SUPPLEMENTARY MATERIAL: EXPERIMENTAL DESING FOR DISCRIMINATION OF LOGICAL INPUT-OUPUT BEHAVIORS IN CANCER PATHWAYS

SANTIAGO VIDELA[1,2,3], CARITO GUZIOLOWSKI[4], JULIO SAEZ-RODRIGUEZ[5], TORSTEN SCHAUB[3], ANNE SIEGEL[1,2,*]

[1]*CNRS, UMR 6074 IRISA, Campus de Beaulieu, 35042 Rennes, France,* [2]*INRIA, Dyliss project, Campus de Beaulieu, 35042 Rennes, France,* [3]*Universität Potsdam, Institut für Informatik, D-14482 Potsdam, Germany,* [4]*École Centrale de Nantes, IRCCyN UMR CNRS 6597, 44321, Nantes, France,* [5]*European Molecular Biology Laboratory, European Bioinformatics Institute, Hinxton CB10 1SD, UK,* E-mail: anne.siegel@irisa.fr*

1. Answer Set Programming at a glance

Our encodings are written in the input language of *gringo* 4 series. Such a language implements most of the so-called *ASP-Core-2* standard.* In what follows, we introduce its basic syntax and we refer the reader to the available documentation for more details. An *atom* is a predicate symbol followed by a sequence of terms (e.g. `p(a,b)`,`q(X,f(a,b))`). A *term* is a constant (e.g. `c`, `42`) or a function symbol followed by a sequence of terms (e.g. `f(a,b)`, `g(X,10)`) where uppercase letters denote first-order variables. Then, a rule is of the form

$$h \texttt{:-}\ b_1, \ldots, b_n.$$

where h (head) is an atom and any b_j (body) is a literal of the form a or `not` a for an atom a where the connective `not` corresponds to default negation. The connectives `:-` and `,` can be read as *if* and *and*, respectively. Furthermore, a rule without body is a *fact*, whereas a rule without head is an *integrity constraint*. A logic program consists of a set of rules, each of which is terminated by a period. An atom preceded with default negation, `not`, is satisfied unless the atom is found to be true. In ASP, the semantics of a logic program is given by the *stable models semantics* [3]. Intuitively, the head of a rule has to be true whenever all its body literals are true. This semantics requires that each true atom must also have some derivation, that is, an atom cannot be true if there is no rule deriving it. This implies that only atoms appearing in some head can appear in answer sets.

We end this quick introduction by three language constructs particularly interesting for our encodings. First, the so called *choice rule* of the form,

$$\{h_1; \ldots; h_m\} \texttt{:-}\ b_1, \ldots, b_n.$$

allows us to express choices over subsets of atoms. Any subset of its head atoms can be included in an answer set, provided the body literals are satisfied. Note that using a choice rule one can easily *generate* an exponential search space of candidate solutions. Second, a conditional literal is of the form

$$l : l_1, \ldots, l_n$$

*`http://www.mat.unical.it/aspcomp2013/ASPStandardization`

The purpose of this language construct is to govern the instantiation of the literal l through the literals $l_1, \ldots, l_n$. In this respect, the conditional literal above can be regarded as the list of elements in the set $\{l \mid l_1, \ldots, l_n\}$. Finally, for solving (multi-criteria) optimization problems, ASP allows for expressing (multiple) cost functions in terms of a weighted sum of elements subject to minimization and/or maximization. Such objective functions are expressed in *gringo* 4 in terms of (several) optimization statements of the form

$$\texttt{\#opt}\{w_1@l_1, t_{1_1}, \ldots, t_{m_1} : b_1, \ldots, b_{n_1}; \ldots; w_k@l_k, t_{1_k}, \ldots, t_{m_k} : b_{1_k}, \ldots, b_{n_k}\}.$$

where *#opt* $\in$ {"*#minimize*", "*#maximize*"}, $w_i, l_i, t_{1_i}, \ldots, t_{m_i}$ are terms and $b_{1_i}, \ldots, b_{n_i}$ are literals for $k \geq 0, 1 \leq i \leq k, m_i \geq 0$ and $n_i \geq 0$. Furthermore, w_i and l_i stand for an integer *weight* and priority *level*. Priorities allow for representing lexicographically ordered optimization objectives, greater levels being more significant than smaller ones.

2. Problem instance

Let $\mathcal{B}$ be a finite set of input-output behaviors represented by logical networks $(V, \phi_1), \ldots, (V, \phi_n)$. We represent the variables V as facts over the predicate `variable/1`, namely `variable(`v`)` for all $v \in V$. Recall that we assume $\phi_j(v)$ to be in disjunctive normal form for all $v \in V$ and $j = 1, \ldots, n$. Hence, $\phi_j(v)$ is a set of clauses and a clause a set of literals. We represent formulas using predicates `formula/3`, `dnf/2`, and `clause/3`. The facts `formula(`j`,`v`,`$s_{\phi_j(v)}$`)` map variables $v \in V$ to their corresponding formulas $\phi_j(v)$. Facts over predicates `stimulus`/1, `inhibitor`/1, and `readout`/1 denote nodes in V_S, V_K, and V_R respectively. Finally, we consider two constants, namely, `maxstimuli=`s and `maxinhibitors=`k, in order to represent the search space of experimental conditions $\mathcal{C}(s, k)$. By default, we assign such constants to $|V_S|$ and $|V_K|$, respectively. Afterwards, at grounding time, we can use command-line options `-c maxstimuli=`s `-c maxinhibitors=`k in order to overwrite these values with arbitrary s and k.

Below we show the instance representation for our toy example. That is, the three input-output behaviors represented by the Boolean logic models $(V, \phi_1), (V, \phi_2)$, and (V, ϕ_3) shown in Figure 1.

```
variable("f").  variable("e").  variable("a").  variable("g").
variable("b").  variable("c").  variable("d").

formula(1,"d",1). formula(1,"e",0). formula(1,"f",3). formula(1,"g",2).
formula(2,"d",4). formula(2,"e",0). formula(2,"f",3). formula(2,"g",2).
formula(3,"d",5). formula(3,"e",0). formula(3,"f",3). formula(3,"g",2).

dnf(0,2). dnf(0,0). dnf(1,7). dnf(1,8). dnf(2,14).
dnf(3,16).  dnf(4,0). dnf(4,7). dnf(5,8). dnf(4,8).

clause(0,"b",1).  clause(2,"c",1).  clause(7,"c",-1). clause(8,"a",1).
clause(14,"e",1). clause(14,"c",-1).  clause(16,"d",1). clause(16,"e",1).

stimulus("a").  stimulus("c").  stimulus("b").
readout("g"). readout("f").   inhibitor("d").

#const maxstimuli = 3.
#const maxinhibitors = 1.
```

3. Problem encoding

Next we describe our encoding relying on *incremental* ASP [1, 2], for finding an optimal experimental design as described in the previous section. The idea is to consider one problem instance after another by gradually increasing the number of experimental conditions such that, if the program is satisfiable at the step ε, then there exists an ε-tuple $(C_1, \ldots, C_\varepsilon)$ satisfying (1). Our ASP encoding is shown below.

```
1  #include <iclingo>.
2  #const imax = 20.
3
4  model(M) :- formula(M,_,_).
5
6  #program cumulative(k).
7
8  {clamped(k,V, 1) : clause(_,V,_), stimulus(V) } maxstimuli.
9  {clamped(k,V,-1) : clause(_,V,_), inhibitor(V)} maxinhibitors.
10  clamped(k,V,-1) :- stimulus(V); not clamped(k,V,1).
11
12 clamped(k,V)  :- clamped(k,V,_).
13 free(k,M,V,I) :- formula(M,V,I); not clamped(k,V).
14
15 eval(k,M,V, S) :- clamped(k,V,S); model(M).
16 eval(k,M,V, 1) :- free(k,M,V,I); eval(k,M,W,T) : clause(J,W,T); dnf(I,J).
17 eval(k,M,V,-1) :- not eval(k,M,V,1); model(M); variable(V).
18
19 diff(k,M1,M2)   :- diff(k,M1,M2,_).
20 diff(k,M1,M2,V) :- eval(k,M1,V,S); eval(k,M2,V,-S);
21                     M1 < M2; readout(V); model(M1;M2).
22
23 #minimize{1@1,clamped,k,V    : clamped(k,V,-1), inhibitor(V)}.
24 #minimize{1@2,clamped,k,V    : clamped(k,V, 1),  stimulus(V)}.
25 #maximize{1@3,diff,k,M1,M2,V : diff(k,M1,M2,V)}.
26
27 #program volatile(k).
28 #external query(k).
29
30 :- not diff(K,M1,M2) : K=1..k; model(M1;M2); M1<M2; query(k).
31
32 #show clamped/3.
33 #show diff/4.
```

Preliminaries. Line 1 declares the required inclusion in order to use incremental solving embedded in *clingo* 4. Importantly, the remainder of the encoding must follow a specific structure in order to work properly. More precisely, we need to define two "subprograms", namely, `cumulative(k)`, and `volatile(k)`. To this end, we use the directive `#program` with a name, e.g., `cumulative`, and an optional list of parameters, e.g., `(k)`. Each subprogram comprise all rules up to the next such directive (or the end of file) and their grounding is controlled via the embedded script. Line 2 declares the constant `imax` which is used as the maximum number of incremental steps to be considered. Recall that, the number of incre-

mental steps represents the number of experimental conditions. Thus, by default, we assign it to 20 but we can easily overwrite this value afterwards using the command-line option `-c imax=`e. Next, Line 4 defines auxiliary domain predicates `model/1`, namely, `model(`j`)` for all Boolean logic model $(V, \phi_j) \in \mathcal{B}$.

Experimental conditions, fixpoints, differences, and optimization. Lines 6-25 define the subprogram `cumulative(k)`. The purpose of this subprogram, is to declare all logic rules which have to be grounded at every incremental step `k`. The representation of clamping assignments is straightforward. Note that we use 1 and -1 for truth assignments to $\boldsymbol{t}$ and $\boldsymbol{f}$, respectively. Let C_i be a clamping assignment, we represent the assignments in C_i as facts over the predicate `clamped/3`, namely `clamped(`i`,`v`,`s`)` with $s = 1$ if $C_i(v) = \boldsymbol{t}$ and $s = -1$ if $C_i(v) = \boldsymbol{f}$. Thus, Lines 8-10 define rules in order to guess the experimental condition at step `k`. Note the usage of constants `maxstimuli` and `maxinhibitors` as upper bounds in the corresponding choice rules. Also, we restrict the choices to experimental conditions clamping a variable $v \in V_S \cup V_K$ only if v occurs in some clause. Otherwise, clamping the variable v does not make any "downstream" difference and hence, it is not useful in order to discriminate models. For instance, following our toy example, the experimental conditions $C_1 = \{b \mapsto \boldsymbol{t}\}$ and $C_2 = \{b \mapsto \boldsymbol{t}, c \mapsto \boldsymbol{t}\}$ are represented by predicates `clamped(1,b,1)`, `clamped(2,b,1)`, and `clamped(2,c,1)`.[†] Line 12 simply projects clamped variables regardless of their value. In Line 13 we use the predicate `free/4` to represent the fact that the value of every variable not clamped in the experiment k, is subject to the corresponding mapping in the logical network. Next, in Lines 15-17 we introduce the predicate `eval/4` to describe the evaluation of each variable with respect to every experiment and logical network. To be more precise, the response for each logical network under each experimental condition is represented over predicates `eval/4`, namely, `eval(`i`,`j`,`v`,`s`)` for a logical network (V, ϕ_j) and experimental condition C_i, if the variable v is assigned to s, i.e. $F_i^j(v) = s$. Let us illustrate this with our example. The corresponding predicates `eval/4` describing the response over variables f and g as in Table 1 are:

```
eval(1,1,"f", 1) eval(1,1,"g",1) eval(2,1,"f",-1) eval(2,1,"g",-1)
eval(1,2,"f", 1) eval(1,2,"g",1) eval(2,2,"f", 1) eval(2,2,"g",-1)
eval(1,3,"f",-1) eval(1,3,"g",1) eval(2,3,"f",-1) eval(2,3,"g",-1)
```

Then, in Lines 19-21 we derive predicates `diff/3` and `diff/4` in order to represent differences among every pair of logical input-output behavior for the experimental conditions under consideration. That is, we derive `diff(`i`,`j`,`j'`,`v`)` provided that for some models $(V, \phi_j), (V, \phi_{j'})$ with $j < j'$, we have that $F_i^j(v) \neq F_i^{j'}(v)$, i.e., `eval(`$i$`,`$j$`,`$v$`,`$s$`)` and `eval(`$i$`,`$j'$`,`$v$`,`$-s$`)`. For our example, we derive predicates `diff(1,1,3,"f")`, `diff(1,2,3,"f")`, `diff(2,1,2,"f")`, and `diff(2,2,3,"f")`. Hence, predicates `diff/3`, namely, `diff(`i`,`j`,`j'`)` indicate the existence of at least one (output) difference between (V, ϕ_j) and $(V, \phi_{j'})$ under experimental condition C_i. Finally, Lines 23-25 declare the three optimization criteria in lexicographic order. Notably,

[†]We note that permutations, e.g., `clamped(2,b,1)`, `clamped(1,b,1)`, and `clamped(1,c,1)`, are allowed in our encoding, but one can encode additional symmetry-breaking constraints in order to avoid them.

when grounding and solving `cumulative(k)` for successive values of `k`, the solver's objective functions are gradually extended accordingly. Lines 23-24 declare the minimization of functions Θ_{V_S} and Θ_{V_K}, as defined in (4). Notice the corresponding priority levels, namely `@1` and `@2`, meaning that as defined in (5), first we minimize Θ_{V_S}, and then with lower priority Θ_{V_K}. As discussed earlier, this is a rather arbitrary decision. However, in practice it can be revisited very easily thanks to the declarativeness of ASP. Finally, Line 25 declares with the greatest priority, namely, `@3`, the maximization of the function Θ_{diff} as defined in (2).

Discriminating among every pair of input-output behaviors. Lines 27-33 define the subprogram `volatile(k)`. The purpose of this subprogram, is to declare logic rules specific for each value of `k`. Typically, in the form of integrity constraints forcing the incremental solving to continue until such constraints are satisfied at a given step. Line 28 declares the external atoms `query(k)` for every incremental step `k`. Internally, the incremental solving embedded in *clingo* 4 assigns these atoms to either true or false in order to indicate the current step. Then, in Line 30 we define an integrity constraint in order to eliminate answer sets describing an ε-tuple $(C_1, \ldots, C_\varepsilon)$ such that (1) does not hold. In fact, the body of the integrity constraint is precisely the negation of the expression given in (1). Therefore, the incremental solving continues until its reach either the maximum number of steps `imax`, or the step ε such that there exists an ε-tuple satisfying (1).

4. Solving

Next we show the optimum answer set for the toy instance described above. Such an answer set represents the experimental conditions $C_1 = \{b \mapsto \boldsymbol{t}\}$ and $C_2 = \{b \mapsto \boldsymbol{t}, c \mapsto \boldsymbol{t}\}$. Notice that, we can see the number of steps in the incremental solving by looking at the number of calls (`Calls: 2` or `Solving...` printed twice). Furthermore, the value for each optimization criterion is given in the corresponding order (`Optimization 8 3 0`). Actually, the grounder performs an internal transformation for maximize statements and hence, the reported value (8) for the first optimization criterion is not exactly as one would expect. In order to recover the optimum for Θ_{diff}, we need to count the number of predicates `diff/4` in the answer set. Thus, in this case, the optimum value for Θ_{diff} is 4. On the other hand, minimization statements are treated without any transformation. Therefore, the optimum value for Θ_{V_S} is 3, whereas the optimum value for Θ_{V_K} is 0, as reported directly by the solver.

```
$ clingo design.lp toy.lp --quiet=1
clingo version 4.3.1
Reading from design.lp ...
Solving...
Solving...
Answer: 1
clamped(1,"a",-1) clamped(1,"c",-1) clamped(1,"b",1)\
clamped(2,"a",-1) clamped(2,"c",1)  clamped(2,"b",1)\
diff(1,1,3,"f") diff(1,2,3,"f") diff(2,1,2,"f") diff(2,2,3,"f")
Optimization: 8 3 0
OPTIMUM FOUND

Models       : 1
```

```
  Optimum     : yes
Optimization : 8 3 0
Calls        : 2
Time         : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.010s
```

References

[1] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer-Verlag, 2008.

[2] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[3] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988.