Logic programming and bisimulation

Logic Programming and Bisimulation∗

Agostino Dovier
University of Udine, DIMI, ITALY
(*e-mail:* agostino.dovier@uniud.it)

submitted 29 April 2015; accepted 5 June 2015

Abstract

The logic programming encoding of the set-theoretic graph property known as *bisimulation* is analyzed. 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 fixpoint procedure that we implemented in Prolog. Similar graph problems which are instead NP hard or not yet perfectly classified (e.g., graph isomorphism) can inherit most from the declarative encodings presented.

KEYWORDS: Logic Programming modeling, Bisimulation

1 Introduction

Graph bisimulation is the key notion for stating equality in non-well-founded-set theory (Aczel 1988). The notion is used extensively whenever cyclic properties need to be checked either theoretically (e.g., in coinductive reasoning (Sangiorgi 2009)) and in the semantics of communicating systems (Milner 1980), or practically, in minimizing graphs for hardware verification, and in model checking in general (Fisler and Vardi 1999). Some recent practical applications of bisimulation are concerned with contracting graphs for optimal planning (Nissim et al. 2011), preventing side-channel leaks in Web Traffic (Backes et al. 2013) and, in general, with reasoning on semistructured data (Fletcher et al. 2015), with airplane turning control systems (Bae et al. 2015), and with phylogenetics, where lumpability (bisimulation on Markov chains) is used (Vera-Ruiz et al. 2014).

The problem of establishing whether two graphs are bisimilar (hence, the sets 'represented' by those graphs are equivalent) admits fast polynomial time algorithms that optimize a naive maximum fixpoint algorithm (Paige and Tarjan 1987; Dovier et al. 2004) (the problem of establishing whether there exists a linear-time algorithm for the general case is still open) and it is equivalent to the problem of finding the maximum bisimulation of a graph into itself. The latter problem has the beauty of having two (equivalent) declarative formalizations. The first one is the definition of a particular *morphism* that is similar to the

∗ The work is partially supported by INdAM GNCS 2014 and 2015 projects.

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 a way as declarative way as possible in some dialects of logic programming, namely, Prolog, Constraint Logic Programming on Finite Domains, Answer Set Programming, Co-inductive Logic Programming, and the set-based constraint logic programming language $\{\log\}$ (read setlog). Although the contribution of this paper is not on the direction of improving existing polynomial time algorithms, the problem is also encoded in Prolog as a (polynomial-time) maximum fixpoint algorithm.

The paper contributes to the series of papers on "Set Graphs" (e.g., (Omodeo and Tomescu 2014)) and to the series of papers aimed at comparing relative expressiveness of logic programming paradigms on families of problems (e.g., (Dovier et al. 2009; Zhou and Dovier 2013)). The models proposed in this paper (available at (Dovier 2015)) can be slightly modified to address modifications of the problem, like detecting similarity/isomorphism equivalence of graphs, possibly with further restrictions on edge labels or on topological properties, that typically do not admit polynomial-time implementations. Therefore, a programmer can exploit for them the declarative programming style of logic languages and the speed of their implementations.

A preliminary version of this work appeared in (Dovier 2014).

2 Sets, Graphs, and Bisimulation

Familiarity with basic notions of set theory, first-order logic, and Logic Programming is assumed. We introduce some notions needed for understanding the contribution of the paper; the reader is referred, e.g., to (Aczel 1988) or (Kunen 1980), for details.

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 $x = \{x\}$ cannot 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 hand, cyclic phenomena are rather common in our experience. For instance, in Knowledge Representation, a cyclic argument is introduced for modeling the *frame problem* in presence of static causal laws (Gelfond and Lifschitz 1998). In Argumentation Theory (Dung 1995), cyclic graphs are introduced to model attacks between opposing

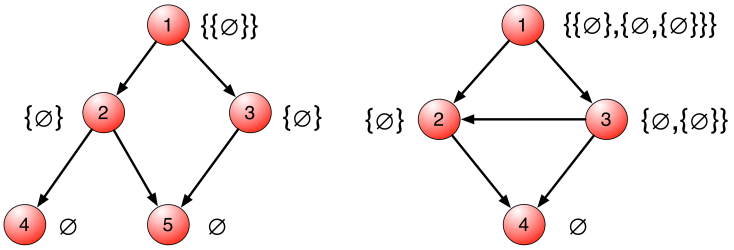


Fig. 1. Two acyclic pointed graphs and their decoration with well-founded sets

entities. Several other examples come from operating systems design, concurrency theory, and so on.

Representing and reasoning on these problems leads us to (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 (Aczel 1988).

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 $\ni$. The above idea is used to *decorate* an apg, namely, assigning a (possibly non-well-founded) set to each of the nodes (Aczel 1988); the set represented by the apg is the decoration of the apg's point. *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* (Kunen 1980)—that works for well-founded sets only). See Figure 1 for two examples; in particular observe that redundant nodes and edges can occur in a graph.

If the graph contains cycles, interpreting edges as membership implies that the set which decorates the graph is no longer well-founded. Non-well-founded sets are often referred to as *hypersets*. To allow their existence, axiom (FA) is removed from the theory and it is replaced by the *Anti Foundation Axiom (AFA)* (Aczel 1988) that 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. For instance, let us try to show, using (E) that the set decorating the leftmost apg of Figure 2 is equal to that decorating the center one. Denoting with ^{1} and ^{2} the nodes of the former and the latter set, respectively, using (E) we would have that $X_1^1 = X_1^2$ iff $X_1^1 = X_2^2$ iff $X_1^1 = X_3^2$ iff $X_1^1 = X_1^2$.

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:

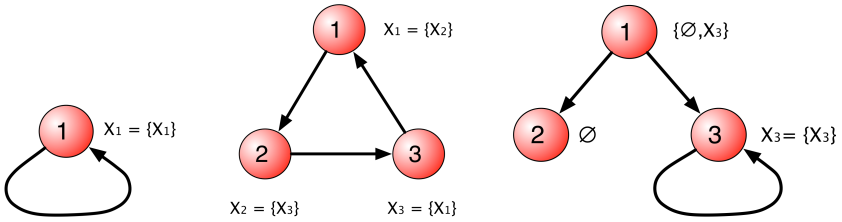


Fig. 2. Three cyclic pointed graphs and their decoration with hypersets

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 said to be *bisimilar*.

Let us consider again the leftmost and center apgs of Figure 2. It is immediate to check that the relation $\{(X_1^1, X_1^2), (X_1^1, X_2^2), (X_1^1, X_3^2)\}$ is a bisimulation between them.

Remark 2.1 (*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 (Cook 1971). Establishing whether G_1 is bisimilar to a subgraph of G_2 (subgraph bisimulation) is NP-complete (Dovier and Piazza 2003). Instead, establishing whether G_1 is bisimilar to G_2 is in P (actually, $O(|E_1 + E_2| \log |N_1 + N_2|)$— (Paige and Tarjan 1987)).

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., (Dovier et al. 2004) 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, it is an equivalence relation, and it contains all other bisimulations on G. Therefore, we may restrict our search to bisimulations on G that are *equivalence relations* on N such that:

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

Looking 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:

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

The graph obtained by collapsing nodes according to the equivalence relation is the one that allows us 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 $\bigcup_i B_i = N$. If the number of blocks is k, we say that the partition has size k. Let E be a binary relation on the set N, with $E^{-1} = \{(b,a) : (a,b) \in E\}$ we denote its inverse relation. If $B \subseteq N$, we define $E^{-1}(B) = \{a \in N : (a,b) \in E, b \in B\}$.

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 stating that there do not exist two blocks $B_1 \in P$ and $B_2 \in P$ such that:

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

A partition P *refines* a partition Q if each block of P is contained in a block of Q. A block B_2 of P *splits* a block B_1 of P if B_1 is replaced in P by $C_1 = B_1 \cap E^{-1}(B_2)$ and $C_2 = B_1 \setminus E^{-1}(B_2)$ and both of them are not empty. In this case B_2 is called a *splitter* (operationally, the split operation can be made even if one of them is empty; in this case it is not added in P and B_1 is replaced by itself, leading to a fix point). The split operation produces a refinement of a partition P; if P is stable with respect to E, no split operation 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 purposes, 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 1987).

3 Logic Programming Encoding of Bisimulation

We first focus on the logic programming declarative encoding of the definition of bisimulation (1) or $(1')$ and of the part needed for the maximum bisimulation on an input apg. We impose that 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). ...` 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.[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

[1] Complete codes are available in (Dovier 2015).

the morphism property (1) is checked. As usual in Prolog, the "for all" property is implemented by a recursive predicate (although a slightly 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 its code has been omitted. 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.

CLP(FD). The programming style is constraint & generate. In this case the bisimulation is stored in a 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 pairs of nodes i and j. Let us focus on the morphism requirements (1). `morphism`/2 collects all edges and nodes and calls `morphismcheck`/4 (Figure 4). This predicate scans each edge $(U1, V1)$ and each node $U2$ and adds the property that if $B[U1,U2] = 1$ then $\sum_{(U2,V2)\in E} B[V1,V2] \geq 1$. 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 firstly, is similar to the strategy used in the Prolog code.

ASP. ASP encodings allow us 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 allow us 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 node of smallest index among those 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` range over all edges and another free node: this generates a grounding of size $O(|E||N|)$.

co-LP. In this section we exploit a less standard logic programming dialect. Coinductive Logic Programming (briefly co-LP) was introduced by (Simon et al. 2006) and presented in a concise way in (Ancona and Dovier 2013; Ancona and Dovier 2015), where computability results and a simple operational semantics are provided. The difference with respect to classical logic programming lays in the semantics: the maximum fixpoint of a coinductive predicate is looked for, as opposite to the least fixpoint. Although this leads to a non-recursively-enumerable semantics, the finiteness of the graphs makes this option

```
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. Excerpt 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;B), bis(A,B), B < A.
rep_node(A)     :- node(A),  not non_rep_node(A).
numbernodes(N) :- N = #count{A : rep_node(A)}.
#minimize {N : numbernodes(N)}.
```

Fig. 5. ASP encoding of the bisimulation definition

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 fixpoint, 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. Complete co-LP 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.

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.

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 requirements of (2) are not naturally implemented by a constraint & generate style. As in the previous CLP(FD) encoding 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). In Figure 7 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.

ASP. Also in this case ASP allows a concise encoding (Figure 8). The assignment is implemented defining the predicate (used as function) `inblock/2`. The possibility of reasoning "a posteriori" and the availability of the constraint rule allow us 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.

```
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
non_empty(N) :- N = #count{A:non_empty_block(A)}.
#minimize { N: non_empty(N)}.
```

Fig. 8. Complete ASP 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

{log}. The CLP language {log}, originally presented in (Dovier et al. 1991), populated with several set-based constraints such as the disjoint constraint (`disj`—imposing empty intersection) in (Dovier et al. 2000) and later augmented with Finite Domain constraints in (Dal Palù et al. 2003), 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 (see Figure 9). `subset` ($\subseteq$), `disj` ($\cap = \emptyset$), `in` ($\in$) are built-in constraints. Similarly, restricted universal quantifiers (`forall(X in S, Goal)`) (i.e., $\forall$`X` (`X` $\in$ `S` $\rightarrow$ `Goal`) and intensional set formers ({`X : Goal(X)`}) are accepted.

4.1 Computing the coarsest stable partition

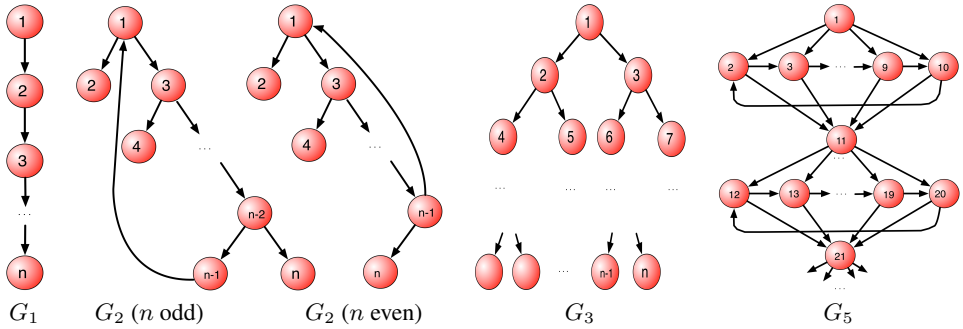
We have implemented the maximum fixpoint procedure for computing the coarsest stable partition in Prolog. Initially nodes are split into (at most) two blocks: internal and not internal nodes (i.e., sinks). For each node U, a list of pairs U-I is computed by stating that U is assigned to block I. Then a possible splitter is found and, in case, a split is executed. The procedure terminates in at most $n-1$ splitting steps where n is the number of nodes. The Prolog code is available in (Dovier 2015); we refer to it as MAXFIXPOINT in Figure 11.

5 Experiments

Although the focus of this work is on the expressivity of the declarative encoding (this problem is solved by fast algorithms in literature, such as (Paige and Tarjan 1987; Dovier et al. 2004)), 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 give us some additional information on the possibilities and on the intrinsic limits of the logic programming dialects analyzed. All experiments were performed on a laptop 2.4GHz Intel Core i7, 8GB memory 1600MHz DDR3, OSX 10.10.3. The systems used are B-Prolog Version 7.8#5 (Zhou 2012), clingo 4.4.0 (Gebser et al. 2007), and SWI Prolog Version 6.4.1 (Wielemaker et al. 2012). In particular, SWI Prolog was used in the co-LP tests, thanks to its rational terms handling. On the other Prolog encodings, B-Prolog proved to be 2 to 3 times faster than SWI and it has been therefore used. Speed-up increased still further using tabling for the predicate `edge`. We tested the encodings 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 blocks are needed.
- G_2 is a cyclic graph with n nodes and edges. If n is even, just two blocks are sufficient; if n is odd, $\frac{n+1}{2}$ blocks 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 block is sufficient.
- G_5 is a multilevel (cyclic) graph.

In Figure 11 the results for the graph G_2 (which has been selected as a good representative of the overall situation; e.g., both grounding and search times are sensible



G_1 G_2 (n odd) G_2 (n even) G_3 G_5

Fig. 10. The graphs G_1, G_2, G_3, and G_5. G_4 is the complete n-nodes graph (not drawn).

in this family of graphs) are graphically summarized. Running times include constraint-generation/grounding time and search time. As far as the declarative, non procedural encodings presented, the ASP encoding is a clear winner, while, as one might expect, the Prolog implementation of the polynomial time algorithm is still better.

All details for the five families of graphs are reported in the tables in (Dovier 2015).

As anticipated, 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 of the complete graph G_4 where the n^2 edges generate too many constraints for the stack size when $n \geq 50$ ($O(|E||N|)$ constraints are added: in this case they are $O(n^5)$; moreover each of those constraints includes a sum of n elements). Let us observe that the complete graph G_4 produces the highest grounding times in the ASP case. 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 our results, the time needed for grounding is shown to be proportional to n^6 for graph G_4.

With the enconding of the coarsest stable partition definition (2) the graphs that can be handled by all approaches are smaller, and ASP is still a clear winner. 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 testing 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 of the results is $O(|N|^3)$ in all columns, which is rather good, considering the purely declarative nature of the encoding (fast solvers such as (Dovier et al. 2004) 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.

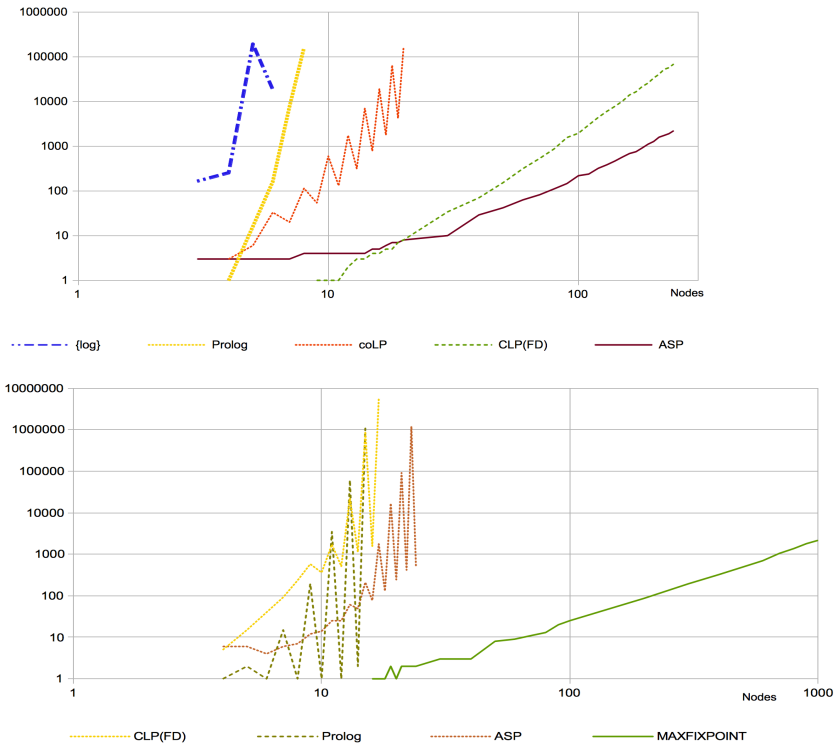


Fig. 11. An overall picture on the computational results on graph G_2. Encoding (1)—top, encoding (2)—bottom. Logarithmic scales for axes have been used. y-axis reports runtimes in ms.

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 us 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 improves the conciseness of the encoding (and readability in general); recursion must be used instead to it in Prolog and CLP. co-LP (resp., $\{\log\}$) allows us to write excellent code for property (1) (resp., property (2)). However, since they are implemented adopting 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 a NP property on graphs is concerned. This would suggest to 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 for the maximum bisimulation problem where polynomial time algorithms for computing the coarsest stable partition can be employed. The one implemented in Prolog (available in (Dovier 2015)) proved also to be the fastest approach presented in this paper.

References

ACZEL, P. 1988. *Non-well-founded sets*. CSLI Lecture Notes, 14. Stanford University, Center for the Study of Language and Information.

ANCONA, D. AND DOVIER, A. 2015. A theoretical perspective of coinductive logic programming. *Fundamenta Informaticae*. To appear.

ANCONA, D. AND DOVIER, A. 2013. co-LP: Back to the Roots. TC at ICLP. *TPLP 13,* 4-5-Online-Supplement.

BACKES, M., DOYCHEV, G., AND KÖPF, B. 2013. Preventing side-channel leaks in web traffic: A formal approach. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society.

BAE, K., KRISILOFF, J., MESEGUER, J., AND ÖLVECZKY, P. C. 2015. Designing and verifying distributed cyber-physical systems using multirate PALS: an airplane turning control system case study. *Sci. Comput. Program. 103*, 13–50.

COOK, S. A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, M. A. Harrison, R. B. Banerji, and J. D. Ullman, Eds. ACM, 151–158.

DAL PALÙ, A., DOVIER, A., PONTELLI, E., AND ROSSI, G. 2003. Integrating finite domain constraints and CLP with sets. In *PPDP*. ACM, 219–229.

DOVIER, A. 2015. Supplemental material on this paper. University of Udine, CLP LAB. `http://clp.dimi.uniud.it/cobis/`

DOVIER, A. 2014. Set graphs VI: logic programming and bisimulation. In *Proceedings of the 29th Italian Conference on Computational Logic, Torino, Italy, June 16-18, 2014.*, L. Giordano, V. Gliozzi, and G. L. Pozzato, Eds. CEUR Workshop Proceedings, vol. 1195. CEUR-WS.org, 14–29.

DOVIER, A., FORMISANO, A., AND PONTELLI, E. 2009. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *J. Exp. Theor. Artif. Intell. 21,* 2, 79–121.

DOVIER, A., OMODEO, E. G., PONTELLI, E., AND ROSSI, G. 1991. {log}: A Logic Programming Language with Finite Sets. In *Proc of ICLP*, K. Furukawa, Ed. The MIT Press, 111–124.

DOVIER, A. AND PIAZZA, C. 2003. The subgraph bisimulation problem. *IEEE Trans. Knowl. Data Eng. 15,* 4, 1055–1056.

DOVIER, A., PIAZZA, C., AND POLICRITI, A. 2004. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science 311,* 1-3, 221–256.

DOVIER, A., PIAZZA, C., PONTELLI, E., AND ROSSI, G. 2000. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst. 22,* 5, 861–931.

DUNG, P. M. 1995. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell. 77,* 2, 321–358.

FISLER, K. AND VARDI, M. Y. 1999. Bisimulation and model checking. In *CHARME*, L. Pierre and T. Kropf, Eds. Lecture Notes in Computer Science, vol. 1703. Springer, 338–341.

FLETCHER, G. H. L., GYSSENS, M., LEINDERS, D., SURINX, D., DEN BUSSCHE, J. V., GUCHT, D. V., VANSUMMEREN, S., AND WU, Y. 2015. Relative expressive power of navigational querying on graphs. *Inf. Sci. 298*, 390–406.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. *clasp* : A conflict-driven answer set solver. In *LPNMR*, C. Baral, G. Brewka, and J. S. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer, 260–265.

GELFOND, M. AND LIFSCHITZ, V. 1998. Action languages. *Electron. Trans. Artif. Intell. 2*, 193–210.

KUNEN, K. 1980. *Set Theory*. North Holland.

MILNER, R. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, vol. 92. Springer.

NISSIM, R., HOFFMANN, J., AND HELMERT, M. 2011. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, T. Walsh, Ed. IJCAI/AAAI, 1983–1990.

OMODEO, E. G. AND TOMESCU, A. I. 2014. Set Graphs. III. Proof Pearl: Claw-Free Graphs Mirrored into Transitive Hereditarily Finite Sets. *J. Autom. Reasoning 52,* 1, 1–29.

PAIGE, R. AND TARJAN, R. E. 1987. Three partition refinement algorithms. *SIAM J. Comput. 16,* 6, 973–989.

SANGIORGI, D. 2009. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst. 31,* 4(15), 1–41.

SIMON, L., MALLYA, A., BANSAL, A., AND GUPTA, G. 2006. Coinductive logic programming. In *ICLP*, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, 330–345.

VERA-RUIZ, V. A., LAU, K. W., ROBINSON, J., AND JERMIIN, L. S. 2014. Statistical tests to identify appropriate types of nucleotide sequence recoding in molecular phylogenetics. *BMC Bioinformatics 15,* 2.

WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. 2012. SWI-Prolog. *TPLP 12,* 1-2, 67–96.

ZHOU, N.-F. 2012. The language features and architecture of B-prolog. *TPLP 12,* 1-2, 189–218.

ZHOU, N.-F. AND DOVIER, A. 2013. A tabled prolog program for solving sokoban. *Fundam. Inform. 124,* 4, 561–575.