistic models. Here, we introduce a compilation from Jani (Budde et al. 2017), an input language for quantitative model-checkers, into PPDDL (Younes et al. 2005). We complement the compilation by an empirical comparison of methods used in model-checking – variants of value iteration (VI) – with the heuristic search algorithms developed by the AI community.

Tools from probabilistic model-checking have become very popular in the robotics and motion planning communities due to their support of expressive formal modeling languages, e. g., (Johnson and Kress-Gazit 2011; Lacerda, Parker, and Hawes 2015). Recent works in decision-theoretic planning started to use temporal logics to encode history-dependent reward functions (Camacho et al. 2017; Brafman, Giacomo, and Patrizi 2018). Also they introduced compilations back to standard formalisms, allowing to use well-established planning algorithms for Markov decision processes (MDP). Teichteil-Königsbuch (2012) has presented an overarching framework connecting decision-theoretic planning with formalisms from model-checking. The result is a very general class of problems, which however falls out of scope of existing algorithms. Despite those works, a direct connection between models and algorithms in probabilistic model-checking and that in probabilistic planning has so far not been explored. We start to close that gap through our compilation and accompanied experiments.

Jani is a powerful language that can express models of distributed and concurrent systems in the form of networks of automata decorated with variables, clocks and probabilities. A large spectrum of case studies exists. In full generality, Jani models are networks of stochastic timed automata. But the core formalism are MDPs, as in PPDDL probabilistic planning. We consider a relevant fragment of Jani that allows for structure-preserving translation into PPDDL.

We focus on the verification of safety properties, which translates in PPDDL to goal probability analysis. In this class of MDPs, heuristic search algorithms like HDP (Bonet and Geffner 2003a) cannot be run as-is, but require an outer loop of iterations known as the FRET framework: find, revise, eliminate traps (Kolobov et al. 2011). Traps are sets of states that are closed under probabilistic branching in the subgraph of the state space induced by the current value function approximation v. While FRET as per Kolobov et al. considers in this subgraph *all* actions that are optimal according to v, Steinmetz et al. (2016) devised a variant which considers only the actions chosen by the greedy policy π. We will refer to these FRET variants as FRET-v and FRET-π respectively. I-Dual is a recently introduced heuristic search algorithm which does not require the FRET outer loop (Trevizan, Teichteil-Königsbuch, and Thiébaux 2017). In essence, this algorithm interleaves linear program (LP) evaluations with state space exploration. The LPs solved are encodings of goal probability MDPs (Altman 1999), restricted to the part of the state space explored so far. By considering in the exploration step only states touched by the last LP solution, one can often find a solution of the MDP while considering only a small fraction of the state space.

In our experiments, we compare PRISM (Kwiatkowska, Norman, and Parker 2011), Modest (Hartmanns and Hermanns 2014), and Fast Downward (Helmert 2006) based on two landmark problems in quantitative model-checking: the dining cryptographers protocol (DCP) and the randomized consensus shared coin protocol (RCSCP). It turns out that FRET-π excels in DCP, being surpassed only by PRISM's symbolic engine. In RCSCP heuristic search is inferior to VI. Nevertheless, with the help of a heuristic identifying 0 goal probability states, FD's VI achieves better scaling. Beyond this case study, our research motivates further cross-fertilization, pertaining to the exchange of algorithms. Our study also initiates the creation of new benchmarks for goal

probability analysis.

Background

Probabilistic PDDL (PPDDL) and FD. PPDDL extends PDDL with the possibility to define probabilistic action effects, i.e., a probability distribution over multiple possible outcomes. Fast Downward (FD) is a wide-spread code base in classical planning. It has been extended by Steinmetz et al. (2016) for goal probability analysis. That extension encompasses topological value iteration (TVI) (Dai et al. 2011), along with several heuristic search algorithms of which here we consider HDP along with FRET-v and FRET-π. We have extended their code by an implementation of the I-Dual algorithm (Trevizan, Teichteil-Königsbuch, and Thiébaux 2017). There are no inherently probabilistic heuristic functions in FD yet. As suggested by Bonet and Geffner (2005), we use the all-outcomes determinization with classical-planning heuristic functions. In our context, this comes down to goal probability estimate 0 for detected dead-ends, and estimate 1 elsewhere. We use $h^{\max}$ (Bonet and Geffner 2001) which provides strong dead-end detection capabilities at a comparatively small computational cost.

The Modest toolset (MT) and PRISM. Probably the most widespread probabilistic model checker is PRISM (Kwiatkowska, Norman, and Parker 2011). MT's core MDP-algorithms are known to be competitive PRISM (Hahn and Hartmanns 2016). Both support the analysis of hybrid, real-time, distributed and stochastic systems. PRISM offers multiple model-checking engines, including variants of VI based on an *explicit* state space representation as well as a *symbolic* variant. In a nutshell, the symbolic version represents the current value function estimation and the transition probabilities as multi-terminal binary decision diagrams (MTBDD). This data structure has builtin support of arithmetic operations, allowing to directly express value updates as MTBDD operations. Furthermore, PRISM optionally identifies states with goal probability 0, respectively 1 in a preprocessing step, which can then be filtered out inside VI. MT provides several components to solve various forms of quantitative model-checking problems. For our MDP setting, we consider one such component, called Mcsta. Mcsta provides MT's variants of value iteration, featuring a preprocessing step similar to PRISM. An external-memory VI (EVI) variant caters for MDPs whose state space is too large to fit into main memory (Hartmanns and Hermanns 2015).

Jani. Jani is an overarching language conceived to foster verification tool interoperation and comparability. It allows to model a rich variety of quantitative automata networks with variable decorations. Properties to be checked are temporal formulas based on computation tree logic (CTL).

Each automaton has locations connected by directed edges. The edges may be taken with a given probability provided the edge's *guard* is satisfied. Then an effect may modify the variable values, and a new location is occupied.

Jani allows variables of various types (e.g. *int* and *bool*). They can be restricted to a finite range. Expressions over these variables support many standard arithmetic operations, as well as conjunction and disjunction. The possible initial variable values are mostly specified via a *restrict-initial* expression but can also be given directly when declaring a new variable.

```
"variables": [
 {"name":"res", "type": "bool"},
 {"name":"coin1", "type": {"kind":"bounded",
    "base":"int", "lower−bound":0, "upper−bound":2}}, ...],
"restrict−initial": {"exp": {"op":"^",
 "left": {"op":"^",
     "left": {...},
     "right": {"op":'=', "left":"coin1", "right":0}},
 "right": {"op":'=', "left":"res", "right":false}}},
"automata": [
 { "name":"aut1",
 "locations": [ {"name":"loc0"}, {"name":"loc1"} ],
 "initial−location": ["loc0"],
 "edges": [ {  "location":"loc0",
    "guard": { "exp": {"op":"=", "left":"coin1", "right":0} },
    "destinations": [
     { "probability": {"exp":0.5},  "location":"loc1",
       "assignments": [ { "ref":"coin1", "value":1 } ] },
     { "probability": {"exp":0.5},  "location":"loc1",
       "assignments": [ { "ref":"coin1", "value":2 } ] } ]}]},
 ...
 { "name":"aut3",
 "locations": [ {"name":"loc0"}, {"name":"loc1"} ],
 "initial−location": ["loc0"],
 "edges": [ { "location":"loc0",
    "guard": { "exp": {"op":"^",
        "left": {"op":"=", "left":"res", "right":false},
        "right": {"op":"=",
          "left":{"op":"+", "left":"coin1", "right":"coin2"},
          "right":3}}},
    "destinations": [
     { "probability": {"exp":1},  "location":"loc1",
       "assignments": [ { "ref":"res", "value":true } ] }]}]}]
```

Figure 1: A Jani specification example.

A Jani file lists the occurring constants (objects), and the variable definitions as well as the *restrict-initial* block. In addition, a list of automata along with their locations and edges is provided.

An example snippet of Jani is given in Figure 1. It specifies a Boolean variable *res* with initial value *false* and a bounded integer variable *coin1* with initial value 0. Two automata are depicted. The first automaton *aut1* consists of two locations *loc0* and *loc1*. From its initial location *loc0*, one edge can be taken provided that the variable *coin1* has value 0, as expressed by the guard. The edge has two potential outcomes, each weighted by a probability. Both outcomes lead to the same destination, *loc1*, but they reassign the variable *coin1* to different values. The automaton *aut3* also consists of two locations *loc0* and *loc1*. *aut3* contains one edge, starting from *loc0*, and requiring that *res* is set to *false* and that the sum of both *coin* variables is 3. The edge moves the automaton location deterministically to *loc1* and changes the value of *res* to *true*.

Compilation from Jani to PPDDL

We outline the main design decisions in our compilation. The source code of our compiler, along with all Jani and corresponding PPDDL benchmarks, is available online.[1]

Fragment of Jani considered. To match PPDDL planning, we consider probabilistic safety properties, i.e., reachability queries of the form $\Diamond\ goal$ (eventually $goal$) where the maximum probability is sought. We do not consider temporal goals, as our design rationale in this work is to stick to a Jani fragment that allows for a largely structure-preserving translation into PPDDL. Hence, we avoid compiling temporal formulas into propositional goals (Edelkamp 2006; Baier, Bacchus, and McIlraith 2009), and can focus on the translation of Jani's automata networks into PDDL.

We consider the finite-state model-checking fragment of Jani, where the definition of each integer variable specifies a finite range, i.e., a lower bound and an upper bound on the variable's values. Moreover, we consider only *restrict-initial* blocks specifying exactly one value for each variable. One can in principle encode, in PPDDL, different possible initial states through actions applied at the start. However, Jani specifies the set of possible initial states in terms of a set of constraints. The naïve approach would enumerate all solutions to that constraint set. To combine the results for multiple initial states, Jani adopted the *filter* functionality of Prism (Kwiatkowska, Norman, and Parker 2011). In a nutshell, along the property to be checked, the *filter* expression defines a set of states for which the property should be verified as well as an operator to combine the individual results (e.g., $\min$, $\max$). As our focus lies on planning for maximal goal probability, we only consider *filter* expressions over the initial state, calculating the maximal probability to reach states satisfying the property. The aforementioned restrictions are not limiting in the sense that we still cover most existing Jani models.

A more limiting restriction we make regards synchronization, where we only consider shared-variable synchronization, via guards referring to variables written on by other automata. Jani also supports handshake synchronization over multiple automata. This can, in principle, be realized in PPDDL by adding additional small protocols, similarly to Edelkamp's (2003) approach. But restricting focus on shared-variable synchronization allows a much more direct, more structure-preserving, compilation, and is still practically relevant. Unfortunately, many Jani benchmarks make use of handshake synchronization, hence restricting the benchmark set in the empirical evaluation.

Predicates, Types, Objects. Variable types in Jani directly translate into (P)PDDL types, with an additional type *loc* for automata locations. The values associated with a type in Jani are encoded as PDDL objects of that type. Jani variables are also encoded as PDDL objects, variable-value assignments are represented by the PDDL predicate $(value\ var\ val)$. We list all Jani constants, variables and locations as PDDL constants, so we can use them in action descriptions. Current automata locations are encoded through an $(at\_X\ loc)$ predicate for each automaton X. The goal is encoded through an additional Jani Boolean variable $goal\_condition$. This auxiliary variable is introduced as part of the compilation into PPDDL only. Changes to the Jani input model are not required.

We handle finite-range arithmetics in PDDL via finite enumeration of arithmetic-operation outcomes, similarly as in previous works encoding finite-range integer variables into PDDL, e.g., (Nakhost, Hoffmann, and Müller 2012). To do so, we determine from the bounds specified in Jani's variable definitions, the minimal L and maximal number U required to safely represent the domains of all variables and all arithmetic operations used in the Jani model. The numbers within the interval $[L, U]$ are then represented by PDDL objects $nL, \ldots, nU$.

Arithmetic operations are hard-coded into the initial state, using additional predicates, and enumerating e.g. all triples *x*, *y*, and $z = x * y$ within the precomputed range, via the list of corresponding static ground facts of the form $(multiply\ x\ y\ z)$. The variables and automata of the Jani snippet in Figure 1 are translated into PDDL types, constants, and predicates as follows:

```
(:types loc int bool)
(:constants
  aut1_loc0 aut1_loc1 aut3_loc0 aut3_loc1 - loc
  goal_condition res true false - bool
  coin1 n0 n1 n2 n3 n4 - int)
(:predicates
  (value ?x ?v - bool)
  (value ?x ?v - int)
  (at_aut1 ?l - loc)
  (at_aut3 ?l - loc)
  (sum ?x ?y ?z - int))
```

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 PDDL object. For example, the outcome value z of the expression $(x_1 + x_2) * y$ is encoded through the conjunction of $(add\ x_1\ x_2\ z')$ with $(multiply\ z'\ y\ z)$.

Actions. Jani edge descriptions translate into PPDDL actions. The action parameters are chosen to represent variables affected by the edge's guards or assignments. In particular, this pertains to numeric variables whose value may change: the respective values before and after the action application become parameters constrained by the precondition to match the necessary value computation. For example, an edge guard $z = (x_1 + x_2) * y$ is encoded into parameters $?x_1$, $?x_2$, $?y$, $?z'$, $?z$, along with the preconditions $(add\ ?x_1\ ?x_2\ ?z')$ and $(multiply\ ?z'\ ?y\ ?z)$.

Preconditions and effects can now be directly compiled from Jani guards and assignments. In addition, a precondition and effect of the form $(at\_X\ start)$ and $(at\_X\ destination)$ encodes the automaton location before and after the action application. For example, the edge of *aut3* from Figure 1 is translated into the following PPDDL action:

[1] http://fai.cs.uni-saarland.de/downloads/jani-ppddl.zip

```
(:action aut3_loc0_to_loc1
:parameters (?v1 ?v2 − int)
:precondition (and (at_aut3 aut3_loc0) (value res false)
          (value coin1 ?v1) (value coin2 ?v2) (sum ?v1 ?v2 n3))
:effect (and (not (at_aut3 aut3_loc0)) (at_aut3 aut3_loc1)
           (not (value res false)) (value res true)))
```

Jani permits disjunctive edge guards, which we compile away using standard techniques for DNF transformation (Gazen and Knoblock 1997), followed by splitting the action into one copy per disjunct. This means, the guard is first translated into DNF and then one action per clause is created in PPDDL having this clause as precondition. All other components of the action stay the same.

The compilation of effects is the only place where we require the modeling power of probabilistic PDDL, to encode probabilistic edge outcomes. The automaton *aut1* from Figure 1 results in the following PPDDL fragment:

```
(:action aut1_loc0_to_loc1_or_loc1
:parameters ()
:precondition (and (at_aut1 aut1_loc0) (value coin1 n0))
:effect (probabilistic
  0.5 (and (not (at_aut1 aut1_loc0)) (at_aut1 aut1_loc1)
           (not (value coin1 n0)) (value coin1 n1))
  0.5 (and (not (at_aut1 aut1_loc0)) (at_aut1 aut1_loc1)
           (not (value coin1 n0)) (value coin1 n2))))
```

In Jani, edges that may alter bounded integer variables to values outside their domains are considered a modeling flaw. Assuming a properly designed Jani model as input, the translation of edge guards into action preconditions is hence already sufficient to ensure that variable values remain within their specified bounds.

Initial State and Goal. In Jani, the model initialization is spread over the *restrict-initial* expression, the definition of constants and variables, and for each automaton its *initial-location*. These fragments can directly be translated into the *init* part of PPDDL (along with arithmetic operations as outlined above). For the example in Figure 1, we obtain

```
(:init
  (value goal_condition false)
  (value res false) (value coin1 n0)
  (at_aut1 aut1_loc0) (at_aut3 aut3_loc0)
  (sum n0 n0 n0) (sum n0 n1 n1) (sum n0 n2 n2) ...)
```

We translate the property to be checked into a goal, through an action whose precondition is the property expression, and whose effect is *(value goal_condition true)*. The latter ground fact becomes the PPDDL goal condition.

As a simple example, consider the property $\Diamond res$, i.e., that res is eventually set to *true*. The Jani snippet encoding this property looks as follows:

```
"properties": [ {"name":"eventually_res", "expression":{
  "op":"filter",
  "fun":"max",
  "values":{
     "op":"Pmax", "exp":{ "op":"U", "left":true, "right":"res" } },
  "states": { "op":"initial" } }}]
```

In PPDDL this property is translated into the following action:

```
(:action achieve_goal_condition
:parameters ()
:precondition (and (value goal_condition false) (value res true))
:effect (and (not (value goal_condition false))
          (value goal_condition true)))
```

In general, properties expressed in Jani might again contain disjunctions and arbitrary algebraic operations. In the translation into PPDDL, we follow the lines of the translation of edge guards into PPDDL action preconditions, transforming the property into DNF, creating one action per clause, and introducing auxiliary action parameters representing current variable assignments and intermediate results if necessary.

Compilation Size. Obviously, the size of our 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. Note also that the parameter-value combinations for ground actions are constrained by the static predicates giving the semantics of the arithmetic operations. Planning systems like FD exploit this property to not even generate obviously unreachable ground actions in the first place.

We provide detailed compilation size, and grounded encoding size, results as part of our experiments.

Experiments

All experiments were conducted on an Intel Xeon E5-2660 machine with time (memory) cut-offs of 1h (6GB). We used Cplex as LP solver. We compare runtime and search space statistics of various configurations of FD, MT, and PRISM:

- We ran FD using TVI, I-Dual, FRET-v and FRET-π with LRTDP (Bonet and Geffner 2003b) and HDP (Bonet and Geffner 2003a), all with and without $h^{\max}$.
- For MT, we ran VI and EVI with and without the preprocessing option.
- For PRISM, we ran both explicit (Exp) and symbolic engines (Sym) with and without the preprocessing step. Additionally, we ran PRISM's hybrid engine (Hyb) which combines the explicit and symbolic approach by storing the value function approximation in a table with one entry for each state, and the MDP's transition function as an

MTBDD. The intention is to benefit both from faster numerical operations on flat data structures as well as a more compact, symbolic, representation of the state space.

PRISM does not support Jani directly, but there exist automatic tools translating Jani to PRISM's input language (Hahn et al. 2014). Unfortunately, we encountered several technical complications during this translation, forcing us to fall back on manually translated models. All benchmark files are available online.[2]

We have also run the original I-Dual implementation (Trevizan, Teichteil-Königsbuch, and Thiébaux 2017) based on mGPT (Bonet and Geffner 2005). mGPT-IDual turned out to be consistently slower than FD-IDual due to inefficiencies in mGPT's grounding procedure. We will only report results for the FD version.

Most Jani models make use of handshake synchronization, which is currently not supported by our translation, limiting the selection of domains. We consider two case studies, which belong to the most popular benchmarks in probabilistic model-checking. We introduce the case studies, and discuss the respective results.

Dining Cryptographers Protocol (DCP). In DCP (Chaum 1988) $n \geq 3$ cryptographers want to check if the NSA is paying for their dinner while respecting each others privacy, i.e., without advertising who has paid exactly. For that, everyone throws a coin, the outcomes being encoded as 0 and 1. Then each cryptographer C_i looks at the outcome o and the outcome o_N of his left-hand neighbor. If C_i does not pay for dinner, then C_i announces $o \oplus o_N$ to the table. If C_i does pay for dinner, then C_i announces $\overline{o \oplus o_N}$. The intended guarantee is that the xor over all announced values is 0 if the NSA paid, and is 1 if one of the cryptographers paid. What we wish to check is that the protocol is correct. This is encoded as reaching a target condition with probability 1.

A specific property of this domain is the absence of 0 goal-probability states. This particularly qualifies the use of contingent planners. As contingent planners do not need to reason about probabilities, hence avoiding costly numerical operations, they may handle instances more efficiently than fully-fledged probabilistic planners. Thus, we also provide PDDL versions of the compiled PPDDL instances, replacing probabilistic effects by non-deterministic ones. We include the state-of-the-art contingent planner PRP (Muise, McIlraith, and Beck 2012) in the results below.

Consider first Figure 2 (a), which compares the different FD configurations. Note that, due to the aforementioned structure of this domain, $h^{\max}$ merely constitutes an additional overhead here. In all FD configurations but I-Dual, this overhead is reflected in runtime across all instances and grows exponentially in n. In FD-IDual, the overhead of $h^{\max}$ is overshadowed by the runtime spent on LP evaluations. Having to solve increasingly large LPs, renders FD-IDual overall uncompetitive – despite offering small advantages in search space size compared to TVI, as indicated by Figure 2 (d). Both LRTDP and HDP perform identical in this domain. FD-FRET-π benefits from the ability of the underlying heuristic search algorithm to find a solution while visiting only a small fraction of the state space. The resulting reduction in the number of visited states grows exponentially in n. Since in this domain every (reachable) state has goal probability 1, FRET-v, in contrast, always has to consider the entire state space in the trap removal procedure. Hence, the similar performance of FD-FRET-v and FD-TVI.

Consider next Figure 2 (b), comparing the different MT and PRISM configurations. With the exception of PRISM-Hyb, the preprocessing option (referred to by the "-Pre" suffix) of MT and PRISM did not help anywhere. Clearly, PRISM-Exp is not competitive at all, scaling only to n=5, and running out of memory afterwards. PRISM-Hyb performs similarly to MT-VI, both scaling to n=10 and then running out of memory. The additional overhead induced by MT-EVI compared to MT-VI shows in runtime, but its smaller memory demand improves scaling to n=11. Being able to compactly represent extremely large numbers of states turns PRISM's symbolic approaches to the best configurations in this comparison. As the goal probability is 1 in this domain, PRISM-Hyb-Pre and PRISM-Sym-Pre solve all instances already in the preprocessing step. By storing the value function in a symbolic data structure, PRISM-Sym achieves the same performance without executing the preprocessing step.

Consider finally Figure 2 (c). FD-TVI outperforms MT-VI, presumably due to more efficient internal data structures. Of the configurations operating on flat data structures, FD-FRET-π clearly stands out. It solves instances up to n=17, while the next best (non-symbolic) VI configuration only scales to n=11. Interestingly, the limiting factor in FD-FRET-π is not the number of visited states, but lies in the grounding process. Figure 2 (e) illuminates this with compilation-size data. For the models themselves, i.e. Jani vs. PPDDL, the difference is small. But at the grounded level, where FD enumerates action parameter instantiations, matters are different. While no operation equivalent to grounding is needed within the MT or PRISM, in FD the number of ground actions increases steeply in n.

PRP shows a very volatile performance footprint. Surprisingly, it cannot beat the shown FRET-π configuration even in a single instance. The clear winner in this case study is PRISM-Sym. Solving the instances up to n=26 makes it far beyond the reach of all other non-BDD-based configurations.

Randomized Consensus Shared Coin Protocol (RCSCP). In this protocol (Aspnes and Herlihy 1990) n tourists wish to reach consensus of which place to visit next. In a nutshell, each tourist keeps tossing a coin; depending on the outcome of this toss, a global counter is either incremented or decremented; when the counter is within a certain desired range, the tourist stops. What we want to check is the probability that all tourists stopped at the same counter value. In difference to the DCP, that probability is < 1.

Figure 3 shows the performance results. Let's again turn our attention to the FD configurations first, Figure 3 (a).

[2] http://fai.cs.uni-saarland.de/downloads/jani-ppddl.zip

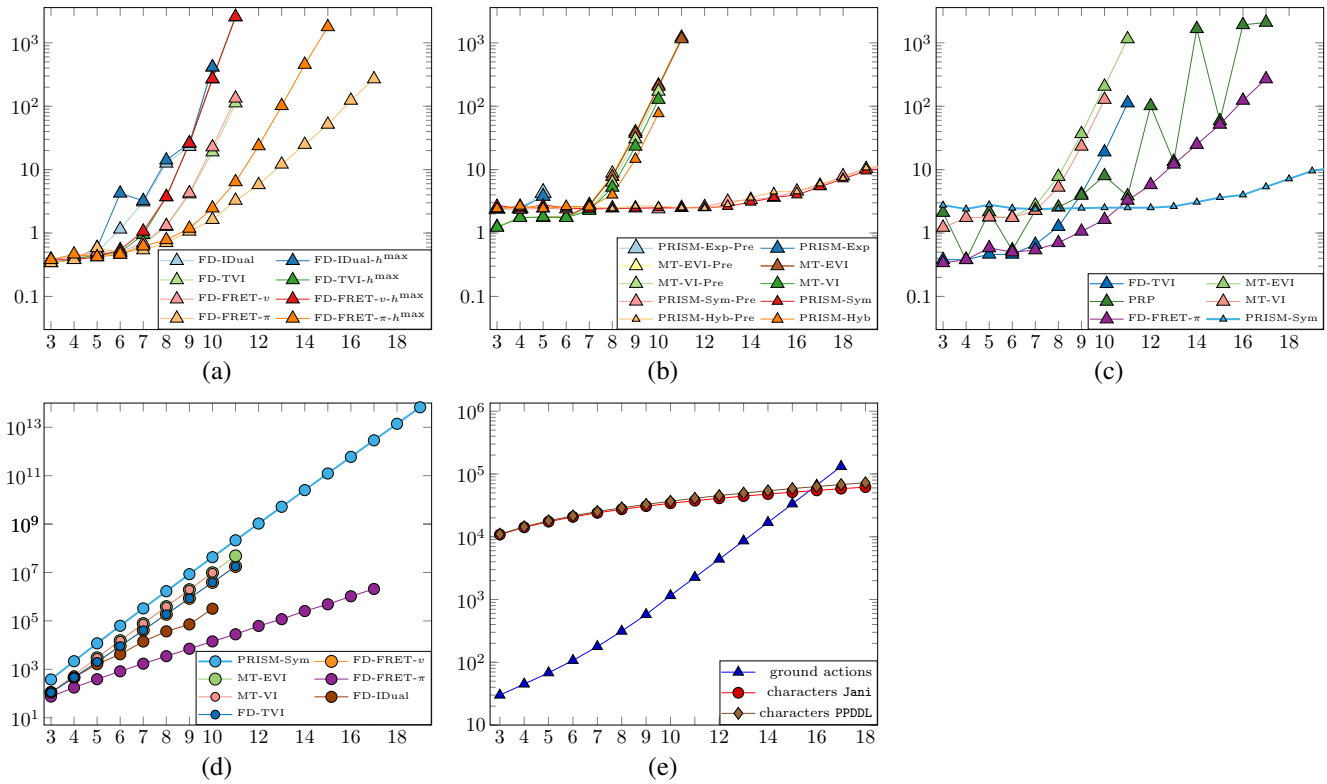

Figure 2: Empirical results in DCP, as a function of the number n of cryptographers. (a) – (c) runtime in CPU seconds; (d) number of visited states; (e) compilation size measures.

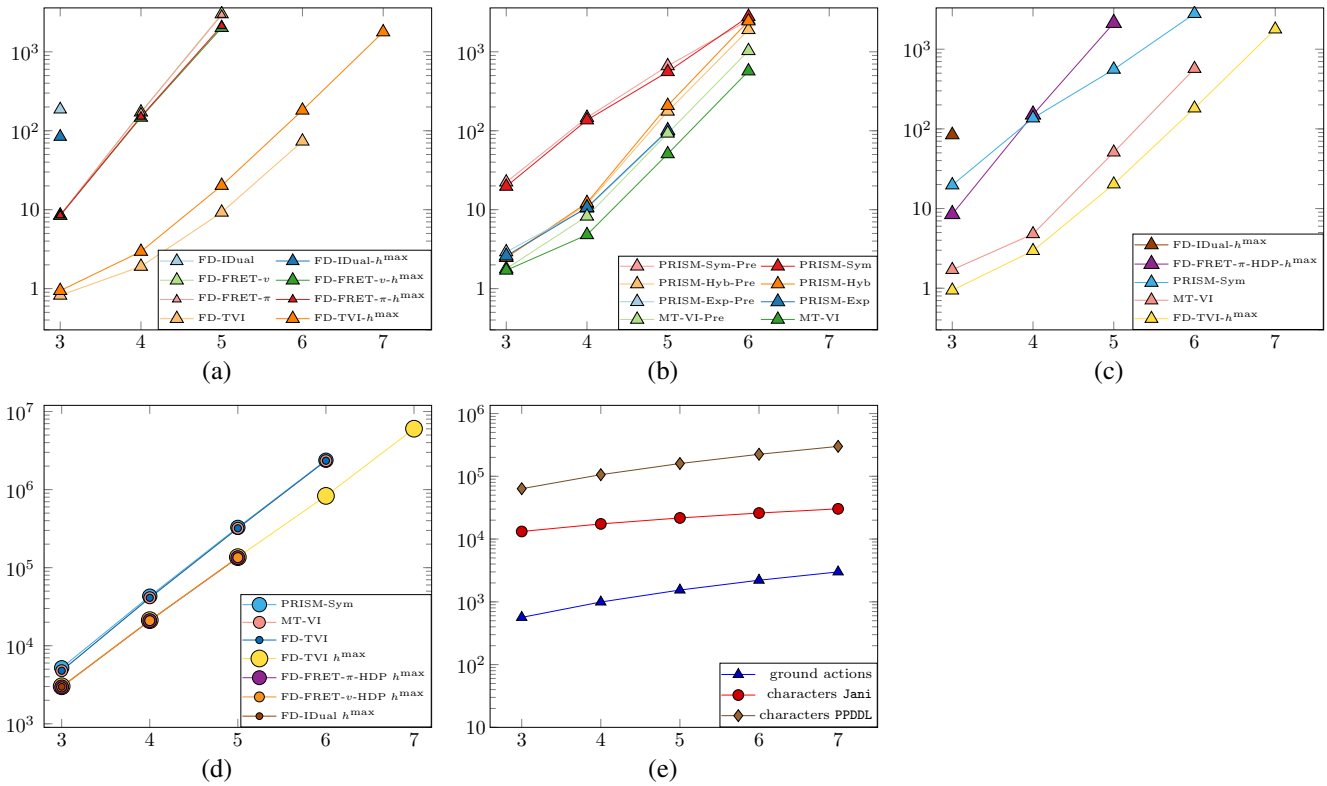

Figure 3: Empirical results in RCSCP, as a function of the number n of tourists. (a) – (c) runtime in CPU seconds; (d) number of visited states; (e) compilation size measures.

Contrary to DCP, $h^{\max}$ helps in all FD configurations. For the heuristic search algorithms, runtime is reduced slightly, but the runtime overhead of $h^{\max}$ is still visible in TVI. Due to the reduction in the number of states visited, FD-TVI-$h^{\max}$ is able to scale up to n=7, one step further than FD-TVI. Similarly to DCP, FD-IDual again suffers from having to solve many LPs. The FRET configurations are not competitive at all in this model. Both HDP and LRTDP show a slow convergence behavior, requiring significantly more value updates than, e.g., TVI. While HDP is able to reach a fixpoint for the instances up to $n = 5$, LRTDP does not terminate even in the smallest instance before the time limit. In contrast to DCP, neither one of the heuristic search algorithms provides any reduction in number of visited states.

Consider Figure 3 (b). As before, the preprocessing option of both PRISM and MT has no considerable effect on performance. PRISM's symbolic approaches are not able to compactly represent the state space / the value function. Numerical operations on the symbolic value function representation in PRISM-Sym are however more expensive, which is reflected in runtime. PRISM-Hyb can evade this overhead through its flat value function representation, but it is still beaten by PRISM-Exp in terms of runtime. Similar to DCP, MT's VI implementation outperforms PRISM-Exp. MT-EVI highly depends on a good state-partitioning function organizing external-memory access. In DCP it is not possible to provide such a function, preventing the use of MT-EVI.

In the cross-comparison, Figure 3 (c), FD's VI implementation is again more efficient than MT-VI. This still holds when taking into account the additional $h^{\max}$ evaluations. FD-TVI-$h^{\max}$'s scaling to n=7 makes it to the overall best configuration in this case study.

Consider finally Figure 3 (e). In difference to DCP, the Jani models are much smaller than the PPDDL ones. This is due to the size of the initial state, which needs to encode several arithmetic operations. In terms of the number of ground actions though, RCSCP is harmless.

Conclusion and Future Work

We have established a new connection between probabilistic model-checking and probabilistic planning. The empirical results show advantages for techniques from both areas, depending as always on the domain. To foster compilability of model-checking models into probabilistic planning, support for numeric state variables, and for more general goals, would be desirable. Further directions suggested by our experiments include the development of hybrid methods between grounding and lifted methods in planning, combining the benefits of both sides. Porting probabilistic heuristic search algorithms to model-checking tools directly, would give an opportunity to tackle all the syntactic elements that cannot be easily compiled. Moreover, this would allow to profit from techniques developed in this context already, notably the wealth of abstraction techniques that can be used to obtain better goal probability bounds.

Acknowledgments. This work was partially supported by the ERC Advanced Investigators Grant 695614 (POWVER) and by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA, grant no. 16KIS0656).

References

Altman, E. 1999. *Constrained Markov Decision Processes*. Chapman and Hall.

Aspnes, J., and Herlihy, M. 1990. Fast randomized consensus using shared memory. *J. Algorithms* 11(3):441–461.

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

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Bonet, B., and Geffner, H. 2003a. Faster heuristic search algorithms for planning with uncertainty and full feedback. In Gottlob, G., and Walsh, T., eds., *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, 1233–1238. Morgan Kaufmann.

Bonet, B., and Geffner, H. 2003b. Labeled RTDP: Improving the convergence of real-time dynamic programming. In Giunchiglia, E.; Muscettola, N.; and Nau, D., eds., *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS'03)*, 12–21.

Bonet, B., and 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.; and Patrizi, F. 2018. LTL$_f$ / LDL$_f$ non-markovian rewards. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence AAAI*.

Budde, C. E.; Dehnert, C.; Hahn, E. M.; Hartmanns, A.; Junges, S.; and Turrini, A. 2017. JANI: quantitative model and tool interaction. In Legay, A., and Margaria, T., eds., *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, volume 10206 of *Lecture Notes in Computer Science*, 151–168.

Camacho, A.; Chen, O.; Sanner, S.; and 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)*, 279–283.

Chaum, D. 1988. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptology* 1(1):65–75.

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

Edelkamp, S. 2003. Promela planning. In Ball, T., and Rajamani, S., eds., *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN-03)*, 197–212.

Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In Long, D., and Smith, S., eds., *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS'06)*, 374–377.

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

Hahn, E. M., and Hartmanns, A. 2016. A comparison of time- and reward-bounded probabilistic model checking techniques. In Fränzle, M.; Kapur, D.; and Zhan, N., eds., *Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium, SETTA 2016, Beijing, China, November 9-11, 2016, Proceedings*, volume 9984 of *Lecture Notes in Computer Science*, 85–100.

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

Hartmanns, A., and Hermanns, H. 2014. The modest toolset: An integrated environment for quantitative modelling and verification. In Ábrahám, E., and Havelund, K., eds., *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, 593–598. Springer.

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

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

Johnson, B., and Kress-Gazit, H. 2011. Probabilistic analysis of correctness of high-level robot behavior with sensor error. In *Robotics: Science and Systems*.

Kolobov, A.; Mausam; Weld, D. S.; and Geffner, H. 2011. Heuristic search for generalized stochastic shortest path MDPs. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS'11)*. AAAI Press.

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

Lacerda, B.; Parker, D.; and Hawes, N. 2015. Optimal policy generation for partially satisfiable co-safe LTL specifications. In *Proc. 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*. IJCAI/AAAI.

Muise, C. J.; McIlraith, S. A.; and Beck, J. C. 2012. Improved non-deterministic planning by exploiting state relevance. In Bonet, B.; McCluskey, L.; Silva, J. R.; and Williams, B., eds., *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*. AAAI Press.

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

Steinmetz, M.; Hoffmann, J.; and Buffet, O. 2016. Goal probability analysis in mdp probabilistic planning: Exploring and enhancing the state of the art. *Journal of Artificial Intelligence Research* 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.; and 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*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, 744–749. IOS Press.

Trevizan, F. W.; Teichteil-Königsbuch, F.; and Thiébaux, S. 2017. Efficient solutions for stochastic shortest path problems with dead ends. In Elidan, G.; Kersting, K.; and 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.

Younes, H. L. S.; Littman, M. L.; Weissman, D.; and Asmuth, J. 2005. The first probabilistic track of the international planning competition. *Journal of Artificial Intelligence Research* 24:851–887.