Enhancing Answer Set Programming with Templates

Giovambattista Ianni
Giuseppe Ielpa
Adriana Pietramala
Maria Carmela Santoro
Francesco Calimeri
Mathematics Dept., Università della Calabria,
Via Pietro Bucci, 30B
87036 Rende (CS), Italy
E-mail: {lastname}@mat.unical.it

Abstract

The work aims at extending Answer Set Programming (ASP) with the possibility of quickly introducing new predefined constructs and to deal with compound data structures: we show how ASP can be extended with 'template' predicate's definitions. We present language syntax and give its operational semantics. We show that the theory supporting our ASP extension is sound, and that program encodings are evaluated as efficiently as ASP programs. Examples show how the extended language increases declarativity, readability, compactness of program encodings and code reusability.[1].

Introduction

Research on Answer Set Programming (ASP, in the following) produced several, mature, implemented systems featuring clear semantics and efficient program evaluation (Faber *et al.* 1999; Faber, Leone, & Pfeifer 2001; Niemelä 1999; Simons 2000; Anger, Konczak, & Linke 2001; Egly *et al.* 2000; McCain & Turner 1998; Rao *et al.* 1997; East & Truszczyński 2000). ASP has recently found a number of promising applications: several tasks in information integration and knowledge management require complex reasoning capabilities, which are explored, for instance, in the INFOMIX and ICONS projects (funded by the European Commission)(INFOMIX ; ICONS). It is very likely that this new generation of ASP applications require the introduction of repetitive pieces of standard code. Indeed, a major need of complex and huge ASP applications such as (Nogueira *et al.* 1999) is dealing efficiently with large pieces of such a code and with complex data structures, more sophisticated than the simple, native ASP data types.

Indeed, the non-monotonic reasoning community has continuosly produced, in the past, several extensions of nonmonotonic logic languages, aimed at improving readability and easy programming through the introduction of new constructs, employed in order to specify classes of constraints, search spaces, data structures, new forms of reasoning, new special predicates (Cadoli *et al.* 2000; Eiter, Gottlob, & Leone 1997; Kuper 1990), such as aggregate predicates (Dell'Armi *et al.*).

[1]This work has been partially funded by the EU research project IST-2002-33570 (INFOMIX)

The language DLP^T we propose here has two purposes. First, DLP^T moves the ASP field towards industrial applications, where code reusability is a crucial issue. Second, DLP^T aims at minimizing developing times in ASP system prototyping. ASP systems developers wishing to introduce new constructs are enabled to fast prototype their languages, make their language features quickly available to the scientific community, and successively concentrate on efficient (and long lasting) implementations. To this end, it is necessary a sound specification language for new ASP constructs. ASP itself proves to fit very well for this purpose.

The proposed framework introduces the concept of 'template' predicate, whose definition can be exploited whenever needed through binding to usual predicates: this approach is somehow similar to the clause view approach, common in Object Oriented Logic Programming (Davison 1993).

Template predicates can be seen as a way to define intensional predicates by means of a subprogram, where the subprogram is generic and reusable. This eases coding and improves readability and compactness of ASP programs:

Example 1.1 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 generic unary predicate `p`. A template definition may be instantiated as many times as necessary, through *template atoms*, like in the following sample 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 the definition of the template predicate `max` on a ternary predicate, discarding the second attribute of `student`, and grouping by values of the first attribute. □

The operational semantics of the language is defined through a suitable algorithm which is able to produce, from a set of nonrecursive 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

algorithm, and the expressiveness of the DLP^T language. Indeed, we prove that it is guaranteed that DLP^T program encodings are as efficient as plain ASP encodings, since unfolded programs are just polynomially larger with respect to the originating program.
The DLP^T language has been successfully implemented and tested on top of the DLV system (Faber, Leone, & Pfeifer 2001). Anyway, the proposed paradigm does not rely at all on DLV special features, and is easily generalizable. In sum, benefits of the DLP^T language are: improved declarativity and succinctness of the code; code reusability and possibility to collect templates within libraries; capability to quickly introduce new, predefined constructs; fast language prototyping.
The paper is structured as follows: 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 . Section formally introduces the semantics of DLP^T. Theoretical properties of DLP^T are discussed in section . In section we describe architecture and usage of the implemented system. Eventually, in section , conclusions are drawn.

Syntax of the DLP^T language

We give a quick definition of the syntax and informal semantics of DLP programs[2]. We assume the reader to be familiar with basic notions concerning with DLP semantics. A thorough definition of concepts herein adopted can be found in (Eiter *et al.* 2000). A (DLP) *rule* r is a construct

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

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 $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k,$ not $b_{k+1}, ...,$ 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*).
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 (Lifschitz 1996). 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 1.2 A template definition D consists of:
- a template header,

$$\#\textbf{template } n_D[f_1(b_1)\,,\,...\,,\,f_n(b_n)](b_{n+1})$$

where each $b_i (1 \leq i \leq n+1)$ is a nonnegative integer value, and $f_1, \ldots, f_n$ are predicate names, said in the following *formal predicates*. n_D is called *template name*;
- an associated DLP^T subprogram enclosed in curly braces; n_D may be used within the subprogram as predicate of arity b_{n+1}, whereas each predicate $f_i (1 \leq i \leq n)$ is intended to be of arity b_i. At least a rule having n_D within its head must be declared. For instance, the following is a valid template definition:

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

Definition 1.3 A template atom t is of the form:

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

where $p_1, \ldots, p_n$ are predicate names (namely *actual* predicates), and n_t is a template name. Each $\mathbf{X}_i (1 \leq i \leq n)$ is a list of *special* terms (referred in the following as *special* list of terms). A special list of terms can contain either a variable name, a constant name, a dollar '$' symbol (from now on, *projection term*) or a '*' (from now on, *parameter term*). Variables and constants are called *standard* terms. Each $p_i(\mathbf{X}_i)(1 \leq i \leq n)$ is called *special* atom. $\mathbf{A}$ is a list of usual terms (i.e. either variables or constants) called *output list*. Given a template atom t, let $D(t)$ be the corresponding template definition having the same template name. It is assumed there is a unique definition for each template name. □

An example of template atom is

```
max[company(\$,State,*)](Income).
```

Intuitively, projection terms ('$' symbols) are intended in order to indicate attributes of an actual predicate which have to be ignored. A standard term (a constant or a variable) within an actual atom indicates a 'group-by' attribute, whereas parameter terms ('*' symbols) indicate attributes to be considered as parameter. The intuitive meaning of the above template atom is to define a predicate computing the companies with maximum value of the 'income' attribute (the third attribute of the *company* predicate), grouped by the 'state' attribute (the second one), ignoring the first attribute. The computed values of $Income$ are returned through the output list.

Knowledge Representation by DLP^T

In this section we show by examples the main advantages of template programming. Examples point out the provision of a succinct, elegant and easy-to-use way for quickly introducing new constructs through the DLP^T language.

Aggregates. Aggregate predicates (Ross & Sagiv 1997), allow to represent properties over sets of elements. Aggregates or similar special predicates have been already studied and implemented in several ASP solvers (Dell'Armi *et al.* ; Simons 2000): the next example shows how to fast prototype aggregate semantics without taking into account of the efficiency of a built-in implementation. Here we take advantage of the template predicate `max`, defined in Example 1.1. The next template predicate defines a general program to count distinct values of a predicate `p`, given an order relation `succ` defined on the domain of `p`. We assume the domain of integers is bounded to some finite value.

[2]Disjunctive Logic Programming. Throughout this paper, we will adopt the first historical definition of ASP (Lifschitz 1999) as synonym of Disjunctive Logic Programming.

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

The above template definition is conceived in order to count, in a iterative-like way, values of the p predicate through the $partialCount$ predicate. A ground atom $partialCount(i, a)$ means that at the stage i, the constant a has been counted up. The predicate $count$ takes the value which has been counted at the highest (i.e. the last) stage value. The above program is somehow involved and shows how difficult could be to simulate aggregate constructs in Answer Set Programming. Anyway, the use of templates allows to write it once, and reuse it as many times as necessary.
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`. The last rule is equivalent to the piece of code:

```
partialCount'(X) :- partialCount(_,X).
count(M) :- max[partialCount'(*)](M).
```

Definition of ad hoc search spaces. Template definitions can be employed to introduce and reuse constructs defining the most common search spaces. This improves declarativity of ASP programs to a larger extent. 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 (Greco & Saccà 1997; Cadoli *et al.* 2000).

```
#template
subset[p(1)](1)
    {
    subset(X) v -subset(X) :- p(X).
    }
#template permutation[p(1)](2).
    {
    permutation(X,N) v npermutation(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).
    }
```

The explanation of the `subset` template predicate is quite straightforward. As for the `permutation` definition, a ground atom $permutation(x, i)$ tells that the element x (taken from the domain of p), is in position i within the currently guessed permutation. The rest of the template subprogram forces permutations properties to be met.
Next we show how `count` and `subset` can be exploited to succinctly encode the *k-clique* problem (Garey & Johnson 1979), i.e., given a graph G (represented by predicates `node` and `edge`), find if there exists a complete subgraph containing at least k nodes (we consider here the 5-clique problem):

```
in_clique(X) :- subset[node(*)](X).
:- count[in_clique(*),>(*,*)](K), K < 5. :-
in_clique(X),in_clique(Y), X <> Y, not edge(X,Y).
```

The first rule of this example guesses a clique from a subset of nodes. The first constraint forces a candidate clique to be at least of `5` nodes, while the last forces a candidate clique to be strongly connected. The `permutation` template can be employed, for instance, to encode the Hamiltonian Path problem: given a graph G, find a path visiting each node of G exactly once:

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

Handling of complex data structures. DLP^T can be fruitfully employed to introduce operations over complex data structures, such as 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 (Kuper 1990; Leone & Rullo 1993)), 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`, can 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 symmetricdifference[a(1),b(1)](1)
    {
    symmetricdifference(X)
        :- union[a(*),b(*)](X),
           not intersection[a(*),b(*)](X).
    }
```

Dates: managing time and date data types is another important issue in engineering applications of DLP. For instance, in (Ianni *et al.* 2003 Reggio Calabria Italy), it is very important to reason on compound records containing date values. The following template shows how to compare dates represented through a ternary relation ⟨day, month, year⟩.

```
#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.
}
```

Semantics of DLP^T

The semantics of the DLP^T language is given through a suitable ''explosion'' algorithm. It is given a DLP^T program P. The aim of the *Explode* algorithm, introduced next, is to remove template atoms from P. Each template atom t is replaced with a standard atom, referring to a fresh intensional predicate p_t. The subprogram d_t, defining the predicate p_t, is computed taking into account of the template definition $D(t)$ associated to t. Actually, many template atoms may be grouped and associated to the same subprogram. The concept of atom signature, introduced next, helps in finding groups of equivalent template atoms. The final output of the algorithm is a DLP program P'. Answer sets of the originating program P are constructed, *by definition*, from answer sets of P'. Throughout this section, we will refer to Example 1.1 as running example. By little abuse of notation, $a \in P$ (resp. $a \in r$) means that the atom a appears in the program P (the rule r, respectively).

Definition 1.4 Given a template atom t, the corresponding *template signature* $s(t)$ is obtained from t by replacing each standard term with a conventional (mute variable) '_' symbol. Let $D(s(t))$ be the template definition associated to the signature $s(t)$; Given a DLP^T program P, let $A(P)$ be the set of template atoms occurring in P. Let $S(A(P))$ be the set of signatures $\{s(t) : t \in A(P)\}$. □

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

The Explode algorithm

The **Explode** algorithm ($\mathcal{E}$ in the following) is sketched in Figure 1. It is given a DLP^T program P and a set of template definitions T. The output of $\mathcal{E}$ is a DLP program P'. $\mathcal{E}$ takes advantage of a stack of signatures S, which contains the set of signatures to be processed; a set U contains the already processed signatures. S is initially filled up with each template signature occurring within P, while U is initially empty.

The purpose of the main loop of $\mathcal{E}$ is to iteratively apply the $\mathcal{U}$ (Unfold) operation to P, until S is empty. Given a signature s, the $\mathcal{U}$ operation generates from the template definition $D(s)$ a DLP^T program P^s which defines a fresh predicate t^s, where t is the template name of s. In case s is being processed for the first time ($s \notin U$), P^s is appended to P; furthermore, each template atom $a \in P$, such that a has signature s, is replaced with a suitable atom $a^s(\mathbf{X}')$. It is important pointing out that, if P^s contains template atoms, the unfolding operation updates S with new template signatures.

We show next how P^s is constructed and template atoms are removed.

Explode(Input: a DLP^T program P, a set of template definitions T.
Outputs: an updated version of P' of P in DLP form.
Data Structures: a stack S, a set U)
begin
push $S(A(P))$ in S;
$U = \emptyset$; $P' = P$
while (S is not empty) **do begin**
pop a template signature s from S;

//Start of the $\mathcal{U}$ (Unfold) operation;
if ($s \notin U$)
construct P^s (see Subsection), then set $P = P \cup P^s$;
push $S(A(P^s))$ in S;
for each template atom $a \in P$
if a has signature s
construct the standard atom $a^s(\mathbf{X}')$ (see Subsection);
replace a with $a^s(\mathbf{X}')$;
//End of the $\mathcal{U}$ operation;

$U = U \cup \{s\}$.
end;
end.

Figure 1: The Explode ($\mathcal{E}$) Algorithm

Let the header of $D(s)$ be

$$\#\mathbf{template}\ t[f_1(b_1)\ ,\ \ldots\ ,\ f_n(b_n)](b_{n+1})$$

Let s be of the form

$$t[p_1(\mathbf{X}_1)\ ,\ \ldots\ ,\ p_n(\mathbf{X}_n)](\mathbf{X}_{n+1})$$

Given a special list $\mathbf{X}$ of terms, let $\mathbf{X}[j]$ denote the j^{th} term of $\mathbf{X}$; let $fr(\mathbf{X})$ be a list of $|\mathbf{X}|$ fresh variables $F_{\mathbf{X},1}, \ldots, F_{\mathbf{X},|\mathbf{X}|}$; let $st(\mathbf{X}), pr(\mathbf{X})$ and $pa(\mathbf{X})$ be the sublist of (respectively) standard, projection and parameter terms within $\mathbf{X}$. Given two lists $\mathbf{A}$ and $\mathbf{B}$, let $\mathbf{A}\&\mathbf{B}$ be the list obtained appending $\mathbf{B}$ to $\mathbf{A}$.

How P^s is constructed.

The program P^s is built in two steps. On the first step, P^s is enriched with a set of rules, intended in order to deal with projection variables.

For each $p_i \in s$, we introduce a predicate p_i^s and we enrich P^s with the auxiliary rule $p_i^s(\mathbf{X}'_i) \leftarrow p_i(\mathbf{X}''_i)$, where:

- $\mathbf{X}''_i$ is built from $\mathbf{X}_i$ substituting $pr(\mathbf{X}_i)$ with $fr(pr(\mathbf{X}_i))$, substituting $pa(\mathbf{X}_i)$ with $fr(pa(\mathbf{X}_i))$, and substituting $st(\mathbf{X}_i)$ with $fr(st(\mathbf{X}_i))$;
- $\mathbf{X}'_i$ is set to $fr(st(\mathbf{X}_i))\&fr(pa(\mathbf{X}_i))$.

For instance, given the signature

$s_2 = max[student(\_, \$, *)](\_)$

and the example template definition given in Example 1.1, let $\mathbf{L}$ be the list $\langle$`_, $, *`$\rangle$; it is introduced the rule:

$$student^{s_2}(F_{st(\mathbf{L}),1}, F_{pa(\mathbf{L}),1}) \\ : -student(F_{st(\mathbf{L}),1}, F_{pr(\mathbf{L}),1}, F_{pa(\mathbf{L}),1}).$$

Note that projection variables are filtered out from $student^s$.

In the second step, for each rule r belonging to $D(s)$, we create an updated version r' to be put in P^s, where each atom $a \in r$ is modified this way:

- if a is $f_i(\mathbf{Y})$ where f_i is a formal predicate, it is substituted with the atom $p_i^s(\mathbf{Y}')$. $\mathbf{Y}'$ is set to $fr(st(\mathbf{X}_i))\&\mathbf{Y}$;

- if a is a either a standard (included atoms having t as predicate name) or a special atom (in this latter case a occurs within a template atom) $p(\mathbf{Y})$, it is substituted with an atom $p^s(\mathbf{Y}')$, where
$Yvect' = fr(st(\mathbf{X}_1))\& \ldots \&fr(st(\mathbf{X}_n))\&\mathbf{Y}$.

Example 1.5 For instance, consider the rule
$max(X) \text{:-} p(X), not\ exceeded(X).$
from Example 1.1, and the signature
$s_2 =$ `max[student(_,$,*)](_);`
let $\mathbf{L}$ be the special list $\langle \_, \$, * \rangle$; according to the steps introduced above, this rule is translated to

$$max^{s_2}(F_{\mathbf{L},1}, X) \text{:-} student^{s_2}(F_{\mathbf{L},1}, X), \\ not\ exceeded^{s_2}(F_{\mathbf{L},1}, X).$$ □

How template atoms are replaced[3].

Consider a template atom in the form
$t[p_1(\mathbf{X}_1), \ldots, p_n(\mathbf{X}_n)](\mathbf{X}_{n+1})$.
It is substituted with
$t^s(\mathbf{X}')$
where
$\mathbf{X}' = st(\mathbf{X}_1)\& \ldots \&st(\mathbf{X}_n)\&\mathbf{Y}$.

Example 1.6 The complete output of $\mathcal{E}$ on the constraint
$\text{:-} max[student(\_, \$, *)](M), M > 25.$
coupled with the template definition of `max` given in Example 1.1 is:

$$\begin{array}{rcl} student^{s_2}(S_1, P_1) & :- & student(S_1, \_, P_1). \\ exceeded^{s_2}(F_{\mathbf{L},1}, X) & :- & student^{s_2}(F_{\mathbf{L},1}, X), \\ & & student^{s_2}(F_{\mathbf{L},1}, Y), Y > X. \\ max^{s_2}(F_{\mathbf{L},1}, X) & :- & student^{s_2}(F_{\mathbf{L},1}, X), \\ & & not\ exceeded^{s_2}(F_{\mathbf{L},1}, X). \\ & :- & max^{s_2}(Sex, M), M > 25. \end{array}$$ □

We are now able to give the formal semantics of DLP^T. It is important highlighting that stable models of a DLP^T program are, by definition, constructed in terms of stable models of an equivalent DLP program.

Definition 1.7 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 $H(P)$ be the Herbrand base of P' restricted to those atoms having predicate name appearing in P. Given a stable model $m \in M(P')$, then we define $H(P) \cap m$ as a stable model of P. □

Note that the Herbrand base of a DLP^T program is defined in terms of the Herbrand base of a DLP program *which is not* the output of $\mathcal{E}$.

Theoretical properties of DLP^T

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

[3]Depending on the form of $D(s)$, some template atom might not to be allowed, since some atom with same predicate name but with mismatched arities could be generated. We do not discuss here these syntactic restriction for space reasons.

- Finding whether and when $\mathcal{E}$ terminates; in general, we observe that $\mathcal{E}$ might not terminate, for instance, in case of recursive template definitions. Anyway, we prove that it can be decided in polynomial time whether $\mathcal{E}$ terminates on a given input.
- Establishing whether DLP^T programs are encoded as efficiently as DLP programs. In particular, we are able to prove that P' is polynomially larger than P. Thus DLP^T keeps the same expressive power as DLP. This way, we are guaranteed that DLP^T program encodings are as efficient as plain DLP encodings, since unfolded programs are always reasonably larger with respect to the originating program.

Definition 1.8 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 referring 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 1.9 It is given a DLP^T program P, and a set of template definitions T. It can be decided in polynomial time whether $\mathcal{E}$ terminates when P and T are taken as input.
Proof. (Sketch). It is easy to see that $\mathcal{E}$ 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, $\mathcal{E}$ behaves like an in-depth, arc visit algorithm, where no arc is visited twice.
Vice versa, 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 1.10 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. □

It is useful to deal with nonrecursive sets of template definition, since they may be safely employed with any program. Checking whether a set of template definitions is nonrecursive is quite easy.

Proposition 1.11 A set of template definitions T is nonrecursive iff $G_{T,\emptyset}$ is acyclic.

Theorem 1.12 It is given a DLP^T program P, and a nonrecursive set of template definitions T. The output P' of $\mathcal{E}$ on input $\langle P, T \rangle$ is polynomially larger than P and T.
Proof. (Sketch). We simply observe that each execution of $\mathcal{U}$ 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 $\mathcal{U}$ operations carried out by $\mathcal{E}$ 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 $O(|T|(|T| + |P|))$. □

Figure 2: Architecture of the DLP^T compiler

Corollary 1.13 DLP^T has the same expressive power as DLP.

Proof. (Sketch). It is proved in (Dantsin *et al.* 2001) that plain DLP programs (under the brave reasoning semantics) 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 $\mathcal{E}$ 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. □

System architecture and usage

The DLP^T language has been implemented on top of the DLV system (Faber *et al.* 1999; Faber, Leone, & Pfeifer 2001; Faber & Pfeifer since 1996). The current version of the language is available through the DLP^T Web page (DLPT). The overall architecture of the system is shown in Figure 2. The DLP^T system work-flow is as follows. A DLP^T program 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 performs the *Explode* Algorithm and produces an equivalent DLV program P'; P' is piped towards the DLV system. The models $M(P')$ of P', computed by DLV, are then converted in a readable format through the Post-parser module; the Post-parser filters out from $M(P')$ informations about internally generated predicates and rules.

Conclusions

We presented the DLP^T language, an extension of ASP allowing to define template predicates. 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 semantics 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 language using standard languages such as C++. As far as performances are concerned, we point out that these are strictly tied to performances of resulting DLP programs. Nonetheless, this work aims at introducing fast prototyping techniques, and does not consider time performances as a primary target[4].

[4]We would like to thank Nicola Leone and Luigi Palopoli for their fruitful remarks.

References

Anger, C.; Konczak, K.; and Linke, T. 2001. 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), 406–410. Springer Verlag.

Cadoli, M.; Ianni, G.; Palopoli, L.; Schaerf, A.; and Vasile, D. 2000. NP-SPEC: An executable specification language for solving all the problems in NP. *Computer Languages, Elsevier Science, Amsterdam (Netherlands)* 26(2-4):165–195.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys* 33(3):374–425.

Davison, A. 1993. A survey of logic programming-based object-oriented languages. In Wegner, P.; Yonezawa, A.; and Agha, G., eds., *Research Directions in Concurrent Object Oriented Programming*, 42–106. MIT Press.

Dell'Armi, T.; Faber, W.; Ielpa, G.; Leone, N.; and Pfeifer, G. Aggregate functions in disjunctive logic programming: Semantics,complexity,and implementation in DLV. *International Joint Conference on Artificial Intelligence (IJCAI 2003)*.

The DLP^T web site. http://dlpt.gibbi.com.

East, D., and Truszczyński, M. 2000. dcs: An implementation of DATALOG with Constraints. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*.

Egly, U.; Eiter, T.; Tompits, H.; and Woltran, S. 2000. 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*, 417–422. AAAI Press / MIT Press.

Eiter, T.; Faber, W.; Leone, N.; and Pfeifer, G. 2000. Declarative Problem-Solving Using the DLV System. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers. 79–103.

Eiter, T.; Gottlob, G.; and Leone, N. 1997. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science* 189(1–2):129–177.

Faber, W., and Pfeifer, G. since 1996. DLV homepage. http://www.dlvsystem.com/.

Faber, W.; Leone, N.; Mateis, C.; and Pfeifer, G. 1999. Using Database Optimization Techniques for Nonmonotonic Reasoning. In *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, 135–139. Prolog Association of Japan.

Faber, W.; Leone, N.; and Pfeifer, G. 2001. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, 635–640. Seattle, WA, USA: Morgan Kaufmann Publishers.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company.

Greco, S., and Saccà, D. 1997. NP optimization problems in datalog. *International Symposium on Logic Programming. Port Jefferson, NY, USA* 181–195.

Ianni, G.; Calimeri, F.; Lio, V.; Galizia, S.; and Bonfà, A. 2003. Reggio Calabria, Italy. Reasoning about the semantic web using answer set programming. In *APPIA-GULP-PRODE 2003. Joint Conference on Declarative Programming*, 324–336.

The ICONS web site. http://www.icons.rodan.pl/.

The Infomix web site. http://www.mat.unical.it/infomix.

Kuper, G. M. 1990. Logic programming with sets. *Journal of Computer and System Sciences* 41(1):44–64.

Leone, N., and Rullo, P. 1993. Ordered logic programming with sets. *Journal of Logic and Computation* 3(6):621–642.

Lifschitz, V. 1996. Foundations of Logic Programming. In Brewka, G., ed., *Principles of Knowledge Representation*. Stanford: CSLI Publications. 69–127.

Lifschitz, V. 1999. Answer set planning. In *International Conference on Logic Programming*, 23–37.

McCain, N., and Turner, H. 1998. Satisfiability Planning with Causal Theories. In *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, 212–223. Morgan Kaufmann Publishers.

Niemelä, I. 1999. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3–4):241–273.

Nogueira, M.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 1999. 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, 169–183. Springer.

Rao, P.; Sagonas, K. F.; Swift, T.; Warren, D. S.; and Freire, J. 1997. 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), 2–17. Dagstuhl, Germany: Springer Verlag.

Ross, K. A., and Sagiv, Y. 1997. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences* 54(1):79–97.

Simons, P. 2000. *Extending and Implementing the Stable Model Semantics*. Ph.D. Dissertation, Helsinki University of Technology, Finland.