# Explorations of causal probabilistic programming approaches for rule-based models of biological signaling pathways

Devon Kohler[1], Jeremy Zucker[2], Vartika Tewari[1], Karen Sachs[3], Robert Ness[4], Olga Vitek[1]

[1]Khoury College of Computer Sciences, Northeastern University, Boston, MA, USA;
[2]Pacific Northwest National Laboratory, Richland, WA;
[3]Next Generation Analytics, Palo Alto, CA 94301; [4]Altdeep.ai

## 1 Introduction

Signaling pathways are the molecular circuits that underlie cellular processes, such as proliferation and cell-cell communication. Dysregulation of these pathways is the source of most morbidities. Since the size and complexity of biological systems defy human intuition, and since exhaustive experimental profiling of system states under perturbation is infeasible, there is an increasing need to simulate signaling pathways, and to forecast the causal effects of experimental or clinical perturbations.

Signaling pathways can be thought of as causal stochastic generative processes. Historically, they are characterized by forward simulation, using algorithms such as Gillespie [MSD04], which generates a trace for a set of stochastic differential equations with known rates. The algorithm works well for finding the average effect over all traces in a population due to a perturbation, however it falls short of understanding the causal effect of one unique trace.

Recently, signaling pathways have been represented using *rule-based models*, where agents and their interactions are viewed as events that cause the system to change to a new state. A transition event is defined by a set of rules applied whenever the system state matches a pattern defined by the rule's prerequisites. Since the transition events are represented at the abstract rule level, they avoid complexities at the concrete reaction level that obscure the underlying causal process [LYF18]. The history of rule applications in a single run generate a trace of transition events over time.

In many cases, we are interested in *counterfactual* queries about signaling pathway behavior, where the chance of the occurrence of an event is defined not only by what happened, but also by what did not happen. We argue that probabilistic programming languages (PPL) offer a robust framework for deriving such counterfactual traces in rule-based models of signaling pathways. Specifically, we explore for this task with Omega, a causal probabilistic programming language, probability trees, which can represent a causal generative or stochastic process, and Kappa, a language for rule-based modeling. We highlight advantages, limitations, and practical implementation challenges of each language for this application. The code is publicly available [1].

## 2 Background

**Kappa** is explicitly designed for rule-based modeling of signaling pathways. It is based on a formalism known as process algebra, and thus has a clear semantics for representing parallel processes. Kappa is

[1]https://github.com/devonjkohler/causal-prog-rule-models

different from most modelling techniques traditionally used by biologists (such as differential equations systems or Petri nets) in that its rules can feature underspecified agents. Kappa is based on Ocaml, a general purpose programming language that emphases expressiveness and safety. It has an efficient implementation and is equipped with parametric polymorphism and type inference, which makes it fast. Recently, Kappa was extended with causal semantics to allow counterfactual resimulation [LYF18], which marries the concurrency community's notion of causality as enablement with Pearl's notion of causality as counterfactual prevention [Pea09], and allows us to evaluate counterfactual statements.

**Omega** is designed for general counterfactual inference [Tav+19a]. Similarly to structured causal models [Pea11], it relies on an abduction-action-prediction style workflow. This workflow involves implementing conditional sampling to learn exogenous variables, performing interventions using a replace operator, and simulating to make predictions in the counterfactual world. In practice, this means conditioning on the entire observed trace, including the time points after the intervention.

For abduction, Omega implements a variety of sampling algorithms. In particular, in complex sample spaces Omega implements the idea of a softened predicate in the interval [0,1]. To converge to exact inference Omega utilizes a temperature parameter and Replica exchange Markov Chain Monte Carlo, varying the temperature across multiple chains to achieve the best results [Tav+19b]. Additionally, Omega utilizes a higher order implementation of Judea Pearl's **do** operator, implemented as **replace** in the code. This operator enables the replacement of a variable's dependencies with not only a constant value, but with any arbitrary function. Finally, Omega transforms how code is evaluated, switching from eager to lazy evaluation, thus enabling the rewiring of variable's dependencies after it has been defined. These program transformations allow the user to represent counterfactual queries by composing intervention and conditioning operators [Tav+19a; Tav+19c].

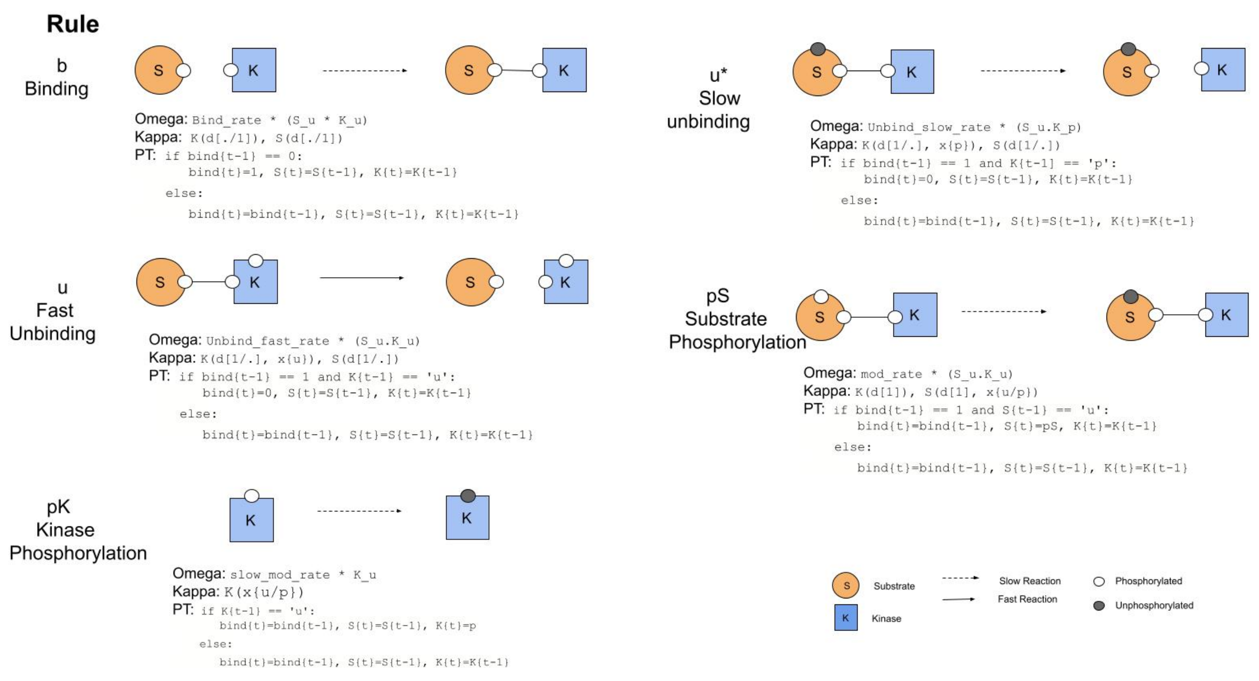
**Probability Trees** A probability tree is a simple model for representing the causal generative process of a random experiment or stochastic process [Sha96]. Each node in the tree corresponds to a potential state of the process, and the arrows indicate both the probabilistic transitions, and the temporal ordering of those transitions. Recently, [Gen+20] developed algorithms that endowed probability trees with causal semantics, enabling them to answer probabilistic and deterministic questions at all 3 levels of Pearl's causal hierarchy [Bar+21]. In this manuscript, we show that probability trees, implemented in python, are useful for interpreting factual simulations and counterfactual resimulations of causal generative models.

## 3 Methods

**Case study** Signaling pathways are typically comprised of chains of *kinases*, i.e. enzymes that chemically attach a phosphate group to substrate proteins. This alters the substrate's configuration, and activates the substrate. If the substrate is itself a kinase, it in turn activates downstream kinases, propagating the activation signal towards a desired biological endpoint.

This case study, previously described in [LYF18], consists of two biological agents, a kinase $K$ and a substrate $S$. Each agent may be in one of two phosphorylation states, phosphorylated or unphosphorylated, and can either be in a free state or bound to each other. Thus at any time the system can be in one of eight different states. Furthermore, five different rules may cause the system to transition to a new state (Figure 1). Each rule has a set of prerequisites described as a pattern over potential states. The system starts in an initial state. After an interval of time has passed, a rule is selected. If the rule's preconditions are satisfied, the rule will fire, and the state will transition. The model can be extended to any number of $S$ and $K$ agents, however for simplicity we will only consider an initial state with one agent of each type.

The observed (factual) trace used for all evaluations is as follows: $init, b, u, pK, b, pS, u^*$. Using this trace as our observed data, we ask the counterfactual question *"Given we observed $pK$ occurring at time $t = 3$ in trace $T$, what would $T$ have been been had $pK$ not occurred at $t = 3$?"*. To answer this counterfactual we conditioned on the observed trace and intervened on $pK$, forcing it to evaluate to any rule other than $pK$.



```
Omega: Bind_rate * (S_u * K_u)
Kappa: K(d[./1]), S(d[./1])
PT: if bind{t-1} == 0:
        bind{t}=1, S{t}=S{t-1}, K{t}=K{t-1}
    else:
        bind{t}=bind{t-1}, S{t}=S{t-1}, K{t}=K{t-1}
```

```
Omega: Unbind_fast_rate * (S_u.K_u)
Kappa: K(d[1/.], x{u}), S(d[1/.])
PT: if bind{t-1} == 1 and K{t-1} == 'u':
        bind{t}=0, S{t}=S{t-1}, K{t}=K{t-1}
    else:
        bind{t}=bind{t-1}, S{t}=S{t-1}, K{t}=K{t-1}
```

```
Omega: slow_mod_rate * K_u
Kappa: K(x{u/p})
PT: if K{t-1} == 'u':
        bind{t}=bind{t-1}, S{t}=S{t-1}, K{t}=p
    else:
        bind{t}=bind{t-1}, S{t}=S{t-1}, K{t}=K{t-1}
```

```
Omega: Unbind_slow_rate * (S_u.K_p)
Kappa: K(d[1/.], x{p}), S(d[1/.])
PT: if bind{t-1} == 1 and K{t-1] == 'p':
        bind{t}=0, S{t}=S{t-1}, K{t}=K{t-1}
    else:
        bind{t}=bind{t-1}, S{t}=S{t-1}, K{t}=K{t-1}
```

```
Omega: mod_rate * (S_u.K_u)
Kappa: K(d[1]), S(d[1], x{u/p})
PT: if bind{t-1} == 1 and S{t-1} == 'u':
        bind{t}=bind{t-1}, S{t}=pS, K{t}=K{t-1}
    else:
        bind{t}=bind{t-1}, S{t}=S{t-1}, K{t}=K{t-1}
```

Figure 1: Five rules in the case study. These are the binding of agents $b$ if the two agents are currently unbound, the fast unbinding of agents $u$ if the kinase is unphosphorylated, the slow unbinding of agents $u^*$ if the kinase is phosphorylated, the phosphorylation of the substrate $pS$ if the substrate is unphosphorylated and bound to the kinase, and the phosphorylation of the kinase $pK$ if the kinase is unphosphorylated. The corresponding code used to calculate the rate at which the rule will fire is included for each implementation.

**Case study implementation in Kappa** We first generated a factual trace using stochastic Kappa Simulator, which inherently supports rule-based models and is equipped with stochastic firing rates [**Kappatools**]. The stochastic simulator samples from the model with a version of the Gillespie algorithm, which accurately mimics the noise and dynamics of a real system. It weighs each rule with respect to the rate and the number of present agents to which it could apply, and uses the weights to randomly choose a rule to apply. Then, it samples from an exponential distribution the time before the rule is applied. The simulator repeats this procedure until there are no more rules to apply, or until the time limit set by the user is elapsed. The generated trace logged every observed event. Further, we used counterfactual re-simulation, another variation of the Gillespie algorithm, to sample a counterfactual trace given the reference trace and the intervention [Gil77; LYF18]. Both KaSim and the counterfactual re-simulation were implemented in OCaml [LYF18].

**Case study implementation in Omega** Similarly to the above, we implemented the Gillespie algorithm in Omega. Since the algorithm contains both probabilistic and deterministic components, the implementation was straightforward. The probabilistic components include the selection of the rule, modeled as a categorical distribution, and the time between reactions, an exponential distribution. Once the rule is selected the state of the system is deterministically updated using a transition matrix. An important distinction between Kappa and Omega is that Omega allows us to intervene on the time component in addition to the rule selection.

First, we used forward simulation to generate a prior trace that matches the observed trace used in all the evaluations. We then ran a new simulation conditioning on the observed trace to obtain a conditioned posterior distribution. To condition on the data we used Replica exchange Markov Chain Monte Carlo sampling with the default temperature values in Omega. With the conditioned posterior we used Omega's "replace" function to intervene on $pK$, and set it to take any value other than $pK$. We then ran one final

```
def KappaModel(bvars, b = 1,
               u = 17, u_star = 2,
               pS = 2, pK = 1,
               tmax=6, semantics='trace'):
    t=0
    if 'Rule0' not in bvars:
        return init( t )
    while f'Rule{t}' in bvars:
        t += 1
    if t <= tmax:
        return apply_rules( bvars, t, b, u,
                  u_star, pS , pK, semantics )
    return None
```

(a)

```
def apply_rules( bvar, t, b, u, u_star, pS, pK,
                          semantics='trace' ):
    rate_state = []
    rate_state.append( binding( bvar, b, t ) )
    rate_state.append( fast_unbinding(bvar, u, t ) )
    rate_state.append( slow_unbinding( bvar, u_star, t ) )
    rate_state.append( substrate_phosphorylation( bvar, pS, t))
    rate_state.append( kinase_phosphorylation( bvar, pK, t))
    rate_sum = sum([rate for rate, state in rate_state])
    if semantics=='trace':
        return [(rate/rate_sum, state)
                for rate, state in rate_state
                if rate > 0]
    else:
        return [(rate/rate_sum, state)
                for rate, state in rate_state]
```

(b)

Figure 2: Python implementation of Probability Trees (a) `KappaModel()` takes as input weights for each rule, corresponding to the probability of them firing, and the number of steps the tree should sample. The function then initializes the defined tree. (b) `apply_rules()` takes the variables that have been bound so far, along with reaction rule firing rates. Each rule is applied to the previous state, and returns the rate at which it will fire, and the next state. The probability that a reaction will fire is equal to its normalized firing rate, and if the previous state does not match the precondition pattern, the rule will return a rate of 0 and the previous state. `semantics='trace'` generates nodes for only those rules that can be fired given the previous state, otherwise it generates a node for every rule, including those with a zero rate rule.

simulation on the conditioned and intervened trace to answer the counterfactual question.

Omega allowed us to express the time component of the model, leading to a prior that better represents the underlying system and stronger inference. These advantages come at the cost of increased complexity. Using time in the model doubles the number of random variables and adds a continuous component. This makes conditioning on the prior more challenging and time consuming.

**Case study implementation in Probability Trees** We used the probability tree factory functions given by [Gen+20] for our implementation. The advantage of using this method is that we can exploit symmetries (e.g. conditional independencies) to code a much more compact description of the probability tree. The probability factory function binds the root node to the initial state and recursively descends the probability tree, binding each node to the application of an event transition and the corresponding new state, given the previous state, up to a pre-specified maximum tree depth. The implementation was similar to Gillespie, in that for each rule it calculated weights that determine selection probabilities. However, unlike Gillespie probability trees do not explicitly select a rule, unless they are conditioned on an observed trace, in which case the observed rule will be updated to a probability of 100%. This has the advantage of allowing us to visually express and track all the potential rule traces and their chance of occurring. Another difference between this method and the others is that it has no notion of the time between rule choices. An example of our implementation can be seen in Figure 2.

## 4 Results

**Kappa** Our implementation was able to reproduce the results from the original manuscript [LYF18]. If we just look at the factual trace we see that $b$ enabled $pS$ as seen in Figure 3(a). But it doesn't tell us about the the importance of $pK$. The counterfactual trace when rule $pK$ was blocked helps understand that had it not been for $pK$, $u$ would have happened, and had $u$ happened $pS$ would not have happened Figure 3(b). Thus, by setting the same seed as in the factual trace and also keeping the events and probabilities as they had exactly happened in the factual trace we make the counterfactual world deterministic and obtain the corresponding counterfactual trace using the resimulation algorithm. The advantage of this causal analysis is that it is based on sequences of events that have really happened during the simulation, and thus more useful than static analysis like contact maps which represent a static summary of all possible states in a model and their potential binding interactions. We also notice that the implementation is highly efficient and runs extremely fast. In addition Kappa is highly intuitive to use for biological problems.

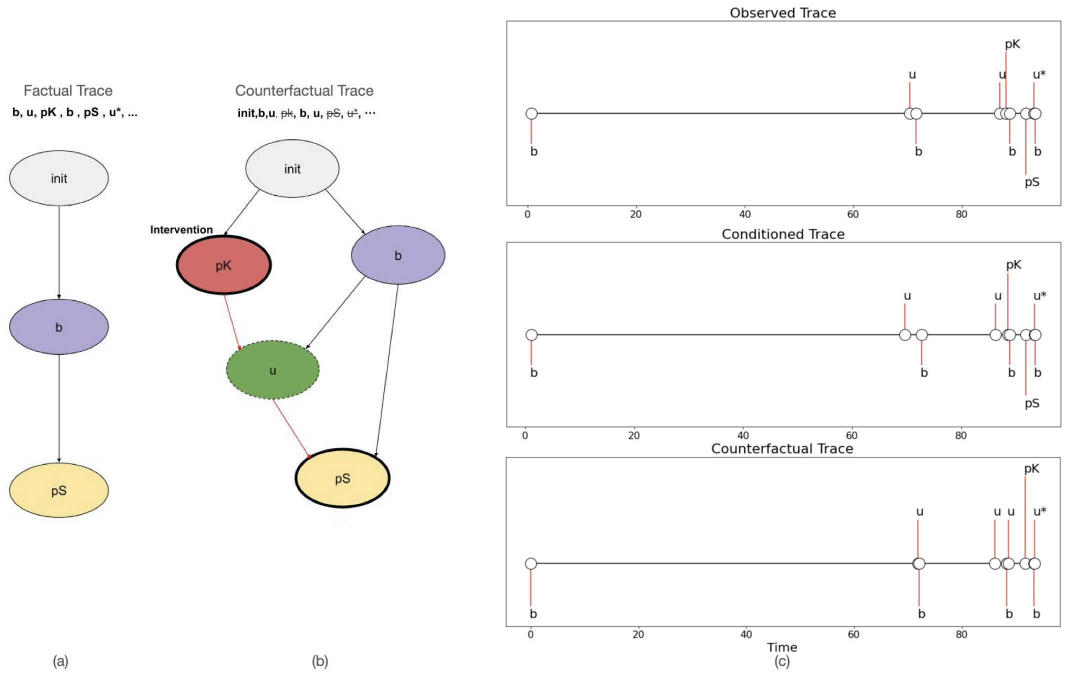


Figure 3: Factual and counterfactual traces, visualized as the output of Kappa and Omega. The factual, or observed, trace is $init, b, u, pK, b, pS, u^*$. The counterfactual trace is $init, b, u, b, u, pK, b, u^*$. (a) Factual trace as implemented in Kappa. The start of the trace, $init, b, u, pK, b$ are included in the $init$ node. (b) Counterfactual trace as implemented in Kappa. Here the prevention arrows, colored red, note events that could not happen after the intervention was applied. (c) Factual, conditional, and counterfactual traces as implemented in Omega. Time is an important variable in the Omega implementation and is included here. Ideally the factual and conditional traces would be identical, however in the in this implementation the two traces have the same rules but the times of the reactions are slightly different due to time being a continuous variable. The counterfactual trace shows the intervention of blocking $pK$ at $t = 3$. In this case $pK$ would still happen, but it would be delayed.

**Omega** The Omega implementation was able to answer our counterfactual question. The conditioned trace converged to the observed trace, learning the endogenous variables that generated the data (Figure 3). There was a visible trade-off in the speed and accuracy of convergence. When the number of replicate exchanges was small convergence was fast, but the conditioned trace was visibly different from the observed data. The difference was especially pronounced in the continuous time component. We decided converging closer to exact inference was worth the increased processing time.

After conditioning, the replace operator was used to block $pK$ at $t = 3$. This resulted in the trace $init, b, u, b, u, pK, b, u^*$ (Figure 3). Omega used information from the factual trace to generate the counterfactual, leading to a unique answer to the our counterfactual question. This is in contrast to intervening and running a forward simulation which would generate a different answer every simulation. In comparison to the other two languages the Omega counterfactual trace is the same as the counterfactual trace for Kappa and it follows one of the likelier outcomes when looking at the interventional probability tree.

**Probability Trees** Four generated probability trees corresponded to the unconditional trace, the factual trace, the intervened trace, and the counterfactual trace Supplementary Figure 4. The unconditional probability tree represented all possible traces starting with the initial conditions, up to a depth of 6. The factual probability tree was the result of conditioning on the factual trace, its only nonzero probability branch corresponding to the factual trace. The interventional probability tree blocked $pK$ from firing at $t = 3$. The counterfactual tree is the result of conditioning on the factual trace and then blocking $pK$ at $t = 3$.

The counterfactual tree resulted in probabilities of 100% for the rules up until the intervention. After the intervention blocked $pK$, a binding rule $b$ took it's place at $t = 3$ with a probability value of 100%. Rule $b$ at $t = 3$ had a 100% chance to fire due to prevention, i.e. if $pK$ could not happen the only other event that could occur was $b$. After $t = 3$, multiple traces that could have occurred. At $t = 4$ the unbinding event $u$ had the highest probability to fire at 85%. Next in $t = 5$ rule $pK$ has the highest chance to fire at 42.5%. These highest probability events coincide with what we saw in both the Kappa and Omega counterfactual traces. While these events had the highest chance of occurring, the probability tree shows all possible events and gives a better idea of the chance of each trace.

## 5 Discussion

In this work the results of Omega and Kappa were very comparable, both outputting the same counterfactual trace. A major distinction is that, while Kappa was designed for rule-based models only, Omega can be extended beyond this model class. This allows Omega to answer queries beyond rule interventions, such as *"what rule would be selected if the time between reactions was different?"*. However, this flexibility comes at a cost. First, in this small toy model Kappa took only 2-3 minutes to answer the counterfactual question, Omega took about 10 minutes. Second, since Omega was not built specifically to address this problem and model type, there are increased implementation challenges for the user. It is possible to both express a model in an inefficient way, and build a model which compiles but makes inference on the model intractable. For example it is possible to implement the rule based Omega model recursively, however it is impossible to reference the recursively called random variables (i.e. they cannot be intervened on). Some of these issues could be fixed with a more refined and efficient implementation by the user.

Omega and Kappa are distinct from the probability trees in two major aspects. First, unlike Omega and Kappa, probability trees do not include a notion of time. More broadly probability trees cannot easily represent continuous variables in this setting. Second, while Omega and Kappa use the information beyond the intervention to predict a single answer to the counterfactual query, probability trees provide an array of potential outcomes with their associated probabilities. The size of tree can quickly become intractable as the potential outcome space increases. With that being said, probability trees are straightforward to implement as they do not require a complicated abduction step to make counterfactual inferences.

Overall, we found that Kappa is best applicable to rule-based problems where the queries of interest involve intervening directly on the rules. Omega is applicable to both rule-based models and more general problems, although it involves a more complex implementation challenge. Probability trees may be best suited for smaller-scale problems where modelers are interested in the range of potential outcomes and their explicit probabilities, without including inference about the state of the system after the intervention occurred.

## 6 Acknowledgments

We thank Zenna Tavares and Jonathan Laurent for their valuable input into the development of this work.

## References

[Gil77] Daniel T. Gillespie. "Exact stochastic simulation of coupled chemical reactions". In: *The Journal of Physical Chemistry* 81 (1977), p. 2340.

[Sha96] Glenn Shafer. *The Art of Causal Conjecture*. 1996.

[MSD04] T. C. Meng, S. Somani, and P. Dhar. "Modeling and simulation of biological systems with stochasticity". In: *In Silico Biology* 4 (2004), p. 293.

[Pea09] J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2009.

[Pea11] J. Pearl. "The algorithmization of counterfactuals". In: *Annals of Mathematics and Artificial Intelligence* 61 (2011), p. 293.

[LYF18] J. Laurent, J. Yang, and W. Fontana. "Counterfactual resimulation for causal analysis of rule-based models". In: *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. 2018, p. 1882.

[Tav+19a] Z. Tavares et al. *A language for counterfactual generative models*. Tech. rep. MIT, 2019.

[Tav+19b] Z. Tavares et al. "Soft constraints for inference with declarative knowledge". In: *arXiv:1901.05437* (2019).

[Tav+19c] Z. Tavares et al. "The random conditional distribution for uncertain distributional properties". In: *arXiv* (2019).

[Gen+20] T. Genewein et al. "Algorithms for causal reasoning in probability trees". In: *arXiv:2010.12237* (2020).

[Bar+21] E. Bareinboim et al. "On Pearl' hierarchy and the foundations of causal inference". In: *Probabilistic and Causal Inference: The Works of Judea Pearl*. 2021.

## 7 Supplementary Figures

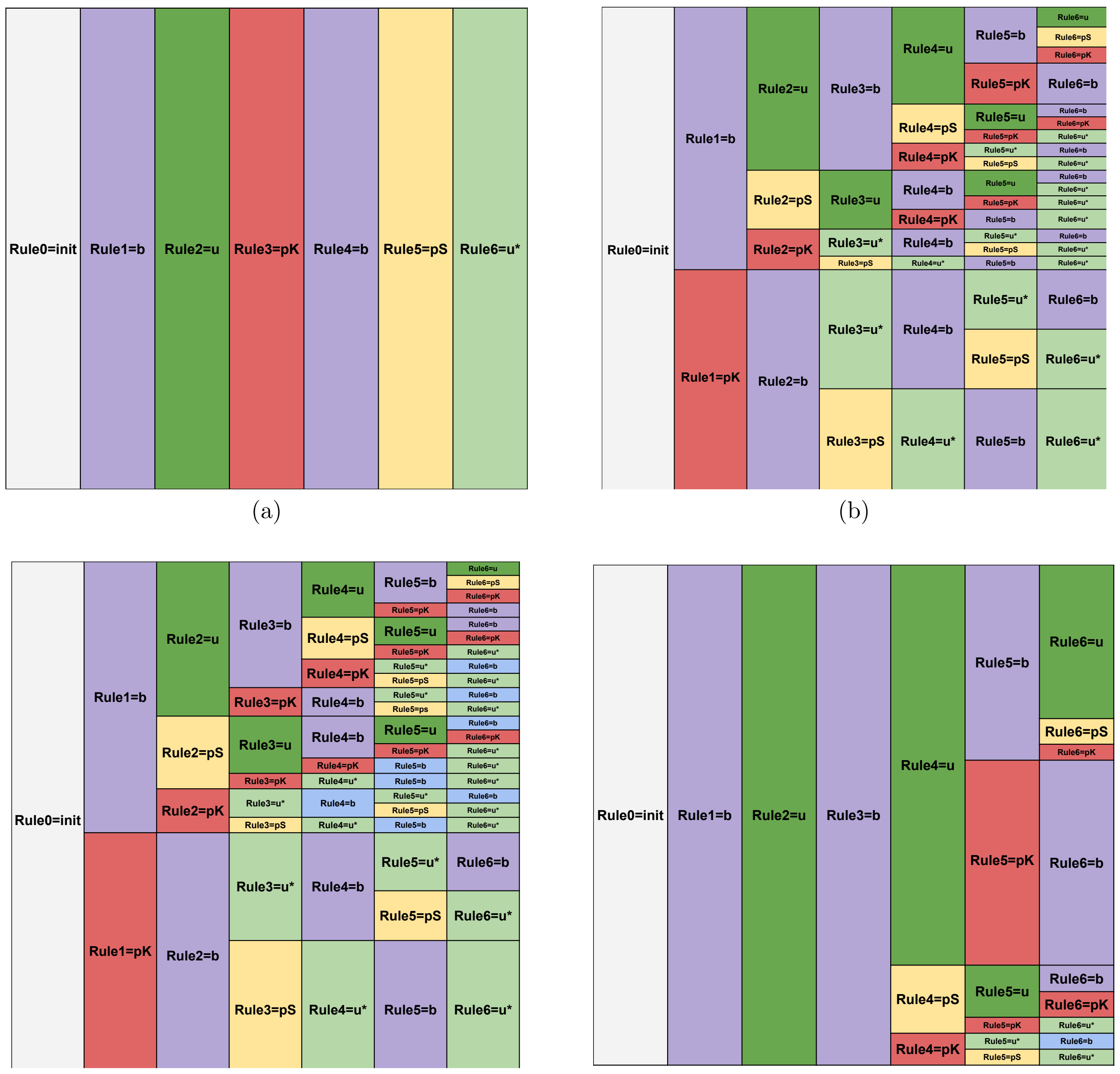


Figure 4: Four probability trees. (a) Factual tree, the probability of each event is 100% and only one rule is possible. (b) Intervened tree, all possible events do not include the intervention rule $pK$ at $t = 3$. (c) The unconditioned tree, all rules are possible with different probabilities. (d) In the counterfactual tree, the events up to the intervention have probability of 100% and then diverge into all the potential paths the trace can take.