Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 64–73
http://www.floc-conference.org/ICLP-home.html

EFFICIENT SOLVING OF TIME-DEPENDENT ANSWER SET PROGRAMS

TIMUR FAYRUZOV [1] AND JEROEN JANSSEN [2] AND DIRK VERMEIR [2] AND CHRIS CORNELIS [1] AND MARTINE DE COCK [1,3]

[1] Dept. of Appl. Math. and Comp. Sc., Ghent University, Krijgslaan 281 (S9), 9000 Gent, Belgium
E-mail address: {timur.fayruzov,chris.cornelis}@ugent.be

[2] Dept. of Computer Science, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
E-mail address: {jeroen.janssen,dvermeir}@vub.ac.be

[3] Institute of Technology, University of Washington, 1900 Commerce St., Tacoma, WA-98402, USA
E-mail address: mdecock@u.washington.edu

Abstract. Answer set programs with time predicates are useful to model systems whose properties depend on time, like for example gene regulatory networks. A state of such a system at time point t then corresponds to the literals of an answer set that are grounded with time constant t. An important task when modelling time-dependent systems is to find steady states from which the system's behaviour does not change anymore. This task is complicated by the fact that it is typically not known in advance at what time steps these steady states occur. A brute force approach of estimating a time upper bound t_{max} and grounding and solving the program w.r.t. that upper bound leads to a suboptimal solving time when the estimate is too low or too high. In this paper we propose a more efficient algorithm for solving Markovian programs, which are time-dependent programs for which the next state depends only on the previous state. Instead of solving these Markovian programs for a long time interval $\{0, \ldots, t_{max}\}$, we successively find answer sets of parts of the grounded program. Our approach guarantees the discovery of all steady states and cycles while avoiding unnecessary extra work.

1. Introduction

Answer Set Programming (ASP) is a form of non-monotonic reasoning based on the stable-model semantics [Gel88]. The number of ASP application domains is growing fast (see e.g. [Dwo08, Tra06, Sch09]). Some of these require an adaptation of the general-purpose solving process to their specific needs to allow for faster answer set computation. One broad domain of ASP applications uses programs that depend on a parameter that bounds the size of a solution. Consider e.g. the following time-dependent answer set program, for which the

1998 ACM Subject Classification: I.2.4 [Artificial Intelligence] Relation systems; J.3 [Computer Applications]: Biology and genetics.

Key words and phrases: answer set programming, time-dependent programs, gene regulation networks.

Jeroen Janssen is funded by a joint Research Foundation-Flanders (FWO) project. Chris Cornelis is a postdoctoral fellow of the Research Foundation-Flanders (FWO).

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.64

grounding size depends on the parameter t_{max}. This program uses a non-standard notation of the form $p@T$ which is equivalent to $p(T)$. The purpose of this notation is described in Section 3.

Example 1.1. Program P consists of the following rules:

$$
\begin{array}{lll}
time(0 \dots t_{max}). & & \\
q@T & \leftarrow & p@(T-1), time(T), time(T-1). \\
v@T & \leftarrow & q@(T-1), not\ w@T, time(T), time(T-1). \\
w@T & \leftarrow & q@(T-1), not\ v@T, r(X), time(T), time(T-1). \\
p@T & \leftarrow & time(T). \\
r(str). & &
\end{array}
$$

where $time(0 \dots t_{max})$ is a shorthand for the facts $time@0, time@1, \dots, time@t_{max}$ and T is a time-bound variable. This program describes the behaviour of a system whose properties depend on time. The answer sets for this program change as the time boundary t_{max} increases. When $t_{max} = 0$ there is only one answer set[1], namely $A = \{r(str), time(0), p(0)\}$. The unique answer set for $t_{max} = 1$ is $B = A \cup \{time(1), p(1), q(1)\}$, which is twice the size of the previous one. For $t_{max} = 2$, negation as failure comes into play, resulting in two different answer sets $C = B \cup \{time(2), p(2), q(2), v(2)\}$ and $D = B \cup \{time(2), p(2), q(2), w(2)\}$. For $t_{max} = 3$ there are already four different answer sets:

$$
\begin{array}{lllllll}
E & = & C \cup \{time(3), p(3), q(3), v(3)\} & \qquad & G & = & D \cup \{time(3), p(3), q(3), w(3)\} \\
F & = & C \cup \{time(3), p(3), q(3), w(3)\} & & H & = & D \cup \{time(3), p(3), q(3), v(3)\}
\end{array}
$$

As this example illustrates, the number of answer sets as well as the size of the answer sets of a time-dependent program can increase exponentially in the time boundary.

An important task when modelling and simulating a time-dependent system is to find its steady states. Answer set E in Example 1.1 contains one steady state of the system described by program P, as $time, p, q$, and v (and no other time-dependent predicates) belong to E both for time step 2 and 3.

The main problem in finding these steady states is that it is typically not known in advance at what time steps they occur. Furthermore a system may converge to several steady states (e.g. E and F in Example 1.1) or may even oscillate among several states repeatedly (e.g a cycle between v and w in Example 1.1 that manifests itself in some of the answer sets of P for $t_{max} = 4$), and we may want to find them all. A brute force approach of estimating a time upper bound and grounding and solving the program w.r.t. that upper bound may lead to a suboptimal solving time: if the upper bound is estimated too high, the grounded program is larger than necessary to find the steady states, hence requiring unnecessary work, and if it is estimated too low, not all steady states are found, meaning the process needs to be redone for a larger estimate.

In this paper we propose a technique that allows to find all steady states and cycles efficiently. To this end, we define the notion of Markovian programs, which can be grounded for one time step at a time. We introduce a way of solving these programs by solving one-step grounded versions. These programs can be used to model protein interaction networks as described e.g. in [Fay09]. We proceed by recalling ASP notions in Section 2, formally defining time-dependent programs and Markovian programs in Section 3, and proposing a method to solve these programs efficiently in Section 4. We explain the difference with other approaches in Section 5 and finally conclude in Section 6.

[1]See Section 2 for preliminaries w.r.t. answer set programming.

2. Preliminaries

Answer set programs are built from a signature $\sigma = \langle \gamma, \upsilon, \pi \rangle$, where γ is a set of *constant symbols*, υ is a set of *variable symbols*, and $\pi = \bigcup_{i=1}^{m} \pi_i (m \in \mathbb{N})$ is the union of sets π_i of i-ary *predicate symbols*. We define a set of *variable expressions* ϵ containing expressions of the form $t' \pm t''$ where $t' \in \upsilon$ and $t'' \in \gamma$. An *atom* over σ is an object of the form $p(t_1, \ldots, t_n)$, where $p \in \pi_n$ and $t_i \in \gamma \cup \upsilon \cup \epsilon$ for each $i \in 1 \ldots n$. We implicitly assume that if t_i is a symbol starting with a capital, it denotes a variable; otherwise it is a constant. A *literal* over σ is either an atom, or an atom preceded by $\neg$, which denotes *classical negation*. *Naf-literals* over σ, denoting negation-as-failure, are of the form $\mathbf{not}\, l$, where l is a literal over σ. For a set of literals X we introduce the notation $\mathbf{not}\, X = \{\mathbf{not}\, l | l \in X\}$. For a literal or a naf-literal l, we use $\mathbf{vars}(l)$ to denote the set of variables contained in l. If $\mathbf{vars}(l) = \emptyset$ then l is called *ground*.

A *rule* r over σ is an object of the form $l_0 \leftarrow l_1, \ldots, l_m, \mathbf{not}\, l_{m+1}, \ldots, \mathbf{not}\, l_n$, where l_i, for $i \in 0 \ldots n$, are literals over σ. If each l_i is ground, it is called a *ground rule*. We refer to l_0 as the *head* of r, denoted as $head(r)$, to the set $\{l_1, \ldots, l_m, \mathbf{not}\, l_{m+1}, \ldots, \mathbf{not}\, l_n\}$ as the body of r, denoted as $body(r)$, to $\{l_1, \ldots, l_m\}$ as the positive part of the body, denoted as $pos(r)$, and to the set $\{l_{m+1}, \ldots, l_n\}$ as the negative part of the body, denoted as $neg(r)$. We denote $Lit(r) = pos(r) \cup neg(r)$. If the head of the rule is empty, the rule is called a *constraint*[2]; if the body of the rule is empty, the rule is called a *fact*.

An *answer set program* P over a signature σ is a finite set of rules over σ. If all rules in P are ground, it is called a *ground program*. The process of *grounding* constructs a ground program $Gnd(P)$ from an answer set program P over a signature σ by replacing each rule r by the set of rules obtained from r by all possible substitutions of the constants of σ for the variables in r. If any of the predicate arguments takes on a composite form $t' \pm t''$ with t', t'' grounded as numbers, they are substituted with the resulting value. A rule r that does not contain negation-as-failure, i.e. $neg(r) = \emptyset$, is called a *simple rule*. A program that contains only simple rules is called a simple program.

Turning to the semantics, a set of ground literals I over a signature σ is called an *interpretation* if it is *consistent*, i.e. there is no literal l such that both $l \in I$ and $\neg l \in I$. An interpretation I is a model of a simple rule r iff $pos(r) \not\subseteq I \vee head(r) \in I$. An interpretation that is a model of all rules of a simple program P is called a *model* of P. The minimal model of a simple program P is called an *answer set* of P. If P contains negation-as-failure, then an interpretation I of P is called an *answer set* of P iff I is the answer set of the *reduct program* P^I, where $P^I = \{head(r) \leftarrow pos(r) \mid r \in P, I \cap neg(r) = \emptyset\}$. The set of all answer sets of a program P is denoted as $AS(P)$.

3. Theoretical underpinnings

3.1. Time-dependent programs

In the remainder of this paper, we designate certain predicates as *time-dependent predicates* and denote atoms built with these predicates as $p(t_1, \ldots, t_{n-1})@\theta$ where θ is called a *time argument*. This is a convenience notation that allows to separate the (semantic)

[2]We do not consider constraints in our formal language, since a constraint $\leftarrow \beta$ can be simulated by a rule $l \leftarrow \mathbf{not}\, l, \beta$, where l is a literal not occurring in the program.

notion of time from the underlying syntactic representation. This notation is translated to a conventional atom of the form $p(t_1, \ldots, t_{n-1}, \theta)$ at grounding time.

Definition 3.1 (Time-dependent program)**.** A *time-dependent program* $\mathbf{P}$ is a tuple $\langle P, \tau \rangle$ over a signature $\sigma = \langle \gamma, \upsilon, \pi \rangle$, such that P is an answer set program over σ and $\tau \subseteq \pi$ is a set of *time-dependent predicates*. We denote the set of n-ary time-dependent predicates as τ_n. We define the set of *free time-dependent literals*

$$\mathcal{F}_{\mathbf{P}} = \bigcup \left\{ \{p(t_1, \ldots, t_{n-1})@\theta, \neg p(t_1, \ldots, t_{n-1})@\theta\} \;\middle|\; \begin{array}{l} t_1, \ldots, t_{n-1} \in \gamma \cup \upsilon \cup \epsilon, \\ \theta \in \upsilon \cup \epsilon, p \in \tau_n, 1 \leq n \leq m \end{array} \right\}$$

and the set of *bound time-dependent literals*

$$\mathcal{B}_{\mathbf{P}} = \bigcup \left\{ \{p(t_1, \ldots, t_{n-1})@\theta, \neg p(t_1, \ldots, t_{n-1})@\theta\} \;\middle|\; \begin{array}{l} t_1, \ldots, t_{n-1} \in \gamma \cup \upsilon \cup \epsilon, \\ \theta \in \gamma, p \in \tau_n, 1 \leq n \leq m \end{array} \right\}.$$

The literals from $\mathcal{F}_{\mathbf{P}}$ contain a variable or a variable expression as the time argument, while the literals from $\mathcal{B}_{\mathbf{P}}$ contain a constant as the time argument. The set of *time-dependent literals* of a time-dependent program $\mathbf{P}$ is defined as $Lit(\mathbf{P})^\tau = \mathcal{F}_{\mathbf{P}} \cup \mathcal{B}_{\mathbf{P}}$. It is a subset of the set of all literals of $\mathbf{P}$, which is defined as $Lit(\mathbf{P}) = \bigcup_{r \in P} Lit(r)$. Furthermore, for $l \in Lit(\mathbf{P})^\tau$, we use $t_{arg}(l)$ to refer to the time argument θ of l. A time-dependent program $\mathbf{P}$ is called *well-typed* iff

$$\forall r \in P \cdot (Lit(\mathbf{P})^\tau \cap (pos(r) \cup neg(r)) \neq \emptyset) \Rightarrow (head(r) \in Lit(\mathbf{P})^\tau)$$

Intuitively, if a rule in a well-typed time-dependent program contains a time-dependent literal in its body, it should contain a time-dependent literal in its head. In the remainder we will only consider well-typed time-dependent programs.

Definition 3.2 (t-grounding of a time-dependent literal)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and $t \in \mathbb{N}$. The t-grounding of a literal $l \in Lit(\mathbf{P})$, denoted as $Gnd(l)_t$, is obtained as follows: 1) if $l \in Lit(\mathbf{P}) \setminus \mathcal{F}_{\mathbf{P}}$ then $Gnd(l)_t = l$; 2) if $l \in \mathcal{F}_{\mathbf{P}}$ then the variable in $t_{arg}(l)$ is replaced by t, and in case of a variable expression the resulting value is calculated. In all cases, the obtained literal $Gnd(l)_t$ is subsequently translated to the conventional ASP notation. For a set of literals L, we define the t-grounding of this set as $Gnd(L)_t = \bigcup_{l \in L} Gnd(l)_t$, i.e. we take the pointwise t-grounding of its elements.

Example 3.3. The 2-grounding of literal $l = p(X, a)@(T + 1)$ is $Gnd(l)_2 = p(X, a, 3)$.

Definition 3.4 (t-grounding of a rule)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and $t \in \mathbb{N}$. The t-grounding of a rule $r \in P$ is defined as

$$Gnd(r)_t = Gnd(head(r))_t \leftarrow Gnd(pos(r))_t, \mathbf{not}\, Gnd(neg(r))_t$$

Definition 3.5 (t-grounding of a time-dependent program)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and $t_{max} \in \mathbb{N}$. The t_{max}-grounding of $\mathbf{P}$ is defined as

$$Gnd(\mathbf{P})_{t_{max}} = Gnd(\{Gnd(r)_{t'} \mid r \in P, t' \in \mathbb{N}, t' \leq t_{max}\})$$

Intuitively, to obtain $Gnd(\mathbf{P})_{t_{max}}$ we instantiate all time-dependent literals with a set of time points $\{t' | 0 \leq t' \leq t_{max}\}$ and then ground the resulting program in the conventional way.

Example 3.6. Let $\mathbf{P} = \langle P, \{v, w, q, p, time\} \rangle$ where P is the program from Example 1.1. Its 2-grounding $Gnd(\mathbf{P})_2$ is obtained by setting $t_{max} = 2$ and is defined as

$$
\begin{array}{lllll}
1: & time(0). & & & \\
2: & time(1). & & & \\
3: & time(2). & & & \\
4: & q(0) & \leftarrow & p(-1), time(0), time(-1). & \\
5: & q(1) & \leftarrow & p(0), time(1), time(0). & \\
6: & q(2) & \leftarrow & p(1), time(2), time(1). & \\
7: & v(0) & \leftarrow & q(-1), not\ w(0), time(0), time(-1). & \\
8: & v(1) & \leftarrow & q(0), not\ w(1), time(1), time(0). & \\
9: & v(2) & \leftarrow & q(1), not\ w(2), time(2), time(1). & \\
10: & w(0) & \leftarrow & q(-1), not\ v(0), r(str), time(0), time(-1). & \\
11: & w(1) & \leftarrow & q(0), not\ v(1), r(str), time(1), time(0). & \\
12: & w(2) & \leftarrow & q(1), not\ v(2), r(str), time(2), time(1). & \\
13: & p(0) & \leftarrow & time(0). & \\
14: & p(1) & \leftarrow & time(1). & \\
15: & p(2) & \leftarrow & time(2). & \\
16: & r(str). & & &
\end{array}
$$

Definition 3.7 (State of an answer set)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and I be an answer set of the t_{max}-grounding $Gnd(\mathbf{P})_{t_{max}}$ for $t_{max} \in \mathbb{N}$. Furthermore let $t \in \mathbb{N}$ with $t \leq t_{max}$. The *state* of I at time point t is defined as

$$I^t = \{l \mid l \in I, t_{arg}(l) = t\}$$

Intuitively, the state of answer set I of $Gnd(\mathbf{P})_{t_{max}}$ at time point t is the set of ground time-dependent literals in I that were grounded with t in the time argument. Two states are called equivalent if the only difference between literals in these states is in the values of the time points (see Example 3.12). We denote state equivalence as $I^{t'} =_{time} I^{t''}$.

Example 3.8. Consider the program $Gnd(\mathbf{P})_2$ from Example 3.6. The answer sets of this program are C and D as defined in Example 1.1. The states of answer set C at time points 0, 1 and 2 are $C^0 = \{time(0), p(0)\}$, $C^1 = \{time(1), p(1), q(1)\}$ and $C^2 = \{time(2), p(2), q(2), v(2)\}$.

Definition 3.9 (Trajectory of an answer set)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and I an answer set of the t_{max}-grounding $Gnd(\mathbf{P})_{t_{max}}$ for $t_{max} \in \mathbb{N}$. The *trajectory* of I is defined as

$$T^I = \langle I^0 \dots I^{t_{max}} \rangle$$

Example 3.10. The trajectory of answer set C of program $Gnd(\mathbf{P})_2$ from Example 3.6 is

$$T^C = \langle \{time(0), p(0)\}, \{time(1), p(1), q(1)\}, \{time(2), p(2), q(2), v(2)\} \rangle$$

Definition 3.11 (Steady state, steady cycle)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a time-dependent program and I be an answer set of the t_{max}-grounding $Gnd(\mathbf{P})_{t_{max}}$ for $t_{max} \in \mathbb{N}$. The state of I at time point t, with $t < t_{max}$, is called a steady state iff $I^t =_{time} I^{t+1}$. The sequence $\langle I^k \rangle_{t_1 \leq k \leq t_2}$, with $t_1 \in \mathbb{N}$, $t_2 \in \mathbb{N}$ and $t_1 < t_2 \leq t_{max}$, is called a *steady cycle* iff $I^{t_1} =_{time} I^{t_2}$.

Note that to define whether a state is a steady state it is enough to check the next state, because if it does not change in the next step it will not change in the following steps as well due to the deterministic nature of the model.

Example 3.12. The 3-grounding $Gnd(\mathbf{P})_3$ of $\mathbf{P} = \langle P, \{v, w, q, p, time\} \rangle$ where P is the program from Example 1.1, has answer sets E, F, G, and H as defined in Example 1.1. The states of answer set E are $E^0 = \{time(0), p(0)\}$, $E^1 = \{time(1), p(1), q(1)\}$, $E^2 = \{time(2), p(2), q(2),\ v(2)\}$, and $E^3 = \{time(3), p(3), q(3), v(3)\}$. E^2 is a steady state, as $E^2 =_{time} E^3$.

When solving time-dependent programs, one is usually interested in finding steady states, steady cycles and trajectories leading to these states, as they can help to verify the model's correctness and/or provide new hypotheses about the behaviour of the underlying system. An important problem is that it is in general impossible to accurately estimate an upper time bound t_{max} that suffices to find all steady states. Thus, one should manually

adjust the bound and recompute answer sets over and over, which is very inefficient. In the following section we narrow down time-dependent programs to Markovian programs and propose an approach that does not require a time bound estimation for trajectory computation.

3.2. Markovian programs

In this section we define a subclass of time-dependent programs, called Markovian programs. This type of time-dependent programs is defined in such a way that every next state directly depends only on the previous state, and does not depend on any of the future states (hence the name Markovian). This is a reasonable assumption as real-world models are normally unaware of any future events and make their decisions based on the information directly available.

Recall that steady states and steady cycles for a time-dependent program $\mathbf{P}$ can be found by grounding the program for a manually chosen time upper bound t_{max} (see Definition 3.5), solving the resulting ground program $Gnd(\mathbf{P})_{t_{max}}$ to obtain its answer sets, and verifying whether the answer sets reveal steady states or cycles (see Definition 3.11). The Achilles' heel in this procedure is in the manual choice of t_{max}. Iteratively incrementing it and repeating the above process until reaching a time point t_{max} at which a steady state or cycle is encountered is inefficient, because that would require solving $Gnd(\mathbf{P})_0$, $Gnd(\mathbf{P})_1$, $Gnd(\mathbf{P})_2$, ..., $Gnd(\mathbf{P})_{t_{max}}$, or, in other words, grounded versions of the original time-dependent program for time intervals $\{0,1\}$,$\{0,1,2\}$,...,$\{0,\ldots,t_{max}\}$. Instead, we propose to consecutively solve smaller programs for intervals $\{0,1\}$, $\{1,2\}$,..., $\{t_{max}-1,t_{max}\}$. This approach is more efficient because we ground only for one time step at a time and solve smaller programs in every iteration. Further in this section we show that by doing so we obtain the same answer sets as by solving the initial program for interval $\{0,\ldots,t_{max}\}$.

Definition 3.13 (Markovian program)**.** A time-dependent program $\mathbf{P} = \langle P, \tau \rangle$ is called *Markovian* iff it satisfies the following conditions for every $r \in P$ with $head(r) \in Lit(\mathbf{P})^\tau$ and $t \in \mathbb{N}$:

(1) $t_{arg}(head(r)) \in \gamma \cup \upsilon$
(2) $t_{arg}(Gnd(head(r))_t) = t_{arg}(Gnd(l)_t)$ or $t_{arg}(Gnd(head(r))_t) = t_{arg}(Gnd(l)_t) + 1$ for all $l \in Lit(r) \cap Lit(\mathbf{P})^\tau$

Rules in a Markovian program $\mathbf{P}$ can be divided into two subsets: a program that describes temporal relationships $P^\tau = \{r | r \in P, (head(r) \cup Lit(r)) \cap Lit(\mathbf{P})^\tau \neq \emptyset\}$ and a program that describes the rest of the relationships $P^e = P \setminus P^\tau$. Program P^e can be interpreted as environmental conditions that are invariant over time. By definition, P^e is independent from the program's temporal part, thus it can be solved separately to obtain its answer sets that represent the values of these conditions. Note that if P^e does not have an answer set, then for any $t_{max} \in \mathbb{N}$, $Gnd(\mathbf{P})_{t_{max}}$ does not have an answer set either.

Example 3.14. Consider Markovian program $\mathbf{P}$ and its 2-grounding $Gnd(\mathbf{P})_2$ as defined in Example 3.6. Here the program P^τ contains rules 1-15, while the program P^e contains rule 16. The program $Gnd(\mathbf{P})_2$ has two answer sets, namely C and D as defined in Example 1.1. The unique answer set of P^e is $\{r(str)\}$.

Definition 3.15 (Partial temporal grounding)**.** Let $\mathbf{P} = \langle P, \tau \rangle$ be a Markovian program and $t \in \mathbb{N}$. The *partial temporal grounding* of $\mathbf{P}$ for time point t is defined as

$$P_t = \{Gnd(r)_t | r \in P, head(r) \in Lit(\mathbf{P})^\tau, t_{arg}(Gnd(head(r))_t) = t\}$$

In other words, a partial temporal grounding for a time point t is the set of t-grounded rules whose head depends on time point t.

Example 3.16. The partial temporal grounding of $\mathbf{P}$ built in Example 3.6 for time point 2 is the program P_2 that is defined as follows

$$\begin{array}{rlll} 3: & time(2). & & \\ 6: & q(2) & \leftarrow & p(1), time(2), time(1). \\ 9: & v(2) & \leftarrow & q(1), not\ w(2), time(2), time(1). \\ 12: & w(2) & \leftarrow & q(1), not\ v(2), r(X), time(2), time(1). \\ 15: & p(2) & \leftarrow & time(2). \end{array}$$

Assume that the t_{max}-grounding $Gnd(\mathbf{P})_{t_{max}}$ of a Markovian program $\mathbf{P}$ has an answer set A. Once A is known, using Definition 3.7, we can straightforwardly find its states at time points 0, 1, ..., t_{max}, i.e., A^0, A^1, ..., $A^{t_{max}}$. Below we show that it is also possible to find states of an answer set without prior knowledge of an answer set itself. In particular, the state A^t of an (unknown) answer set of $Gnd(\mathbf{P})_{t_{max}}$ at time point t can be computed based on knowledge of the state A^{t-1} at time point $t-1$, as well as knowledge of an answer set A^{-1} of P^e. This means that from the answer sets of P^e, the set of states at time point 0 can be found, and from this the set of states at time point 1, etc. This is done by building $P'_t = P_t \cup \{l \leftarrow . | l \in A^{t-1} \cup A^{-1}\}$ and then grounding P'_t and transforming it by replacing literals from $A^{t-1} \cup A^{-1}$ with true values, which is formally defined in Definition 3.17. Solving the resulting (small) program yields as answer sets the possible states at time point t given the state A^{t-1} and the environmental conditions A^{-1}.

Definition 3.17 (Partial reduct)**.** Let P be a ground program, I an interpretation of P and $P_I = \{l \leftarrow . | l \in I\}$, such that $P_I \subseteq P$ and $head(P \setminus P_I) \cap I = \emptyset$. The *partial reduct* of P w.r.t. I is the program $R^I(P)$ defined as

$$R^I(P) = \{head(r) \leftarrow (pos(r) \setminus I), \mathbf{not}\, neg(r). | r \in P \setminus P_I, neg(r) \cap I = \emptyset\}$$

Example 3.18. Assume we known that $\{time(1), p(1), q(1)\}$ is the state at time point 1 of a (possibly unknown) answer set of program $Gnd(\mathbf{P})_2$ from Example 3.6. We also know an answer set of P^e, namely $\{r(str)\}$. Let P_2 be the partial temporal grounding of $\mathbf{P}$ for time point 2 as described in Example 3.16. We construct the set $I = \{time(1), p(1), q(1)\} \cup \{r(str)\}$ and the program $P'_2 = Gnd(P_2 \cup \{l \leftarrow . | l \in I\})$ as follows:

$$\begin{array}{llll} time(2). & & & time(1). \\ q(2) & \leftarrow & p(1), time(2), time(1). & p(1). \\ v(2) & \leftarrow & q(1), not\ w(2), time(2), time(1). & q(1). \\ w(2) & \leftarrow & q(1), not\ v(2), r(str), time(2), time(1). & r(str). \\ p(2) & \leftarrow & time(2). & \end{array}$$

The partial reduct $R^I(P'_2)$ is then defined as

$$\begin{array}{llllll} time(2). & & & & & \\ q(2) & \leftarrow & time(2). & w(2) & \leftarrow & not\ v(2), time(2). \\ v(2) & \leftarrow & not\ w(2), time(2). & p(2) & \leftarrow & time(2). \end{array}$$

By applying the partial reduct we remove the literals from I that appear positively in rule bodies as well as the facts that appear as literals in I. The answer sets of the resulting program are $\{time(2), p(2), q(2), v(2)\}$ and $\{time(2), p(2), q(2), w(2)\}$ which correspond to C^2 and D^2 with C and D as in Example 1.1.

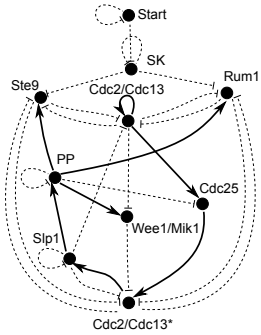

Figure 1: The gene regulatory network of Fission Yeast that can be modelled using Markovian programs (illustration from [Dav08]).

The theorem below states that instead of computing answer sets of a Markovian program $Gnd(\mathbf{P})_{t_{max}}$ directly, we can compute answer sets of smaller programs for every time step $0 \leq t' \leq t_{max}$ consecutively and obtain the same result. This fact has as an important implication that we can arrive at answer sets of a Markovian program without considering t_{max} at all, which allows to impose another stopping condition. This is the technique that is used to implement an algorithm for computing steady states explained in the following section.

Theorem 3.19. *Let* $\mathbf{P} = \langle P, \tau \rangle$ *be a Markovian program and* $Gnd(\mathbf{P})_{t_{max}}$ *be a* t_{max}*-grounding of* $\mathbf{P}$ *for* $t_{max} \in \mathbb{N}$*, then*

$$AS(Gnd(\mathbf{P})_{t_{max}}) = \left\{ B^{-1} \cup \ldots \cup B^{t_{max}} \;\middle|\; \begin{array}{l} B^{-1} \in AS(P^e), , B^t \in AS(R^{B^{t-1} \cup B^{-1}}(P'_t)), \\ t \in 0 \ldots t_{max} \end{array} \right\}$$

with $P'_t = Gnd(P_t \cup \{l \leftarrow . | l \in B^{t-1} \cup B^{-1}\})$.

4. Practical application

The results from the previous section give rise to an algorithm for finding all steady states and cycles of Markovian programs. It can be summarized as:

(1) Solve program P^e with the environmental conditions and initialize $t = 0$.
(2) Obtain the partial temporal grounding for t and find the system's states at time t.
(3) Update the list of trajectories with the new states found in step (2).
(4) Increment t.
(5) If any of the trajectories did not reach steady state or cycle, go to step (2).

This algorithm can be applied to model gene regulatory networks. An example regulatory network of Fission Yeast is presented in Figure 1, where nodes stand for genes and proteins, pointed edges define the activation of one node by another and blunt edges define the inhibition of one node by another. The semantics of the network can be expressed as a program P, while the actual network structure can be defined independently in a separate program P' as described in [Fay09]. The resulting program is a Markovian program $\mathbf{P} = \langle P \cup P', \tau \rangle$. A trajectory in the network is found w.r.t. an initial state of the network. The state of the network is defined as a combination of states of its nodes, where the state

of a node is defined as $active(a, T)$ or $inhibited(a, T)$ where a is a protein and T is a time variable, i.e. $active, inhibited \in \tau$. Looking at the network it is not possible to estimate how many time steps it would take to find all network steady states, and setting the time boundary too high would result in significant computation overheads, while the approach we propose does not involve an explicit time boundary and thus avoids these overheads. Solving the program with the algorithm outlined above allows to obtain trajectories and all steady states and cycles of the network that are reported in [Dav08].

5. Related work

In Section 4 we proposed a method to find all steady states of a Markovian program efficiently. However, our approach is not the only way to deal with the problem. Gebser et al. have recently proposed an incremental program solving approach and a specially constructed solver *iclingo* that allows for solving incremental programs [Geb08]. Even though this solver, when used for Markovian programs, terminates as soon as the first steady state is encountered, and hence unlike our approach does not find all steady states, Gebser et al.'s proposal is relevant to our work. An incremental program includes a special incremental parameter k and consists of three parts (base, cumulative and volatile) that allow to reduce the efforts required for solving this type of programs. Due to the space limitation we refer to [Geb08] for details. The advantage of this approach compared to the usual solving process is that it reduces the effort of computing the answer set for unknown k.

If we regard the incremental parameter k as time, we can simulate a Markovian program $\mathbf{P} = \langle P, \tau \rangle$ by putting P^e in the base part and P^τ in the cumulative part. However, implementing the volatile part is not straightforward. Given the set τ of time-dependent predicates we can write rules to capture steady states or cycles and define a constraint over the occurrence of such a state or cycle in the volatile part, as illustrated below.

Example 5.1. Let $\mathbf{P} = \langle P, \tau \rangle$ be a Markovian program over a signature $\sigma = \langle \gamma, \upsilon, \pi \rangle$ and $\tau = \{u, v\}$ where $u, v \in \pi$ are unary time-dependent predicates. We define an incremental program P' from $\mathbf{P}$ as explained above, i.e. by putting P^e in the base part of P' and P^τ in the cumulative part of P'. The exact contents of P^e and P^τ do not matter for the sake of this example. Next, we add the following set of rules to the cumulative part of P':

$$
\begin{array}{lcl}
int(0..k-1). & & \\
h(k) & \leftarrow & not\ u(k), not\ u(k-T_1), not\ v(k), not\ v(k-T_1), int(T_1). \\
h(k) & \leftarrow & u(k), u(k-T_1), not\ v(k), not\ v(k-T_1), int(T_1). \\
h(k) & \leftarrow & v(k), v(k-T_1), not\ u(k), not\ u(k-T_1), int(T_1). \\
h(k) & \leftarrow & v(k), v(k-T_1), u(k), u(k-T_1), int(T_1).
\end{array}
$$

Finally, we initialize the volatile part of P' with the rule $\leftarrow \mathbf{not}\, h(k)$. Intuitively, the appearance of $h(k)$ in an answer set of P' indicates that a steady state or cycle is found. The constraint in the volatile part only allows answer sets that contain $h(k)$.

However, there are two pitfalls associated with the above encoding. First, the number of rules that needs to be added to the cumulative part grows exponentially with the number and the arity of time-dependent predicates; recall that we do not only need all combinations of time-dependent predicates, but also all their possible groundings. Secondly, the solver terminates as soon as the first steady state is encountered, and hence does not generate all steady states of the program. It is not obvious how to encode the program in order to deal with these problems. For these reasons, the approach we propose in this paper is a

more suitable candidate to tackle the steady state search problem in Markovian programs. Applying a meta-procedure similar to the algorithm from Section 4 is not a solution as the incremental program still cannot find all steady states that stem from the same initial state without adjusting the termination condition $h(k)$.

Action languages [Gel92], another set of formalisms applicable to solve time-dependent programs, provide a high-level description language that can be adopted to model time-dependent systems. However, they suffer from the same drawback as incremental programs: it is not possible to define a set of constraints that allows to find all steady states and cycles.

6. Conclusions

In this paper, we introduced time-dependent answer set programs, which are useful to model systems like gene regulatory networks whose behaviour depends on time. An important task when modelling such systems is to find their steady states and cycles. Unfortunately, it is typically not known in advance at what time steps these steady states manifest themselves. A brute force approach of estimating a time upper bound and grounding and solving the program w.r.t. that upper bound may lead to a bad solving time: if the upper bound's estimate is too high, the grounded program is larger than necessary to find the steady states, hence requiring unnecessary work, and if it is too low, not all steady states (if any) are found and the process needs to be redone for a larger estimate.

We proposed an efficient algorithm for solving Markovian programs, i.e. time-dependent programs for which the next state of the program depends only on the previous state of the program. This is a reasonable assumption as real-world models are normally unaware of any future events and make their decisions based on the information directly available. Instead of solving Markovian programs for some long time interval $\{0, \ldots, t_{max}\}$ we consecutively solve smaller programs for intervals $\{0, 1\}$, $\{1, 2\}$,..., $\{t_{max} - 1, t_{max}\}$, which can be done more efficiently. We showed that by doing so we obtain the same answer sets as by solving the initial program for interval $\{0, \ldots, t_{max}\}$. We successfully applied our algorithm to find the steady states of a gene regulatory network for fission yeast.

References

[Dav08] Maria I. Davidich and Stefan Bornholdt. Boolean network model predicts cell cycle sequence of fission yeast. *PLoS ONE*, 3(2), 2008.

[Dwo08] Steve Dworschak, Susanne Grell, Victoria J. Nikiforova, Torsten Schaub, and Joachim Selbig. Modeling biological networks by action languages via answer set programming. *Constraints*, 13(1-2):21–65, 2008.

[Fay09] Timur Fayruzov, Martine De Cock, Chris Cornelis, and Dirk Vermeir. Modeling protein interaction networks with answer set programming. In *BIBM*, pp. 99–104. 2009.

[Geb08] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental asp solver. In *ICLP*, pp. 190–205. 2008.

[Gel88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. pp. 1070–1080. MIT Press, 1988.

[Gel92] Michael Gelfond and Vladimir Lifschitz. Representing actions in extended logic programming. In *JICSLP*, pp. 559–573. 1992.

[Sch09] Torsten Schaub and Sven Thiele. Metabolic network expansion with answer set programming. In *ICLP*, pp. 312–326. 2009.

[Tra06] Nam Tran. *Reasoning and hypothesizing about signaling networks*. Ph.D. thesis, Arizona State University, 2006.