ROYAL SOCIETY
OPEN SCIENCE

royalsocietypublishing.org/journal/rsos

Perspective

Cite this article: Belle V. 2023 Logic + probabilistic programming + causal laws. *R. Soc. Open Sci.* **10**: 230785.
https://doi.org/10.1098/rsos.230785

Received: 20 June 2023
Accepted: 29 August 2023

Subject Category:
Computer science and artificial intelligence

Subject Areas:
artificial intelligence

Keywords:
statistical relational learning, first-order logic, probabilistic programming

Author for correspondence:
Vaishak Belle
e-mail: vaishak@ed.ac.uk

Contribution to special collection 'Making the most of AI's potential: cross-disciplinary perspectives on the role of AI in science and society'.

THE ROYAL SOCIETY
PUBLISHING

Logic + probabilistic programming + causal laws

Vaishak Belle

University of Edinburgh & Alan Turing Institute, Edinburgh, UK

VB, 0000-0001-5573-8465

Probabilistic planning attempts to incorporate stochastic models directly into the planning process, which is the problem of synthesizing a sequence of actions that achieves some objective for a putative agent. Probabilistic programming has rapidly emerged as a key paradigm to integrate probabilistic concepts with programming languages, which allows one to specify complex probabilistic models using programming primitives like recursion and loops. Probabilistic logic programming aims to further ease the specification of structured probability distributions using first-order logical artefacts. In this article, we briefly discuss the modelling of probabilistic planning through the lens of probabilistic (logic) programming. Although many flavours for such an integration are possible, we focus on two representative examples. The first is an extension to the popular probabilistic logic programming language PROBLOG, which permits the decoration of probabilities on Horn clauses—that is, prolog programs. The second is an extension to the popular agent programming language GOLOG, which permits the logical specification of dynamical systems via actions, effects and observations. The probabilistic extensions thereof emphasize different strengths of probabilistic programming that are particularly useful for non-trivial modelling issues raised in probabilistic planning. Among other things, one can instantiate planning problems with growing and shrinking state spaces, discrete and continuous probability distributions, and non-unique prior distributions in a first-order setting.

1. Introduction

Automated planning is a major topic of research in artificial intelligence, and enjoys a long and distinguished history [1]. The classical paradigm assumes a distinguished initial state, comprising a set of facts, and is defined over a set of actions which change that state in one way or another. Actions are further characterized in terms of their applicability conditions, that is, things that must be true for the agent to be able to execute it, and effects, which procedurally amounts to adding new facts to a state while removing others. The scientific agenda is then to design algorithms that synthesize a sequence of actions that takes the agent from an initial state to a desired goal state.

From the early days, automated planning was motivated by robotics applications. But it was observed that the real world—or more precisely, the robot's knowledge about the world—is almost never simply a set of facts that are true, and actions that the agent intends to execute never operate the way they are supposed to. One way to make sense of this complication is to separate the 'high-level reasoning', in our case the planner's search space, from the low-level sensor-motor details. On the positive side, such a move allows the plan representation to be finite, discrete and simple. On the negative side, significant expert knowledge has to go into materializing this separation of concerns, possibly at the loss of clarity on the behaviour of the system as a whole.

Incidentally, by testing the robot's effectors repeatedly in a controlled environment, one can approximate the uncertain effects of an action in terms of a probability distribution. Similarly, based on minimalistic assumptions about the environment, expressed as a probabilistic prior, by repeated sampling, the robot can update its prior to converge on a reasonable posterior that approximates the environment [2]. To that end, probabilistic planning attempts to incorporate such models directly into the planning process. There are, to date, numerous languages and algorithmic frameworks for probabilistic planning, e.g. [3–6].

In this article, we briefly discuss the modelling of probabilistic planning through the lens of probabilistic (logic) programming. Probabilistic programming has rapidly emerged as a key paradigm to integrate probabilistic concepts with programming languages, which allows one to specify complex probabilistic models using programming primitives like recursion and loops [7]. Probabilistic logic programming aims to further ease the specification of structured probability distributions using first-order logical artefacts.

In general, probabilistic programming languages developed so as to enable modularity and reuse in probabilistic machine learning applications. Their atomic building blocks incorporate stochastic primitives, and the formal representation also allows for compositionality [7,8].

Although many flavours for such an integration are possible, we focus on two representative examples. The first is an extension to the popular probabilistic logic programming language PROBLOG, which permits the decoration of probabilities on Horn clauses—that is, prolog programs. The second is an extension to the popular agent programming language GOLOG, which permits the logical specification of dynamical systems via actions, effects and observations. The probabilistic extensions thereof emphasize different strengths of probabilistic programming that are particularly useful for non-trivial modelling issues raised in probabilistic planning. Among other things, one can instantiate planning problems with growing and shrinking state spaces, discrete and continuous probability distributions, and non-unique prior distributions in a first-order setting. More precisely, we provide an overview of the features of two systems:

— HYPE [9]: a planning framework based on *distributional clauses* [10]; and
— ALLEGRO [11]: a high-level control programming framework that extends GOLOG [12].

These two systems emphasize different strengths of probabilistic programming, which we think are particularly useful for complex modelling issues raised in probabilistic planning. HYPE can easily describe growing and shrinking state spaces owing to uncertainty about the existence of objects, and thus is closely related to BLOG models [13,14]. Since HYPE is an extension of PROBLOG [15], it stands to benefit from a wide range of applications and machine learning models explored with PROBLOG.[1] The dynamical aspects of the domain are instantiated by reifying time as an argument in the predicates, and so it is perhaps most appropriate for finite horizon planning problems.

ALLEGRO treats actions as first-class citizens and is built on a rich model of dynamics and subjective probabilities, which allows it to handle context-sensitive effect axioms, and non-unique probability measures placed on first-order formulae. GOLOG has also been widely used for a range of applications that apply structured knowledge (e.g. ontologies) in dynamical settings [16], and ALLEGRO stands to inherit these developments. GOLOG has also been shown as a way to structure search in large plan spaces [17]. Finally, since there are constructs for iteration and loops, such programs are most appropriate for modelling non-terminating behaviour [18].

In what follows, we describe the essential formal and algorithmic contributions of these systems before concluding with open computational issues. Clearly, these two systems are not the only

[1]dtai.cs.kuleuven.be/problog

languages with elements of logic and probability for planning: relational Markov decision processes (MDPs) [19], first-order partially observable Markov decision processes (POMDPs) [20], and to a much lesser extent dynamic Bayesian networks over action models [21] are part of the larger family. However, these systems are built on powerful and general logical foundations: HYPE is an extension of PROBLOG, and so allows the specification of probabilistic assertions with logic programming. ALLEGRO, on the other hand, is an extension of GOLOG, which is a dialect of first-order logic with some second-order logical features. Thus, they are indicative of what is possible when probabilities and first-order logic are unified in a dynamic setting. This sets our agenda apart from proposals such as BLOG and its dynamic version too [13,14], which support some first-order features but do not permit arbitrary logical connectives and quantification.

To prepare for our discussion, let us briefly reflect on why probabilistic logic programming is a powerful paradigm.

2. Why probabilistic programming + logic?

Probabilistic programming is a growing field that involves representing probabilistic models as executable code [8,13,22–25]. This approach has enabled researchers to formalize, automate and scale up many aspects of modelling and inference, making them more accessible to a broader audience of developers and domain experts. By integrating modelling and inference approaches from multiple domains, this technology has also led to the development of new programmable AI systems. Probabilistic programming is widely used for formulating and solving problems in statistics and data analysis. Stochastic programming languages, such as STAN and BUGS [26], provide a formal language that gives simple, uniform and re-usable descriptions of a wide range of models, supporting generic inference techniques. Pyro, a probabilistic programming language written in Python and supported by PyTorch, enables flexible and expressive deep probabilistic modelling, unifying deep learning and Bayesian modelling [27].

Much of the mainstream discussion falls into two camps. First, for 'conventional' programming languages such as Python, the effort centres around support for wrapping neural computations in program code, as in the case of Pyro (and STAN to some extent). On the other extreme, support for functional programming and higher-order programming languages, and how a denotational semantics should be assigned to operational constructs in such languages, remains a challenge, especially constructs such as 'sampling' from a, say, normal distribution [28]. The correct sampling strategy for programmatic code, especially with loops and higher-order functions, and how these samples converge for robust inference, is also a challenging problem [24].

Amidst this, an interesting entrant is probabilistic logic programming (PLP) [10,29–35]. This concept involves the integration of probabilities into the rules of a logic program. (Although we largely focus on PROBLOG here, there is a history of interesting proposals for combining logic programming and probability, as hinted above; see [36].) Initially, the idea was to generate all possible proofs for each random choice and determine the probabilities of these 'worlds' [15]. However, recent advancements have led to the development of encoding strategies of programs to the task of model counting [37]. Interestingly, this problem task can be applied to many other representations, including factor graphs, relational Bayesian networks and Markov logic networks [38]. All this instantiates an incredibly powerful pipeline that offers exact inference.

In our view, four key features set probabilistic logic programming apart from other probabilistic programming techniques. Firstly, probabilistic logic programming exhibits elaboration tolerance, which means that it can easily incorporate new knowledge without having to change the entire program. Secondly, it allows for the incorporation of relational knowledge, which enables modelling of random objects and their properties. Thirdly, it can handle constraints, which makes it useful for solving optimization problems. Lastly, it supports modular non-probabilistic computation, which makes it easier to develop larger and more complex systems.

We do not mean to suggest that PLPs alleviates the problems or the representational challenges of classical probabilistic programs (PPs). To a large extent, the foci and applications of classical probabilistic programming are orthogonal to those of PLP. While it is possible to reconstruct some simple examples from PPs to PLPs, a deeper study is clearly needed. Additionally, the integration of functional artefacts and higher-order programming in the context of PLPs requires further study, despite existing work on integrating logic programming and higher-order programming [39].

Let us make these aforementioned features concrete using a few examples. Recall that declarative programming is a programming paradigm that expresses the logic of a computation without explicitly defining how the individual steps are to be executed. We describe the 'what' of the program rather than the 'how to accomplish it'. What might this paradigm mean in a probabilistic context? We take this to mean that (i) the instructions and logic for decision making are fixed, but (ii) the probabilities, which are learned from data anyway, can be updated without needing to manipulate the logical instructions. Additionally, (iii) computing with these probabilities can be done without requiring input from the user. As we will show, a language like PROBLOG (but also other probabilistic logical languages) can accomplish this easily.

Suppose there is an infectious disease spreading through a population. If two people are in regular contact and one is infected, there is a 0.6 probability that the other person will also become infected. The goal is to predict the spread of the disease given a set of initially infected people and a graph of connections between individuals in the population. We might use the following program:

```
person(a).
person(b).
0.1::inf(X) :- person(X).
0.1::contact(X,Y) :- person(X), person(Y).
0.6::inf(X) :- contact(X, Y), inf(Y).
query(inf(_)).
```

This statement indicates that a person chosen at random has a 0.1 probability of being infected. Two people chosen at random have a 0.1 probability of being in contact with each other. Finally, given a randomly chosen infected person and another person in contact with that infected person, there is a 0.6 probability that the second person will become infected.

It is clear that if we need to update the probabilities or make any of them deterministic, the logical rules are unaffected. We would only need to update the numerical values. For example, to make the contagion deterministic, we may replace the fifth line above with:

```
inf(X) :- contact(X, Y), inf(Y).
```

Indeed, McCarthy's notion of elaboration tolerance was defined as a property of a formalism that makes it convenient to modify a set of facts to accommodate new phenomena or changed circumstances. This means that the formalism can easily incorporate new knowledge without having to change the entire program. He envisoned that there are different kinds of elaboration, with the simplest being the addition of new formulae, which he called 'additive elaborations'. Additionally, a second type of elaboration is changing the values of parameters. Although he ultimately explored these notions using a different set of examples, it is easy to see that in a very concrete sense, elaboration tolerance is seen to be supported in PLPs. Updating the probabilities or even making probabilistic assertions deterministic is easily accomplished, as seen below (example adapted from [40,41]):

```
0.2::inf(X) :- person(X).
0.2::contact(X,Y) :- person(X), person(Y).
inf(X) :- contact(X, Y), inf(Y).
```

In this case, the priors on infection and contact were increased to 0.2—probably due to new data—and the spread of infection among people in contact with each other is categorical.

If we are now to incorporate multiple sources of infection (so-called inhibition effects [40]):

```
0.1::inf(X) :- person(X).
0.1::contact(X,Y) :- person(X), person(Y).
0.1::groundzero(X) :- person(X).
0.6::inf(X) :- contact(X, Y), inf(Y).
0.2::inf(X) :- groundzero(X).
```

We need then a noisy-or structure where the parents independently influence a joint effect. All of this is accomplished without manipulating the remaining rules, and the computing of the probabilities is completely hidden from the user.

Finally, we can further contextualize influence, by allowing for, say, a distinguished group of vulnerable people, while not adjusting the weights or the rules previously constructed.

```
0.1::inf(X) :- person(X).
0.1::contact(X,Y) :- person(X), person(Y).
0.1::groundzero(X) :- person(X).
0.1::susceptible(X) :- person(X).
0.6::inf(X) :- contact(X, Y), inf(Y), \+ susceptible(X).
0.8::inf(X) :- contact(X, Y), inf(Y), susceptible(X).
0.2::inf(X) :- groundzero(X).
```

The program now additionally says that a vulnerable person has an increased chance of catching the infection.

It is worth noting that such languages could serve as an interface for other AI systems, including natural language interactions as well as deep learning models. For example, in [42], questions of the following sort can be parsed into PROBLOG programs.

2.1. You roll a **fair six-sided die twice**. What is the probability that the first roll shows a five and the second roll shows a six?

Here, the bold text is clearly a probabilistic model, and the rest a query. Correspondingly, the text is parsed and tokenized into a PROBLOG program of the following sort (with some additional syntax for probability and combinatorics constructs, including taking with replacement):

```
group(die). size(die, 6).
given(exactly(1, die, one)).
…
given(exactly(1, die, six)).
take_wr(die, rolls, 2).
probability(and(nth(1, rolls, five), nth(2, rolls, six))).
property(outcome(0),[ one, two, three, four, five, six] ).
```

Running this program then returns the answer to the question.

Conversely, in [30,43], logical reasoning is used to signal deep learning models to learn distributions that respect constraints. Underlying these models is an extremely simple but ubiquitous computational task called *weighted model counting* [44]. An extension to that task for continuous models [45] has enabled a logic-based solver strategy for a range of 'non-logical' (i.e. classical) probabilistic programming languages in the recent years [46–48].

3. HYPE

PROBLOG aims to unify logic programming and probabilistic specifications, in the sense of providing a language to specify distributions together with the means to query about the probabilities of events. As a very simple example, to express that the object c is on the table with a certain probability, and that all objects on the table are also in the room, we would write (free variables are assumed to be quantified from the outside):

```
0.6::onTable(c).
inRoom(x) :- onTable(x).
```

This then allows us to query the probability of atoms such `inRoom(c)`.

A more recent extension [10] geared for continuous distributions and other infinite event-space distributions allows the head atom of a logical rule to be drawn from a distribution directly, by means of the following syntax:

```
h ~ D :- b1,…, bn.
```

For example, suppose there is an urn with an unknown number of balls [13]. Suppose we pull a ball at a time and put it back in the urn, and repeat these steps (say) six times. Suppose further we have no means of identifying if the balls drawn were distinct from each other. A probabilistic program for this situation might be as follows:

```
n ~ poisson(6).
pos(x) ~ uniform(1,10) :- between(1,~(n),x).
```

For simplicity, we assume here that the physical form of the urn is a straight line of length 10, and the position of a ball is assumed to be anywhere along this line.

HYPE is based on a dynamic extension that allows us to temporally index the truth of atoms, and so can be used to reason about actions. For example, the program

```
numBehind(x,t+1) ~ poisson(1) :- removeObj(x,t).
```

says that on removing the object x at t, we may assume that there are objects—typically one such object—behind x. Such programs can be used in object tracking applications to reason about occluded objects.

A common declaration in many robotics applications [2] is to define actions and sensors with an error profile, such as a Gaussian noise model. These can be instantiated in HYPE using:

```
pos(x, t+1) ~ gaussian(~(pos(x,t)) + 1, var) :- move(x,t).
obs(x,t+1) ~ gaussian(~(pos(x,t)), var).
```

The first rule says that the position of x on doing a move action is drawn from a normal distribution whose mean is x's current position incremented by one. The second one says that observing the current position of x is subject to additive Gaussian noise.

As an automated planning system, HYPE instantiates a MDP [49]. Recall that MDPs are defined in terms of states, actions, stochastic transitions and reward functions, which can be realized in the above syntax using rules such as:

```
poss(act, t) ~ conditions(t).
reward(num, t) :- conditions(t).
```

As an example, imagine writing a program to instantiate an agent whose goal is to find certain types of objects (say, boxes), and move them to a specific location. (The example is adapted from [9].) Firstly, we would need to capture the frame problem [50] to say that type of the object does not change on moving it to another location:

```
type(X,t+1) ~ val(T) :- ~(type(X,t)) = T, not(removeObj(X)).
```

We might also say that boxes usually have one or more objects behind them:

```
numObjBehind(X,t+1) ~ poisson(1) :- ~(type(X,t)) = box,removeObj(X).
```

Finally, we might give a reward to the agent when it finally has the can at hand:

```
stop(t) :- ~ (type(X,t)) = can.
reward(20,t) :- stop(t).
reward(-1,t)
:- not(stop(t)).
```

To compute a policy, which is a mapping from states and time points to actions, HYPE combines importance sampling and SLD resolution to effectively bridge the high-level symbolic specification and the probabilistic components of the programming model. HYPE allows states and actions to be discrete or continuous, yielding a general planning system. Empirical evaluations are reported in [9,51].

In sum, HYPE benefits from all the advantages and modelling capabilities that have demonstrated through PROBLOG. But as can be seen, it does not quite have a distinct action language or an easy way to represent actions other than as constants indexed by time variables. This is where the situation calculus differs slightly, as it directly models a dynamic world. This allows for more flexible language for describing effects and actions, discussed below.

4. ALLEGRO

The GOLOG language has been successfully used in a wide range of applications involving control and planning [16], and is based on a simple ontology that all changes are a result of named actions. It is a high-level agent programming language built on the situation calculus [12], which is a dialect of first-order logic with (some) second-order logical features. An initial state describes the truth values of properties, and actions may affect these values in non-trivial context-sensitive ways. In particular, GOLOG is a programming model where executing actions are the simplest instructions in the program, upon which more involved constructions for iteration and loops are defined. For example, a program to clear a table containing an unknown number of blocks would be as follows: $([\pi \texttt{x}\ \texttt{onTable(x)?};\ \texttt{removeObj(x)}])^*;\ \neg\exists \texttt{x}\ \texttt{onTable(x)?}$

Here, π is the non-deterministic choice of argument, semi-colon denotes sequence, ? allows for test conditions, and * is unbounded iteration. The program terminates successfully because the sub-program before the final test condition removes every object from the table.

As argued in [16], the rich syntax of GOLOG allows us, on the one hand, to represent policies and plans in an obvious fashion; for example: $(\texttt{a}_1;\ \ldots;\ \texttt{a}_n;\ \texttt{P?})$ ensures that the goal P is true on executing the sequence of actions. However, the syntax also allows open-ended search; for example: $(\texttt{while}\ \neg\texttt{P}\ \pi\texttt{a}.\ \texttt{a})$ tries actions until P is made true. The benefit of GOLOG then is that it allows us to explore plan formulations between these two extremes, including partially specified programs that are completed by a meta-language planner.

ALLEGRO augments the underlying ontology to reason about probability distributions over state properties, and allow actions with uncertain (stochastic) effects. In logical terms, the semantical foundations rest on a rich logic of belief and actions. Consequently, it can handle partial probabilistic specifications. For example, one can say c is on the table with a certain probability as before: $\texttt{pr(onTable(c))} = 0.6$, but it is also possible to express the probability that there is an object on the table without knowing which one: $\texttt{pr}(\exists \texttt{x onTable(x)}) = 0.6$. We can go further and simply say that there is a non-zero probability of that statement: $\texttt{pr}(\exists \texttt{x onTable(x)}) > 0$, which means that any distribution satisfying the formula is admissible. Such a feature can be very useful: for example, in [52], it is argued that when planning in highly stochastic environments, it is useful to allow a margin of error in the probability distributions defined over state properties.

To model the case of Gaussian error models, actions with uncertain effects are given a general treatment. For one thing, the effects of actions are axiomatized using the notion of successor state axioms which incorporate Reiter's solution to the frame problem [12]. So, for example, changing the position of an object using a move action can be expressed as:

$$\texttt{pos(x, do(a, s))} = \texttt{u} \equiv (\texttt{a} = \texttt{move(x, y)} \wedge \texttt{pos(x, s)} = \texttt{u} + \texttt{y}) \vee (\texttt{a} \neq \texttt{move(x, y)} \wedge \texttt{pos(x, s)} = \texttt{u}).$$

This says that if the action of moving x was executed, its position (along a straight line) is decremented by y units, and for all other actions, the position is unaffected. To deal with uncertain effects, we will distinguish between what the agent intends and what actually happens. That is, let *move*(x, y, z) be a new action type, where y is what the agent intends, and z is what happens. Then, the successor state axiom is rewritten as follows:

$$\texttt{pos(x, do(a, s))} = \texttt{u} \equiv (\texttt{a} = \texttt{move(x, y, z)} \wedge \texttt{pos(x, s)} = \texttt{u} + \texttt{z}) \vee (\texttt{a} \neq \texttt{move(x, y, z)} \wedge \texttt{pos(x, s)} = \texttt{u}).$$

The story remains essentially the same, except that z determines the actual position in the successor state, but it is not in control of the agent. A Gaussian error profile can be accorded to this action by means: $\texttt{l(move(x, y, z), s)} = \texttt{gaussian(z; y, var)}$. That is, the actual value is drawn from a Gaussian whose mean is the intended argument y. Analogously, attributing additive Gaussian noise in a sensor for observing the position is defined using: $\texttt{l(obs(x, z), s)} = \texttt{gaussian(z; pos(x, s), var)}$. That is, the observation z is drawn from a Gaussian whose mean is the actual position of the object x.

As hinted above, as an extension to GOLOG, the syntax of ALLEGRO is designed to compactly represent full or partial plans and policies in a general way. Previously, GOLOG programs have been shown to control the plan search space [17], and recent work has shown that probabilistic action programs are a more compact way to represent recursive policies. On termination, ALLEGRO programs can be tested for, say, the (subjective) probability that the goal is believed to hold.

For example, imagine you wish to get a robot with noisy sensors and effectors close to a wall. The robot would move some distance, but because the action is noisy, it would need to repeatedly sense to ensure it did actually get closer. This sense-act loop would repeat until it believes with high probability that it is actually close to the wall. An ALLEGRO program for such a protocol might look like this:

```
WHILE Pr(distance <= 2)
    forward(1) | sonar;
ENDWHILE
```

So we recur on either moving or sensing (declared using the non-deterministic branch operator) until the robot believes the distance to the wall is less than, say, 2 units. Here, the action `forward(1)` denotes an action *forward(x,y)*, where the intended argument is ($x = 1$), but the actual argument is, as discussed above, unknown to the agent. Since the well-defined terms in the language will need a distinct value for y, we will need the value for y, as determined by, say, nature or the environment. However, the belief of the agent will not be affected by this value, since the value is not observed. So any randomly chosen value for y would also suffice to understand program correctness.

Somewhat analogously, `sonar` might denote the action *sonar(z)*, where z is the value read on the sensor. But unlike the actual argument for actions above, the value read does matter to the beliefs of the agent, because it adjusts its beliefs based on what was observed. So we need to incorporate these read values to ascertain if the program terminates.[2] The semantical foundations of ALLEGRO incorporating these ideas was established in [11] with a discussion on its empirical behaviour.

The above program fully specifies the behaviour, but at the same time, it is capturing a very simple example. A more complex problem setting might involve unspecified sub-programs (which might involve planning), as discussed in [16]. Alternatively partially specified programs might be used to define and constrain the search space, as seen in [17].

[2]It might be possible, of course, to define a notion of program correctness that works for any observed value, in a style similar to the offline computation of policies [53–55]. We gloss over this issue for now.

5. A qualitative comparison

Before concluding the article, it is an interesting exercise to briefly compare the features and properties of the two formalisms. Of course, readers may have the impression that such a comparison would be richer with an empirical evaluation, perhaps by benchmarking the two formalisms against each other in a set of domains, to compare their computational performance. Although this is feasible for a particular problem space (and worth considering for the future), the languages are not fully comparable, as the discussion below will demonstrate.

5.1. Static modelling

We refer to static modelling here to mean capturing a probability distribution over a finite set of random variables. Given a number of atoms in a propositional language (for simplicity), assume atoms are accorded probabilities and these are taken together with non-probabilistic definitive clauses, of the form $h \leftarrow b_1, \ldots, b_k$. For example, here is a PROBLOG program for the burglary-earthquake-alarm Bayesian network example by Pearl [56]:

```
0.1::b.
0.2::e.
a :- b.
a :- e.
query(a).
```

That is:

— b is a proposition standing for burglary and has a probability of 0.1 of occurrence;
— e is a proposition standing for earthquake and has a probability of 0.2 of occurrence;
— $a \leftarrow b$ is a non-probabilistic rule that says that burglary can trigger the alarm, captured using proposition a; and
— $a \leftarrow e$ is a non-probabilistic rule that says that earthquake can trigger the alarm too.

The reader may now construct a set of possible worlds to work out the marginal probabilities. With three propositions, we have eight possible worlds in principle. However, there are some caveats to how we read these atoms and rules in PROBLOG. The atoms for which we have provided probabilities are called *probabilistic facts* and should be treated separately from the heads of rules. In fact, the rules would be used in conjunction with each of the possible worlds (defined purely over the probabilistic atoms) to determine if the head can be derived. Based on that, the success probability of the head is calculated.

Putting that together, we have four worlds over the probabilistic facts b and e. The probability of the worlds is obtained by taking the product of the probabilities of literals built from b and e, with the understanding that the probability of any proposition p and its negation must sum up to 1. So, for example, in a world where b is true but e is false, we would obtain the probability of $0.1 \times (1 - 0.2)$ which is 0.08. By extension we have

1. the world $b \wedge \neg e$ with probability 0.08 as just determined;
2. the world $b \wedge e$ with probability $0.1 \times 0.2 = 0.02$;
3. the world $\neg b \wedge e$ with probability $(1 - 0.1) \times 0.2 = 0.18$; and
4. finally, the world $\neg b \wedge \neg e$ with probability $(1 - 0.1) \times (1 - -0.2) = 0.72$.

To now calculate the probability of triggering the alarm, we look at each possible word together with the non-probabilistic rules and see if a is entailed. It is easy to see that this only happens with the first three worlds: here, either burglary or alarm is true and so it immediately follows that a holds. Thus, the probability of triggering the alarm is 0.28. See [37] for the full semantic treatment.

Thus, the semantics of logic programs seem particularly conducive to modelling Bayesian networks [44]. By contrast, the semantics of ALLEGRO is based on probabilities on first-order structures. So it is not quite the exact same set of formulae needed to replicate that network. We need to slightly adapt it. For one thing, there is nothing in semantics that suggests that certain atoms should not have probabilities, as it is implicitly assumed in the semantics of PROBLOG programs. So, for instance, we could use the following formulae in ALLEGRO for the network:

— $\Pr(b) = 0.1$;

— $\Pr(e) = 0.2$;
— $a \equiv b \vee e$ and $\Pr(a) = 1$.

In ALLEGRO, then, because there are three propositions we indeed have eight possible worlds in principle. The last formula, however, ensures that worlds where b and e are false but a is true are impossible.

If we now contrast this with the four possible worlds we had for PROBLOG, we can indeed see that it is the same set of truth-settings for b and e (the first three) that would be considered for the probability of a. The fourth world $\neg b \wedge \neg e$ cannot be one where a is true, as established. However, the calculation of the probabilities of worlds is obtained in a different manner, but amounting to the same specification. The formulae, for example, state that all the worlds where b is true need to be assigned a total probability of 0.1. This essentially means the first and second worlds, (1) and (2) above, need to have their probabilities sum to 0.1. Likewise, the set of all worlds where e is true needs to be assigned a total probability of 0.2. So the second and third worlds, (2) and (3) above, need to have their probabilities sum to 0.2. By working through these, it can be shown that the probability of the alarm being triggered is still 0.28.

The fact that the formulae need to be mapped this way between the formalisms may seem strange at first glance, but it is not surprising. A translation of PROBLOG programs to probabilistic propositional theories [37] relies on these kinds of steps to assign probabilities to propositional worlds, which as we just established, needs to take all the propositions in the language into account.

Of course, it is entirely reasonable to expect modelling languages to be as suitable as needed for the problem at hand. Clearly, when capturing Bayesian networks and other related types of graphical models, PROLOG and HYPE are much more straightforward than, say, specifying probabilities over first-order formulae. For the latter, we need to implicitly determine how these probabilities correspond to a Bayesian network or some other factored representation [57].

However, given that PROBLOG programs are then translated into probabilistic propositional theories, one could imagine that there is a more natural modelling language for users who prefer graphical models. Then an intermediate step would translate it into an ALLEGRO model so that the initial knowledge can be used with a rich theory of actions. Moreover, there is one case where the static knowledge base between these formalisms may not require much reworking. Consider probabilistic databases [58]: the simplest setting is that of *tuple-independent* probabilistic databases, where each proposition is assigned a distinct probability, and assumed to be probabilistically independent from others. In such a setting, the representation in both formalisms would be virtually identical because the probability of each world is essentially specified explicitly: it is obtained from the product of the probability of the propositions true at the world.

There is one aspect in which the formalisms differ deeply. Logic programs allow the use of inductive definitions [59]. For example, we might use the following two formulae to capture an inductive definition for paths in a graph:

— $\forall x, y(E(x, y) \supset P(x, y))$; and
— $\forall x, y, z(E(x, y) \wedge P(y, z) \supset P(x, z))$.

This says that every edge is a path, and if there is an edge between nodes x and y and a path from y to z, then there is also a path from x to z.

ALLEGRO is based on the situation calculus, which allows for some second-order features, including an inductive definition for the set of situations [12]. However, it is rarely the case that inductive definitions are provided as part of the initial knowledge base. (Nonetheless, there is some work regarding this in the literature [60].) Here too, PROLOG and HYPE may seem like more attractive formalisms if one is mostly concerned with reasoning about static knowledge.

5.2. Observations and actions

We already discussed how these languages do allow the modelling of dynamics, but mainly as captured by dynamic Bayesian networks, hidden Markov models, MDPs, POMDPs and other related frameworks [29,36]. This is achieved by introducing a time argument for the predicates, and reasoning about a certain number of time steps.

Nonetheless, such an approach makes these languages more limited compared with dynamic languages like the situation calculus. In the situation calculus, not only is it possible to have durative actions where time is an additional argument [61], but the dynamics of states can be captured directly using named actions. As a result, it becomes natural to capture numerous planning languages [12].

The situation calculus has also been extended to reason about decision-theoretic models [62] and relational extensions to POMDPs [20]. Its first-order nature also makes it a candidate for reasoning about unbounded numbers of objects and unbounded time sequences. For example, it is possible to ask *projection queries* of the following sort:

— Is ϕ true after doing actions a_1 to a_k?
— Was ϕ true at every step of the sequence, or conversely was ϕ made false at any point between the start and the end of the sequence?
— Will ϕ be true for every sequence executed from here on?

Thus, not only can one write programs in the language, but one can also verify the properties of these programs directly within the language [63].

It is also worth noting that the action types themselves are probably more powerful than what can be provided in a formalism such as a dynamic Bayesian network [64]. For example, in the context of the Bayesian network used previously on the alarm getting triggered, one can write actions that can potentially change the edges between nodes in an arbitrary fashion and add new nodes. We can also reason about the ramifications and indirect effects of actions [65].

5.3. Synthesis versus policy execution

As was made clear from the presentation of HYPE, that language is primarily for synthesizing a policy. That is, it specifies actions and effects and the noise models, and the purpose of the formalism is to compute a policy from that specification. We can contrast that to ALLEGRO in two ways. Firstly, belief-based programs are policies in themselves. So, executing the program is tantamount to executing a policy for a POMDP under the appropriate representational considerations [66]. But of course, such a policy could also be synthesized. There is considerable work on the induction of recursive program-like structures [67]. Thus, the situation calculus provides a comprehensive framework to reason about different sorts of policies and verify their properties.

5.4. Incomplete and partial specifications

Perhaps all of these different dimensions are essentially a statement about the differences in expressiveness. While PROBLOG and HYPE mainly provide a framework for specifying probabilistic models, the situation calculus provides a language for reasoning about probabilistic knowledge and its dynamics more generally. For instance, it is possible to express multiple belief distributions, specify programs either completely or partially, and combine probabilistic knowledge with non-probabilistic, incomplete knowledge (which then necessitates the need for multiple distributions) [68].

All of this means that the situation calculus language is far more expressive because it cannot be reduced to a propositional model. Recall that PROBLOG reduces its inference to a weighted propositional formula, whereas the situation calculus corresponds to a modal first-order logic [69].

Consequently, due to this expressiveness, it can be shown that verifying properties in the full language is highly undecidable [63]. In the case of ALLEGRO, verifying properties would correspond to checking the properties of possibly infinitely many POMDPs.

Of course, the full expressiveness need not be embraced. Depending on the application, we might choose a fragment to work with, perhaps a fragment that closely resembles PROBLOG and HYPE. This is the argument the knowledge representation community have been making all along [12,16]: sometimes it is useful to have a more expressive language to study the properties of the desired problems and then we would choose a restricted fragment to build the domain application at hand.

6. Conclusion

Automated planning is often deployed in an application context, and in highly stochastic and uncertain domains, the planning model may be derived from a complex learning and reasoning pipeline, or otherwise defined over non-trivial state spaces with unknowns. In this article, we reported on two probabilistic programming systems to realize such pipelines. Indeed, combining automated planning and probabilistic programming is receiving considerable attention recently, e.g. [14]. These languages are general purpose, and their first-order expressiveness can not only enable a compact codification of the domain but also achieve computational leverage.

One of the key concerns with the use of probabilistic programming and stochastic specifications generally is that most systems perform inference by Monte Carlo sampling. As is well known, one is able to only obtain asymptotic guarantees with such methods, and moreover, handling low-probability observations can be challenging. In that regard, there have been recent logical approaches for inferring in mixed discrete-continuous probability spaces with tight bounds on the computed answers [45,46,70]. Since HYPE, ALLEGRO and many such systems use probabilistic inference as a fundamental computational backbone, the question then is whether the aforementioned approaches can enable robust planning and programming frameworks in stochastic domains.

There may also be added value in finding deeper connections with conventional probabilistic programming languages. For example, quite a bit of work has been done on trying to learn in such conventional languages [71]. Additionally, deep learning techniques can be integrated as computational primitives in such languages, as seen in Pyro. It would be interesting to explore whether such ideas can be incorporated into HYPE and ALLEGRO to unify first-order probabilistic reasoning and learning.

Data accessibility. This article has no additional data.
Declaration of AI use. I have not used AI-assisted technologies in creating this article.
Author's contributions. V.B.: conceptualization, formal analysis, investigation, writing—original draft, writing—review and editing.
Conflict of interest declaration. I declare I have no competing interests.
Funding. No funding has been received for this article.

References

1. Fikes R, Nilsson NJ. 1971 STRIPS: a new approach to the application of theorem proving to problem solving. *Artif. Intell.* **2**, 189–208. (doi:10.1016/0004-3702(71)90010-5)
2. Thrun S, Burgard W, Fox D. 2005 *Probabilistic robotics*. New York, NY: MIT Press.
3. Boutilier C, Dean T, Hanks S. 1999 Decision-theoretic planning: structural assumptions and computational leverage. *J. Artif. Intell. Res.* **11**, 94. (doi:10.1613/jair.575)
4. Domshlak C, Hoffmann J. 2007 Probabilistic planning via heuristic forward search and weighted model counting. *J. Artif. Intell. Res.* **30**, 565–620. (doi:10.1613/jair.2289)
5. Kaelbling LP, Littman ML, Cassandra AR. 1998 Planning and acting in partially observable stochastic domains. *Artif. Intell.* **101**, 99–134. (doi:10.1016/S0004-3702(98)00023-X)
6. Ong SCW, Png SW, Hsu D, Lee WS. 2010 Planning under uncertainty for robotic tasks with mixed observability. *Int. J. Rob. Res.* **29**, 1053–1068. (doi:10.1177/0278364910369861)
7. Goodman ND, Mansinghka VK, Roy DM, Bonawitz K, Tenenbaum JB. 2008 Church: a language for generative models. In *Proc. of the 24th Conf. on Uncertainty in Artificial Intelligence, Helsinki, Finland, 9–12 July*, pp. 220–229.
8. Gordon AD, Henzinger TA, Nori AV, Rajamani SK. 2014 Probabilistic programming. In *Proc. Int. Conf. on Software Engineering*, pp. 167–181. New York, NY: ACM.
9. Nitti D, Belle V, Raedt LD. 2015 Planning in discrete and continuous Markov decision processes by probabilistic programming. In *European Conf. on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, Porto, Portugal, 7–11 September*, pp. 327–342. Springer International Publishing. (doi:10.1007/978-3-319-23525-7_20)
10. Gutmann B, Thon I, Kimmig A, Bruynooghe M, De Raedt L. 2011 The magic of logical inference in probabilistic programming. *Theory Pract. Log.* **11**, 663–680. (doi:10.1017/S1471068411000238)
11. Belle V, Levesque HJ. 2015 ALLEGRO: belief-based programming in stochastic dynamical domains. In *IJCAI'15: Proc. of the 24th Int. Conf. on Artificial Intelligence, July*, pp. 2762–2769.
12. Reiter R. 2001 *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. New York, NY: MIT Press.
13. Milch B, Marthi B, Russell S, Sontag D, Ong D, Kolobov A. 2007 BLOG: probabilistic models with unknown objects. In *Introduction to statistical relational learning* (eds L Getoor, B Taskar), pp. 373–298. Cambridge, MA: MIT Press.
14. Srivastava S, Russell SJ, Ruan P, Cheng X. 2014 First-order open-universe POMDPs. In *UAI'14: Proc. of the 30th Conf. on Uncertainty in Artificial Intelligence, Quebec, Canada, July*, pp. 742–751. Arlington, VA: AUAI Press.
15. Raedt LD, Kimmig A, Toivonen H. 2007 ProbLog: a probabilistic prolog and its application in link discovery. In *Proc. of the 20th Int. Joint Conf. on Artificial Intelligence, Hyderabad, India, 6–12 January*, pp. 2462–2467.
16. Lakemeyer G, Levesque HJ. 2007 Cognitive robotics. In *Handbook of knowledge representation* (eds F van Harmelen, V Lifschitz, B Porter), pp. 869–869. Amsterdam, The Netherlands: Elsevier.
17. Baier JA, Fritz C, McIlraith SA. 2007 Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proc. of the 17th Int. Conf. on Automated Planning and Scheduling, ICAPS 2007, Providence, RI, 22–26 September*, pp. 26–33.
18. Claßen J, Lakemeyer G. 2008 A logic for non-terminating golog programs. In *Principles of Knowledge Representation and Reasoning: Proc. of the 11th Int. Conf., KR 2008, Sydney, Australia, 16–19 September*, pp. 589–599.
19. Sanner S, Delgado KV, de Barros LN. 2011 Symbolic dynamic programming for discrete and continuous state MDPs. In *UAI 2011, Proc. of the 27th Conf. on Uncertainty in Artificial Intelligence, Barcelona, Spain, 14–17 July*, pp. 643–652.
20. Sanner S, Kersting K. 2010 Symbolic dynamic programming for first-order POMDPs. In *Proc. of the 24th AAAI Conf. on Artificial Intelligence, AAAI 2010, Atlanta, GA, 11–15 July*, pp. 1140–1146.
21. Sanner S. 2011 *Relational dynamic influence diagram language (rddl): language description*. Technical report, Australian National University.
22. McCallum A, Schultz K, Singh S. 2009 FACTORIE: probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 22: 23rd Annual Conf. on Neural Information Processing Systems 2009, Vancouver, Canada, 7–10 December*, pp. 1249–1257.
23. Jones C, Plotkin G. 1989 A probabilistic powerdomain of evaluations. In *Proc. of the 4th Annual Symp. on Logic in Computer Science, Pacific Grove, CA, June*, pp. 186–195. IEEE.
24. Mansinghka V. 2009 Natively probabilistic computation. PhD thesis, Massachusetts Institute of Technology. MIT/EECS George M. Sprowls Doctoral Dissertation Award.
25. Pfeffer A. 2001 IBAL: a probabilistic rational programming language. In *Proc. of the 17th Int.Joint Conf. on Artificial Intelligence - Volume 1 IJCAI'01*, pp. 733–740. San Francisco, CA: Morgan Kaufmann Publishers Inc.

26. Korner-Nievergelt F, Roth T, Von Felten S, Guélat J, Almasi B, Korner-Nievergelt P. 2015 *Bayesian data analysis in ecology using linear models with R, BUGS, and Stan*. New York, NY: Academic Press.
27. Bingham E *et al.* 2019 Pyro: deep universal probabilistic programming. *J. Mach. Learn. Res.* **20**, 973–978.
28. Staton S, Yang H, Wood F, Heunen C, Kammar O. 2016 Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proc. of the 31st Annual ACM/IEEE Symp. on Logic in Computer Science*, pp. 525–534.
29. Nitti D. 2016 Hybrid probabilistic logic programming. PhD thesis, KU Leuven, Leuven, Belgium.
30. Manhaeve R, Dumancic S, Kimmig A, Demeester T, De Raedt L. 2018 Deepproblog: neural probabilistic logic programming. *In Advances in neural information processing systems* **31**.
31. Sanghai S, Domingos P, Weld D. 2005 Relational dynamic Bayesian networks. *J. Artif. Intell. Res.* **24**, 759–797. (doi:10.1613/jair.1625)
32. Ng R, Subrahmanian V. 1992 Probabilistic logic programming. *Inf. Comput.* **101**, 150–201. (doi:10.1016/0890-5401(92)90061-J)
33. Ngo L, Haddawy P. 1995 Probabilistic logic programming and Bayesian networks. In *Algorithms, Concurrency and Knowledge, ACSC 1995* (eds K Kanchanasut, JJ Lévy), pp. 286–300. Lecture Notes in Computer Science, vol 1023. Berlin, Germany: Springer. (doi:10.1007/3-540-60688-2_51)
34. Poole D. 2011 Logic, probability and computation: foundations and issues of statistical relational AI. In *Logic Programming and Nonmonotonic Reasoning: 11th Int. Conf., LPNMR 2011, Vancouver, Canada, 16–19 May*, vol. 6645, pp. 1–9. Berlin, Germany: Springer.
35. Raedt LD, Kersting K. 2008 Probabilistic inductive logic programming. In *Probabilistic inductive logic programming* (eds L De Raedt, P Frasconi, K Kersting, S Muggleton). Lecture Notes in Computer Science, vol 4911. Berlin, Germany: Springer. (doi:10.1007/978-3-540-78652-8_1)
36. De Raedt L, Kimmig A. 2015 Probabilistic (logic) programming concepts. *Mach. Learn.* **100**, 5–47. (doi:10.1007/s10994-015-5494-z)
37. Fierens D, den Broeck GV, Thon I, Gutmann B, Raedt LD. 2011 Inference in probabilistic logic programs using weighted CNF's. In *Conf. on Uncertainty in Artificial Intelligence*, pp. 211–220.
38. Van den Broeck G, Meert W, Darwiche A. 2014 Skolemization for weighted first-order model counting. In *KR'14: Proc. of the 14th Int. Conf. on Principles of Knowledge Representation and Reasoning, July*, pp. 111–120.
39. Nadathur G, Mitchell DJ. 1999 System description: teyjus-a compiler and abstract machine based implementation of lambda-prolog. In *Automated Deduction — CADE-16*. Lecture Notes in Computer Science, vol. 1632, pp. 287–291. Berlin, Germany: Springer. (doi:10.1007/3-540-48660-7_25)
40. Meert W, Struyf J, Blockeel H. 2008 Learning ground CP-Logic theories by leveraging Bayesian network learning techniques. *Fundamenta Informaticae* **89**, 131–160.
41. Meert W, Vennekens J. 2014 Inhibited effects in CP-logic. In *Probabilistic Graphical Models: 7th European Workshop, PGM 2014, Utrecht, The Netherlands, 17–19 September. Proceedings 7*, pp. 350–365. Berlin, Germany: Springer.
42. Dries A, Kimmig A, Davis J, Belle V, De Raedt L. 2017 Solving probability problems in natural language. In *Proc. of the 26th Int. Joint Conf. on Artificial Intelligence, Melbourne, Australia*, pp. 3981–3987. (doi:10.24963/ijcai.2017/556)
43. Hoernle N, Karampatsis RM, Belle V, Gal K. 2022 Multiplexnet: towards fully satisfied logical constraints in neural networks. In *Proc. of the 36th AAAI Conf. on Artificial Intelligence*, pp. 5700–5709. (doi:10.1609/aaai.v36i5.20512)
44. Chavira M, Darwiche A. 2008 On probabilistic inference by weighted model counting. *Artif. Intell.* **172**, 772–799. (doi:10.1016/j.artint.2007.11.002)
45. Belle V, Passerini A, Van den Broeck G. 2015 Probabilistic inference in hybrid domains by weighted model integration. In *Proc. of the 24th Int. Joint Conf. on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, 25–31 July*, pp. 2770–2776. IJCAI Inc.
46. Chistikov D, Dimitrova R, Majumdar R. 2017 Approximate counting in SMT and value estimation for probabilistic programs. *Acta Informatica* **54**, 729–764 . (doi:10.1007/s00236-017-0297-2)
47. Albarghouthi A, D'Antoni L, Drews S, Nori AV. 2017 Fairsquare: probabilistic verification of program fairness. *Proc. ACM Program. Lang.* **1**, 1–30. (doi:10.1145/3133904)
48. Holtzen S, Van den Broeck G, Millstein T. 2020 Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.* **4**, 1–31. (doi:10.1145/3428208)
49. Puterman ML. 1994 *Markov decision processes: discrete stochastic dynamic programming*, 1st edn. New York, NY: John Wiley & Sons Inc.
50. Reiter R. 1991 The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, pp. 359–380. New York, NY: Academic Press.
51. Nitti D, Belle V, De Laet T, De Raedt L. 2017 Planning in hybrid relational MDPs. *Mach. Learn.* **106**, 1–28. (doi:10.1007/s10994-017-5669-x)
52. Kaelbling LP, Lozano-Pérez T. 2013 Integrated task and motion planning in belief space. I. *J. Rob. Res.* **32**, 1194–1227. (doi:10.1177/0278364913484072)
53. Chatterjee K, Chmelík M. 2015 POMDPs under probabilistic semantics. *Artif. Intell.* **221**, 46–72. (doi:10.1016/j.artint.2014.12.009)
54. Corona G, Charpillet F. 2010 Distribution over beliefs for memory bounded Dec-POMDP planning. In *Proc. of the Conf. on Uncertainty in Artificial Intelligence*, pp. 135–142.
55. Pajarinen JK, Peltonen J. 2011 Periodic finite state controllers for efficient POMDP and DEC-POMDP planning. In *Advances in Neural Information Processing Systems 24 (NIPS 2011)*, pp. 2636–2644.
56. Pearl J. 1988 *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Cambridge, MA: Morgan Kaufmann.
57. Richardson M, Domingos P. 2006 Markov logic networks. *Mach. Learn.* **62**, 107–136. (doi:10.1007/s10994-006-5833-1)
58. Suciu D, Olteanu D, Ré C, Koch C. 2011 Probabilistic databases. *Synthesis Lect. Data Manage.* **3**, 1–180. (doi:10.2200/S00362ED1V01Y201105DTM016)
59. Denecker M, Bruynooghe M, Marek V. 2001 Logic programming revisited: logic programs as inductive definitions. *ACM Trans. Comput. Logic* **2**, 623–654. (doi:10.1145/383779.383789)
60. Denecker M, Ternovska E. 2007 Inductive situation calculus. *Artif. Intell.* **171**, 332–360. (doi:10.1016/j.artint.2007.02.002)
61. Pinto J, Reiter R. 1995 Reasoning about time in the situation calculus. *Ann. Math. Artif. Intell.* **14**, 251–268. (doi:10.1007/BF01530822)
62. Boutilier C, Reiter R, Soutchanski M, Thrun S. 2000 Decision-theoretic, high-level agent programming in the situation calculus. In *Proc. of the 17th National Conference on Artificial Intelligence and 12th Conf. on Innovative Applications of Artificial Intelligence, July*, pp. 355–362. AAAI Press.
63. Liu D. 2023 Projection in a probabilistic epistemic logic and its application to belief-based program verification. PhD thesis, RWTH Aachen University, Aachen, Germany.
64. Vassos S, Sardina S, Levesque H. 2009 Progressing basic action theories with non-local effect actions. In *Proc. Commonsense – Proceedings of the 9th Int. Symp. on Logical Formalizations of Commonsense Reasoning*, pp. 135–140.
65. McIlraith SA. 2000 Integrating actions and state constraints: a closed-form solution to the ramification problem (sometimes). *Artif. Intell.* **116**, 87–121. (doi:10.1016/S0004-3702(99)00087-9)
66. Lang J, Zanuttini B. 2012 Knowledge-based programs as plans - the complexity of plan verification. In *ECAI'12: Proc. of the 20th European Conf. on Artificial Intelligence, August*, pp. 504–509.
67. Treszkai L, Belle V. 2020 A correctness result for synthesizing plans with loops in stochastic domains. *Int. J. Approx. Reason.* **119**, 92–107. (doi:10.1016/j.ijar.2019.12.005)
68. Belle V, Levesque HJ. 2018 Reasoning about discrete and continuous noisy sensors and effectors in dynamical systems. *Artif. Intell.* **262**, 189–221. (doi:10.1016/j.artint.2018.06.003)
69. Belle V, Lakemeyer G. 2017 Reasoning about probabilities in unbounded first-order dynamical domains. In *Proc. of the 26th Int. Joint Conf. on Artificial Intelligence (IJCAI-17), Melbourne*, pp. 828–836. (doi:10.24963/ijcai.2017/115)
70. Belle V, Van den Broeck G, Passerini A. 2015 Hashing-based approximate probabilistic inference in hybrid domains. In *Proc. of the 25th Int. Joint Conf. on Artificial Intelligence, IJCAI 2016, New York, NY, 9–15 July*, pp. 4115–4119..
71. Lake BM, Salakhutdinov R, Tenenbaum JB. 2015 Human-level concept learning through probabilistic program induction. *Science* **350**, 1332–1338. (doi:10.1126/science.aab3050)