Answer Set Programming with Templates

Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, and Maria Carmela Santoro

Mathematics Dept., Università della Calabria,
Via Pietro Bucci, 30B
87036 Rende (CS), Italy

Abstract. This work[1] aims at introducing a new form of code reusability in Answer Set Programming languages. It is shown how ASP can be extended with 'template' predicate's definitions by introducing a well-suited form of second order logics, and an unfolding semantics. A primary feature of the language is the possibility to quickly introduce and prototype new constructs, and new data structure primitives extending ASP languages. We present language syntax and give its operational semantics. We show that the theory supporting our ASP extension is sound, and that expressiveness of ASP is preserved. Examples show how the extended ASP language greatly increases declarativity, readability, compactness of program encodings and code reusability.

1 Introduction

Answer Set Programming (ASP, in the following) research produced several, mature, implemented systems featuring clear semantics and efficient program evaluation [10, 11, 24, 27, 1, 7, 23, 26, 6]. Answer set programming has recently found a number of promising applications: several tasks in information integration require complex reasoning capabilities, which are explored in the INFOMIX project (funded by the European Commission, project IST-2002-33570)[19]. Another EC-funded project, ICONS [18] (IST-2001-32429), employs a DLP system as intelligent query engine for knowledge management. The Polish company Rodan Systems S.A. integrates a DLP system in a tool for the detection of price manipulations and unauthorized uses of confidential information, which is used by the Polish securities and exchange commission. ASP solvers are used also for decision support in the Space Shuttle [25], for product and software configuration tasks [28, 29], for model checking applications [16], and more.
Such engineering applications of answer set programming often require the introduction of very repetitive pieces of standard code. Indeed, a major need of complex and huge ASP applications such as [13, 25] is dealing efficiently with large pieces of such code. Furthermore, the non-monotonic reasoning community has continuosly produced, in the past, several extensions of nonmonotonic

[1] This work has been partially funded by the EU project IST 2001-37004 (WASP), IST 2001-32426 (ICONS) and IST 2001-33570 (INFOMIX).

logic languages, aimed at improving readability and easy programming through the introduction of new constructs, employed, e.g., in order to specify standard constraints, search spaces, data structures, new forms of reasoning, new special predicates [2, 9, 20], like, for instance, aggregate predicates [4].
The language DLP^T we propose here has two purposes. First of all, DLP^T moves the ASP field towards industrial applications, where code reusability is a crucial issue. Second, DLP^T aims at minimizing develop time in ASP system prototyping. ASP Systems' developers wishing to introduce new constructs are made able to fast prototype their languages, make their language features quickly available to the scientific community, and successively concentrate on efficient (and long lasting) implementations. Therefore, it is necessary a sound specification language for ASP extensions. This kind of language should fulfill two important properties: first, it should be highly declarative, and second, its expressiveness should not be higher than ASP itself. ASP *itself* proves to fit very well for this purpose.
To these ends, the DLP^T language introduces a form of second order logic programming, intended in order to ease the job of extending Answer Set Programming by means of new constructs. The proposed framework introduces the concept of 'template' predicate, whose definition can be instantiated whenever needed through binding to usual predicates.
Template predicates can be seen as a way to define intensional predicates by means of a subprogram, where the subprogram is generic and reusable as many times as necessary. This eases coding and improves readability and compactness of ASP programs. For instance, the following template definition

```
template max[p(1)](1)

{
    exceeded(X) :- p(X),p(Y), Y > X.
    max(X) :- p(X), not exceeded(X).
}
```

introduces a generic template program, defining the predicate `max`, intended to compute the maximum value over the domain of a given unary predicate `p`. A template definition may be invoked as many times as necessary, through *template atoms*, like in the following program

```
:- max[weight(*)](M), M > 100.
:- max[student(Sex,$,*)](M), M > 25
```

Template definitions may be unified with a template atom in many ways. The above program contains a plain invocation (`max[weight(*)](M)`), and a compound invocation (`max[student(Sex,$,*)](M)`). The latter allows to employ `max` definition on a ternary predicate, discarding the second attribute of `student`, and grouping values on the first attribute.
The semantics of the language is introduced through a suitable algorithm (the **Explode** algorithm) which is able to produce, from a set of template definitions and a DLP^T program, an equivalent ASP program.
There are some important theoretical questions to be addressed, such as the termination of the **Explode** algorithm, and the expressiveness of the DLP^T

language. It is a desirable property of the language to keep the same expressive power and complexity of DLP. Indeed, we prove that it is guaranteed that DLP^T program encodings are as efficient as plain DLP encodings, since unfolded programs are just polynomially larger with respect to the originating program. Benefits introduced by means of the proposed language can be resumed in the following points:

- Improved declarativity and succinctness of the code;
- Code reusability and possibility to collect templates within libraries;
- Capability to quickly introduce new, predefined constructs;
- Rapid language prototyping: a language designer is made able to quickly implement his framework, and then implement a final, efficient version, in a second moment.

In sum, the main contributions of this work are, therefore:

- introducing the DLP^T language, featuring 'template' predicate definitions.
- introducing the DLP^T semantics, based on a suitable transformation algorithm.
- studying DLP^T theoretical properties, such as termination of the transformation algorithm, and expressiveness;
- addressing knowledge representation issues, showing how predicate templates may be employed to easily introduce new constructs in ASP languages.
- providing an implementation of the DLP^T language, based on a suitable ASP solver.

The paper is structured as follows: the next section briefly gives syntax and semantics of ASP and syntax of the language DLP^T. Features of DLP^T are then illustrated by examples in section 3. Section 4 formally introduces the semantics of DLP^T. Theoretical properties of DLP^T are discussed in section 5. In section 6 we describe architecture and usage of the implemented system. Eventually, in section 7, conclusions are drawn.

2 Syntax of the DLP^T language

We give a quick definition of the syntax and informal semantics of DLP [2] programs. We assume the reader to be familiar with basic notions regarding ASP semantics. A thorough definition of concepts herein adopted can be found in [8]. A *(*DLP*) rule* r is a construct

$$a_1 \ \lor \ \cdots \ \lor \ a_n :- b_1, \cdots, b_k, \ \texttt{not} \ \ b_{k+1}, \cdots, \ \texttt{not} \ \ b_m.$$

where $a_1, \cdots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are literals, and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \ \lor \ \cdots \ \lor \ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k$, $\texttt{not} \ \ b_{k+1}, ..., \texttt{not} \ \ b_m$ is the *body* of r. A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule.* A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint* (or *strong constraint*).

[2] Disjunctive Logic Programming. With a slight loss of precision, from now on we will employ DLP as a synonym for ASP.

A DLP program is a set of DLP rules. The semantics of a DLP program is introduced through the Gelfond-Lifschitz transform as defined in [22]. Given a DLP program P, we denote $M(P)$ the set of stable models of P computed according to the Gelfond-Lifschitz transform.
A DLP^T program is a DLP program, where rules and constraints may contain (possibly negated) *template atoms*. Definition of template atoms is provided in the following of this section.

Definition 1. A template definition consists of two parts:

– A template header,

$$\#\textbf{template}\ t[p_1(a_1)\,,\,\ldots,\,p_n(a_n)](a_{n+1})$$

where a_i are nonnegative integer values, and $p_1, \ldots, p_n$ are predicate names, said in the following *formal predicates*. t is the *template name*.
– An associated DLP^T subprogram; at least a rule having t within the head must be declared; t may be used within the subprogram as predicate name of atoms of arity a_{n+1}, whereas each predicate $p_i (1 \leq i \leq n)$ may be used within atoms of arity a_i.

For instance, the following is a valid template definition:

```
template subset[p(1)](1)
{
   subset(X) v -subset(X) :- p(X).
}
```

Definition 2. A template atom is of the form:

$$t[p_1(\mathbf{X}_1)\,,\,\ldots\,,\,p_n(\mathbf{X}_n)](\mathbf{A})$$

where $p_1, \ldots, p_n$ are predicate names, and t is a template name. Each $\mathbf{X}_i (1 \leq i \leq n)$ is a list of terms (referred in the following as *compound* list of terms). A compound list of terms can contain either a variable name, a constant name, a dollar '\$' (from now on, *projection term*) or a '*' (from now on, *parameter term*). Variable and constants are called *standard* terms. Each $p_i(\mathbf{X}_i)(1 \leq i \leq n)$ is called *compound* or *actual* atom. $\mathbf{A}$ is a list of standard terms.
For instance, `max[color($,*)](N)` and `c[e(X,Y,$),f(*)](A,B)` are template atoms.

3 DLP^T by examples

In this section we show by examples the main advantages of template programming. Examples put in evidence the easiness of providing a succinct and elegant way for quickly introducing new constructs using the DLP^T language.

3.1 Aggregates

Aggregate predicates allow to represent properties over sets of elements. There are several applications where aggregates problems encoding provide to the end user an easier way to write programs with accuracy. Aggregate or similar special predicates have been already built in several ASP solvers [4, 27]: the next example shows how to fast prototype aggregate semantics without taking into account of the efficiency of a built-in implementation.
We recall here how to define a template which computes the maximum over the set of values of a given unary predicate p, using the definition:

```
template max[p(1)](1)
{
    max(X) :- p(X), not overcome(X).
    overcome(X) :- p(X),p(Y),Y > X.
}
```

To show how max can be fruitfully employed, the next template defines a general program able to count values of a given predicate p, provided it given an order relation succ defined on the domain of p.

```
template count[p(1),succ(2)](1)
{
 partialCount(0,0).
 partialCount(I,V) :- not p(Y), succ(Y,I), partialCount(Y,V).
 partialCount(I,V2):- p(Y), succ(Y,I), partialCount(Y,V), V2 = V+1.
 count(M) :- max[partialCount($,*)](M).
}
```

It is worth noting how max is employed over the binary predicate partialCount, instead of an unary one. Indeed, the '$' and '*' symbols are employed to project out the first argument of partialCount.

3.2 Defining ad hoc search spaces

Template definitions may be employed to define at once the most common search spaces, improving readability and succinctness of the resulting encoding. The next two examples show how to define a predicate subset and a predicate permutation, ranging, respectively, over subsets and permutations of the domain of a given predicate p. Such kind of constructs enriching plain Datalog languages have been proposed, for instance, in [15, 2].

```
template subset[p(1)](1)
{
    subset(X) v -subset(X) :- p(X).
}

template permutation[p(1)](2).
{
  permutation(X,N) v -permutation(X,N) :- p(X),#int(N), count[p(*),>(*,*)](N1), N <= N1.
  :- permutation(X,A),permutation(Z,A), Z <> X.
  :- permutation(X,A),permutation(X,B), A <> B.
  covered(X) :- permutation(X,A).
  :- p(X), not covered(X).
}
```

Next we show how `count` and `subset` may be employed to succinctly encode the *k-clique* problem [14], that is, to find if there exists in a graph G a complete subgraph containing at least k nodes. We assume a graph G is encoded through the predicates `node` and `edge`.

```
in(X) :- subset[node(*)](X).
:- count[in(*),>(*,*)](K), K < k.
:- in(X),in(Y), X <> Y, not edge(X,Y).
```

As for the `permutation` template, it can be employed, for instance, in this encoding of the Hamiltonian Path problem (given a graph G, find a path touching each node exactly once).

```
path(X,N) :- permutation[node(*)](X,N).
:- path(X,M), path(Y,N), not edge(X,Y), M = N+1.
```

3.3 Dealing with complex data structures

DLP^T can be fruitfully employed in order to manage with complex data structures, such as, for instance, sets, dates, trees, etc.

Sets. Extending Datalog with Set programming is another matter of interest for the ASP field. This topic has been already discussed (e.g. in [20, 21]), proposing some formalisms aiming at introducing a suitable semantics with sets. It is fairly quick to introduce set primitives using DLP^T; a set S is modeled through the domain of a given unary predicate s. Intuitive constructs like `intersection`, `union`, or `symmetricdifference`, may be modeled as follows.

```
#template intersection[a(1),b(1)](1).
{
   intersection (X) :- a(X),b(X).
}
#template union[a(1),b(1)](1).
{
   union(X) :- a(X).
   union(X) :- b(X).
}
#template difference[a(1),b(1)](1)
{
   difference(X) :- a(X), not b(X).
}
#template symmetricdifference[a(1),b(1)](1)
{
   symmetricdifference(X) :- union[a(*),b(*)](X),
                             not intersection[a(*),b(*)](X).
}
```

Dates. Managing data types dealing with actual values of time is another important matter in engineering applications of DLP. For instance, in [17], it is very important to reason on compound records containing date values. The following example template shows how to introduce a reusable construct dealing with dates stored using three different attributes of a given relation (an usual ⟨day, month, year⟩ 3-tuple).

```
#template before[date1(3),date2(3)](6)
{
    before(D,M,Y,D1,M1,Y1)  :- date1(D,M,Y), date2(D1,M1,Y1), Y < Y1.
    before(D,M,Y,D1,M1,Y1) :- date1(D,M,Y), date2(D1,M1,Y1), Y == Y1,
                              M < M1.
    before(D,M,Y,D1,M1,Y1) :- date1(D,M,Y), date2(D,M1,Y1), Y == Y1,
                              M = M1, D < D1.
}
```

3.4 Reusing code

Template programming can become a very useful feature in applications where it is necessary to compact repetitive pieces of code. We consider a planning program for the Space Shuttle Reaction Control System presented in [25] and written in SModels [27]. The DLV team is experimenting a porting of this program under the DLV system [13], and under DLP^T. Here is a sketch from the program:

```
#template ready_to_fire_conditions[propulsor_type(2)](2)
{
    ready_to_fire_conditions(J,T) :- propulsor_type(J,R), time_slot(T),
                                     tank(TK1,R), tank(TK2,R), TK1 != TK2,
                                     is_pressurized_by(J,TK1,T),
                                     is_pressurized_by(J,TK2,T),
                                     not is_damaged(J).
}
```

the above template is employed to remove a set of repeated rules, which are changed this way

```
fire_jet(J,T) :- ready_to_fire_conditions[jet(*,*)](J,T).
fire_vernier(J,T) :- ready_to_fire_conditions[vernier(*,*)](J,T).
```

4 Semantics of DLP^T

The semantics of the language is given through a suitable explosion algorithm. The **Explode** algorithm carries out the job of replacing groups of template atoms, having the same signature (whose definition is provided next), and occurring in a given DLP^T program, with corresponding atoms referred to a new intensional predicate, whose definition is provided through the introduction of new rules computed from the corresponding template definition.

Definition 3. Given a template atom t, its *template signature* s_t is given by replacing each standard term with a conventional (mute variable) '_' symbol.

For instance, `max[p(*,S,$)](M)` has the same signature (`max[p(*,_,$)](_)`) as `max[p(*,a,$)](H)`.

4.1 The Explode algorithm

It is given a DLP^T program P and a set of template definitions T. The **Explode** algorithm updates P by adding new rules and eliminating template atoms. The output of the algorithm is a DLP program. Let R the set of template atoms occurring within P. It is given a stack of signatures S, which will contain the set of signatures to be processed, and a set of signatures U, which will contain the set already processed signatures. S is initially filled up with each template signature occurring within P, while U is initially empty.

```
**Explode**(Input: a DLP$^T$ program $P$, a set of template definitions $R$.
        Outputs: $P'$ in DLP form )
**begin**
        push all the template signatures occurring in $P$ in $S$;
        $U = \emptyset$;
        **while**( $S$ is not empty ) **do**
            pop a template signature $s$ from $S$;
            **Unfold**( s );
            $U = U \cup \{s\}$.
**end**
```

Given a signature s, the **Unfold** algorithm generates from the template definition t associated to s a program $P_{t,s}$ which is added to P. S is updated with new template signatures that may be introduced in P, during the **Unfold** operation.

```
**Unfold** (Input: a signature $s$. Updates $P$ and $S$)
**begin**
    **if** ( $s \in U$ )
      given the template definition $t$ associated to $s$, $P = P \cup P_{t,s}$;
    **for** each $r \in P$
      **for** each template atom $a \in r$
          **if** $a$ has signature $s$
            **replace** $a$ with the standard atom $a_{t,s}$.
**end**
```

We show next how $P_{t,s}$ and $a_{t,s}$ are built.
Let the t header be

$$t[f_1(a_1)\,,\,\dots\,,\,f_n(a_n)](a_{n+1}) \tag{1}$$

and τ be its subprogram. Let s be

$$t[a_1(\mathbf{X}_1)\,,\,\dots\,,\,a_n(\mathbf{X}_n)](\mathbf{X}_{n+1}) \tag{2}$$

Given a compound list $\mathbf{X}$ of terms, let $\mathbf{X}_j$ denote the j^{th} term of $\mathbf{X}$; let $fr(\mathbf{X})$ a list of $|\mathbf{X}|$ fresh variables $F_{\mathbf{X},1}, \dots, F_{\mathbf{X},|\mathbf{X}|}$; let $st(\mathbf{X}), pr(\mathbf{X})$ and $pa(\mathbf{X})$ be the sublist of (respectively) standard, projection and parameter terms within $\mathbf{X}$.

Building $P_{t,s}$. The program $P_{t,s}$ is built in two steps. On the first step, $P_{t,s}$ is enriched with a set of rules, intended in order to deal with projection variables. For each i, we introduce a predicate a_i^s and we enrich $P_{t,s}$ with the auxiliary rule $a_i^s(\mathbf{X}'_i) \leftarrow a_i(\mathbf{X}''_i)$, where:

- $\mathbf{X}_i''$ is built from $\mathbf{X}_i$ substituting each term belonging to $pr(\mathbf{X}_i)$ with a '_' symbol, substituting $pa(\mathbf{X})$ with a fresh variable $P_h(1 \leq h \leq pa(\mathbf{X}))$, and substituting each term $X_{i,j}$ belonging to $st(\mathbf{X})$ with a fresh variable $S_k(1 \leq k \leq st(\mathbf{X}))$;
- $\mathbf{X}_i'$ is then set to $S_1, \ldots, S_{|st(\mathbf{X})|}, P_1, \ldots, P_{|pa(\mathbf{X})|}$.

For instance, given the signature s' =`max[student($,_,*)](_)` and the example template definition given in Definition 1, it is introduced the rule:
`student_s'(X,Y) :- student(_,X,Y).`
In the second step, for each rule or constraint c_τ belonging to τ, we create a new clause of $P_{t,s}$ where each atom a of c_τ is modified this way:

1. If a is $f_i(\mathbf{Y})$ where f_i is a formal predicate, it is substituted with the atom $a_i^s(\mathbf{Y}')$. $\mathbf{Y}'$ is set to $fr(st(a_i))|\mathbf{Y}$;
2. if a is a standard atom $p(\mathbf{Y})$, it is substituted by an atom $p^s(\mathbf{Y}')$, where $\mathbf{Y}' = fr(st(\mathbf{X}_1))|\ldots|fr(st(\mathbf{X}_n))|\mathbf{Y}$.
3. if a is a template atom, each compound atom in it is substituted according with point 2 above, obtaining a template atom a', then the signature of a' is pushed in S.

Building $a_{t,s}$. Assume a template atom referred to the definition t is in the form $t[a_1(\mathbf{X}_1)\,, \ldots\,, a_n(\mathbf{X}_n)](\mathbf{X}_{n+1})$. Then it is substituted with $a^s(\mathbf{X}')$, where $\mathbf{X}' = st(\mathbf{X}_1)|\ldots|st(\mathbf{X}_n)|\mathbf{Y}$.
An execution example of the **Explode** algorithm is given in Section 6. We are now able to give the formal semantics of DLP^T.

Definition 4. Given a DLP^T program P, and a set of template definitions T, let P' the output of the *Explode* algorithm on input $\langle P, T\rangle$. Let H be the Herbrand base of P. Given a stable model $M \in M(P')$, then we define $H \cap M$ as a stable model of P.

5 Theoretical properties of DLP^T

The explosion algorithm eliminates template atoms from a DLP^T program P, producing a DLP program P'. It is very important to investigate about two theoretical issues:

- Finding whether and when the **Explode** algorithm terminates; in general, we observe that, the explosion algorithm may not terminate. Anyway, we are able to prove that it can be decided in polynomial time whether **Explode** terminates on a given input.
- Establishing whether DLP^T programs are more expressive than DLP. In particular, we are able to prove that P' is polynomially larger than P. Thus DLP^T keeps the same expressive power as DLP. We observe that it is a desirable property of the language to keep the same expressive power and complexity of DLP. Indeed, we are guaranteed that DLP^T program encodings are as efficient as plain DLP encodings, since unfolded programs are reasonably larger with respect to the originating program.

Definition 5. It is given a DLP^T program P, and a set of template definitions T. The *dependency graph* $G_{T,P} = \langle V, E\rangle$ encoding dependencies between template atoms and template definitions is built as follows. Each template definition $t \in T$ will be represented by a corresponding node v_t of V. V contains a node u_P associated to P as well. E will contain a direct edge $(u_t, v_{t'})$ if the template t contains a template atom referred to the template t' inside its subprogram (as for the node referred to P, we consider the whole program P). Let $G_{T,P}(u) \subseteq G_{T,P}$ be the subgraph containing nodes and arc of $G_{T,P}$ reachable from u.

Theorem 1. *It is given a* DLP^T *program* P*, and a set of template definitions* T*. It can be decided in polynomial time whether the* **Explode** *algorithm terminates when* P *and* T *are taken as input.*

Proof. It is easy to see that **Explode** terminates iff $G_{T,P}(u_P)$ is acyclic. Indeed, consider that each operation of unfolding corresponds to the visit of an arc of $G_{T,P}(u_P)$. If $G_{T,P}(u_P)$ acyclic, **Explode** behaves like an in-depth, arc visit algorithm, where no arc is visited twice.
Viceversa, if $G_{T,P}(u_P)$ contains some cycle $u, v_1, \ldots, v_n, u$, an infinite series of new signatures will be produced and queued for processing. Indeed, assume each arc (u, v_1), $(v_1, v_2), \ldots, (v_n, u)$ has been processed. After the (v_n, u) processing, the arc (u, v_1) will be re-enqueued with a new signature, not present in the set of used signatures U, thus causing an infinite loop.

Definition 6. A set of template definitions T is said *nonrecursive* if for any DLP^T program P, the subgraph $G_{T,P}(u_P)$ is acyclic.

Proposition 1. *A set of template definitions* T *is nonrecursive iff* $G_{T,\emptyset}$ *is acyclic.*

Theorem 2. *It is given a* DLP^T *program* P*, and a nonrecursive set of template definitions* T*. The output* P' *of the* Explode *algorithm on input* $\langle P, T\rangle$ *is polynomially larger than* P *and* T*.*

Proof. We simply observe that each **Unfold** operation adds to P a number of rules/constraints whose overall size is bounded by the size of T. If T is nonrecursive, the number of **Unfold** operations carried out by the **Explode** algorithm corresponds to the number of arcs of $G_{T,P}$. The number of arcs of $G_{T,P}$ is bounded by the overall size of T and P. Thus the size of P' is $O(|T|(|T| + |P|))$.

Corollary 1. DLP^T *has the same expressive power as* DLP.

Proof. (Sketch). It is proved in [3] that plain DLP programs capture the Σ_2^P complexity class. DLP^T programs may allow to express more succinct encodings of problems. Anyway, since unfolded program produced by the **Explode** algorithm are polynomially larger only, and DLP^T semantics is defined in term of the equivalent, unfolded, DLP program, DLP^T has the same expressiveness properties as DLP.

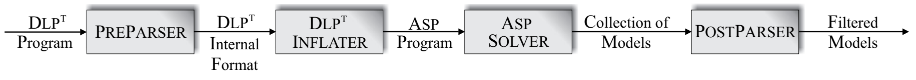

Fig. 1. Architecture of the DLP^T compiler

6 System architecture and usage

The DLP^T language has been successfully implemented on top of the DLV system [10–12], although other ASP solver may fruitfully serve the purpose. The current version of the language is available through the DLP^T Web page [5]. The overall architecture of the system is shown in Figure 1. The DLP^T system work-flow is as follows. A DLP^T program P, containing template atoms and template definitions is sent to a DLP^T pre-parser, which performs syntactic checks (included nonrecursivity checks), and builds an internal representation of the DLP^T program. The DLP^T Inflater implements the *Explode* Algorithm and produces an equivalent DLV program P'; P' is piped towards the DLV system. The set of models $\mathcal{M}$ of P', computed using DLV, is then converted in a more natural format through the Post-parser module; the Post-parser filters out from $\mathcal{M}$ informations about internally generated predicates and rules.
For instance, assume the following DLP^T program is given in a text file named `example.dlt`:

```
person(riccy,f,29). person(gibbi,m,25). person(peppe,m,28).
person(kali,m,27). person(paddy,f,26).

#template max[p(1)](1)
{
    exceeded(X) :- p(X),p(Y), Y > X.
    max(X) :- p(X), not exceeded(X).
}
oldest(Name,Sex,Age) :- max[person($,$,*)](Age), person(Name,Sex,Age).
older_sex(Name,Sex,Age) :- max[person($,Sex,*)](Age), person(Name,Sex,Age).
```

The DLP^T system is invoked through a shell command like

```
$ dlvt example.dlt
```

The `dlvt` executable accepts any command line option proper of the DLV executable. The DLV executable is automatically invoked with these options on a text file containing the output of the *Explode* algorithm, which contains pieces of code like

```
...
person_3(Var0) :- person(_,_,Var0).
max_2(X) :- person_3(X), not exceeded_1(X).
exceeded_1(X) :- person_3(X), person_3(Y), >(Y,X).
oldest(Name,Sex,Age) :- max_2(Age), person(Name,Sex,Age).
...
person_1(Sex,Var0) :- person(_,Sex,Var0).
max_0(Sex,X) :- person_1(Sex,X), not exceeded_0(Sex,X).
exceeded_0(Sex,X) :- person_1(Sex,X), person_1(Sex,Y), >(Y,X).
older_sex(Name,Sex,Age) :- max_0(Sex,Age), person(Name,Sex,Age).
```

The former group of rules represents the inline expansion of the template atom `max[person($,$,*)](Age)`, whereas the latter group corresponds to the expansion of the atom `max[person($,Sex,*)](Age)`. Note that another interesting feature of DLP^T is here put in evidence, i.e. the capability to employ a template definition grouping by different values of an attribute (e.g. grouping ages by sex). The output of the call to the DLV system contains one model containing a lot of redundant information:

```
{person(riccy,f,29), ..., person(paddy,f,26), oldest(riccy,f,29),
older_sex(riccy,f,29), older_sex(peppe,m,28), max_2(29),
max_0(f,29), ..., person_1(m,28), exceeded_0(f,26), ..., exceeded_0(m,27),
person_3(25), ..., person_3(29), exceeded_1(25), ..., exceeded_1(28)}
```

Anyway, models are presented to the user after the filtering step, which outputs:

```
{person(riccy,f,29), ..., person(paddy,f,26),
oldest(riccy,f,29),  older_sex(riccy,f,29), older_sex(peppe,m,28)}
```

7 Conclusions

We presented the DLP^T language, an extension of ASP allowing to define template predicates. The semantics of the language has been introduced through a suitable and sound transformation algorithm. The proposed language is, in our opinion, very promising: we plan to further extend the framework, and, in particular, we are thinking about *a)* generalizing template definition in order to allow safe forms of recursion between template definitions, *b)* introducing new forms of template atoms in order to improve reusability of the same template definition in different contexts, *c)* extending the template definition scheme using standard languages such as `C++`. We would like to thank Francesco Calimeri, Nicola Leone and Luigi Palopoli for their fruitful remarks.

References

1. C. Anger, K. Konczak, and T. Linke. `NoMoRe`: A System for Non-Monotonic Reasoning. In *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI), pages 406–410. Springer Verlag, September 2001.
2. M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all the problems in NP. *Computer Languages, Elsevier Science, Amsterdam (Netherlands)*, 26(2-4):165–195, 2000.
3. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
4. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programming: Semantics,complexity,and implementation in DLV. *International Joint Conference on Artificial Intelligence (IJCAI 2003). To appear.*
5. The DLP^T web site. `http://dlpt.gibbi.com`.

6. D. East and M. Truszczyński. dcs: An implementation of DATALOG with Constraints. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000.
7. U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00), July 30 – August 3, 2000, Austin, Texas USA*, pages 417–422. AAAI Press / MIT Press, 2000.
8. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.
9. T. Eiter, G. Gottlob, and N. Leone. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1–2):129–177, December 1997.
10. W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan, September 1999.
11. W. Faber, N. Leone, and G. Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, August 2001. Morgan Kaufmann Publishers.
12. W. Faber and G. Pfeifer. DLV homepage, since 1996. http://www.dlvsystem.com/.
13. S. Galizia. Generazione automatica di manovre per lo space shuttle mediante la programmazione logica disgiuntiva. *Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003. To appear.*
14. M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.
15. S. Greco and D. Saccà. NP optimization problems in datalog. *International Symposium on Logic Programming. Port Jefferson, NY, USA*, pages 181–195, 1997.
16. K. Heljanko and I. Niemelä. Bounded LTL Model Checking with Stable Models. In *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI), pages 200–212. Springer Verlag, September 2001.
17. G. Ianni, F. Calimeri, V. Lio, S. Galizia, and A. Bonfà. A reasoning system for managing web ontologies and agent information exchange. *Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003. To appear.*
18. The ICONS web site. `http://www.icons.rodan.pl/`.
19. The Infomix web site. `http://www.mat.unical.it/infomix`.
20. G. M. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41(1):44–64, 1990.
21. N. Leone and P. Rullo. Ordered logic programming with sets. *Journal of Logic and Computation*, 3(6):621–642, 1993.
22. V. Lifschitz. Foundations of Logic Programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford, 1996.
23. N. McCain and H. Turner. Satisfiability Planning with Causal Theories. In *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 212–223. Morgan Kaufmann Publishers, 1998.
24. I. Niemelä. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

25. M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog Decision Support System for the Space Shuttle. In *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, number 1551 in Lecture Notes in Computer Science, pages 169–183. Springer, 1999.
26. P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.
27. P. Simons. *Extending and Implementing the Stable Model Semantics.* PhD thesis, Helsinki University of Technology, Finland, 2000.
28. T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, number 1551 in Lecture Notes in Computer Science, pages 305–319. Springer, 1999.
29. T. Syrjänen. A Rule-Based Formal Model for Software Configuration. Technical Report A55, Digital Systems Laboratory, Department of Computer Science, Helsinki University of Technology, Espoo, Finland, 1999.