Mach Learn (2018) 107:1171–1206
https://doi.org/10.1007/s10994-018-5709-1

CrossMark

Identification of biological transition systems using meta-interpreted logic programs

Michael Bain[1] · **Ashwin Srinivasan**[2]

Received: 31 March 2017 / Accepted: 5 April 2018 / Published online: 22 May 2018
© The Author(s) 2018

Abstract We adopt the principal idea from Plotkin's Structural Operational Semantics (SOS), in which computation by a system is to be understood using: (a) a signature of configurations, Γ; (b) a binary relation ($\rightarrow$) defined over $\Gamma \times \Gamma$; and (c) a meta-interpreter for general transition systems, defined at the level Γ and $\rightarrow$. Using specific definitions for configurations and transition rules, the meta-interpreter generates an operational explanation of a system's behaviour in the form of the stepwise computations (transitions) involved. This setting is of special interest to inductive logic programming (ILP), given recent developments in meta-interpretive learning. We focus here on the specific application of obtaining automatically Petri net models of biological system behaviour. Using a simple logic program as a meta-interpreter with a meta-rule for guarded transitions we show that using definitions of biologically-known transitions, proofs constructed by the meta-interpreter allow us, just as in SOS, to explain system behaviour as stepwise transitions in Petri nets. In the meta-interpretive learning setting, the proofs identify hypotheses that together with the meta-interpreter and domain-knowledge logically entail the observed behaviour. Meta-interpretive learning enables us to go beyond the explanations available in SOS, which are purely deductive, since the meta-interpreter is allowed abductive steps in the proof. This enables us to "invent" transitions which have not been specified in domain-knowledge. We use this facility to deal with noisy data by constructing first a hypothesis that includes abduced transitions, followed by the use of a Viterbi-style computation to find the most likely sequence of transitions for a system with a specified initial and final state. Extensive experiments with some well-known biological systems show that this approach can reliably identify the correct set of transitions even with fairly high levels of noise and with moderate amount of missing values.

Editors: James Cussens and Alessandra Russo.

✉ Ashwin Srinivasan
ashwin.srinivasan@wolfson.oxon.org

[1] School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

[2] Department of Computer Science and Information Systems, BITS-Pilani, Goa Campus, Sancoale, India

1 Introduction

How should we describe a biological system (or, for that matter, any system)? In principle, the mathematical language of differential equations, the differential calculus, and data about initial conditions provide a uniform way to describe relations between variables of interest, and how these change as a function of time, given the initial conditions. But, with biological systems, the equations involved are often highly non-linear, making it difficult to obtain analytical solutions. Inevitably therefore, simplifications of the equations result, and the model progressively reflects less of the underlying biological process being studied. So, a biologist has little choice but to resort to verbal or pictorial descriptions, often in the form of annotated graphs, with much of the kinetics being lost in translation.

Is it possible to obtain the benefits of modeling system kinetics, while retaining the expressivity of more qualitative representations? The use of Petri nets attempts to achieve this to some extent. Comprising a bipartite graph representation, some computational rules, and data consisting of (discrete) initial values of state variables, system-behaviour is a sequence of states obtained by a repeated application of the computation rules. The basic Petri net model has been extended in a number of ways that are of interest for biological networks [Koch et al. (2011) provides an excellent summary].

These extensions incorporate timing (timed Petri nets), concentrations (continuous Petri nets), stochasticity (stochastic Petri nets), multiple levels of organisation (hierarchical Petri nets), activation and inhibition relations needed for signalling networks (Petri nets with “read” and “inhibitor” arcs), and so on [see David and Alla (2010) for descriptions of these aspects of Petri nets]. Mathematically, the power of Petri nets ranges from simple qualitative producer-consumer models to that of ordinary differential equations. Computationally, the range is from above regular languages to Turing machines (Peterson 1981).

Petri nets are one form of a more general computational view of systems. In this view, a system is comprised of a program and data. The behaviour of the system is described by how the program computes in stepwise fashion, and the possible state-transformations that result, given the data. Readers familiar with the semantics of programming languages will recognise this as the basis of the Stuctural Operational Semantics (SOS) introduced by Plotkin (1981), although the idea relating a program’s semantics to the operations it can perform can be found nearly twenty years earlier in McCarthy (1963). For our purposes, it is sufficient to know that Petri nets are a kind of transition system, and the work in SOS gives us one way of describing and understanding the behaviour of general transition systems (more on this below). We will use this basic SOS idea as a springboard to address the main point of interest for this paper: the automatic identification of models for biological systems from data. We propose to achieve this by adopting some of the recent developments in Inductive Logic Programming (ILP) on meta-interpretive learning. Specifically:

- We define a general transition system, much as it done in SOS, using: (a) a program; (b) an interpreter for the program; and (c) data in the form of observed state-pairs. In this paper, (a), (b) and (c) are all logic-programs. In meta-interpretive ILP terms, (a) and (b) constitute background knowledge B_D and B_M (denoting domain-knowledge and the meta-interpreter), and (c) are the positive examples E.
- We investigate empirically the identification of some well-known metabolic and signalling networks, using a meta-interpreter B_M, domain knowledge B_D consisting of Petri net transitions, and data E that range from being complete and correct through varying amounts of incompleteness and incorrectness, by extending the meta-interpreter and moving to probabilistic transition systems.

The principal contribution of this paper is to system identification; in particular, we provide a logical formulation for the identification of transition systems from data by the use of an ILP meta-interpreter. The second main contribution is to introduce into meta-interpretive ILP a method for the identification of systems with discrete state spaces using a formalism expressive enough to describe computations, namely Plotkin's SOS. Furthermore, the underlying formalism of transition systems is sufficiently general to capture the semantics of non-deterministic and probabilistic representations, and is modular, enabling a form of problem decomposition. The third contribution is to biology, where our approach enables incremental identification in which learning can scale with the size of the system being studied by the re-use of networks constructed previously as transitions at the next level of abstraction.

The rest of the paper is organised as follows. In Sect. 2 we describe general transition systems in the manner used by SOS, and an implementation of a meta-interpreter for such a transition system using logic programs. This section also shows some simple examples of "system identification" using the meta-interpreter along with specific programs and data. Depending on the program, the system identified changes: we show examples of finite-state automatons, Turing machines, and Petri nets. Section 3 describes the emerging area of meta-interpretive ILP, and a small change to the meta-interpreter. This results in a specialised form of meta-interpretive ILP for transition systems. This section also describes how hypotheses obtained can be specialised to account for noisy data. Experiments without and with noise are in Sect. 4. Section 5 describes how the work here is related to other work both on system-identification in biology and more generally, and work in meta-interpretive ILP. Section 6 concludes the paper.

2 Transition systems

We adopt the principal idea from Plotkin's Structural Operational Semantics (SOS) (Plotkin 1981), in which computations are modelled by a transition system or TS. A TS is defined formally as a structure $(\Gamma, \rightarrow)$ where:

- Γ is a set, denoting *configurations*; and
- $\rightarrow \subseteq \Gamma \times \Gamma$ is a binary relation.

The notation $\gamma \rightarrow \gamma'$ is used to denote pairs $\gamma, \gamma' \in \Gamma$ that are in the binary relation $\rightarrow$. This is to be understood as "there is a transition between γ and γ'." The definition of a TS can be extended to include a set of labels A, giving a labelled transition system, or LTS, defined as a structure $(\Gamma, \rightarrow, A)$, where $(\Gamma, \stackrel{a}{\rightarrow})$ is a transition system for each $a \in A$ (Keller 1976). An LTS allows us to associate additional meaning to the transitions (like names, actions, operations and so on). For the biological applications addressed in this paper, we will assume that all transitions will have labels, denoting known chemical reaction, or a specific action (like phosphorylation), and so on: we may sometimes even use a ground first-order terms for labels. It is useful also to distinguish further subsets I and T of Γ, to denote initial and final configurations.

A computation that changes a system from configuration γ to γ' is described by a sequence of transitions. Mathematically, the pair (γ, γ') is an element of the transitive- (correctly, the reflexive-transitive) closure of $\rightarrow$. Adapting the terminology in Plotkin (1981), we will call a sequence of transitions $\langle a_1, a_2, \ldots, a_n \rangle$ the *trace* for (γ, γ') if $\gamma \in I$, $\gamma' \in T$) and $\gamma \stackrel{a_1}{\longrightarrow} s_1 \stackrel{a_2}{\longrightarrow} s_2 \cdots \stackrel{a_n}{\longrightarrow} s_n = \gamma'$ for $s_1, s_2, \ldots, s_n \in \Gamma$.

In Plotkin (1981), a distinction is made between transitions used to explain a system's behaviour *internally* and those needed to explain it *externally*, based on the observation that not all the transitions needed for a computation may actually produce an observable effect. This leads to a qualititative notion of explaining behaviour using transitions of the "correct granularity". No prescription is made on how to achieve this correct granularity—just that it is inevitable that it would be needed. Much work has been done in the area to describe notions of behavioural equivalence, in the usual mathematical sense (of being reflexive, symmetric and transitive) leading naturally to the formalisation of simulation and bisimulation (Van Glabbeek 2011). In the ILP setting of this paper, we side-step the granularity issue, assuming this is resolved by the domain-knowledge provided. Also, we are not concerned, as yet, with equivalence of transition systems. The system identification task we are concerned with in this setting is as anticipated by Plotkin in the second half of his observation: "Often two or more definitions of behavior (or having the same behavior) are possible for a given transition system. Indeed on occasion one must turn the problem around and look for a transition system which makes it possible to obtain the expected notion of behaviour."

The main concern of this paper differs from that of providing operational semantics to programs. In operational semantics the focus is usually on translating constructs in a programming language (like loops, conditional statements, assignments and the like) to an abstract machine. Abstract machines can be formulated as transition systems (Plotkin shows examples of a variety of different machines that can be seen as TS's and LTS's), and computations by the program are traces from specific initial states. SOS is thus concerned predominantly with the abstract machine that acts as an interpreter for a programming language. However, we are concerned instead with the interpreter for a general transition system that constructs the trace semantics of behaviour.

2.1 An interpreter for transition systems

We will adopt the language of logic programs for transition systems, and develop an interpreter that returns traces between pairs of states (γ, γ'). We will in fact develop a *meta-interpreter*: an interpreter also written in the same language used for transition systems, here logic programs, since this will allow us to extend naturally to the ILP setting.

The reflexive-transitive closure of the binary relation in a transition system can be implemented quite straightforwardly as a Prolog program:[1]

```
ts((S,S)).
ts((Si,Sf)):-
    trans(T,Si,S),
    ts((S,Sf)).
```

We will refer to this program, and definitions related to `trans/3`, as B_D (domain- or background-knowledge). Given a configuration-pair (γ_x, γ_y), we would like the meta-intepreter for B_D to logically entail a trace T (if one exists) such that $B_D \cup T \models (\gamma_x, \gamma_y)$. One way to obtain T is with a meta-interpreter that constructs refutation-proofs.

[1] We use standard Prolog syntax: the ":-" is to be read as "if"; variables are in upper-case, predicates and function symbols (including constants) are in lower case; a term is a variable, constant, or a function symbol applied to terms; a literal is a predicate symbol applied to terms. A clause is of the form `Head :- Body`, where `Head` consists of 0 or 1 literal and `Body` is a conjunction of 0 or more literals. All variables in `Head` are universally quantified, and those that appear only in `Body` are existentially quantified. `not` denotes a special meta-predicate denoting "not provable": `not(P)` is true if `P` is not provable.

$Trace((\gamma_x, \gamma_y), B_D, B_M)$:

Given: A pair of configurations γ_x, γ_y; an object-level program B_D for `ts/1` and any predicates related to the definition of `trans/3`; and B_M, a meta-interpreter for B_D.

Find: A conjunction of ground atoms T s.t. $B_D \cup T \models ts((\gamma_x, \gamma_y))$

1. Let $\neg P$ be a partial refutation-proof for $B_D \cup \{\leftarrow ts((\gamma_x, \gamma_y))\}$ using B_M
2. Let $P = \exists \mathbf{y}(A_1 \wedge A_2 \wedge \cdots \wedge A_k)$ and $T_{xy} = (A'_1 \wedge A'_2 \wedge \cdots \wedge A'_k)$, where the A_i is A_i with the $\mathbf{y}_i$ replaced with appropriate Skolem constants
3. Return T_{xy}

Fig. 1 Computation of a transition system's trace from proofs obtained using a meta-interpreter

Definition 1 (*SLD-refutation proof*) An SLD-refutation proof for a goal G consists of a sequence of triples $\langle (G_1, C_1, \theta_1), (G_2, C_2, \theta_2), \ldots, (\Box, C_n, \theta_n)\rangle$ where $G_1 = G$, G_i is a goal, C_i is a clause and θ_i is a substitution.

Definition 2 (*Partial refutation-proof*) Let S be a refutation proof for a goal G. Let $P = \exists \mathbf{y}(A_1 \wedge A_2 \wedge \cdots \wedge A_k)$ where $(\neg \exists \mathbf{y}_i A_i, C_i, \theta_i) \in S$ $(1 \leq i \leq k)$. Then $\neg P$ is a partial refutation-proof for G (or simply, a partial proof for G).

We will say that the goals in the partial proof are obtained by *marking* goals in the SLD refutation-proof.

Definition 3 (*Meta-interpreter*) Let B_D be a definite-clause logic program and G a definite goal. Let $P = \exists \mathbf{y}(A_1 \wedge A_2 \wedge \cdots \wedge A_k)$ s.t. $B_D \cup P \cup G \models \Box$, where the A_i are atoms and $\mathbf{y}$ are variables in $(A_1 \wedge \cdots \wedge A_k)$. Let B_M be a higher-order logic program s.t. $B_M \cup B_D \models prove(G, \neg P)$ where $prove(G, \neg P)$ is true if $\neg P$ is a partial refutation-proof for G. Then B_M is a meta-interpreter for B_D.

Definition 4 (*Transition system trace*) Given an object-level program B_D for `ts/1` (above) and any predicates related to the definition of `trans/3`, we define a trace between a pair of configurations $\gamma_{x,y}$ from a finite set of configurations Γ as a conjunction of ground atoms T s.t. $B_D \cup T \models ts((\gamma_x, \gamma_y))$

We are now able to formulate a procedure for computing traces using the proofs returned by a meta-interpreter (see Fig. 1).

We note that there may be more than one proof for a goal, and, correspondingly, more than one trace. The computation in Fig. 1 is non-deterministic and does not specify which of these traces is returned. It is straightforward to see that the procedure is correct. From Definitions 2 and 4, $P = \exists \mathbf{y}(A_1 \wedge A_2 \wedge \cdots \wedge A_k)$ and $B_D \wedge P \wedge \{\neg ts((\gamma_x, \gamma_y))\} \models \Box$. That is $B_D \cup P \models ts((\gamma_x, \gamma_y))$. Further, since $A'_i \models A_i$, clearly $B_D \cup \{A'_1, A'_2 \cdots, A'_k\} \models B_D \cup P \models ts((\gamma_x, \gamma_y))$. We call the ground atoms T_{xy} a trace for the pair (γ_x, γ_y). We note that there may be more than one trace for a pair of configurations.

Example 5 (SLD-refutation proof for a transition system) Let B_D be the program for `ts/1`, and additionally contain:

```
trans(t1,s1,s2).
trans(t2,s2,s3).
```

Let T be a trace for $(s1, s3)$, then $B \wedge T \models ts((s1, s3))$. An SLD-refutation proof consists of a sequence of triples of the form (G_i, C_i, θ_i), where G_i is a goal, C_i is a clause and θ_i is a substition. A SLD-refutation-proof for $B_D \cup \{\leftarrow ts((s1, s3))\}$ is as follows:

1. $((\leftarrow ts((s1, s3))), (ts(Si, Sf) \leftarrow trans(T, Si, S), ts(S, Sf)), (\{Si/s1, Sf/s3\}))$
2. $((\leftarrow trans(T, s1, S), ts(S, s3)), (trans(t1, s1, s2) \leftarrow), (\{T/t1, S/s2\}))$
3. $((\leftarrow ts((s2, s3))), (ts(Si, Sf) \leftarrow trans(T, Si, S), ts(S, Sf)), (\{Si/s2, Sf/s3\}))$
4. $((\leftarrow trans(T, s2, S), ts(S, s3)), (trans(t2, s2, s3) \leftarrow), (\{T/t2, S/s3\}))$
5. $((\leftarrow ts((s3, s3))), (ts(S, S) \leftarrow), (\{S/s3\}))$
6. $(\Box, (), \emptyset)$

Example 6 (Transition system trace) A marking of $trans/3$ goals derived in the SLD refutation-proof in Example 5 would mark $\leftarrow trans(t1, s1, s2)$ and $\leftarrow trans(t2, s2, s3)$. Let B_M be a meta-interpreter s.t. $B_M \wedge B_D \models prove(\leftarrow ts((s1, s3)), \neg P)$ where $\neg P = \neg(trans(t1, s1, s2) \wedge trans(t2, s2, s3))$. There are no variables in P, so no further Skolemisation is needed. A trace for $(s1, s3)$ is therefore $T = (trans(t1, s1, s2) \wedge trans(t2, s2, s3))$.

Clearly, to compute traces in the manner shown, we need a meta-interpreter B_M. We use as starting point the simple meta-interpreter for pure Prolog in Sterling and Shapiro (1994):

```
prove(true).
prove((A,B)) :- prove(A), prove(B).
prove(A) :- clause(A,B), prove(B).
```

We enhance it to return the derivation a pair of configurations (if it exists). Further, we would like the meta-interpreter to be independent of the abstract automaton used to provide semantics to the behaviour (different automata will result in changes to the set of configurations Γ and the transition relation $\rightarrow$). This latter requirement is achieved by the use of *meta-rules* to specify a generic pattern to transitions. The enhanced meta-interpreter is shown below (for simplicity, we omit some ancillary definitions, and leave out checks for duplicates in proof):

```
prove(Goal,neg(Proof)):-
        meta_prove([Goal],[],Proof).

meta_prove([],Proof,Proof).
meta_prove([Goal|Goals],P0,P):-
        meta_rule(builtin,Goal), !,
        call(Goal),
        meta_prove(Goals,P0,P).
meta_prove([Goal|Goals],P0,P):-
        meta_rule(Type,(Goal:-Body)),
        goals_to_list(Body,BodyL),
        meta_prove(BodyL,P0,P1),
        update_proof(P1,Goal,P2),
        meta_prove(Goals,P2,P).

meta_rule(builtin,Head):- built_in(Head).
meta_rule(user,(Head:-Body)):- clause(Head,Body).
meta_rule(trans/3,(trans(T,S1,S2):-
       pre(T,S1), succ(T,S1,S2), post(T,S2))).

update_proof(Proof,Goal,[Goal|Proof]):-
       functor(Goal,Name,Arity),
       proof_goal(Name/Arity), !.
update_proof(Proof,_,Proof).
```

proof_goal specifies the goals that are to be marked in the SLD refutation-proof. We augment B_M with a definition for extracting traces from goals marked in proofs:

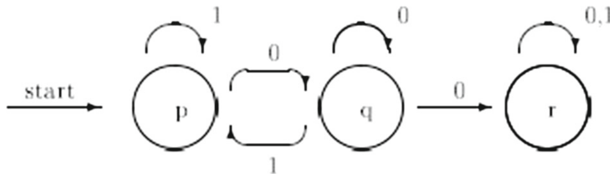
```
proof_to_trace(neg(Lits),Trace):-
        skolemise(Lits,Trace).
```

(Some minor changes will be needed if there are meta-rules with names other than just trans/3). The computation of a trace in Fig. 1) is now straightforward:

```
trace(ConfigPair,Trace):-
       prove(ts(ConfigPair),Proof), proof_to_trace(Proof,Trace).
```

The meta-rule for trans/3 encodes the template for all transitions consisting of a precondition, a successor relation, and a post-condition. We show examples of several kinds of automata that can be encoded as special cases.

Example 7 (Finite State Machine) The following example is from Plotkin (1981). We have the following abstract machine:



The machine is a special case of a transition system, with the following definitions for configurations and transitions. Each configuration is a 2-tuple (State,InputString), where State is one of p, q, r and InputString is a binary string. Transitions are defined by the following pre, post and succ definitions:

```
pre(pt1,(p,_)).
post(pt1,(p,_)).
post(pt1,(q,_)).

pre(pt2,(q,_)).
post(pt2,(p,_)).
post(pt2,(r,_)).

pre(pt3,(r,_)).
post(pt3,(r,_)).

succ(pt1,(p,[0|L]),(q,L)).
succ(pt1,(p,[1|L]),(p,L)).

succ(pt2,(q,[0|L]),(q,L)).
succ(pt2,(q,[0|L]),(r,L)).
succ(pt2,(q,[1|L]),(p,L)).

succ(pt3,(r,[_|L]),(r,L)).
```

Then the query:

```
trace((p,[0,1,0,0,1]),(r,[])),Trace)
```

returns the following answer-substitution:

```
Trace = [
    trans(pt3,(r,[1]),(r,[])),
    trans(pt2,(q,[0,1]),(r,[1])),
    trans(pt1,(p,[0,0,1]),(q,[0,1])),
    trans(pt2,(q,[1,0,0,1]),(p,[0,0,1])),
    trans(pt1,(p,[0,1,0,0,1]),(q,[1,0,0,1]))
```

which is the behaviour of the FSM in Plotkin (1981).

Example 8 (Turing Machine) A pair of numbers M and N can be represented by a sequence of M 1's, a 0, and a sequence of N 1's. The Adding Machine is to scan such a string, and print a sequence of M+N 1's and halt. A simple way to accomplish this is shift the string of M 1's one step right. Once finished a single string of M+N 1's will result. As an example, consider adding 1 and 1. Initially the tape has ...01010.... After shifting the 1 the tape halts with ...00110....

A 3-state machine is sufficient for this adder. The instruction table for the machine is:

Input	State		
	ts1	ts2	ts3
0	$[0, ts1, r]$	$[1, ts3, r]$	*halt*
1	$[0, ts2, r]$	$[1, ts2, r]$	*halt*

Each instruction-triple is $[WriteSymbol, NextState, MoveTape]$. The Turing Machine is a special case of a transition system in which each configuration is a 3-tuple denoting `(State,Tape,Head)` where `Head` is the location of the reading head. It is more efficient to represent configurations as the 2-tuple `(State,TapeComponents)` where `TapeComponents` is a list consisting of 3 parts: `LeftOfHead, Head, RightOfHead`.

```
pre(tm1,(ts1,Tape)):- current_symbol(Tape,0).
post(tm1,(ts1,Tape)).

pre(tm2,(ts1,Tape)):- current_symbol(Tape,1).
post(tm2,(ts2,Tape)).

pre(tm3,(ts2,Tape)):- current_symbol(Tape,0).
post(tm3,(ts3,Tape)).

pre(tm4,(ts2,Tape)):- current_symbol(Tape,1).
post(tm4,(ts2,Tape)).

pre(tm5,(ts3,Tape)).
post(tm5,(halt,Tape)).

pre(tm6,(halt,Tape)).
post(tm6,(halt,Tape)).
```

```
succ(tm1,(ts1,CurrentTape),(ts1,NewTape)):-
        current_symbol(CurrentTape,0),
        write_symbol(0,CurrentTape,Tape),
        move(r,Tape,NewTape).
succ(tm2,(ts1,CurrentTape),(ts2,NewTape)):-
        current_symbol(CurrentTape,1),
        write_symbol(0,CurrentTape,Tape),
        move(r,Tape,NewTape).
succ(tm3,(ts2,CurrentTape),(ts3,NewTape)):-
        current_symbol(CurrentTape,0),
        write_symbol(1,CurrentTape,Tape),
        move(r,Tape,NewTape).
succ(tm4,(ts2,CurrentTape),(ts2,NewTape)):-
        current_symbol(CurrentTape,1),
        write_symbol(1,CurrentTape,Tape),
        move(r,Tape,NewTape).
succ(tm5,(ts3,Tape),(halt,Tape)).
succ(tm6,(halt,T),(halt,T)).

current_symbol([Left,[S],Right],S).

write_symbol(S,[Left,[_],Right],[Left,[S],Right]).

move(l,[Left,[C],Right],[L1,[C1],R1]):-
        app(L1,[C1],Left),
        R1 = [C|Right].
move(r,[Left,[C],Right],[L1,[C1],R1]):-
        app(Left,[C],L1),
        Right = [C1|R1].
```

Then the query:

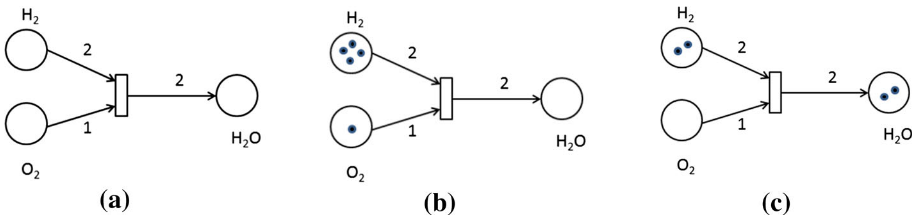
```
trace(((ts1,[[],[0],[1,0,1]]),(halt,_T)),Trace)
```

returns the answer-substitution:

```
Trace = [
    trans(tm5,(ts3,[[0,0,1],[1],[]]),(halt,[[0,0,1],[1],[]])),
    trans(tm3,(ts2,[[0,0],[0],[1]]),(ts3,[[0,0,1],[1],[]])),
    trans(tm2,(ts1,[[0],[1],[0,1]]),(ts2,[[0,0],[0],[1]])),
    trans(tm1,(ts1,[[],[0],[1,0,1]]),(ts1,[[0],[1],[0,1]]))
       ]
```

which describes the behaviour of the adding machine.

Example 9 (Petri Net) A simple Petri net representing the reaction $2H_2 + O_2 \rightarrow 2H_2O$ is shown in (a) below. In (b) an “initial marking”, in which molecules of hydrogen and oxygen are shown by tokens (small solid circles); and in (c), a “final marking”, which results in two molecules of water, from the molecules of hydrogen and oxygen in (b).



The Petri net is a special case of a transition system in which configurations are tuples with the tokens with each kind of molecule ("place-vectors"), and the transition relation is defined by:

```
pre(water,(Hydrogen-h2,Oxygen-o2,Water-h2o)):-
      Hydrogen >= 2, Oxygen >= 1.
post(water,(Hydrogen-h2,Oxygen-o2,Water-h2o)):-
      Water >= 2.

succ(water,Pre,Post):-
      Pre  = (H1-h2,O1-o2,W1-h2o),
      Post = (H2-h2,O2-o2,W2-h2o),
      H2 is H1-2, O2 is O1-1, W2 is W1 + 2.
```

Then the query:

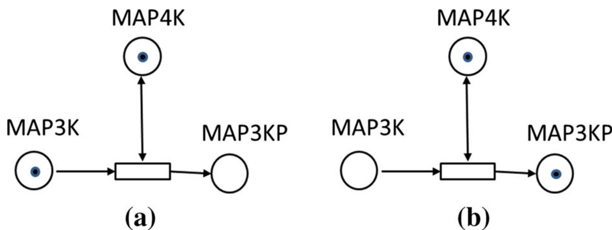
```
trace(((4-h2,2-o2,0-h2o),(2-h2,1-o2,2-h2o)),Trace)
```

returns the answer-substitution:

```
Trace = [
    trans(water,(4-h2,2-o2,0-h2o),(2-h2,1-o2,2-h2o))
        ]
```

which describes the behaviour of the equation $2H_2 + O_2 \rightarrow 2H_2O$.

It is straightforward also to represent extended Petri nets. The Petri net representing the first stage of a MAPK cascade is shown below. This is an extended Petri net with a "read arc" (David and Alla 2010) from MAP4K (tokens in MAP4K need to be present for the reaction to proceed, but do not get used up):



The transition relation for this kind of system can be represented as follows:

```
pre(read(map4k),(MAP4K-map4k,MAP3K-map3k,MAP3KP-map3kp)):-
      NAP4K >= 1, MAP3K >= 1.
post(read(map4k),(MAP4K-map4k,MAP3K-map3k,MAP3KP-map3kp)):-
```

```
    MAP4K >= 1, MAP3KP >= 1.

succ(read(map4k),Pre,Post):-
      Pre  = (M4K1-map4k,M3K1-map3k,M3KP1-map3kp),
      Post = (M4K2-map4k,M3K2-map3k,M3KP2-map3kp),
      M4K2 = M4K1, M3K2 is M3K1 - 1, M3KP2 is M3KP1 + 1.
```

Then the query:

```
trace(((1-map4k,1-map3k,0-map2kp),(1-map4k,0-map3k,1-map3kp)),
      Trace)
```

returns the answer-substitution:

```
Trace = [
    trans(read(map4k),
          (1-map4k,1-map3k,0-map2kp),(1-map4k,0-map3k,1-map3kp))
        ]
```

The representation in fact allows more flexibility than shown above. The transition can be written more generally (we omit some utility definitions):

```
pre(phosphorylates(A,B),Config):-
      place(A), place(B), can_phosphorylate(A,B),
      value(A,Config,ValA), value(B,Config,ValB),
      ValA >= 1, ValB >= 1.
post(phosphorylates(A,B),Config):-
      place(A), place(B), phosphorylate(B,BP), place(BP),
      value(A,Config,ValA), value(BP,Config,ValBP),
      ValA >= 1, ValBP >= 1.

succ(phosphorylates(A,B),Config1,Config2):-
      place(A), place(B), phosphorylate(B,BP), place(BP),
      value(A,Config1,ValA1), value(B,Config1,ValB1),
            value(BP,Config1,ValBP1),
      value(A,Config2,ValA2), value(B,Config2,ValB2),
            value(BP,Config2,ValBP2),
      ValA2 = ValA1, ValB2 is ValB1 - 1, ValBP2 is ValBP1 + 1.

place(map4k). place(map3k). place(map3kp).

can_phosphorylate(map4k,map3k)

phosphorylate(map3k,map3kp).
```

The answer-substitution returned for the same query as before would now be:

```
Trace = [
    trans(phosphorylates(map4k,map3k),
          (1-map4k,1-map3k,0-map2kp),(1-map4k,0-map3k,1-map3kp))
        ]
```

Here an advantage of the declarative representation used in ILP is evident, namely the ability to incorporate higher-level domain knowledge (such as one molecule being able to phosphorylate another) into the definition of transitions.

We note that traces should ideally contain sufficient information to reconstruct the automaton being interpreted. This is useful for reasons of understandability; it is evident from the examples shown that traces are not as easy to understand as their pictorial counterparts. It may correspondingly more helpful, when performing system identification, to extract diagrammatic representations from traces.

Remark 10 (Petri nets from traces) Suppose the trace contains an element `trans(t,s1,s2)` where `t` is a transition label, `s1`, `s2` are configurations. We will assume that configurations are place-vectors. For example, in the case of the "water" reaction `trans(water,(4-h2, 2-o2,0-h2o),(2-h2,1-o2,2-h2o))`, we have `t = water`, `s1 = (4-h2, 2-o2,0-h2o)` and `s2 = (2-h2,1-o2,2-h2o)`. Using the terminology in Durzinsky et al. (2011c), let us define a reaction vector `r` as the place-wise difference between `s2` and `s1`. That is, `r = s2 - s1`. For the example just shown, `r = (-2-h2,-1-o,+2-h2o)`.

It is evident that for every `trans(t,s1,s2)` in a trace, there is a unique `(t,r)` tuple. With some abuse of notation, let $Input(r)) = \{p_i : x_i - p_i \in r \text{ s.t. } x_i < 0\}$ and $Output(r) = \{p_i : x_i - p_i \in r \text{ s.t. } x_i > 0\}$. It is straightforward to obtain a Petri net diagram for transition `t` using $Input(r)$ and $Output(r)$. (This will not work for extended Petri nets, but a method for obtaining diagrams can be devised provided that the transition label contains sufficient information.)

3 Meta-interpretive system identification

The transition-system semantics described so far allows reachability between a pair of system states to be treated as a logical consequence of the domain-knowledge B_D. However there is a difficulty: in practice, B_D can be incomplete, or incorrect, or both. In such cases reachability as logical consequence may fail. Some additional issues need clarification before we tackle these practical problems.

Definition 11 (*Deterministic transition systems*) Let Γ be the set of possible system configurations and A the set of transition labels. A transition system $(\Gamma, \rightarrow, A)$ is deterministic if, for every configuration γ_x, whenever $\gamma_x \xrightarrow{a} \gamma_y$ and $\gamma_x \xrightarrow{a} \gamma_z$, $\gamma_y = \gamma_z$. That is, $\xrightarrow{a}$ is a partial function, where for each configuration γ there is at most one configuration γ' such that $\gamma \xrightarrow{a} \gamma'$ (Keller 1976).

Definition 12 (*Non-deterministic transition systems*) A transition system $(\Gamma, \rightarrow, A)$ is non-deterministic if for some configuration $\gamma_x \in \Gamma$, there exist transitions $\gamma_x \xrightarrow{a} \gamma_y$ and $\gamma_x \xrightarrow{a} \gamma_z$ and $\gamma_y \neq \gamma_z$. That is, $\xrightarrow{a}$ is a relation $\Gamma \times A \times \Gamma$, for $\gamma_x, \gamma_y, \gamma_z \in \Gamma$ and $a \in A$.

Definition 13 (*Probabilistic transition system*) A probabilistic transition system is defined as the 4-tuple $(\Gamma, \rightarrow, A, \pi)$, where π is the conditional probability $Pr(\gamma', a|\gamma)$ for $\gamma, \gamma' \in \Gamma$ and $a \in A$.

It is evident that for deterministic systems, `Proof` (and the corresponding trace) is unique. Otherwise, there can be more than one proof for a pair of configurations. Also, in a probabilistic transition system, a (conditional) probability distribution is defined over the transition

relation. Informally, given a pair of transitions $p_1 : \gamma \xrightarrow{a} \gamma_1$ and $p_2 : \gamma \xrightarrow{a} \gamma_2$, a choice of transition is based on p_1 and p_2.

It is also unlikely in practice that system-behaviour would consist of a transition between a single pair of configurations. Instead, it is more likely to be specified by a sequence of configurations. It is useful therefore to define the following additional functions.

Definition 14 (*Sequences to conjunctions*) Given a sequence of configurations $s = \langle \gamma_1, \gamma_2, \gamma_3, \ldots, \gamma_j \rangle$, we define $Conj(s) = (\gamma_1, \gamma_2) \wedge (\gamma_2, \gamma_3) \wedge \cdots \wedge (\gamma_{j-1}, \gamma_j)$.

Definition 15 (*Conjunctions to sequences*) For a conjunction $c = (\gamma_1, \gamma_2) \wedge (\gamma_2, \gamma_3) \wedge \cdots \wedge (\gamma_{j-1}, \gamma_j)$ we define $Seq(c) = \langle \gamma_1, \gamma_2, \gamma_3, \ldots, \gamma_j \rangle$.

We note that *Seq* is only partially-defined: not all conjunctions of configuration-pairs represent sequences. For example, the conjunction $(\gamma_1, \gamma_2) \wedge (\gamma_3, \gamma_4)$ does not represent a sequence.

With these definitions in place, we turn to addressing practical difficulties of incompleteness and incorrectness. These involve extensions to both the meta-interpreter B_M and the object-level program B_D.

3.1 Extending the meta-interpreter

We now extend the meta-interpreter introduced in Sect. 2.1 to allow traces to be constructed for data *not* entailed by $B_M \cup B_D$. The principal mechanism that allows this is drawn from the field of meta-interpretive ILP, which allows a meta-interpreter to abduce atoms to allow a SLD refution-proof to proceed.[2] This is not dissimilar to the extension proposed to SLD-resolution, to allow "skipping" of failing goals in a proof (Yamamoto 1997). The resulting inference procedure (SOLD-resolution), identifies atoms to be abduced.

```
prove(Goal,Proof):-
        meta_prove([Goal],[],Proof).

meta_prove([],Proof,Proof).
meta_prove([Goal|Goals],P0,P):-
        meta_rule(builtin,Goal), !,
        call(Goal),
        meta_prove(Goals,P0,P).
meta_prove([Goal|Goals],P0,P):-
        meta_rule(Type,(Goal:-Body)),
        goals_to_list(Body,BodyL),
        meta_prove(BodyL,P0,P1),
        update_proof(P1,Goal,P2),
        meta_prove(Goals,P2,P).
meta_prove([Goal|Goals],P0,P):-
        abduce(Goal),
        update_proof(P0,Goal,P1),
```

[2] In principle, our use of the term abduction is consistent with that of Peirce (see, e.g., Psillos (2011)), who introduced the term. It is more useful for our purpose to follow (Kakas et al. 1992), who formulate a computational form of Peirce's philosophical description, specifically for logic programs. In this formulation, atoms are hypothesised from a class of *abducibles*, which may include existentially quantified variables. These *abduced* atoms (Δ) allow derivation of of a sentence G from a theory T (that is, $T \cup \Delta \models G$, or $T \cup \Delta \vdash G$).

```
        meta_prove(Goals,P1,P).

meta_rule(builtin,Head):- built_in(Head).
meta_rule(user,(Head:-Body)):- clause(Head,Body).
meta_rule(trans/3,(trans(T,S1,S2):- pre(T,S1),
          succ(T,S1,S2), post(T,S2))).

update_proof(Proof,Lit,[neg(Lit)|Proof]):-
       functor(Lit,Name,Arity),
       proof_goal(Name/Arity), !.
update_proof(Proof,_,Proof).

abduce(Lit):-
    functor(Lit,Name,Arity),
    abducible(Name/Arity).
```

It is evident that as long as `trans/3` is a known abducible, the meta-interpreter will always abduce a `trans/3` literal, thus allowing refutations to proceed, even if there are no transitions between a pair of configutations.

Example 16 Let B_D be the program for `ts/1`, and additionally contain:

```
trans(t1,s1,s2).
```

Let T be a trace for $(s1, s3)$, then $B_D \cup T \models ts((s1, s3))$. An SLD-refutation proof for $B_D \cup \{\leftarrow ts((s1, s3))\}$ will fail, since there is no known transition between $s2$ and $s3$. The extended meta-interpreter ${B_M}^+$ however constructs an SLD-refutation proof for $B_D \cup \{\leftarrow ts((s1, s3))\}$ as follows:

1. $((\leftarrow ts((s1, s3))), (ts(Si, Sf) \leftarrow trans(T, Si, S), ts(S, Sf)), (\{Si/s1, Sf/s3\}))$
2. $((\leftarrow trans(T, s1, S), ts(S, s3)), (trans(t1, s1, s2) \leftarrow), (\{T/t1, S/s2\}))$
3. $((\leftarrow ts((s2, s3))), (ts(Si, Sf) \leftarrow trans(T, Si, S), ts(S, Sf)), (\{Si/s2, Sf/s3\}))$
4. $((\leftarrow trans(T, s2, S), ts(S, s3)), (trans(T', s2, S') \leftarrow), (\{T/T', S/S'\}))$
5. $((\leftarrow ts((s3, s3))), (ts(S', S) \leftarrow), (\{S'/s3\}))$
6. $(\square, (), \emptyset)$

The abduction step by the interpreter is in Step 4. Now $B_M \wedge B_D \models prove(\leftarrow ts((s1, s3)), \neg P)$ where $\neg P = \neg\exists T'(trans(t1, s1, s2) \wedge trans(T', s2, s3))$. A trace for $(s1, s3)$ is obtained by replacing the existentially quantified variable with a Skolem constant, giving the trace $(trans(t1, s1, s2) \wedge trans(sk_1, s2, s3))$, where sk_1 is a Skolem constant.

The principal difficulty with abducing goals to compute proofs is this: how do we distinguish a genuine abduction (that is, one constructed to correct an incompleteness in B_D) from a spurious one (that is, one constructed simply out of convenience to complete a proof)? The problem is a long-standing one in the field of abductive reasoning, and Kakas et al. (1992) describes a number of remedies, ranging from simple syntactic measures (minimal abductions, for example) to more elaborate ones based on the use of integrity constraints that would rule out spurious additions to the store of abduced explanations. The corresponding concern in system identification is one of structure estimation. The meta-interpreter equipped with abduction performs a simple form of structure enumeration: each trace for a conjunction of configuration-pairs contains sufficient information to reconstruct an automaton capable of deriving all configuration-pairs, given input data (see Remark 10). Some portions of this

structure may be spurious, and the structure estimation problem is concerned with detecting these.

Definition 17 (*Spurious atoms*) Given system behaviour expressed as a conjunction of configuration-pairs c, let T be a trace obtained with a meta-interpreter B_M and domain-knowledge B_D. Then spurious atoms refers to some set of abduced atoms $A \subseteq T$ to be discarded from T.

It is evident that discarding a set of elements A from a trace T may result in some configuration-pairs in c no longer being derivable from $B_D \wedge (T - A)$. This requires a modification to our previous requirement that a system should derive the entire conjunction c (Fig. 4). Instead we relax this to entailing some sub-conjunction c' of c. The structure-estimation task is to find a plausible set $(T - A)$ given c.

How are we to detect (and avoid) spurious atoms? We propose incorporating mechanisms at the object- and the meta-level. The former results in changes to the transition system, and the latter to the meta-interpreter. The second change is smaller, and we present that first.

3.1.1 Bounded abduction

The simplest "fix" to the problem of spurious abduction is introduce a bound on the number of abductive steps allowed by the meta-interpreter. This results in the following change (we only show the change to the abductive step: the other definitions are unchanged, except for the obvious inclusion of a bound, which does not change on iterations):

```
meta_prove([Goal|Goals],Bound,P0,P):-
        Bound > 0,
        abduce(Goal),
        update_proof(P0,Goal,P1),
        Bound1 is Bound - 1,
        meta_prove(Goals,Bound1,P1,P).
```

(Clearly, `prove` now will need an additional argument: `prove(Goal,Proof)` will become `prove(Goal,Bound,Proof)`.)

This mechanism is crude, but does put a limit on the number of abductive steps in a proof. In turn, the meta-interpreter is forced to find proofs that contain only a bounded number of spurious atoms. From now on, we will call the meta-interpreter capable of performing bounded abduction as an "extended meta-interpreter".

3.1.2 Stochastic theorem proving

Anticipating the move to a probabilistic transition system, we also introduce the built-in machinery needed for stochastic selection of a goal, based on a probability distribution. We will take the distribution to be specified by weights on (for us, ground) instances of `Goal` (with a uniform distribution taken as the default distribution):

```
sample(Goal):-
        findall(Wt-Goal,(prove(Goal,Proof),wt(Goal,Wt)),WGoals),
        WGoals \= [],
        normalise(WGoals,PGoals),
        select(PGoals,Goal).
```

Here `select/2` randomly selects `Goal` using a distribution over `PGoals`. The definition for `sample/1` requires the meta-interpreter (to prove `Goal`: correctly, the call should now be with an abduction-bound of 0).

We anticipate that a probabilistic transition system will require the definition of a meta-rule for a stochastic transition predicate `strans/3`:

```
meta_rule(strans/3,(strans(T,S1,S2):-
       sample(trans(T,S1,S2))).
```

3.2 Object-level modifications

We extend the transition system in two different ways. First, we restrict ourselves to depth-bounded transition systems (not to be confused with the abduction-bound at the meta-level). This means that computational explanations of system-behaviour are now sequences of transitions of bounded length.

3.2.1 Depth-bounded transition system

The system thus cannot seek long explanations simply by inventing spurious atoms.[3] The definition of the transition system now becomes:

```
ts((Si,Sf),D):-
        D = 1,
        trans(T,Si,Sf).
ts((Si,Sf),D):-
        D > 1,
        trans(T,Si,S),
        D1 is D - 1,
        ts((S,Sf),D1).
```

3.2.2 Probabilistic transitions

The second modification is to move to a probabilistic transition system. The motivation here is that spurious induction is likely to result in transitions with low probability. Using the stochastic transition predicate just introduced, the change is minor:

```
ts((Si,Sf),D):-
    D = 1,
    strans(T,Si,Sf).
ts((Si,Sf),D):-
    D > 1,
    strans(T,Si,S),
    D1 is D - 1,
    ts((S,Sf),D1).
```

From now on, we will refer to the depth-bounded probabilistic transition system as the "extended object-level program". For reasons that will become apparent soon, traces will now be conjunctions of `strans/3` facts.[4]

[3] In fact, this restriction applies to more than just the problem of spurious atoms. The bound prevents unbounded explanations, even without any abductive steps.

[4] This does not require any further changes to the meta-interpreter: all that is needed is the procedure in Fig. 1, using `strans/3` goals marked in SLD-refutation proofs, rather than `trans/3` goals.

$ConjTrace(c, B_D, B_M)$:

Given: A conjunction $c = s_1 \wedge s_2 \wedge \cdots \wedge s_{j-1}$ $(j > 1)$ where each $s_i = (\gamma_i, \gamma_{i+1})$ $(\gamma_{i,j} \in \Gamma)$ an extended object-level program B_D for `ts/1` and any predicates related to the definition of `trans/3`; and an extended meta-interpreter B_M for B_D

Find: A conjunction of ground atoms T s.t. $B_D \cup T \models \bigwedge_{i=1}^{j-1} ts(s_i)$

1. For each s_i in c:
   (a) $\mathcal{T}_i = \{T_i : T_i = Trace(s_i, B_D, B_M)\}$
2. Let $\mathcal{T} = \mathcal{T}_1 \times \mathcal{T}_2 \times \cdots \times \mathcal{T}_{j-1}$
3. Let $(T_1, T_2, \ldots, T_{j-1}) \in \mathcal{T}$ and $T = T_1 \cup T_2 \cdots T_{j-1}$
4. Return T

Fig. 2 Non-deterministic computation of traces for a conjunction of configuration pairs. Here $Trace$ is the procedure in Fig. 1

Example 18 (Probabilistic system trace) The probabilistic system trace for the behaviour in Example 16 is $T = (strans(t1, s1, s2) \wedge strans(sk_1, s2, s3))$ Using a set representation, $T = \{strans(t1, s1, s2), strans(sk_1, s2, s3)\}$. With B_D containing definitions as usual for `ts/1`, $B_D \cup T \models ts((s1, s3))$.

Assuming that the B_M and B_D provided refer to the extended versions (with pre-set abduction- and depth-bounds), it is straightforward to obtain probabilistic traces of conjuctions of configuration-pairs (see Fig. 2).

Each element of $\mathcal{T}$ in Fig. 2 is a tuple $(T_1, T_2, \ldots, T_{j-1})$, where T_i is a trace for each $s_i \in Conj(s)$. It follows by construction that $T = \bigcup_i T_i$ is a trace for $Conj(s)$. We note that for deterministic systems, each $\mathcal{T}_i$ is a singleton set, and $\mathcal{T}$ contains a single tuple. In general though, $\mathcal{T}$ can contain several tuples, each of which gives a trace for the sequence s. We are now able to return to the function $ConjTrace$ in Fig. 2. Assuming that the B_M and B_D provided refer to the extended versions (with pre-set abduction- and depth-bounds), it is straightforward to implement $ConjTrace$ along the lines of:

```
conjtrace([],[]).
conjtrace([ConfigPair|Pairs],Trace):-
    trace(ConfigPair,T0),
    conjtrace(Pairs,T),
    conjoin(T0,T,Trace).
```

(The non-deterministic aspect of $ConjTrace$ is naturally accounted for by the backtracking mechanism of Prolog.) Traces of probabilistic system behaviour give us one way of estimating parameters (probabilities over transitions) of transition systems, which we look at next.

3.3 System identification as meta-interpretive learning

Given a configuration γ, a probabilistic transition system selects a transition a and configuration γ' by sampling from a distribution π. In this paper, we will estimate conditional probabilities for a transition using as data traces of the probabilistic system for a conjunction of configuration-pairs (Fig. 3).

The procedure in Fig. 3 makes some simplifying assumptions. First, transitions in the procedure do not contain labels. This is easily rectified by extending the matrix P to include labelled transitions. Secondly, each trace returned in $\mathcal{T}$ actually results in a separate estimate

$ProbEst(c, B_D, B_M)$:

Given: A conjunction of configuration-pairs c; an extended object-level program B_D for `ts/1` and any predicates related to the definition of `trans/3`; and an extended meta-interpreter B_M for B_D

Find: A transition probability matrix

1. Let $\mathcal{T} = \{T : ConjTrace(c, B_D, B_M)\}$
2. For all transitions (γ, γ') in c
   (a) Let $T = \bigcup_{T_i \in \mathcal{T}} T_i$
   (b) let $T_{\gamma,\gamma'} = \{s : s = sample(trans(a, \gamma, \gamma')$ where $a \in A$ and $s \in T\}$
   (c) Let $T_\gamma = \{s : s = sample(trans(a, \gamma, x)$ where $a \in A, x \in \Gamma$ and $s \in T\}$
   (d) $P_{\gamma,\gamma'} = |T_{\gamma,\gamma'}|/|T_\gamma|$
3. Return P

Fig. 3 Estimating conditional probabilities of transitions from conjunctions of configuration-pairs. Here $ConjTrace$ is the procedure in Fig. 2. That procedure is non-deterministic, and $\mathcal{T}$ is all answers computed by the procedure for a sequence s. We assume a finite set of configurations Γ and a finite set of labels A. $P_{\gamma,\gamma'}$ denotes the entry $Pr(\gamma'|\gamma)$ in the transition probability matrix

$SysId(S, Start, Finish, B_D, B_M)$:

Given: A set of sequences of configurations $S = \{s_1, s_2, \ldots, s_n\}$ where $s_i = \langle \gamma_{i,1}, \gamma_{i,2}, \ldots \rangle$; a start configuration $Start$; a terminal configuration $Finish$; an object-level program B_D for `ts/1` and any predicates related to the definition of `trans/3`; and a meta-interpreter B_M for B_D

Find: The highest probability sequence of transitions from $Start$ to $Finish$

1. Let $c = \bigwedge_{i=1}^{n} Conj(s_i)$
2. Let $P = ProbEst(c, B_D, B_M)$
3. Let $s = Viterbi(P, Start, Finish)$
4. Return s

Fig. 4 A procedure for system identification using multiple sequences of configurations. $ProbEst$ is the function in Fig. 3 that estimates the entries of of the transition probability matrix (that is, $P(\gamma'|\gamma)$ for all transitions $\gamma \rightarrow \gamma'$ that occur in c). *Viterbi* returns the sequence of configurations from $Start$ to $Finish$ with the highest probability

for $Pr(\gamma'|\gamma)$, and correctly, the probability $P_{\gamma,\gamma'} \in [L, U]$ where L and U are the minimum and maximum values of the probability estimates. In Fig. 3, a point-estimate for this probability is obtained by simply taking the union of all the traces in $\mathcal{T}$. This effectively treats all ground atoms in all traces as being independent of each other.

The point-estimate of the transition probability allow us to define a form of system identification as the the highest-probability transition-sequence between any pair of configurations (that is, the "system" is the transition sequence identified by the Viterbi algorithm: see Fig. 4).

We now return to Plotkin's observation of having to look for a transition system that makes it possible to obtain an expected behaviour. The procedure proposed in Fig. 4 provides one way to address Plotkin's requirement.

4 Application: transition system identification of biological networks

Networks are ubiquitous in Biology. They are used to represent biological relationships ranging across all levels of organisation: for example, relationships between organisms, and

between an organism and its environment; the flow of energy and matter in an ecosystem; the pathway of carbon atoms through an ecosystem from producers of organic compounds to consumers that release carbon by respiration; the nitrogen cycle that links the environment to proteins and compounds that form the bodies of living things; the stimulus-response mechanisms in constituting nervous pathways; the regulation and control of endocrine glands; the events related to the division and replication of cells; and intra- and inter-cellular interactions between chemicals.

Computationally, substantial research effort has been, and continues to be invested in developing models of biological networks (Junker and Schreiber 2008). Much of this research has been directed at representation and reasoning, with a focus on models that not only determine the underlying relationships amongst entities, but are also capable of simulating the dynamics of the system. The basic Petri net (PN) structure and its extensions have found widespread use in this regard: Wagler (2011) provides an excellent summary of their use in representing metabolic, signalling and genetic regulatory networks. Most of this work has been concerned with hand-crafted Petri net models, with Durzinsky et al. (2011c) being a notable exception that has looked at automatic identification of Petri nets from data.

In this section we assess the utility of meta-interpretive system identification for biological systems. We will focus on identifying Petri net models, although, as will be seen below, the logical setting allows us to provide extensive amounts of domain-knowledge (if this is available).

4.1 Aims

We would like to assess the use of the procedure described in Fig. 4 as a tool for identifying transition systems in biology. The principal question we aim to address is this: *Can meta-interpretive system identification correctly identify a model for a biological system, given observational data?*. We seek to address this in the form of four empirical studies based on data quality, where the observational data can be either *Noisy* or *Missing* with respect to the target networks, as follows:

		Noisy ?	
		No	Yes
Missing ?	No	Study 1	Study 2
	Yes	Study 3	Study 4

There is, of course, the question of data quantity as well. We defer this for the present, but return to it in Sect. 4.5.

4.2 Materials

All experiments use the following target networks (Petri net models of these networks are in “Appendix A)”:

Network	Type	Biological feature(s)
Glycolysis	Metabolic	Long chain of reactions, in which a substrate is modified step-by-step into a product
MAPK cascade	Signalling	(a) Chain of phosphorylation reactions, each triggered by the presence of a protein earlier in the sequence; (b) sequential signalling results in a cascade
Yeast pheromone	Signalling/gene regulation	(a) Signalling cycle involving transmembrane receptors activating a MAPK cascade; (b) cycle shut down using a negative feedback loop; and (c) concurrent reactions

These networks impose several requirements on a program for identification of models from data:

Network	PN type	Modelling requirements
Glycolysis	Pure PNs	Models with lots of transitions and places (10 or more of each)
MAPK cascade	Extended PNs	(a) Models with activators for signalling ("read" arcs); (b) models with specific ordering of signals
Yeast pheromone	Extended, hierarchical	(a) Re-use of multiple models; (b) models with feedback and (c) deterministic behaviour from models with concurrent transitions

Each network will provide a different problem for our system identification approach. More details of these networks as problems are now described.

4.2.1 Problems

Glycolysis The glycolysis pathway was the first metabolic pathway to be discovered. It is a classic case of a series of metabolic reactions in which products of one reaction form the substrates (reactants) for the next reaction. The glycolysis pathway is comprised of 10 such reactions. The reactions breakdown (metabolize) each molecule of glucose into two molecules of pyruvate. The sequence proceeds in three stages: primary (3 reactions), splitting (2 reactions) and phosphorylation (5 reactions). Altogether, 15 metabolites are involved. The pathway is one of the central metabolic pathways in living organisms: it provides an essential part of the energy required for the functioning of a cell, and is used in several metabolic processes.

MAPK The MAPK pathway is a protein-based sequence of events that translate a signal at the cell-surface to the nucleus. The pathway commences when a protein or a hormone binds to a receptor protein that is usually bound to the cell-membrane. This triggers a sequence of events that stops with the DNA expressing one or more genes that alter cell function. At any one step of the cascade, phosphor groups are attached to proteins. This phosphorylated form of the protein then forms a "switch" for commencing the next step. A total of 5 reactions involve 9 molecules. MAPK is a central signalling pathway that is used in all cell-tissues to communicate extra-cellular events to the cell nucleus. It is used to regulate a variety of responses, like hormone action, cell-cycle progression and cell-differentiation. It is also of immense clinical value, since a defect in the pathway often leads to uncontrolled

growth. Proteins in the pathway are thus natural targets for anti-cancer drugs.

Yeast pheromone Receptors at the cell-surface receive an extra-cellular signal and transmit this signal, by use of a series of proteins and enzymes to the nucleus, in turn triggering the expression of a gene or a set of genes that cause the cell to respond to the external signal. The intra-cellular mechanisms by which yeast (*S. cerevisiæ*) responds to an external mating signal (a pheromone) is one of the best understood signal transmission mechanisms in eukaryotes. Using a combination of genetics, biochemistry, and theoretical biology, the following are now known either completely, or substantially: the proteins, and enzymes involved; the order in which events take place; the protein-protein interactions, enzyme-catalyzed reactions and feedback links; and the rates at which many of the reactions occur. The resulting network is complex with 19 molecules in 6 reactions, but it re-uses several components found in signalling pathways of all eukaryotes. Specifically, many of the proteins found in the pathway have homologs in humans, and the G protein cycle and MAPK pathways are conserved in both yeast and humans.

4.3 Domain knowledge

Domain knowledge can be usefully thought of in two parts: the first specifying transitions for the meta-interpreter; and the second specifying constraints relevant to the problems described above. The meta-interpreter requires definitions for the following relations: $Pre(t, c)$, which is true if transition t satisfies pre-conditions in configuration c; $Post(t, c)$, which is true if t satisfies the post-conditions in configuration c for a transition t to be applicable in configuration c; and $Succ(t, c1, c2)$, that specifies the successor relation between transitions and configurations. For transition systems based on Petri nets, we will assume that a meta-interpreter transition t will correspond to some Petri net transition. There is a potential source of confusion here between transitions of the meta-interpreter, and transitions of a Petri net. To avoid this, we will refer to the latter as *PN-transitions*, although we will continue to use t to refer to both kinds of transitions. For transition systems based on Petri nets, pre- and post-conditions on a PN-transition t are defined in terms of *input* and *output* places, i.e., places with directed arcs respectively *into* and *out* of t. That is, the pre-condition for PN-transition t is that all input places for t must have a token; and the post-condition is that all output places for t must have a token (David and Alla 2010). For biological networks represented as Petri net, these pre- and post-conditions will simply enforce the basic stoichiometric requirements associated with the corresponding PN-transition. $Succ$ for PNs is a constrained form of the general successor relation for Petri nets defined in Durzinsky et al. (2011c) (the constraints arise from requiring that places that are not inputs or outputs for a transition t remain unchanged in successor states). A partial description of Pre, $Post$ and $Succ$ used in this paper is in "Appendix B".

Problem-specific domain knowledge used here is in the form of specifications of (possibly) relevant PN-transitions. These can be as simple as specifying input and output places, or can encode significantly more biological detail. "Appendix B" shows the different levels of details for the problems studied here. A summary is as follows:

Glycolysis PN-transitions are known reactions along with their inputs and outputs. The transitions are named using the catalyst that enables the corresponding reaction.

Table 1 General form of the data provided for system identification (top), and an example from the MAPK problem (bottom)

Place	States					
	0	1	2	…	…	n
p_1	$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	…	…	$s_{1,k}$
p_2	$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	…	…	$s_{2,k}$
…	…	…	…	…	…	…
…	…	…	…	…	…	…
p_l	$s_{l,0}$	$s_{l,1}$	$s_{l,2}$	…	…	$s_{l,k}$

Place	Time and states					
	$t = 0$	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
	s_0	s_1	s_2	s_3	s_4	s_5
map4k	1	1	1	1	1	1
map3k	1	0	0	0	0	0
map3kp	0	1	1	1	1	1
map2k	1	1	0	0	0	0
map2kp	0	0	1	0	0	0
map2kpp	0	0	0	1	1	1
mapk	1	1	1	1	0	0
mapkp	0	0	0	0	1	0
mapkpp	0	0	0	0	0	1

MAPK PN-transitions now include biological knowledge of known phosphorylation reactions. In Petri net terms, this allows us to construct extended PNs (that is, transitions with "read" arcs).

Yeast pheromone The pheromone response pathway uses some standard signalling building blocks: a G-protein cycle for transmission from the receptor; a MAPK cascade; G-protein formation from sub-units; and pathways for the formation of scaffolds that hold proteins in place. PN-transitions as domain-knowledge allows the re-use of some known sub-nets.

4.3.1 Data

Data are taken to be the result of one or more experiments, each capable of generating a sequence of states.[5] Each state is a marking in Petri net terminology: see Table 1.
For all experiments in this paper, place-values will be Boolean, with a token-value 1 denoting that adequate quantity of the the corresponding place is present, and 0 denoting that an adequate quantity is not present. This results in a qualitative variant of usual Petri nets, in which tokens are allowed to have non-negative integer values.

4.3.2 Algorithms and machines

Simulated data for experiments with noise are obtained using the probabilistic environment provided within the PRISM system (Sato and Kameya 1997) (the details are in Sect. 4.4). The

[5] Datasets can be downloaded from: http://www.cse.unsw.edu.au/~mike/PetriData.tar.

meta-interpreter, the meta-learner used the interpreter, including computation of probabilities for traces, and the program used for computing successors in Petri nets are available from the authors as Prolog programs.

The experiments were conducted on an Intel Core i7 laptop computer, using VMware virtual machine running Fedora 13, with an allocation of 2 GB for the virtual machine. The Prolog compiler used was Yap, version 6.2.2.[6]

4.4 Method

We categorise the amount of noise in the data into 4 categories: $none$, low, $medium$ and $high$ (the quantitative meaning of these qualitative values is clarified below). Similarly we categories the amount of missing data into $none$, low, $medium$, and $high$. Based on the amount of noisy and missing data (namely incomplete and incorrect data), we distinguish between the following case studies.

Complete, Correct:
$Missing = none$; $Noise = none$
Complete, Incorrect:
$Missing = none$; $Noise = low, medium, high$
Incomplete, Correct:
$Missing = low, medium, high$; $Noise = none$
Incomplete, Incorrect:
$Missing = low, medium, high$; $Noise = low, medium, high$

Our method is straightforward. For each problem, in broad outline, the steps followed are these:

1. Let T denote the target (correct) sequence of states (configurations) for the problem with an initial state S and a final state F
2. Repeat R times:
3. For $Missing = none, low, medium, high$
   (a) For $Noise = none, low, medium, high$
      i. Generate a data sample consisting of N configuration sequences using $T, Missing, Noise$
      ii. Provide the data sample to the meta-interpreter and obtain sample estimates of state-transition probabilities from traces obtained from the meta-interpreter
      iii. Obtain the highest-probability sequence of states T^* between S and F using the probability estimates obtained from the data
      iv. Compare T and T^*

The following details are relevant:

- For all problems, we will take the initial state S to be one with all places having the value 0. The final state F is problem-specific.
- For all problems, R is 10 (this is the number of repetitions of the experiments, to account for sampling variations).
- We follow the transition noise model described in Srinivasan et al. (2016). That is, low, $medium$ and $high$ levels of noise are defined in terms of the probability with which a probabilistic transition generates the output-state of the corresponding deterministic

[6] http://www.dcc.fc.up.pt/~vsc/Yap/.

transition. Here, low noise means that this probability is 90%; medium noise means that the probability is 75%; and high noise means that the probability is 50% (put another way, a low value denotes 10% noise, $medium$ denotes 25% noise and $high$ denotes 50% noise).

- Probabilities for missing values are similarly defined. A low value means that there is a 90% probability of state being present, $medium$ means a probability of 75% and $high$, a probability of 50%. will denote 25% and $high$ will denote 50%. For simplicity, initial and final states are never missing. Data that have no missing values and no noise are characterised by $Missing = none$ and $Noise = none$.
- When identifying the system with missing values, it may become necessary to identify intermediate states. This will require providing the transition system with depth-bounds greater than 1. We will consider depth-bounds of 1 and 2 for experiments with missing values. For all other cases, the depth-bound is 1.
- For all problems, the size of the data sample N is 100. That is, we provide 100 discrete time-sequences of data. (We will comment later on lower values of N).
- We compare efficacy of system identification based on a state-by-state comparison of T and T^*. The accuracy of identification is the number of states in T correctly identified in place by T^*. We will be concerned with average accuracy (or error) over the R repetitions.

4.5 Results

For each of the four cases described in the previous section, we present the results that show the average accuracy of system identification. The results are tabulated in Tables 2 and 3. A summary of these tables is this: (a) Systems are identified reliably when data are complete and noise free; (b) Provided data are not missing, reliable identification is possible even with high noise levels; (c) Provided data are not noisy, reliable identification is possible upto moderate levels of missing data; and (d) When data are noisy and missing, reliable identification is possible at low levels of noise and missing data, if the depth-bound is greater than 1.

It is instructive to examine the extent to which these basic findings are affected by experimental design choices. Specifically, we would like to know: (a) Will poor results improve if more data were provided; and (b) Will good results degrade if less data were provided. Tables 4 and 5 provide some relevant evidence. The tabulations suggest some refinements to our previous findings, namely: (a) with imperfect data, we can expect accuracy of system identification to increase if data are increased, and to fall if they are decreased; and (b) when the data are complete and correct, correct system identification possible with very low sample sizes [here, we obtain the correct system with a single data instance: this is consistent with other results in the area of meta-interpretive ILP, where complex hypotheses can be constructed, often with a single instance (Muggleton et al. 2014b)].

5 Discussion and related work

Computational approaches are increasingly being viewed as critical for knowledge discovery in the natural sciences (Kell 2012). On one hand this may be addressed by using "big data" to reduce the space of possible solutions (Hey et al. 2009). On the other hand, if data is limited, as, for example, is typical for short time-series gene expression experiments, learning can be constrained by relevant domain knowledge (Subramanian et al. 2005). The latter setting is the one addressed in our work. The view that formalisms used to characterise the semantics of computation can have a central role in understanding systems biology is not new (Regev

Table 2 System identification results (part 1)

Problem	Accuracy (%)
(a) Complete and correct data	
Glycolysis	100.0 (0.0)
MAPK	100.0 (0.0)
Yeast	100.0 (0.0)

Problem	Accuracy		
	L	M	H
(b) Noisy data (no missing data)			
Glycolysis	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)
MAPK	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)
Yeast	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)

Problem	Accuracy					
	Depth = 1			Depth = 2		
	L	M	H	L	M	H
(c) Missing data (no noise)						
Glycolysis	100.0 (0.0)	39.2 (3.8)	22.5 (3.8)	100.0 (0.0)	100.0 (0.0)	82.5 (26.7)
MAPK	52.9 (6.5)	51.4 (7.0)	42.86 (0.0)	100.0 (0.0)	100.0 (0.0)	82.9 (26.2)
Yeast	65.0 (22.9)	37.5 (0.0)	37.6 (0.0)	100.0 (0.0)	62.5 (0.0)	37.5 (0.0)

L, M, H denote "low", "medium" and "high" levels of noise or missing values. "Depth" denotes the depth-bound provided for the transition-system and meta-interpreter (where this is not shown, the value is 1) . The entries are mean values obtained over 10 repetitions and the numbers in parentheses are standard deviations

Table 3 System identification results (part 2)

Problem	Accuracy					
	Depth = 1			Depth = 2		
	L	M	H	L	M	H
(a) Missing data (low noise)						
Glycolysis	100.0 (0.0)	40.0 (5.0)	20.0 (4.1)	100.0 (0.0)	100.0 (0.0)	31.7 (8.2)
MAPK	50.0 (7.1)	42.9 (0.0)	42.9 (0.0)	100.0 (0.0)	100.0 (0.0)	42.9 (0.0)
Yeast	96.3 (11.3)	37.5 (0.0)	37.5 (0.0)	100.0 (0.0)	62.5 (0.0)	38.8 (3.8)
(b) Missing data (medium noise)						
Glycolysis	25.0 (0.0)	25.0 (0.0)	16.7 (0.0)	25.0 (0.0)	25.0 (0.0)	16.7 (0.0)
MAPK	57.1 (0.0)	51.4 (7.0)	42.9 (0.0)	100.0 (0.0)	100.0 (0.0)	42.0 (0.0)
Yeast	70.0 (20.3)	36.2 (3.78)	37.5 (0.0)	100.0 (0.0)	48.7 (10.4)	37.5 (0.0)
(c) Missing data (high noise)						
Glycolysis	16.7 (0.0)	16.7 (0.0)	16.7 (0.0)	16.7 (0.0)	16.7 (0.0)	16.7 (0.0)
MAPK	44.3 (10.0)	40.0 (8.6)	40.0 (5.7)	100.0 (0.0)	100.0 (0.0)	40.0 (5.7)
Yeast	46.2 (9.8)	36.2 (8.7)	31.2(6.2)	87.5 (19.4)	43.7 (10.1)	32.5 (6.1)

L, M, H denotes low, medium and high levels of missing values, Depth is the depth-bound, and the entries are mean values obtained over 10 repetitions. The numbers in parentheses are standard deviations

Table 4 Increasing sample size N. The accuracies shown are for noise-free data with high levels of missing data and depth = 2

Problem	Accuracy (%)	
	$N = 100$	$N = 200$
Glycolysis	82.5 (26.7)	100.0 (0.0)
MAPK	82.9 (26.2)	100.0 (0.0)
Yeast	37.5 (0.0)	37.5 (0.0)

Table 5 Decreasing sample size N. The accuracies shown are for noisy data with no missing values (a) and for complete and correct data (b)

Problem	Accuracy (%)					
	Noise = L		Noise = M		Noise = H	
	$N = 100$	$N = 10$	$N = 100$	$N = 10$	$N = 100$	$N = 10$
(a)						
Glycolysis	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)	90.0 (14.8)	100.0 (0.0)	40.8 (16.0)
MAPK	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)	95.7 (12.8)	100.0 (0.0)	54.3 (17.8)
Yeast	100.0 (0.0)	100.0 (0.0)	100.0 (0.0	98.7 (3.7)	100.0 (0.0)	50.0 (14.8)

Problem	Accuracy (%)		
	$N = 100$	$N = 10$	$N = 1$
(b)			
Glycolysis	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)
MAPK	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)
Yeast	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)

and Shapiro 2002). Petri nets have probably been the most widely used such formalism for modelling in systems biology (Koch et al. 2011). Typically, though, such models are hand-crafted from biological knowledge. Although the problem of identification, or reverse engineering, in systems biology has been the subject of many studies [a recent review is in Villaverde and Banga (2014)] the work of Durzinsky et al. (2008) appears to be the first published method for the reconstruction of Petri net models of biological systems from time series data, where a combinatorial method is used to generate the set of all pure Petri nets consistent with discrete time-series data; subsequently this was broadened to handle extended Petri nets, with read and inhibitory arcs (Durzinsky et al. 2011b). In Durzinsky et al. (2011a) their Petri net reconstruction algorithm was reformulated using Answer Set Programming [ASP—see Baral (2003) for an overview]. ASP is an approach to logic programming that has a number of useful features for systems biology modelling, including true negation [as well as default negation or negation as failure (Gelfond 2008)]; efficient solvers; and a number of declarative built-in language constructs for choice and optimization. They note some advantages of ASP for this work: that it allows a declarative reformulation of their previous implementation; the possibility of addition of declarative biological knowledge as constraints; and, since the ASP system used is based on a constraint solvers, the approach was as efficient as a previous special-purpose implementation. Essentially, the method searches for models that conform to a graph of system states, termed the experiment graph, where models are constrained by clauses specifying the network reconstruction algorithm.

There have been several approaches to the identification of Petri nets from data in the area of business process mining. Typical approaches Aalst et al. (2010) use a transition system as

an intermediate representation then apply region theory (Ehrenfeucht and Rozenberg 1990) to generate a Petri net. However, this type of approach in practice can suffer from both over- and under-fitting and may not be able to handle noisy data and incompleteness (Aalst et al. 2010). More recently work using first-order logic representations has been applied. In Bellodi et al. (2016) a two-stage aproach to workflow mining starts by extracting first-order constraints from process logs, then applies Markov logic to learn parameters of a statistical relational model. This improves predictive accuracy on real-world datasets. An enriched representation of events for process mining in first-order logic is presented in Ferrilli (2016) but no results from learning were given.

Abductive logic programming to learn to complete metabolic networks (Reiser et al. 2001) was the basis of the Robot Scientist project (King et al. 2004). This was extended to combine abduction and induction in tasks where general rules assist in prediction (Tamaddoni-Nezhad et al. 2006) where the problem is modelled by a causal network.

Inoue (2011) formulated Boolean networks as (normal) logic programs, and Inoue et al. (2014) showed how this semantics enabled a method of learning from state transition pairs, where each state is an interpretation. However, this does not identify probabilistic transitions. In Inoue et al. (2013) meta-level abduction is applied to learn biological networks represented as causal graphs. Although these are not dynamic models like Petri nets they do capture inhibitory and activatory effects.

The XHAIL system also uses ASP for representation and reasoning in a method that combines abduction with inductive search to revise biological networks (Ray et al. 2010). In this work two domain-specific meta-rules are used to characterise reactions as assertable or retractable. A more recent version was used to revise a model of the yeast metabolic network from the Robot Scientist project with over 1000 reactions (Bragaglia and Ray 2015). The task for XHAIL in these studies is to abduce rules to correctly complete the metabolic network graph, rather than the task of system identification addressed in this paper. That is, XHAIL learns a rule for each product of a reaction, whereas in our approach what is learned for each reaction is a single transition representing the change in state for all the reactants (essentially, a transition is vector-valued). Unlike the approach presented in this paper, to the best of our knowledge XHAIL is unable to include previously learned sub-networks as individual transitions in order to learn a hierarchical model. XHAIL's representation does allow learning to use both classical and default negation, which our approach does not. Both approaches can handle uncertainty by computing over weights defined for data, although XHAIL does not use probabilistic sampling.

Clearly, there is a close link between meta-interpretive system identification as we have described it, and recent research on meta-interpretive ILP (MILP) (Muggleton et al. 2014b). In this paper our approach is in some sense a development of the meta-interpretive learning approach to grammar learning (Muggleton et al. 2014b). Although this does not use probabilistic inference, the MILP approach MetaBayes does, combining meta-interpretive learning with Bayesian inference (Muggleton et al. 2014a). However, in comparison with our approach, where probabilistic sampling is defined at the object level in terms of a probabilistic transition system, in MetaBayes it is at the meta-level, essentially by converting the meta-interpreter to a stochastic logic program that is executed to generate clause refinements stochastically.

We now list some important points of similarity and difference between our approach and MILP in general. First, although both MILP and the work here use a meta-interpreter, the motivations are different. For MILP, the meta-interpreter is intended to be a general-purpose mechanism for providing higher-order templates for learning first-order logic programs. Here, the meta-interpreter is necessary for generating the operational semantics of general transition

systems. Since transition systems can be written as logic programs (as seen in the paper), it follows that the meta-interpreter used by a MILP system can, in principle, also be used for system identification. It is therefore unsurprising that the inspiration for the meta-interpreter here lies in the one described in Muggleton et al. (2014b) (however, see next).

Second, to the best of our knowledge, meta-interpreters used by MILP engines have not employed a specific abduction bound, or have been instrumented in the manner described here to allow a data-driven estimation of probabilities over proofs by sampling at the object-level. These features are necessary for practical system identification, in cases where data on system behaviour can be incorrect or incomplete, or both (see Sect. 4).

Third, the main ideas proposed in this paper are of more importance than our implementation. These ideas are: (a) the use of Plotkin's Structural Operational Semantics (SOS) as a general setting for system identification; (b) the use of a depth-bounded, probabilistic transition system as a modified form of the transition-systems studied in SOS; (c) an abduction-bounded MILP-style meta-interpreter for systems in (b); and (d) data-driven estimation of the most likely sequence of transitions between an initial and a final configuration of the system. All of (b)–(d) can be implemented within a probabilistic logic programming system like PRISM (Sato and Kameya 1997), or its generalisations like Problog (De Raedt 2007). For our purpose, an implementation using standard Prolog programs has been sufficient.

We conclude with answers to some questions that may arise from the experiments described above. First, concerning the representation:

Are we retricted to qualitative values? In all experiments, Boolean values were used to indicate the presence or absence of a sufficient quantity of a chemical. Is this a requirement of the representation? Clearly not, as shown early in the paper (Example 9), in which tokens are integer-valued. We would however expect the transition relation to become increasingly non-deterministic as the cardinality of values allowed increases This can, in turn, lead to multiple proofs, not all of which may make sense, biologically speaking. The most effective antidote for this is again more domain-knowledge.

Are we restricted to Petri nets? Again, early examples in the paper show how we are able to represent a variety of automata (including Turing machines) with the representation of transition systems. In fact, with the extension to probabilistic transition systems, we should, in principle also be able to represent different kinds of probabilistic automata. The system identification procedure we use, for example, is in effect extracting a model from the transition probability matrix of a Markov model. Our specific choice of Petri nets here is motivated by its ability to represent a wide variety of biological networks (see Koch et al. 2011).

Are we restricted to logic programs? In principle, no. The procedures in Figs. 1, 2, 3 and 4 provide a complete specification of the approach we have used. The implementation as a logic program is especially natural, given the built-in facilities for theorem-proving and backtracking.

Next, concerning the usefulness to Biology:

Will the approach scale-up? The experiments here are on real, but modest-sized networks. They have been chosen to highlight some specific features of biological networks, namely: routine metabolic reactions, catalysed reactions, cascades, feedback loops and pathway re-use. We have also seen how domain-knowledge can be naturally included at the object-level, as part of the definition of the transition relation. The experimental results here suggest that construction of large networks is likely to proceed hierarchically (the Yeast network for example, contains several smaller sub-networks, abstracted to the level of a transition), and with strong domain knowledge to constrain proofs by the meta-interpreter.

What about "real" biology? A criticism of the paper is this: all experiments are reconstructions of networks from simulated data. How useful is this? There are in fact two separate

questions here: (a) Broadly speaking, how useful to biologists are re-constructive experiments of the kind reported here?; and (b) Specifically, will the approach work with real experimental data? Standard practice in computational biology to verify that an identification algorithm works as expected is to use controlled experiments where algorithms are applied to reconstruct known networks from simulated data. Quantitative evaluations of the reconstruction performance of such algorithms are a necessary step to assess the algorithm's utility. But clearly, the results of such experiments may not be sufficient to guarantee that the technique will work with experimental data, which can often be noisy, missing or both. In this paper experiments have included such problems with the data, and the results act as a guide of what can be expected when the approach is used in practice. The question of whether the method can be used to identify biologically meaningful networks from experimental data on real systems can only be answered in conjunction with biologists. This we leave for future research.

6 Concluding remarks

System identification pre-supposes an understanding of what constitutes a system. When identifying mathematical models of a system, it is normally accepted that this means finding a set of ordinary, or partial differential equations, that can be used to simulate the dynamic behaviour of a system. But what of other kinds of system models? Here the situation is less clear, with special-purpose identification methods proposed for each kind of model. In this paper, we revisit a general model of system semantics proposed by Plotkin that can act as a template for a variety of different kinds of computational models. The Structural Operational Semantics (SOS) model proposed in Plotkin (1981) allows us to specify system behaviour by defining transition systems. By changing definitions, we are able to model system behaviour as the operation of different kinds of automata. In this paper, we show: (1) how transition systems used in SOS can be implemented as meta-interpreted logic programs; (2) how research in meta-interpretive ILP (MILP) can be used to extend the meta-interpreter for transition systems to allow for abduction of transitions; (3) how a probability estimates can be obtained to justify the preference of some transition sequences over others. Finally, and most importantly, we show using a comprehensive set of experiments how the approach developed can be used to identify Petri-net models of well-known biological systems from data of varying quality and quantity. Taken together, we believe the work makes a substantial case for the meta-interpretive induction of computational systems from data.

In principle, it should be possible to replicate the results that we have shown here with any logic programming system extended with probabilities (in our experience, it was only a technical glitch and efficiency concerns that prevented the use of PRISM, for example). To that extent, the specific implementation that we have described is unimportant. What is important though is that it is possible to learn non-trivial computational models from data. We believe that the meta-interpretive approach offers the possibility of going much further than what we have described here. It may be possible, for example, to provide templates for the structured induction of transition models. By this we mean the specification of transitions that recursively consist of a hierarchy of sub-networks. This kind of specification would bring the work much closer to the forefront of MILP research that has been looking at templates for learning recursive definitions.

On the application front, to the best of our knowledge, this is the first paper that has presented significant empirical evidence for a single approach that can deal with the problem

of learning Petri net models of biological systems from data of varying quality and quantity. The ease with which we were able to identify systems—sometimes even with substantial data deficiencies—is both surprising and promising. It is of interest to see if the approach continues to identify systems correctly with larger networks with greater complexity than the ones we have looked at here. We expect the greatest difficulties to arise from biological experiments yielding only a small amount of data. In such cases, the role of biological theory becomes increasingly important, translating into domain-knowledge for system identification.

One issue that remains to be addressed is that of identification within systems of modular components and their parameters. This is a complex problem since any suitable representation must capture both hierarchical dependencies as well as concurrency in modular interactions (Pedersen and Plotkin 2010).

A Target networks

B Domain knowledge

B.1 Transition specification

```
pre(T,S):-
        ground(S),
        model(M),
        transition(M,T,Input,_),
        \+ violates_stoichiometry(Input,S).

post(T,S):-
        ground(S),
        model(M),
        transition(M,T,_,Output),
        \+ violates_stoichiometry(Output,S).

succ(T,Si,Sj):- reachable(T,Si,Sj).

violates_stoichiometry(Places,State):-
        mem(Place,Places),
        (Place = neg(P) ->
                val(P,State,X), X \= 0;
                val(Place,State,X),
                X =< 0).

% reachable/3 is an implementation of the specification of
  legal
% transitions in a Petri net described in Durzinsky et al.
```

B.2 Problem-specific information

The specification of problem-specific knowledge demonstrates several aspects: (a) a simple declarative specification of named transitions (Glycolysis); (b) a specification in which named

transition can be arbitrary terms (MAPK); and (c) a specification in which inputs and outputs can involve arbitratry computation (Yeast) (Figs. 5, 6, 7).

Glycolysis

```
model(gly).

place(gly,glu). place(gly,atp). place(gly,adp).
place(gly,g6p). place(gly,f6p). place(gly,f16bp).
place(gly,dhap). place(gly,g3p). place(gly,nad).
place(gly,'13bpg'). place(gly,nadh). place(gly,'3pg').
place(gly,'2pg'). place(gly,pep). place(gly,pv).

% needed for constrained generation of successors by
    reachable/3
% allowed_values(Model,Place,Values)
% allowed_changes(Model,Place,Values)
allowed_values(gly,_,[0,1]).
allowed_changes(gly,_,[-1,0,1]).
```

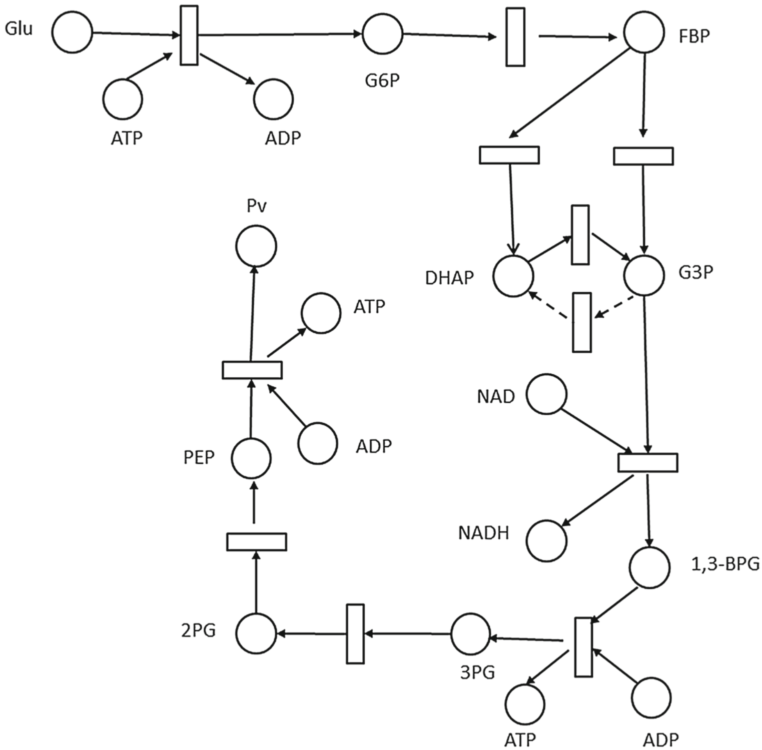


Fig. 5 Network model of the glycolysis pathway. The conversion of DHAP to G3P is taken to be in one-direction only (the reverse is shown by a dashed line, and not identified)

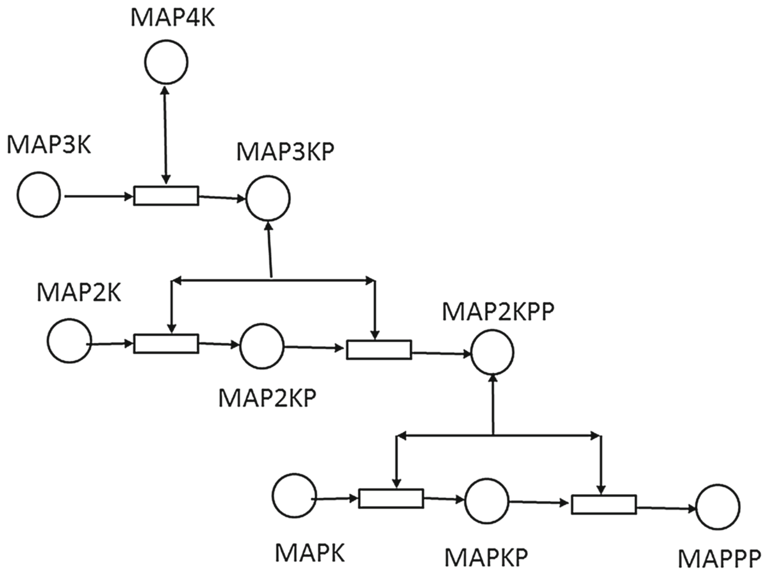


Fig. 6 Network model of the MAPK cascade. Arcs drawn with double arrows indicate "read arcs" (see Example 9)

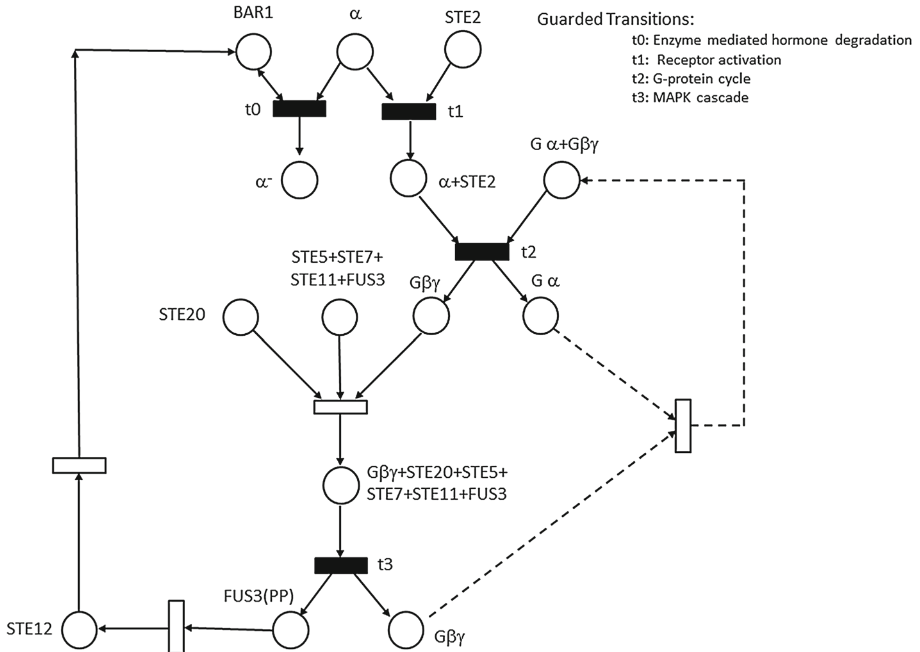


Fig. 7 Network model of yeast pheromone network. Filled (black) transitions denote sub-networks. Dashed lines as in Fig. 5

```
% Named PN-transitions
% transition(Model,Name,Input,Output)
transition(gly,hexokinase,[glu,atp],[g6p,adp]).
transition(gly,phos_gluc_isomerase,[g6p],[f6p]).
transition(gly,phos_fruc_kinase,[f6p,atp],[f16bp,adp]).
transition(gly,aldolase,[f16bp],[dhap,g3p]).
transition(gly,triose_phos,[dhap],[g3p]).
transition(gly,glyceraldehyde,[g3p,nad],['13bpg',nadh]).
transition(gly,phos_kinase,['13bpg',adp],['3pg',atp]).
transition(gly,phos_mutase,['3pg'],['2pg']).
transition(gly,enolase,['2pg'],[pep]).
transition(gly,pv_kinase,[pep],[pv]).
```

MAPK

```
model(mapk).

place(mapk,map4k). place(mapk,map3k). place(mapk,map3kp).
place(mapk,map2k). place(mapk,map2kp). place(mapk,map2kpp).
place(mapk,mapk). place(mapk,mapkp). place(mapk,mapkpp).

allowed_values(mapk,_,[0,1]).
allowed_changes(mapk,_,[-1,0,1]).

transition(mapk,phosphorylates(map4k,map3k),[map4k,map3k],
           [map4k,map3kp]).
transition(mapk,phosphorylates(map3kp,map2k),[map3kp,map2k],
           [map3kp,map2kp]).
transition(mapk,phosphorylates(map3kp,map2kp),[map3kp,map2kp],
           [map3kp,map2kpp]).
transition(mapk,phosphorylates(map2kpp,mapk),[map2kpp,mapk],
           [map2kpp,mapkp]).
transition(mapk,phosphorylates(map2kpp,mapkp),[map2kpp,mapkp],
           [map2kpp,mapkpp]).
```

Yeast pheromone

```
model(pheromone).

place(pheromone,bar1).
place(pheromone,alpha).
place(pheromone,alpha(minus)).
place(pheromone,ste2).
place(pheromone,[alpha,ste2]).
place(pheromone,[g_alpha,g_beta,g_gamma]).
place(pheromone,g_alpha).
place(pheromone,[g_beta,g_gamma]).
place(pheromone,[ste5,ste11,ste7,fus3]).
place(pheromone,ste20).
```

```
place(pheromone,fus3(pp)).
place(pheromone,ste12).

allowed_values(pheromone,_,[0,1]).
allowed_changes(pheromone,_,[-1,0,1]).

pheromone(alpha). receptor(ste2).

g_protein(g_alpha). g_protein(g_beta). g_protein(g_gamma).

map4k(ste20). map3k(ste11). map2k(ste7). mapk(fus3).

scaffold(ste5).

complex(Complex):-
        model(Net),
        place(Net,Complex),
        Complex = [_,_|_].

transition(pheromone,receptor_activation,[P,GPCR],[[P,GPCR]]):-
        pheromone(P),
        receptor(GPCR).
transition(pheromone,g_protein_cycle,[R,GC1],[GP,GC2]):-
        active_receptor(R),
        g_protein_complex(GC1),
        g_protein(GP),
        g_protein_complex(GC2),
        GC1 = [GP|GC2].
transition(pheromone,mapk_cascade,[Ras,Complex,GC],[KPP,GC]):-
        mapk_complex(Complex),
        map4k(Ras),
        g_protein_complex(GC),
        mapk(K),
        KPP =.. [K,pp].
transition(pheromone,pheromone_degradation,[Alpha,bar1],[AM]):-
        pheromone(Alpha),
        AM =..[Alpha,minus].

active_receptor([Pheromone,GPCR]):-
        pheromone(Pheromone),
        receptor(GPCR).

g_protein_complex(Complex):-
        complex(Complex),
        check_g_protein_complex(Complex).

check_g_protein_complex([G1,G2]):-
        !,
        g_protein(G1), g_protein(G2).
check_g_protein_complex([G1,G2|Rest]):-
        g_protein(G1),
```

```
        check_g_protein_complex([G2|Rest]).

mapk_complex(Complex):-
        complex(Complex),
        check_mapk_complex(Complex).

check_mapk_complex(Complex):-
        del(K1,Complex,Rest), mapk(K1),
        del(K2,Rest,Rest1), map2k(K2),
        del(K3,Rest1,_), map3k(K3).
```

References

Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge: Cambridge University Press.

Bellodi, E., Riguzzi, F., & Lamma, E. (2016). Statistical relational learning for workflow mining. *Intelligent Data Analysis*, *20*(3), 515–541.

Bragaglia, S., & Ray, O. (2015). Nonmonotonic learning in large biological networks. In J. Davis & J. Ramon (Eds.), *Proceedings of the 24th international conference on inductive logic programming (ILP-2014), volume LNAI 9046* (pp. 33–48). Berlin: Springer.

David, R., & Alla, H. (2010). *Discrete, continuous, and hybrid Petri nets* (2nd ed.). Berlin: Springer.

De Raedt, L., Kimmig, A., Toivonen, H. (2007). Problog: A probabilistic prolog and its applications in link discovery. In R. de Lopez Mantaras & M. Veloso (Eds.), *IJCAI-07: Proceedings of the 20th international joint conference on artificial intelligence* (pp. 804–809).

Durzinsky, M., Marwan, W., Ostrowski, M., Schaub, T., & Wagler, A. (2011a). Automatic network reconstruction using ASP. *Theory and Practice of Logic Programming*, *11*(4–5), 749–766.

Durzinsky, M., Wagler, A., & Marwan, W. (2011b). Reconstruction of extended Petri nets from time series data and its application to signal transduction and to gene regulatory networks. *BMC Systems Biology*, *5*, 113.

Durzinsky, M., Wagler, A., & Weismantel, R. (2011c). An algorithmic framework for network reconstruction. *Theoretical Computer Science*, *412*, 2800–2815.

Durzinsky, M., Wagler, A., Weismantel, R., & Marwan, W. (2008). Automatic reconstruction of molecular and genetic networks from discrete time series data. *BioSystems*, *93*, 181–190.

Ehrenfeucht, A., & Rozenberg, G. (1990). Partial (set) 2-structures: Part II—State spaces of concurrent systems. *Acta Informatica*, *27*, 343–368.

Ferrilli, S. (2016). Handling complex process models conditions using first-order Horn clauses. In J. Alferes, L. Bertossi, G. Governatori, P. Fodor, & D. Roman (Eds.), *RuleML 2016—Rule technologies. research, tools, and applications, volume 9718 of LNCS*. Berlin: Springer.

Gelfond, M. (2008). Answer sets. In F. van Harmelen, V. Lifschitz, & B. Porter (Eds.), *Handbook of knowledge representation* (pp. 285–316). Amsterdam: Elsevier.

Hey, T., Tansley, S., & Tolle, K. (2009). *The fourth paradigm: Data-intensive scientific discovery*. Redmond: Microsoft Reasearch.

Inoue, K. (2011). Logic programming for Boolean networks. In: *IJCAI 2011: Proceedings of the 22nd international joint conference on artificial intelligence* (pp. 924–930).

Inoue, K., Doncescu, A., & Nabeshima, H. (2013). Completing causal networks by meta-level abduction. *Machine Learning*, *91*, 239–277.

Inoue, K., Ribeiro, T., & Sakama, C. (2014). Learning from interpretation transition. *Machine Learning*, *94*(1), 51–79.

Junker, B. H., & Schreiber, F. (2008). *Analysis of biological networks*. Hoboken, NJ: Wiley.

Kakas, A. C., Kowalski, R. A., & Toni, F. (1992). Abductive logic programming. *Journal of Logic and Computation*, *2*(6), 719–770.

Kell, D. (2012). Scientific discovery as a combinatorial optimisation problem: How best to navigate the landscape of possible experiments? *Bioessays*, *34*, 236–244.

Keller, R. (1976). Formal verification of parallel programs. *Communications of the ACM*, *19*(7), 371–384.

King, R., Whelan, K., Jones, F., Reiser, P., Bryant, C., Muggleton, S., et al. (2004). Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, *427*, 247–252.

Koch, I., Reisig, W., & Schreiber, F. (Eds.). (2011). *Modeling in systems biology: The Petri net approach*. Berlin: Springer.

McCarthy, J. (1963). Towards a mathematical theory of computation. In *Proceedings of the IFIP Congress 62*, Amsterdam. North Holland.

Muggleton, S., Lin, D., Chen, J., & Tamaddoni-Nezhad, A. (2014a). MetaBayes: Bayesian meta-interpretative learning using higher-order stochastic refinement. In G. Zaverucha, V. Santos Costa, & A. Paes (Eds.), *Inductive logic programming: ILP 2013, volume 8812 of LNCS*. Berlin: Springer.

Muggleton, S., Lin, D., Pahlavi, N., & Tamaddoni-Nezhad, A. (2014b). Meta-interpretive learning: Application to grammatical inference. *Machine Learning*, *94*, 25–49.

Pedersen, M., & Plotkin, G. (2010). A language for biochemical systems: Design and formal specification. In C. Priami et al. (Eds.), *Transactions on computational systems biology XII, volume 5945 of lecture notes in bioinformatics* (pp. 77–145). Berlin: Springer.

Peterson, J. (1981). *Petri net theory and the modeling of systems*. Englewood Cliffs, NJ: Prentice-Hall.

Plotkin, G. D. (1981). A structural approach to operational semantics. Technical report, University of Aarhus.

Psillos, S. (2011). An explorer upon untrodden ground: Peirce on abduction. In D. Gabbay, S. Hartmann, & J. Woods (Eds.), *Handbook of the history of logic* (Vol. 10). Amsterdam: Elsevier.

Ray, O., Whelan, K., & King, R. (2010). Automatic revision of metabolic networks through logical analysis of experimental data. In L. De Raedt (Ed.), *Proceedings of the 19th international conference on inductive logic programming (ILP-2009), volume LNAI 5989* (pp. 194–201). Berlin: Springer.

Regev, A., & Shapiro, E. (2002). Cells as computation. *Nature*, *419*(6905), 343.

Reiser, P., King, R., Kell, D., Muggleton, S., Bryant, C., & Oliver, S. (2001). Developing a logical model of yeast metabolism. *Electronic Transactions in Artificial Intelligence*, *5*, 223–244.

Sato, T., & Kameya, Y. (1997). PRISM: A symbolic-statistical modeling language. In *Proceedings of the 15th international joint conference on artificial intelligence (IJCAI97)* (pp. 1330–1335).

Srinivasan, A., Bain, M., Vatsa, D., & Agarwal, S. (2016). Identification of transition models of biological systems in the presence of transition noise. In K. Inoue, H. Ohwada, & A. Yamamoto (Eds.), *ILP-2015—Proceedings of the 25th international conference on inductive logic programming*. Berlin: Springer (to appear).

Sterling, L., & Shapiro, E. (1994). *The art of Prolog*. Cambridge: MIT Press.

Subramanian, A., Tamayo, P., Mootha, V., Mukherjee, S., Eberta, B., Gillette, M., et al. (2005). Gene set enrichment analysis: A knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences*, *102*(43), 15545–15550.

Tamaddoni-Nezhad, A., Chaleil, R., Kakas, A., & Muggleton, S. (2006). Application of abductive ILP to learning metabolic network inhibition from temporal data. *Machine Learning*, *64*, 209–230.

Van Der Aalst, W., Rubin, V., Verbeek, H., Van Dongen, B., Kindler, E., & Gunther, C. (2010). Process mining: A two-step approach to balance between underfitting and overfitting. *Software and Systems Modeling*, *9*, 87–111.

Van Glabbeek, R. (2011). Bisimulation. In D. Padua (Ed.), *Encyclopedia of parallel computing* (pp. 136–139). Berlin: Springer.

Villaverde, A., & Banga, J. (2014). Reverse engineering and identification in systems biology: Strategies, perspectives and challenges. *Journal of the Royal Society Interface*, *11*, 20130505.

Wagler, A. (2011). Prediction of network structure. In I. Koch, W. Reisig, & F. Schreiber (Eds.), *Modeling in systems biology: The Petri net approach* (pp. 307–336). Berlin: Springer.

Yamamoto, A. (1997). Representing inductive inference with SOLD-resolution. In *Proceedings of the IJCAI'97 workshop on abduction and induction in AI*.