Set Graphs VI: Logic Programming and Bisimulation *

Agostino Dovier

University of Udine, DIMI

Abstract. We analyze the declarative encoding of the set-theoretic graph property known as *bisimulation.* This notion is of central importance in non-well founded set theory, semantics of concurrency, model checking, and coinductive reasoning. From a modeling point of view, it is particularly interesting since it allows two alternative high-level characterizations. We analyze the encoding style of these modelings in various dialects of Logic Programming. Moreover, the notion also admits a polynomial-time maximum fix point procedure that we implemented in Prolog. Similar graph problems which are NP hard or not yet perfectly classified (e.g., graph isomorphism) can benefit from the encodings presented.

1 Introduction

Graph bisimulation is the key notion for stating equality in non well-founded-set theory [1]. The notion is used extensively whenever cyclic properties need to be checked (e.g., in conductive reasoning [16]), in the semantics of communicating systems [12], as well as in minimizing graphs for hardware verification, and in model checking in general [9]. The problem of establishing whether two graphs are bisimilar (hence, the sets 'represented' by those graphs are equivalent) is easily shown to be equivalent to the problem of finding a maximum bisimulation of a graph into itself. This problem admits fast polynomial time algorithms that optimize a naive maximum fix point algorithm [14, 7]. As far as we know, the problem of establishing whether there exists or not a linear-time algorithm for the general case is still open.

The maximum bisimulation problem has the beauty of having two (equivalent) declarative formalizations. The first one is the definition of a particular *morhpism* that is similar to the one used for defining other "NP" properties such as graph/subgraph simulation or isomorphism. The second one is based on the notion of *coarsest stable partition* which is itself similar to the property exploited for computing the minimum deterministic finite automata for a given regular language. The focus of the paper is the analysis of the programming style to be used for modeling the maximum bisimulation problem in as much declarative way as possible in some dialects of logic programming, namely, Prolog, Constraint Logic Programming on Finite Domains, Answer Set Programming, the less known, but developed for coinductive reasoning, Co-inductive Logic Programming, and the

* This research is partially supported by INdAM-GNCS.

set-based constraint logic programming language {log} (read setlog). The contribution of this paper is not on the direction of improving existing polynomial time algorithms; however, we also encode in Prolog a polynomial-time max fixpoint algorithm.

The paper is inserted either in the series of papers on "Set Graphs" (e.g., [13]) or in the series of papers aimed at comparing relative expressiveness of logic programming paradigms on families of problems (e.g., [4, 20]). Proposed models can be slightly modified to address the other similar properties recalled above, some of which are not believed to admit a fast implementation and, therefore, they can exploit the declarative style of logic languages and the speed of their implementations, in particular, in the case of ASP modeling.

2 Sets, Graphs, and Bisimulation

We assume the reader has some basic notions of set theory and of first-order logic with equality. We add here a set of notions needed for understanding the contribution of the paper; the reader is referred, e.g., to [1, 11], for details. Basic knowledge of Logic Programming is also assumed.

Sets are made by elements. The *extensionality principle* (E) states that two sets are equal if and only if they contain the same elements:

$$\forall z\Big((z \in x \leftrightarrow z \in y) \rightarrow x = y\Big) \qquad (E)$$

(the $\leftarrow$, apparently missing, direction is a consequence of equality). In "classical" set theory sets are assumed to be well-founded; in particular the $\in$ relation fulfills the so-called *foundation axiom* (FA):

$$\forall x\left(x \neq \emptyset \rightarrow (\exists y \in x)(x \cap y = \emptyset)\right) \qquad (FA)$$

that ensures that a set cannot contain an infinite descending chain $x_0 \ni x_1 \ni x_2 \ni \cdots$ of elements. In particular, let us observe that a set x such that $x = \{x\}$ can not exist since x is not empty, its unique element y is x itself, and $x \cap y = \{y\} \neq \emptyset$ contradicting the axiom.

On the other side, cyclic phenomena are rather common in our experience. For instance in knowledge representation, argumentation theory, operating systems design, concurrency theory, and so on. Representing and reasoning on these problems lead us in working on (cyclic) directed graphs with a distinguished entry point. Precisely, an *accessible pointed graph (*`apg`*)* $\langle G, \nu\rangle$ is a directed graph $G = \langle N, E\rangle$ together with a distinguished node $\nu \in N$ (the *point*) such that all the nodes in N are reachable from ν.

Intuitively, an edge $a \longrightarrow b$ means that the set "represented by b" is an element of the set "represented by a". The graph edge $\longrightarrow$ stands, in a sense, for the Peano symbol $\ni$.[1] The above idea can be used to *decorate* an `apg`, namely,

[1] Let us observe the morphing $\longrightarrow \quad \rightarrow \quad \ni \quad \ni$, pointed out by Carla Piazza.

assigning a (possibly non-well founded) set to each of the nodes. *Sinks*, i.e., nodes without outgoing edges have no elements and are therefore decorated as the empty set $\emptyset$. In general, if the apg is acyclic, it represents a well-founded set and it can be decorated uniquely starting from sinks and proceeding backward to the *point* (theoretically, this follows from the *Mostowski's Collapsing Lemma* [11]). See Figure 1 for two examples; in particular observe that redundant nodes and edges can occur in a graph.

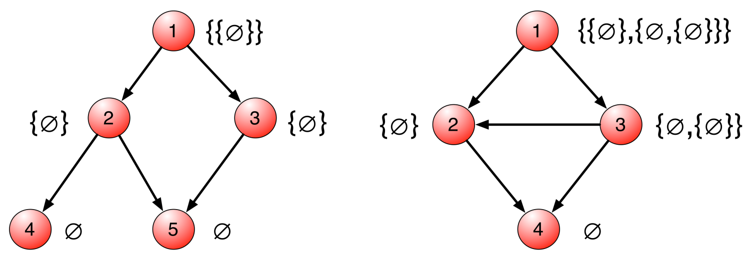

Fig. 1. Two acyclic pointed graphs and their decoration with well-founded sets

If the graph contains cycles, interpreting edges as membership implies that the set that decorates the graph is no longer well-founded. Non well-founded sets are often referred to as *hypersets*. *Anti Foundation Axiom (AFA)* [1] states that every apg has a unique decoration. Figure 2 reports some examples. In particular, the leftmost and the central apgs both represent the hyperset Ω which is the singleton set containing itself. Applying extensionality axiom (E) for verifying their equality would lead to a circular argument.

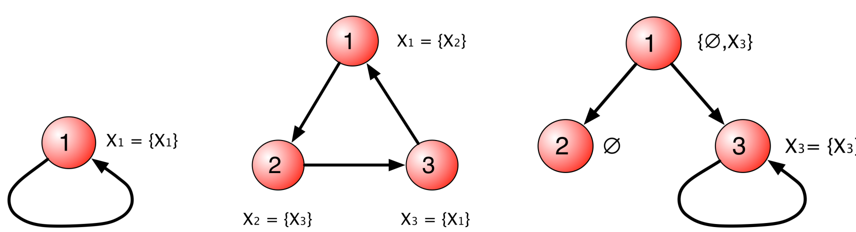

Fig. 2. Three cyclic pointed graphs and their decoration with hypersets

2.1 The notion of Bisimulation

Each apg has a unique decoration. Therefore two apgs denote the same hyperset if and only if their decoration is the same. The notion introduced to establish formally this fact is the notion of *bisimulation*.

Let $G_1 = \langle N_1, E_1 \rangle$ and $G_2 = \langle N_2, E_2 \rangle$ be two graphs, a *bisimulation* between G_1 and G_2 is a relation $b \subseteq N_1 \times N_2$ such that:

1. $u_1 \, b \, u_2 \wedge \langle u_1, v_1 \rangle \in E_1 \Rightarrow \exists v_2 \in N_2 (v_1 \, b \, v_2 \wedge \langle u_2, v_2 \rangle \in E_2)$
2. $u_1 \, b \, u_2 \wedge \langle u_2, v_2 \rangle \in E_2 \Rightarrow \exists v_1 \in N_1 (v_1 \, b \, v_2 \wedge \langle u_1, v_1 \rangle \in E_1)$.

In case G_1 and G_2 are apgs pointed in ν_1 and ν_2, respectively, it is also required that $\nu_1 \, b \, \nu_2$. If there is a bisimulation between G_1 and G_2 then the two graphs are bisimilar.

Remark 1 (Bisimulation and Isomorphism). Let us observe that if b is required to be a bijective function then it is a *graph isomorphism*. Establishing whether two graphs are isomorphic is an NP-problem neither proved to be NP-complete nor in P. Establishing whether G_1 is isomorphic to a subgraph of G_2 (subgraph isomorphism) is NP-complete [15]. Establishing whether G_1 is bisimilar to a subgraph of G_2 (subgraph bisimulation) is NP-complete [6]. Instead, establishing whether G_1 is bisimilar to G_2 is in P (actually, $O(|E_1 + E_2| \log |N_1 + N_2|)$—[14]).

In case G_1 and G_2 are the same graph $G = \langle N, E \rangle$, a *bisimulation on* G is a bisimulation between G and G. It is immediate to see that there is a bisimulation between two apg's $\langle G_1, \nu_1 \rangle$ and $\langle G_2, \nu_2 \rangle$ if and only if there is a bisimulation b on the graph $G = \langle \{\nu\} \cup N_1 \cup N_2, \{(\nu, \nu_1), (\nu, \nu_2)\} \cup E_1 \cup E_2 \rangle$ such that $\nu_1 \, b \, \nu_2$ (see, e.g., [7], for a proof). Therefore, we can focus on the bisimulations on a single graph; among them, we are interested in computing the *maximum bisimulation* (i.e., the one maximizing the number of pairs $u \, b \, v$). It can be shown that it is unique, that is an equivalence relation, and that contains all other bisimulations on G. Therefore, we might restrict our search to bisimulations on G that are *equivalence relations* on N such that:

$$\forall u_1, u_2, v_1 \in N \left(u_1 \, b \, u_2 \wedge \langle u_1, v_1 \rangle \in E \Rightarrow (\exists v_2 \in N)(v_1 \, b \, v_2 \wedge \langle u_2, v_2 \rangle \in E) \right) \quad (1)$$

The fact that we look for equivalence (hence, symmetric) relations makes the case 2 of the definition of bisimulation superfluous. We will use the following logical rewriting of (1) in some encodings:

$$\neg \exists u_1, u_2, v_1 \in N \left(u_1 b u_2 \wedge \langle u_1, v_1 \rangle \in E \wedge \neg \big((\exists v_2 \in N) \, (v_1 b v_2 \wedge \langle u_2, v_2 \rangle \in E) \big) \right) \quad (1')$$

The graph obtained by collapsing nodes according to the equivalence relation is the one that allows to obtain the apg decoration, using the following procedure.

> Let $G = \langle \langle N, E \rangle, \nu \rangle$ be an apg. For each node $i \in N$ assign uniquely a variable X_i, then add the equation $X_i = \{X_j : (i, j) \in E\}$. The set of equations obtained defines the set decorating G, that can be retrieved as the solution of X_ν.

Another characterization of the maximum bisimulation is based on the notion of *stability*. Given a set N, a partition P of N is a collection of non-empty disjoint sets (blocks) $B_1, B_2, \ldots$ such that $\bigcup_i B_i = N$. Let E be a relation on the set N, with E^{-1} we denote its inverse relation.

A partition P of N is said to be *stable* with respect to E if and only if

$$(\forall B_1 \in P)(\forall B_2 \in P)(B_1 \subseteq E^{-1}(B_2) \vee B_1 \cap E^{-1}(B_2) = \emptyset) \tag{2}$$

which is in turn equivalent to state that there do not exist two blocks $B_1 \in P$ and $B_2 \in P$ such that:

$$(\exists x \in B_1)(\exists y \in B_1)\,(x \in E^{-1}(B_2) \wedge y \notin E^{-1}(B_2)) \tag{2$'$}$$

We say that a partition P *refines* a partition Q if each block (i.e., class) of P is contained in a block of Q. A class B_2 of P *splits* a class B_1 of P if B_1 is replaced in P by $C_1 = B_1 \cap E^{-1}(B_2)$ and $C_2 = B_1 \setminus E^{-1}(B_2)$; if C_1 or C_2 is empty, it is not added in P. The split operation produces a refinement of a partition P; if P is stable with respect to E, no split operations changes P.

It can be shown that given a graph $G = \langle N, E \rangle$, starting from the partition $P = \{N\}$, after at most $|N| - 1$ split operations a procedure halts determining the *coarsest stable partition (CSP)* w.r.t. E. Namely, the partition is stable and any other stable partition is a refinement of it. Moreover, and this is relevant to our task, the CSP corresponds to the partition induced by the maximum bisimulation, hence this algorithm can be employed to compute it in polynomial time. Paige and Tarjan showed us the way for fast implementations in [14].

3 Logic Programming Encoding of Bisimulation

We first focus on the logic programming encoding or the definition of bisimulation (1) or (1$'$) and of the part needed to look for the maximum bisimulation on a input `apg`. We impose the relation is symmetric and reflexive. In the remaining part of the paper we assume that `apg`'s are represented by facts `node(1). node(2). node(3). ...` for enumerating the nodes, and facts `edge(u,v).` where `u` and `v` are nodes, for enumerating the edges. For the sake of simplicity, we also assume that node 1 is the point of the `apg`.[2]

3.1 Prolog

The programming style used in the Prolog encoding is generate & test. The core of the encoding is reported in Figure 3. A bisimulation is represented by a list of pairs of nodes (U, V). Assuming a "guessed" bisimulation is given as input, for every guessed pair the morphism property (1) is checked. As usual in Prolog, the "for all" property is implemented by a recursive predicate (although a slightly

[2] Complete codes are available at `http://clp.dimi.uniud.it`

more compact `foreach` statement is available in most Prolog systems and will be used in successive encodings).

`bis`/1 is called by a predicate that guesses a bisimulation of size at least k between nodes, itself called by a meta predicate that increases the value of k until no solution is found. The `guess` predicate forces all identities, all the pairs between nodes without outgoing edges, and imposes symmetries; this extra part of the code is rather boring and we have omitted the code. As a (weak) search strategy, the guess predicate tries first to insert as much pairs as possible: this will explain the difference of computational times on different benchmarks of the same size.

3.2 CLP(FD)

The programming style is constraint & generate. In this case the bisimulation is stored in matrix, say B, of Boolean variables. $B[i,j] = 1$ means that $i\ b\ j$ ($B[i,j] = 0$ means that $\neg(i\ b\ j)$). We omit the definitions of the `reflexivity` predicate that sets $B[i,i] = 1$ for all nodes i and of the `symmetry` predicate that sets $B[i,j] = B[j,i]$ for all pair of nodes i and j. Let us focus on the morphism requirements (1). `morphism`/2 collects all edges and calls `morphism`/3. This predicate scans each edge (U,V) and then each node $U1$ and adds the property that if $B[U,U1] = 1$ then $\sum_{(U1,V1)\in E} B[V,V1] = 1$. Let us observe that $O(|E||N|)$ of these constraints are generated. We omit the definitions of some auxiliary predicates, such as `access(X,Y,B,N,BXY)` that simply sets `BXY = B[X,Y]`. The whole encoding is longer and perhaps less intuitive than the Prolog one. However, the search of the *maximum* bisimulation is not delegated to a meta predicate as in Prolog but it is encoded directly into the `maximize` option of the labeling primitive. The "down" search strategy, trying to assign 1 first, is similar to the strategy used in the Prolog code.

3.3 ASP

ASP encodings allow to define explicitly the bisimulation relation. Two rules are added for forcing symmetry and reflexivity. Then a non-deterministic choice is added to each pair of nodes. The great declarative advantage of ASP in this case is the availability of constraint rules that allows to express universal quantification (negation of existential quantification). The morphism requirement ($1'$) can be therefore encoded as it is, with the unique addition of the `node` predicates needed for grounding (Figure 5). Then we define the notion of representative nodes (the nodes of smaller index among the nodes equivalent to it) and minimize the number of them. This has proven to be much more efficient that maximizing the size of `bis`. A final remark on the expected size of the grounding. Both the constraint and the definition of `one_son_bis` ranges over all edges and another free node: this generates a grounding of size $O(|E||N|)$.

```
bis(B) :-  bis(B,B).          % Recursively analyze B

bis([],_).
bis([ (U1,U2) |RB],B) :-      %%% if U1 bis U2
       successors(U1,SU1),    %%% Collect the successors SU1 of U1
       successors(U2,SU2),    %%% Collect the successors SU2 of U2
       allbis(SU1,SU2,B),     %%% Then recursively consider SU1
       bis(RB,B).

allbis([],_,_).
allbis([V1 | SU1],SU2,B) :-   %%% If V1 is a successor of U1
       member(V2,SU2),        %%% there is a V2 successor of U2
       member( (V1,V2),B),    %%% such that V1 bis V2
       allbis(SU1,SU2,B).

successors(X,SX) :- findall(Y,edge(X,Y),SX).
```

Fig. 3. Prolog encoding of the bisimulation definition. Maximization code is omitted.

```
bis :- size(N), M is N*N,              %%% Define the N * N Boolean
       length(B,M), domain(B,0,1),     %%% Matrix B
       constraint(B,N), Max #= sum(B), %%% Max is the number of pairs
       labeling([maximize(Max),ffc,down],B).  %%% in the bisimulation

constraint(B,N) :- reflexivity(N,B), symmetry(1,2,N,B), morphism(N,B).

morphism(N,B) :-
       findall( (X,Y),edge(X,Y),EDGES),
       foreach( E in EDGES,  U2 in 1..N, morphismcheck(E,U2,N,B)).

morphismcheck( (U1,V1),U2,N,B) :-
       access(U1,U2,B,N,BU1U2),         % Flag BU1U2 stands for (U1 B U2)
       successors(U2, SuccU2),          % Collect all edges (U2,V2)
       collectlist(SuccU2,V1,N,B,BLIST),% BLIST contains all possible flags BV1V2
       BU1U2  #=< sum(BLIST).           % If (U1 B U2) there is V2 s.t. (V1 B V2)
```

Fig. 4. Portion of the CLP(FD) encoding of the bisimulation definition

```
%% Reflexivity and Symmetry
bis(I,I)   :- node(I).
bis(I,J)   :- node(I;J), bis(J,I).
%%% Nondeterministic choice
{bis(I,J)} :- node(I;J).
%%% Morphism requirement (1')
:-  node(U1;U2;V1), bis(U1,U2), edge(U1,V1),  not one_son_bis(V1,U2).
one_son_bis(V1,U2) :-  node(V1;U2;V2), edge(U2,V2), bis(V1,V2).

%% Minimization  (max bisimulation)
non_rep_node(A) :- node(A), bis(A,B), B < A.
rep_node(A)     :- node(A),  not non_rep_node(A).
rep_nodes(N)    :- N=#sum[rep_node(A)].
#minimize [rep_nodes(N)=N].
```

Fig. 5. ASP encoding of the bisimulation definition

3.4 co-LP

In this section we exploit a less standard logic programming dialect. Coinductive Logic Programming (briefly co-LP) was introduced by Gupta et al. [17] and recently presented in a concise way in [2], where computability results and a working SWI interpreter are provided. The same syntax of pure Prolog should be used. The differences lay in the semantics: the maximum fix point of a conductive predicate is looked for, as opposite to the least fix point of classical logic programming. Although this can easily lead to a non recursively enumerable semantics, the finiteness of the graphs makes this option available for this problem. As a matter of fact, the piece of code reported in Figure 6 encodes the problem and, by looking for the maximum fix point, the maximum bisimulation is computed without the need of additional minimization/maximization directives. `bis` and `allbis` are declared as coinductive. The definition of `successors` is the same as in Figure 3 and declared as inductive, as well as the `member` predicate.

```
bis(U,V) :- successors(U,SU), successors(V,SV),
            allbis(SU,SV), allbis(SV,SU).
allbis([],_ ).
allbis([U|R],SV ) :-  member(V,SV), bis(U,V), allbis(R,SV).
```

Fig. 6. co-LP (complete) encoding of the definition of Bisimulation

4 Logic Programming Encoding of CSP

We focus first on the encoding of the definition of stable partition (2) and finally on the (less declarative) computation of the CSP.

4.1 Prolog

The programming style is generate & test. A partition is a list of non-empty lists of nodes (blocks). Sink nodes (if any) are deterministically set in the first block. Possible partitions of increasing size are non-deterministically generated until the first stable one is found. Once the partition is guessed the verify part is made by a double selection of blocks within the list of blocks. The main predicate that encodes property (2) is the following:

```
stablecond(B1,B2) :-  edgeinv(B2,InvB2),
       (subseteq(B1,InvB2)  ;  emptyintersection(B1,InvB2)).
```

where `edgeinv` collects the nodes that enter into `B2` (definable as `findall(X, (edge(X,Y), member(Y,B)), REVB)`) while the two set-theoretic predicates are defined through list operations.

4.2 CLP(FD)

In this case the data structure used is a mapping from nodes to blocks indexes, stored as a list of finite domain variables. The set inclusion and empty intersection requirement of (2) are not naturally implemented by a constraint & generate style. As in the encoding 3.2 maximization is forced by a parameter of the labeling; some symmetry breaking is encoded (e.g.. sink nodes are deterministically forced to stay in partition number one). We only report the excerpt of the encoding, where we made use of the `foreach` built-in. With a rough analysis, the number of constraints needed is $O(|N|^3)$ but each constraint generated by `alledge` can be of size $|N|$ itself.

4.3 ASP

Also in this case ASP allows a concise encoding (Figure 8). The assignment is implemented defining the function `inblock/2`. The possibility of reasoning "a posteriori" and the availability of the constraint rule allows to naturally encode the property ($2'$). The remaining part of the code is devoted to symmetry breaking and minimization of the number of blocks. The bottleneck for the grounding stage is the constraint rule that might generate $O(|N|^4)$ ground instantiations.

4.4 {log}

The CLP language {log}, originally presented in [5], populated with several set-based constraints such as the disjoint constraint (`disj`—imposing empty intersection) in [8] and later augmented with Finite Domain constraints in [3] is a set-based extension of Prolog (and a particular case of constraint logic programming language). Encoding the set-theoretic stable property (2) is rather natural in this case. We report the definition in Figure 9. `subset`, `disj`, `in` are built-in constraints. Similarly, restricted universal quantifiers (`forall(X in S, Goal)`) and intensional set formers ({`X : Goal(X)`}) are accepted.

4.5 Computing the coarsest stable partition

We have implemented the maximum fixpoint procedure for computing the coarsest stable partition in Prolog. Initially nodes are split into (at most) two classes: internal and non internal nodes. For each node U, a list of pairs U-I is computed by stating that U is assigned to block I. Then a possible splitter is found and, in case, a split is executed. The procedure terminates in at most $n-1$ steps where n is the number of nodes. The Prolog code is reported in Appendix (Figure 12).

5 Experiments

Although the focus of this work is on the expressivity of the declarative encoding (being this problem solved by fast algorithms in literature, such as [14,

```
stability(B,N) :-
    foreach( I in 1..N, J in 1..N, stability_cond(I,J,B,N)).

stability_cond(I,J,B,N) :-       % Blocks BI and BJ are considered
    inclusion(1,N,I,J,B, Cincl), % Nodes  in 1..N are analyzed
    emptyintersection(1,N,I,J,B,Cempty), % Cincl and Cempty are reified
    Cincl + Cempty #> 0.         % OR condition

inclusion(X,N,_,_,_, 1) :- X>N,!.
inclusion(X,N,I,J,B, Cout) :-  % Node X is considered
    alledges(X,B,J,Flags),     % Flags stores existence of edge (X,Y) with Y in BJ
    LocFlag #=  ((B[X] #= I) #=> (Flags #> 0)), %% Inclusion check:
    X1 is X+1,                 % If X in BI then X in E-1(BJ)
    inclusion(X1,N,I,J,B,Ctemp), % Recursive call
    Cout #= Ctemp*LocFlag.     % AND condition (forall nodes it should hold)

alledges(X,B,J,Flags) :-       % Collect the successors of X
    successors(X,OutgoingX),   % And use them for assigning the Flags var
    alledgesaux(OutgoingX,B,J,Flags).
alledgesaux([],_,_,0).
alledgesaux([Y|R],B,J,Flags) :- % The Flags variable is created
    alledgesaux(R,B,J,F1),      % Recursive call.
    Flags #= (B[Y] #= J) + F1.  % Add "1" iff there is edge (X,Y) and BY = J
```

Fig. 7. Excerpt of the CLP(FD) encoding of the stable partition property

```
blk(I)     :- node(I).
%%% Function assigning nodes to blocks
1{inblock(A,B):blk(B)}1 :- node(A).
%%% STABILITY (2')
:- blk(B1;B2), node(X;Y), X != Y, inblock(X,B1), inblock(Y,B1),
       connected(X,B2),   not connected(Y,B2).
connected(Y,B) :- edge(Y,Z),blk(B),inblock(Z,B).
%% Basic symmetry-breaking rules (optional)
:- node(A), internal(A), inblock(A,1).
internal(X) :- edge(X,Y).
leaf(X)     :-node(X), not internal(X).
non_empty_block(B) :- node(A), blk(B), inblock(A,B).
empty_block(B) :- blk(B), not non_empty_block(B).
:- blk(B1;B2), 1 < B1, B1 < B2, empty_block(B1), non_empty_block(B2).
%% Minimization
number_blocks(N)  :- N=#sum[non_empty_block(B)].
#minimize [number_blocks(N)=N].
```

Fig. 8. ASP complete encoding of the stable partition property

```
stable(P) :-
       forall(B1 in P, forall(B2 in P, stablecond(B1,B2) ) ).
stablecond(B1,B2) :-
       edgeinv(B2,InvB2) &
       (subset(B1,InvB2) or  disj(B1,InvB2)).
edgeinv(A,B) :-
       B = {X : exists(Y,(Y in A & edge(X,Y)))}.
```

Fig. 9. {log} encoding of the stable partition property

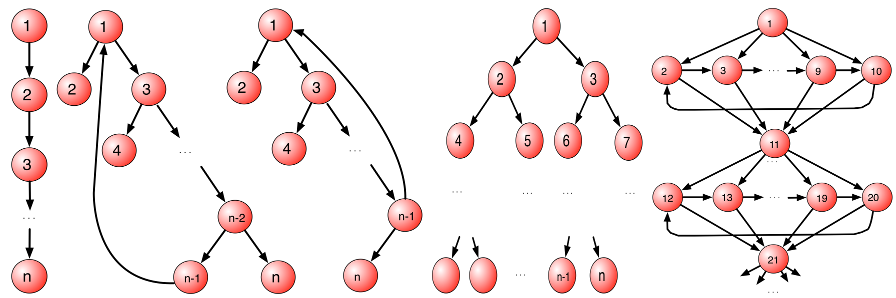

Fig. 10. From left to right, the graphs G_1, G_2 (n odd), G_2 (n even), G_3, and G_5 used in the experiments. G_4 is the complete graph (not reported).

7]), we have reported the excerpt of the running times of the various proposed encodings on some families of graphs, parametric on their number of nodes (Figure 10). Results can give us some additional information on the possibilities and on the intrinsic limits of the logic programming dialects analyzed. All experiments are made on a laptop 2.4GHz Intel Core i7, 8GB memory 1600MHz DDR3, OSX 10.9.2. Systems used are B-Prolog Version 7.8#5 [19], clingo 3.0.5 (clasp 1.3.10) [10], and SWI Prolog Version 6.4.1 [18]. In particular, SWI Prolog has been used in the `co-LP` tests, thanks to its rational terms handling. On the other Prolog encodings B-Prolog proved to be from 2 to 3 times faster than SWI and it has been therefore used. Speed-up increased still further using tabling for the predicate `edge` but we have exploited this additional feature in the experiments in Table 5 only. We tested the codes on five families of graphs G_1–G_5 parametric on the number of nodes n (see Figure 10).

- Graph G_1 is an acyclic graph with $n-1$ edges, where n classes are needed.
- G_2 is a cyclic graph with n nodes and edges. If n is even, just two classes are sufficient; if n is odd, $\frac{n+1}{2}$ classes are needed. This is why in some experiments we have two columns with this family of graphs.
- G_3 is a binary tree populated following a breadth-first visit, with $n-1$ edges.
- G_4 is, in a sense, symmetrical w.r.t. G_1: it is a complete graph with n^2 edges but just one class is sufficient.
- G_5 is a multilevel (cyclic) graph.

The results on the encoding of the bisimulation definition 1 are reported in Tables 1–3. From a quick view one might notice that the ASP encoding is a clear winner. Prolog generate & Test and the `co-LP` interpreter run in reasonable time on very small graphs only (Prolog is used without tabling, tabling the `edge` predicate allows a speed-up of roughly 4 times). The CLP approach becomes unpractical soon in the case for the complete graph G_4 where the n^2 edges generate too many constraints for the stack size when $n \geq 50$ (as reported in Section 3.2, $O(|E||N|)$ constraints are added: in this case they are $O(n^5)$; moreover each of those constraints includes a sum of n elements in this case). Let us observe that the complete graph G_4 produces the highest grounding times in the ASP case. As reported in Section 3.3 grounding size is expected $O(|E||N|) = O(n^3)$ in this case. This has been verified experimentally (table not reported); in particular, for $G_4, n = 200$ the grounded file (obtained with the option `-t`) is of size 275MB. Moreover, by a simple regression analysis of Table 3, the time needed for grounding is shown to be proportional to n^6 for graph G_4.

The results on the enconding of the coarsest stable partition definition 2 are reported in Table 4. Also in this case ASP is a clear winner, although in this case smaller graphs can be handled by all approaches. We have omitted the {log} running times. This system proved to be definitely the slowest; just to have an idea, for G_1, $n = 5$ the computation took roughly 5 hours.

We conclude with the testing of the encoding of the polynomial time procedure of coarsest stable partition computation by maximum fixpoint and splits. In graphs G_2^* and G_3 tabling the edge predicate improved the running time of two orders of magnitude (reported times are those using tabling). As a further consideration, we started finding stack overflow for $n = 5000$. Moreover, the experimental complexity detected by a regression analysis on table 5 is $O(|N|^3)$ in all columns, which is rather good, considering the purely declarative nature of the encoding (fast solvers such as [7] run in $O(|N|)$ in acyclic graphs such as G_1 and G_3, and in the cyclic multi-level graph G_5, while they run in $O(|E| \log |N|) = O(|N|^2 \log |N|)$ in the other cases. By the way, complete graph G_4 could be solved in time $O(1)$ with a simple preprocessing).

6 Conclusions

We have encoded the two properties characterizing the bisimulation definition, and in particular, solving the maximum bisimulation problem, using some dialects of Logic Programming. As a general remark, the guess & verify style of Prolog (and of ASP) allows to define the characterizing properties to be verified 'a posteriori', on ground atoms. In CLP instead, those properties are added as constraints to lists of values that are currently non instantiated and this makes things much more involved, and has a negative impact on code readability. The expressive power of the constraint rule of ASP allows a natural and compact encoding of "for all" properties and this improved the conciseness of the encoding (and readability in general); recursion should be used instead for it in Prolog and CLP. co-LP (resp., {log}) allows to write excellent code for property (1) (resp., property (2)). However, since they are implemented using meta interpreters (naive in the case of co-LP) their execution times are prohibitive for being used in practice.

The ASP encoding is also the winner from the efficiency point of view, as far as a purely declarative encoding of the NP property is concerned. This would suggest the reader that this is the best dialect to be used to encode graph properties if a polynomial time algorithm is not yet available (or it does not exist at all). This is not the case of the maximum bisimulation problem where polynomial time algorithms for computing the coarsest stable partition can be

n	G_1		G_2		G_3		G_4	
	BP	co-LP	BP	co-LP	BP	co-LP	BP	co-LP
4	1	2	1	3	1	1	0	45382
5	18	3	16	6	14	47	1	so
6	508	8	179	33	441	496	1	so
7	33252	14	8240	20	6340	91272	1	so
8	4303576	28	203191	118	884614	47	8	so

Table 1. Property (1). Running time (ms) for the Prolog (BP) and the co-LP encoding on very small graphs. so=stack overflow. G_5 is not considered for these values of n, since its structure requires at least 11 nodes. Let us observe that G_3 is a tree of height 3 for $n = 7$ and of height 4 for $n = 8$. This explain the strange behavior of co-LP.

n	G_1		G_2		G_3		G_4		G_5^*	
	C	S	C	S	C	S	C	S	C	S
10	1	0	1	0	1	0	17	1	3	0
20	5	0	7	1	6	0	529	14	21	0
30	22	0	31	3	29	2	6001	423	69	0
40	59	0	64	6	68	5	33751	3574	202	1
50	147	0	141	12	141	8	so		438	2
60	240	0	277	38	259	18	so		896	2
70	428	0	492	41	460	34	so		1662	2
80	705	0	810	61	762	58	so		2756	3
90	1119	0	1463	99	1179	98	so		4441	4
100	1703	0	1913	158	1803	143	so		6798	4

Table 2. Property (1). Running time (ms) for clp(fd) on small graphs (C=constraint, S=search). so=stack overflow. For G_5 nodes are $n + 1$

n	G_1		G_2		G_3		G_4		G_5^*	
	G	S	G	S	G	S	G	S	G	S
100	670	50	650	100	620	110	998	140	250	20
110	880	80	830	170	820	170	15010	170	330	30
120	1180	100	1090	170	1090	210	23810	240	390	30
130	1510	130	1340	300	1440	280	30860	370	470	30
140	1890	150	1670	290	1690	360	43710	370	560	50
150	2270	180	2030	430	2120	210	55330	460	650	50
160	2740	230	2510	440	2590	260	74030	570	780	50
170	3310	240	2960	760	3120	330	99200	650	860	70
180	3930	280	3520	690	3600	350	123440	810	960	80
190	4600	320	4090	1130	4250	400	151550	950	1110	90
200	5350	350	4910	1140	4850	440	195790	1080	1240	110

Table 3. Property (1). Running time (ms) for clingo on medium graphs (G=grounding+preprocessing, S=search). For G_5 nodes are $n + 1$

n	G_1			G_2			G_3			G_4		
	Prolog and CLP(FD)											
n	BP	C	S	BP	C	S	BP	C	S	BP	C	S
5	3	11	4	2	13	2	0	13	1	0	129	10
6	26	31	58	0	38	1	0	32	5	0	230	8
7	351	78	146	15	79	12	0	79	5	0	372	19
8	5057	115	1257	1	232	8	3	119	11	0	803	47
9	76044	145	29507	196	384	197	5	151	33	0	1895	104
10	1338632	179	222047	1	198	5	8	355	139	0	5352	143
	Clingo											
	G		S	G		S	G		S	G		S
10	20		990	10		0	10		0	20		0
11	20		9980	20		40	10		0	40		0
12	40		103700	0		0	20		0	60		0
13	90		1077220	50		370	30		0	90		0
14	110		12714900	50		0	70		0	120		0

Table 4. Property (2). Running time (ms) for the Prolog (BP) and CLP(FD) (C=constraints, S=search) and ASP (G=grounding+preprocessing, S=search) encodings on very small graphs.

n	G_1	G_2^*	G_2	G_3	G_4	G_5^*
100	24	38	22	15	40	34
200	137	208	87	54	253	244
400	885	968	324	204	1659	1458
600	2769	2804	697	463	5613	4553
800	6544	6169	1365	809	12895	10735
1000	12639	11650	2145	1210	24462	20069

Table 5. Running time (ms) of the B-Prolog encoding of the *fixpoint procedure* for computing the Coarsest Stable Partition on large graphs. $*$ indicates that in those columns the number of nodes is $n + 1$.

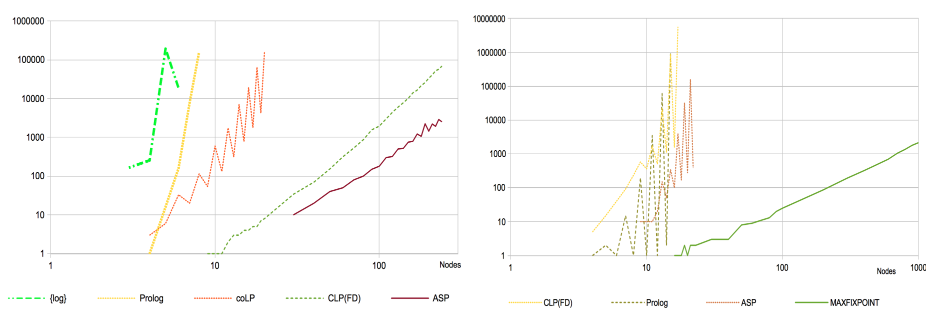

Fig. 11. An overall picture on the computational results on graph G_2. Encoding (1)—left, encoding (2)—right. Logarithmic scales for axis have been used.

employed. The one implemented in Prolog and reported in Appendix proved also to be the fastest approach presented in this paper.

References

1. Aczel, P. *Non-well-founded sets.* CSLI Lecture Notes, 14. Stanford University, Center for the Study of Language and Information, 1988.
2. Ancona, D., and Dovier, A. co-LP: Back to the Roots. *TPLP 13*, 4-5-Online-Supplement (2013).
3. Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. Integrating finite domain constraints and clp with sets. In *PPDP* (2003), ACM, pp. 219–229.
4. Dovier, A., Formisano, A., and Pontelli, E. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *J. Exp. Theor. Artif. Intell. 21*, 2 (2009), 79–121.
5. Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi, G. {log}: A Logic Programming Language with Finite Sets. In *Proc of ICLP* (1991), K. Furukawa, Ed., The MIT Press, pp. 111–124.
6. Dovier, A., and Piazza, C. The subgraph bisimulation problem. *IEEE Trans. Knowl. Data Eng. 15*, 4 (2003), 1055–1056.
7. Dovier, A., Piazza, C., and Policriti, A. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science 311*, 1-3 (2004), 221–256.
8. Dovier, A., Piazza, C., Pontelli, E., and Rossi, G. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst. 22*, 5 (2000), 861–931.
9. Fisler, K., and Vardi, M. Y. Bisimulation and model checking. In *CHARME* (1999), L. Pierre and T. Kropf, Eds., vol. 1703 of *Lecture Notes in Computer Science*, Springer, pp. 338–341.
10. Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. *clasp* : A conflict-driven answer set solver. In *LPNMR* (2007), C. Baral, G. Brewka, and J. S. Schlipf, Eds., vol. 4483 of *Lecture Notes in Computer Science*, Springer, pp. 260–265.
11. Kunen, K. *Set Theory.* North Holland, 1980.
12. Milner, R. *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer, 1980.
13. Omodeo, E. G., and Tomescu, A. I. Set Graphs. III. Proof Pearl: Claw-Free Graphs Mirrored into Transitive Hereditarily Finite Sets. *J. Autom. Reasoning 52*, 1 (2014), 1–29.
14. Paige, R., and Tarjan, R. E. Three partition refinement algorithms. *SIAM J. Comput. 16*, 6 (1987), 973–989.
15. Papadimitriou, C. H. *Computational complexity.* Academic Internet Publ., 2007.
16. Sangiorgi, D. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst. 31*, 4 (2009).
17. Simon, L., Mallya, A., Bansal, A., and Gupta, G. Coinductive logic programming. In *ICLP* (2006), S. Etalle and M. Truszczynski, Eds., vol. 4079 of *Lecture Notes in Computer Science*, Springer, pp. 330–345.
18. Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. SWI-Prolog. *TPLP 12*, 1-2 (2012), 67–96.
19. Zhou, N.-F. The language features and architecture of b-prolog. *TPLP 12*, 1-2 (2012), 189–218.
20. Zhou, N.-F., and Dovier, A. A tabled prolog program for solving sokoban. *Fundam. Inform. 124*, 4 (2013), 561–575.

Appendix

```
stable_comp(Final, Nclasses) :-
       findall(X,node(X),Nodes),
       initialize(Nodes, Initial),
       maxfixpoint(Initial, 2, Final, Nclasses). % start with "2"
%%% maxfixpoint procedure. If possible, split, else stop.
maxfixpoint(AssIn, I, AssOut, C) :-
       split(I,AssIn,AssMid),!,
       I1 is I+1,
       maxfixpoint(AssMid, I1, AssOut, C).
%%% When stop, simply compute the number of classes used
maxfixpoint(Stable,C,Stable,C1) :-
       count_classes(C,Stable,C1).
%%% Split operation.
%%% First locate a block that can be split. Then find the splitter
split(MaxBlock,AssIn,AssMid) :-
        between(1,MaxBlock,I),
        findall(X,member(X-I,AssIn),BI),
        BI = [_, _ | _], %% BI might be split (not empty, not singleton)
        %%% Find potential splitters BJ (and remove duplicates)
        findall(Q,(member(V-Q,AssIn),edge(W,V),member(W,BI)),SP),
        sort(SP,SPS), member(J,SPS),
        findall(Z,(member(Y-J,AssIn),edge(Z,Y)),BJinv),
        my_delete(BI,BJinv,[D|ELTA]), %%% The difference is computed when not empty
        MaxBlock1 is MaxBlock + 1,
        update(AssIn,AssMid,MaxBlock1,[D|ELTA]).

%% Initial partition: Sinks -> B1; Internal -> B2
initialize([],[]).
initialize([A|R], [A-B|Ass]) :- (internal(A), !, B=2; B=1), initialize(R,Ass).

%%% AUXILIARY
count_classes(C,Stable,C1) :-  (C > 3, !, C1 = C;
        C =< 2, member(_-1,Stable),member(_-2,Stable),!,C1=2; C1 = 1).

my_delete([],_,[]).
my_delete([A|R],DEL,S) :- select(A,DEL,DEL1),!, my_delete(R,DEL1,S).
my_delete([A|R],DEL,[A|S]) :- my_delete(R,DEL,S).

update([],[],_,_).
update([X-_|R],[X-I|S],I,D) :- select(X,D,D1),!, update(R,S,I,D1).
update([X-J|R],[X-J|S],I,D) :- update(R,S,I,D).

internal(X) :- edge(X,_).
```

Fig. 12. Prolog computation of the CSP as a maxfixpoint procedure (complete code)