LAYERED MODES

SANDRO ETALLE AND MAURIZIO GABBRIELLI

▷ Modes were introduced in logic programming to differentiate the input arguments of a predicate from its output arguments. This information can be used for verifying the most diverse properties of logic programs, notably absence of run-time errors and absence of deadlocks in presence of delay declarations. We introduce here *layered modes*, an extension of existing mode systems which allow us to enlarge the class of programs which can be verified by using modes. In particular, we show that this extension allows us to better handle programs that employ a dynamic selection rule and programs that use incomplete data structures such as difference-lists. ◁

1. INTRODUCTION

Modes were first introduced in the logic programming literature by Mellish [23] and then more extensively studied by Reddy [26] and by Dembinski and Małuszynski [15]. Informally modes indicate how the arguments of a relation should be used, i.e. which are the input and which are the output arguments for each relation symbol. In Logic Programming modes have been extensively used, mainly for two specific purposes:

(i) *as a validation method.* The information provided by the modes allows one to derive several useful properties for those programs which are "well-moded", i.e. which respect some correctness conditions relating the input arguments to the output arguments. For example, one can prove absence of run-time errors for Prolog built-ins, absence of floundering (for programs with negation) and absence of deadlock (for programs with dynamic scheduling) [4, 2]. Modes are also a basic component of modern tools for the development and the maintenance of large applications (see for instance [18, Section 6]).

Address correspondence to Sandro Etalle, Dept. of Computer Science, Sandro Etalle, Department of Computer Science – University of Maastricht, P.O. Box 616, 6200 MD Maastricht, The Netherlands. etalle@cs.unimaas.nl. Maurizio Gabbrielli, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy gabbri@di.unipi.it.

The research was partly carried out when the first author was employed by the University of Genova first and by the University of Amsterdam later.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010

(ii) *as a tool for improving the performance of programs.* Most compilers encourage the user to specify a mode declaration (see for instance [35]). Notably, in Mercury [30] mode declarations are mandatory and constitute a crucial aspect in obtaining its remarkable performance's achievements .

In this paper we introduce *layered modes*, an extension of the standard definition of mode based on the idea of associating also a *timing* to each atom position. This determines a precedence relation among atoms' positions which generalizes the simple ordering imposed by a fixed selection rule (e.g. leftmost), thus providing a finer control over the input/output dependencies existing in queries and clauses. The main advantage of this extension can be described as follows:

- It improves on standard modes [4] in the fact that it is not bounded to a fixed (left-to-right) selection rule. Thus layered modes are particularly suitable for logic languages with *dynamic scheduling*, where the control over the execution is provided by the so-called *delay declarations* which allow to change dynamically the selection rule. The practical relevance of dynamic scheduling is substantiated by the fact that it is allowed by most of the existing programming systems (e.g. Sicstus Prolog [7], Eclipse [36] and Gödel [19]).
- It allows to deal in a natural manner with programs which use circular structures such as difference-lists. Difference-lists have become a standard tool for logic programmers (see, for instance, the books of Sterling and Shapiro [31] and O'Keefe [25]) and their crucial feature is that they employ a logical variable as a reference to an incomplete part of the structure itself (for difference-lists, the tail of the list). For this reason, difference structures are typically non-ground terms with aliasing, and this makes them difficult fit into a standard moded framework.

Summarizing, layered modes allow us to enlarge the class of well-moded programs while retaining all their classical applications, that is, allowing to prove program's properties such as absence of deadlock and of floundering.

The remaining of this paper is organized as follows. We assume that the reader is acquainted with the terminology and the main results of logic programming theory (see [1, 20]). In the next Section we introduce layered modes and we show that they strictly extend classical modes. In Section 3 we discuss three applications of layered modes: we describe a simple method for proving deadlock-freedom for programs with dynamic scheduling, we show how difference lists can be handled and we consider the proof of absence of floundering for programs with negation. In Section 4 we compare our results to those existing in the literature and Section 5 concludes by illustrating some possible extensions of layered modes, notably to the case of concurrent languages.

2. LAYERED MODES

We start with the following basic definition.

Definition 2.1. Consider an n-ary relation symbol p. By a *mode* for p we mean a function m_p from $\{1,\dots,n\}$ to $\{\mathsf{in},\mathsf{out}\}$. □

If $m_p(i) = \mathsf{in}$ (resp. out), we say that i an *input* (resp. output) position of p (w.r.t. m_p). Intuitively, the mode indicates how the arguments of a relation should be used.

Now, we extend this notion by also giving a *timing* to each atom position as follows.

Definition 2.2. Consider an n-ary relation symbol p. By a (global) *timing* for p we mean a function τ_p from $\{1, \ldots, n\}$ to N. $\square$

Following the notation of [4] we write m_p and τ_p together as $p(m_p(1) : \tau_p(1), \ldots, m_p(n) : \tau_p(n))$. For instance $\mathtt{append}(\mathsf{in} : 1, \mathsf{in} : 1, \mathsf{out} : 2)$ denotes the mode and timing for the relation append in which the first two positions are input positions of timing 1 and the third one is an output position of timing 2. Intuitively, these concepts reflect how a predicate is supposed to be employed: one might regard an atom as a procedure call, which contains input and output positions, and in which timings are specified according to the rule that (the result carried by) an output position of timing α should depend on the data provided in the input positions of timing *smaller than* α. The reason for the presence of timings is that, unlike in imperative programming, in Logic Programming are allowed queries whose input is not completely specified. In situations such as in presence of difference-lists the input actually *cannot* be completely specified.

Definition 2.3. Consider an n-ary relation symbol p. By a *layered mode* for p we mean a couple $\langle m_p, \tau_p \rangle$ consisting of a mode and a timing for p. $\square$

¿From now on we assume that *for each considered relation symbol p there exists a unique fixed layered mode* $\langle m_p, \tau_p \rangle$. Multiple layered modes can be obtained by simply renaming the relations. Moreover we call simply *mode*, denoted by m, the fixed collection of modes for the considered relations, and (global) *timing*, denoted by τ, the fixed collection of (global) timings.

Timings are defined separately on relation symbols, thus the concept extends in a straightforward way to single atoms; however, when we deal with more complex structures such as clauses and queries we have to find a way of combining the timings of the different atoms: for instance in the query $\leftarrow \mathtt{append(list2, list3, Xs)}, \mathtt{append(list1, Xs, Result)}$. we want to be able to state that the timing of the second input position of the second atom is greater than the timing of the third position of the first one (this reflects the fact that the result carried in the third position of the first atom is successively employed by the second atom); clearly if we use fixed timing for each relation symbol this is not possible. Therefore we need to define the concept of *local timing* for a specific clause. A local timing l for a clause (resp. a query) is a function from the clause's (resp. query's) atom positions into N, which is required to be *consistent* with the adopted global timing τ in the sense that l and τ have to determine the same ordering among the positions of each atom of the clause (resp. query). Formally, we have the following definition (where $\mathbf{Q}$ denotes a conjunction of atoms).

Definition 2.4. Let τ be the adopted fixed timing. We say that l is a *local timing* for a clause

$$p_1(t_{1,1},\dots,t_{1,e_1}) \leftarrow p_2(t_{2,1},\dots,t_{2,e_2}), p_3(t_{3,1},\dots,t_{3,e_3}),\dots,p_k(t_{k,1},\dots,t_{k,e_k}).$$

if l is a function from $[1,k]\times[1,\text{Maxargs}]$ into N (where $\text{Maxargs} = \sup\{e_1,\dots,e_k\}$) which satisfies the following *consistency condition*: for each $h \in [1,k]$ and for each $i,j \in [1,e_h]$

$$l(h,i) \leq l(h,j) \text{ iff } \tau_{p_h}(i) \leq \tau_{p_h}(j). \quad (2.1)$$

We say that l is a *local timing* for a query $\mathbf{Q}$ if it is a local timing for $q \leftarrow \mathbf{Q}$, where q is an atom, with no arguments. □

If l is a local timing and $l(i,n) = \alpha$ then we say that the position n of p_i has local timing α. We need one last definition.

Definition 2.5. Let c be a clause (or a query). We call *producing* positions of c the input positions of its head and the output positions of its body atoms. A position of c which is not a producing one will be called *consuming* position. □

We can finally introduce our key concept.

Definition 2.6. Let m and τ be the given mode and timing. We say that a clause (or a query) is well-moded (wrt m and τ) iff there exists a local timing for it which satisfies the following condition

[WM] each variable occurring in a consuming position of local timing α occurs also in a producing position of local timing less than α.

A program is well-moded wrt m if all its clauses are. □

Once a mode m and τ are given it can be checked automatically whether a given program is well-moded wrt m and τ (this issue will be addressed in Section 2.2). Our notion of well-modedness differs from the standard one in [15, 16, 4] in the fact that the ordering among atoms' positions is imposed by the timing, rather than being determined by the left-to-right selection rule. For this reason, as we discuss in the next section, layered modes are more appropriate than standard modes to prove properties of logic languages with dynamic scheduling.

Example 2.1. Consider the following program DELMAX.

```
% del_max(Xs, Zs) ← Zs is obtained from Xs by deleting all the occurrences
%     of its maximum element

  del_max(Xs, Zs) ← find_max_and_del(Xs, Max, Max, Zs).

% find_max_and_del(InList,Max,El,OutList) ← Max is the maximum
%     element of the list of naturals InList, and OutList is obtained from
%     Inlist by deleting all the occurrences of El from it

  find_max_and_del([], 0, _, []).
  find_max_and_del([X | Xs], Max, El, Ys) ←
     find_max_and_del(Xs, Max', El, Zs),
     sup(X, Max', Max),
```

```
    del_if_first([X | Zs], El, Ys).

del_if_first([El | Zs], El, Zs).
del_if_first([X | Zs], El, [X|Zs]) ← X ≠ El.

sup(X, Y, X) ← X ≥ Y.
sup(X, Y, Y) ← X < Y.
```

As usual, the built-ins $\neq$, $\geq$ and $>$ require to be called with ground arguments. If this is not the case in the moment that they are called then a run-time error arise. Now, let us adopt the following layered mode $\langle m_{\mathtt{del}}, \tau_{\mathtt{del}} \rangle$.

```
del_max(in:1, out:2)
find_max_and_del(in:1, out:2, in:3, out:4)
del_if_first(in:1, in:1, out:2)
sup(in:1, in:1, out:2)
≠(in:1, in:1)
```
and the same holds for $\geq$ *and* $<$

In order to check that DELMAX is well-moded wrt $\langle m_{\mathtt{del}}, \tau_{\mathtt{del}} \rangle$. we have to find for each clause in the program a local timing which is consistent with $\tau_{\mathtt{del}}$ and which satisfies [WM] in Definition 6. The task is straightforward and can be done automatically, in particular a local timing for the second clause defining `find_max_and_del` is the following:

```
find_max_and_del([X | Xs]: 1, Max: 6, El: 7, Ys: 12) ←
    find_max_and_del(Xs: 2, Max': 3, El: 8, Zs: 9)
    sup(X: 4, Max': 4, Max: 5)
    del_if_first([X | Zs]: 10, El: 10, Ys: 11).
```

For notational convenience, we have reported the full clause and have inserted in each position the relative local timing. □

We show now that layered modes strictly extend standard modes. On one hand, the notion of well-moded clause (and therefore program) given in [4] can be obtained from ours by assuming that the input and the output positions of any atom always have timing 1 and 2, respectively. The proof of well modedness can be carried out by adopting, for each clause in the program, a local timing l defined as follows (we use here the notation of Definition 4, so k is the number of atoms in the clause while e_h is the arity of the predicate appearing in the h-th atom):

– $l(h,i) = 2 * h$ for each $h \in [1,k]$ and for each $i \in [1, e_h]$, such that $m_{p_h}(i) = \mathsf{in}$,

– $l(h,i) = 2 * h + 1$ for each $h \in [2,k]$ and for each $i \in [1, e_h]$, such that $m_{p_h}(i) = \mathsf{out}$,

– $l(1,i) = 2 * k + 2$ for each $i \in [1, e_1]$ such that $m_{p_1}(i) = \mathsf{out}$.

On the other hand, the program DELMAX cannot be well-moded by using standard modes. In fact, by adopting the definition of well-moded program in [4] we would get the following contradiction: In the first clause defining `del_max` the variable `Max` does not occur in the head and thus the third position of `find_max_and_del` can only be moded as output position; however, this constraint causes the first clause defining `find_max_and_del` to be not well-moded (because of the presence of the variable `_` in the third position of the head).

2.1. Properties of Well-Moded Programs

In the following we extend to layered modes some classical results of well-moded programs. Here and in the sequel we denote by $Var(e)$ the set of variables appearing in the expression e.

Lemma 2.1. An instance of a well-moded query (or clause) is well-moded.

Proof. This follows directly from the definition of well-moded clause and the fact that if $Var(t) \subseteq Var(s_1, \ldots, s_n)$ then $Var(t\theta) \subseteq Var(s_1\theta, \ldots, s_n\theta)$. □

The following result shows the persistence of the property of being well-moded.

Theorem 2.1. Any SLD resolvent obtained from a well-moded query and a well-moded clause (which are variable-disjoint) is well-moded.

Proof. First, we need the following observation.

Claim 1. Let α and β be two functions from $[1, n]$ into N. Let us assume that α and β induce the same ordering over $[1, n]$, i.e. that for all $i, j \in [1, n]$, $\alpha(i) > \alpha(j)$ iff $\beta(i) > \beta(j)$. Then there exists a monotonically increasing injective function $f : Q^+ \to Q^+$ such that, for each i, $\alpha(i) = f(\beta(i))$.

Sketch of the Proof. Since α and β induce the same ordering over $1, \ldots, n$, there exists a permutation $i_1, \ldots, i_n$ of $1, \ldots, n$ such that, for each $j \in [1, n]$, either $\alpha(i_{j+1}) > \alpha(i_j)$ and $\beta(i_{j+1}) > \beta(i_j)$ or $\alpha(i_{j+1}) = \alpha(i_j)$ and $\beta(i_{j+1}) = \beta(i_j)$. The thesis is then immediate from the construction of graph of f.

End of the proof of Claim 1.

We now go on with the proof of the theorem. To simplify the notation in the sequel a clause $q_0(t_1, \ldots, t_{e_0}) \leftarrow q_1(t_{e_0+1}, \ldots, t_{e_1}), q_2(t_{e_1+1}, \ldots, t_{e_2}), \ldots, q_k(t_{e_{k-1}+1}, \ldots, t_n)$ will be denoted by $q_o(t_1, \ldots, t_{e_0}) \leftarrow \mathbf{Q}(t_{e_0+1} \ldots, t_n)$. Moreover, since in the previous notation we use only one index for atom positions, we will consider as local timing one-argument functions of the form $l : [1, n] \to N$ (n is the number of atom positions in the whole clause). Analogously for queries. Further, when we write "a position of (local) timing $< \lambda$" we mean a position whose (local) timing is smaller than the integer number λ.

Let q and cl be respectively the considered query and clause. For the sake of notational simplicity, let us assume that the selected atom is the leftmost one. Let θ be the mgu of the selected atom with the head of cl. By Lemma 2.1, $q\theta$ is well-moded, so there exists a local timing α which satisfies the conditions of Definition 6. Let $q\theta$ be

$$p(t_1 : \alpha(1), \ldots, t_n : \alpha(n)), \mathbf{Q}(r_1 : \alpha(n+1), \ldots, r_m : \alpha(n+m))$$

(here, next to each position we have also reported its local timing, this notation will be used in the sequel as well). Again, $cl\theta$ is well-moded as well, so there exists an appropriate local timing β for it. Let $cl\theta$ be

$$p(t_1 : \beta(1), \ldots, t_n : \beta(n)) \leftarrow \mathbf{B}(s_1 : \beta(n+1), \ldots, s_l : \beta(n+l))$$

Now, we need the following.

Claim 2. It is not restrictive to assume that $\alpha(1) = \beta(1), \ldots, \alpha(n) = \beta(n)$.

Sketch of the Proof. α and β must induce the same ordering on the positions of $p(t_1, \ldots, t_n)$ (by (2.1) in Definition 4) this ordering has to coincide with the ordering induced by the global timing τ). So, by Claim 1 there exists a monotonic function $f : Q^+ \to Q^+$ such that for $i \in [1, n]$, $\alpha(i) = f(\beta(i))$. Now, it follows that for $i \in [1, n]$, $f(\beta(i))$ is an integer while for $i \in [n+1, n+l]$, $f(\beta(i))$ is in Q^+, therefore, if we let δ be the least common denominator of $f(\beta(n+1)), \ldots, f(\beta(n+l))$, we have that, for $i \in [1, n]$, $\delta * \alpha(i) = \delta * f(\beta(i))$, and that, if we multiply each $\alpha(i)$ by δ and we apply $\delta * f$ to each $\beta(i)$, we obtain two local timings which satisfy the claim. *End of the proof of Claim 2.*

Now, for the resolvent of q and cl we'll adopt the following natural local timing

$$\mathbf{B}(s_1 : \beta(n+1), \ldots, s_m : \beta(n+l)), \mathbf{Q}(r_1 : \alpha(n+1), \ldots, r_m : \alpha(n+m))$$

Clearly, since both α and β satisfy (2.1) in Definition 4, this labeling satisfies (2.1) as well, thus, according to Definition 4, it is indeed a local timing. Therefore, now we only have to show that the above local timing satisfies condition [WM] of Definition 6. Recall that

(a) α, when applied to $q\theta$, satisfies condition [WM], and that
(b) β, when applied to $cl\theta$, satisfies condition [WM].

Let x be a variable occurring in an input position of $\mathbf{B}, \mathbf{Q}$ of label λ. We have to prove that

$$x \in \mathit{VarOut}(\mathbf{B}, \lambda) \cup \mathit{VarOut}(\mathbf{Q}, \lambda) \tag{2.2}$$

here and in the sequel the expression $\mathit{VarOut}(\mathbf{B}, \lambda)$ (resp. $\mathit{VarIn}(\mathbf{B}, \lambda)$) denotes the set of variables occurring in the output (resp. input) positions of label less than λ in set of atoms $\mathbf{B}$. We now distinguish two cases.

Case 1. x occurs in an input position of $\mathbf{Q}$ of label λ. Then, by (a) we have that
$x \in \mathit{VarOut}(p(t_1, \ldots, t_n), \lambda) \cup \mathit{VarOut}(\mathbf{Q}, \lambda)$
Now, if x occurs in $\mathit{VarOut}(\mathbf{Q}, \lambda)$ then (2.2) holds and we are finished. Otherwise x occurs in $\mathit{VarOut}(p(t_1, \ldots, t_n), \lambda)$. In this case, let λ' be the least label such that x occurs in an output position of $p(t_1, \ldots, t_n)$ of label λ'. Then, by definition, we know that $\lambda' < \lambda$. By (b) we have that
$x \in \mathit{VarIn}(p(t_1, \ldots, t_n), \lambda') \cup \mathit{VarOut}(\mathbf{B}, \lambda')$
By (a), $\mathit{VarIn}(p(t_1, \ldots, t_n), \lambda') \subseteq \mathit{VarOut}(p(t_1, \ldots, t_n), \lambda') \cup \mathit{VarOut}(\mathbf{Q}, \lambda')$, thus
$x \in \mathit{VarOut}(p(t_1, \ldots, t_n), \lambda') \cup \mathit{VarOut}(\mathbf{Q}, \lambda') \cup \mathit{VarOut}(\mathbf{B}, \lambda')$
By the minimality of λ', $\mathit{VarOut}(p(t_1, \ldots, t_n), \lambda') = \emptyset$, and, since $\lambda' < \lambda$ we have that $\mathit{VarOut}(\mathbf{Q}, \lambda') \subseteq \mathit{VarOut}(\mathbf{Q}, \lambda)$ and that $\mathit{VarOut}(\mathbf{B}, \lambda') \subseteq \mathit{VarOut}(\mathbf{B}, \lambda)$. Hence (2.2) holds.

Case 2. x occurs in an input position of $\mathbf{B}$ of label λ (this case is perfectly symmetrical to the previous one). Then, by (b) we have that
$x \in \mathit{VarIn}(p(t_1, \ldots, t_n), \lambda) \cup \mathit{VarOut}(\mathbf{B}, \lambda)$
Now, if x occurs in $\mathit{VarOut}(\mathbf{B}, \lambda)$ then (2.2) holds and we are finished. Otherwise x occurs in $\mathit{VarIn}(p(t_1, \ldots, t_n), \lambda)$. In this case, let λ' be the least label such that x occurs in an input position of $p(t_1, \ldots, t_n)$ of label λ'. Then, by definition, we know that $\lambda' < \lambda$. Further, by (a) we have that
$x \in \mathit{VarOut}(p(t_1, \ldots, t_n), \lambda') \cup \mathit{VarIn}(\mathbf{Q}, \lambda')$
By (b), $\mathit{VarOut}(p(t_1, \ldots, t_n), \lambda') \subseteq \mathit{VarIn}(p(t_1, \ldots, t_n), \lambda') \cup \mathit{VarOut}(\mathbf{B}, \lambda')$, thus

$x \in VarIn(p(t_1,\ldots,t_n),\lambda') \cup VarOut(\mathbf{B},\lambda') \cup VarOut(\mathbf{B},\lambda')$
By the minimality of λ', $VarIn(p(t_1,\ldots,t_n),\lambda') = \emptyset$, and, since $\lambda' < \lambda$ we have that $VarOut(\mathbf{Q},\lambda') \subseteq VarOut(\mathbf{Q},\lambda)$ and that $VarOut(\mathbf{B},\lambda') \subseteq VarOut(\mathbf{B},\lambda)$. Hence (2.2) holds, thus completing the proof. □

The following result provides an operational explanation, in terms of computed answers, of the intuitive idea that in an atom's definition *"the output positions of timing α depend on the input positions of timing smaller than α"*.

Theorem 2.2. Let P be a well-moded program, Q be a (not necessarily well-moded) unit query, and σ be a computed answer substitution for Q in P. Suppose that t is a term occupying an output position of global timing λ in Q. Then

$$Var(t\sigma) \subseteq Var(t_1\sigma,\ldots,t_n\sigma)$$

where $t_1,\ldots,t_n$ are the terms occupying the input positions of Q whose timing is smaller than λ.

Proof. Note that the previous statement can be equivalently formulated by saying that $Q\sigma$, considered as a unit clause, is well-moded (according to Definition 6). We will then use this terminology to shorten the notation. The proof is given by induction on the length of the derivation ξ that returned the computed answer substitution σ.

Base case: ξ has length 1. In this case, there exists a unit clause H such that $\sigma' = mgu(Q,H)$ and $Q\sigma = Q\sigma'$. Since H is well-moded (as a unit clause), by Lemma 2.1 also $H\sigma'$ is well-moded. The result follows then immediately from the fact that $Q\sigma = Q\sigma' = H\sigma'$ and the fact that the same mode m applies to Q and H, since they share the same predicate symbol.
Inductive step: ξ has length $n > 1$. In this case, there exist:
- a well-moded clause $H \leftarrow A_1,\ldots,A_n$ in P,
- an mgu σ_0 of Q and H,
- derivations $\xi_1,\ldots,\xi_n$, starting in $A_1\sigma_0, A_2\sigma_0\sigma_1,\ldots,A_n\sigma_0\sigma_1\ldots\sigma_{n-1}$, each one of length less than n, with computed answer substitutions $\sigma_1,\sigma_2\ldots,\sigma_n$,

such that $Q\sigma = H\sigma_0\sigma_1\cdots\sigma_n$.

By the inductive hypothesis, for $i \in [1,n]$ $A_i\sigma_0\sigma_1\ldots\sigma_i$ is well-moded if considered as a unit clause. Again by Lemma 2.1 we have that, for each i, $A_i\sigma_0\sigma_1\cdots\sigma_n$ is well-moded (as a unit clause) and the clause and $cl:\ (H \leftarrow A_1,\ldots,A_n)\sigma_0\sigma_1\cdots\sigma_n$ is well-moded. Therefore, from the definition of well-moded clause and a straightforward induction on the positions of the atoms in the clause cl, it follows that also $H\sigma_0\sigma_1\cdots\sigma_n$ is well-moded (as a unit clause). Then, analogously to the previous case, the thesis follows from the fact that $H\sigma_0\sigma_1\cdots\sigma_n = H\sigma = Q\sigma$ and the fact that H and Q share the same mode m. □

In the sequel, in presence of the layered mode $\langle m,\tau\rangle$, we call *minimal positions* of an atom those positions which have the lowest global timing Consequently, the *input minimal positions* of an atom (resp. a query) will be those of its positions which are at the same time input and minimal positions. The following observation is useful.

Proposition 2.1. Let $\mathbf{Q}$ be a well-moded query. Then there exists an atom in $\mathbf{Q}$ whose input minimal positions are filled in by ground terms.

Proof. First notice that the condition is immediately satisfied if there exists an atom in $\mathbf{Q}$ that has *no* input minimal positions. Now, by the definition of well-moded query, there exists a local timing l for $\mathbf{Q}$ which satisfies (2.1) in Definition 4 and the requirements of Definitions 4 and 6. Since the local timing and the global timing induce the same ordering on atoms' positions, from Definition 6 it follows that those positions of $\mathbf{Q}$ that have lowest local timing and that are also input positions are filled in by ground terms. So, let us consider an atom A of $\mathbf{Q}$ such that A contains one of the positions of $\mathbf{Q}$ that have lowest local timing. We have that the input minimal positions of A are also input positions of $\mathbf{Q}$ which have lowest local timing, thus, they are filled in by ground terms. By (2.1), we have the thesis. □

Finally, also for layered modes we have a classical property of well-moded programs, consisting in the fact that the computed instance of a well-moded query is ground.

Corollary 2.1. Let P be a well-moded program and $\mathbf{Q}$ be a well-moded query. Then for every computed answer σ of $P \cup \{\mathbf{Q}\}$, $\mathbf{Q}\sigma$ is ground.

Proof. It follows from Theorem 2.2 by observing that any atom in $\mathbf{Q}\sigma$ is well-moded, if considered as a unit clause. Since $\mathbf{Q}\sigma$ is well-moded as a query, it is straightforward to check that this implies that $\mathbf{Q}\sigma$ is ground. □

2.2. Checking Well-Modedness

Clearly checking whether a program is well-moded is decidable. In the following we an algorithm which, given a mode and a global timing, checks whether a clause is well moded. Even though this algorithm has an exponential upper bound, as we discuss later in almost all practical cases the algorithm runs in polynomial time.

Proposition 2.2. The problem of checking whether a clause is well moded can be solved by an algorithm whose worst case running time is $O(v * n^2 * k)$, where

n *is the number of positions of cl*

v *is the number of variables of cl*

$$k = \prod_{X \in Var(cl)} \text{number of occurrences of } X \text{ in producing positions of } cl$$

Proof. Consider a clause $cl:\ p(t_1, \dots, t_l) \leftarrow \mathbf{B}(t_{l+1}, \dots, t_n)$. In order to check that cl is well-moded we have to check if there exist $\beta_1, \dots, \beta_n$ such that the labeling
$p(t_1 : \beta_1, \dots, t_l : \beta_l) \leftarrow \mathbf{B}(t_{l+1} : \beta_{l+1}, \dots, t_n : \beta_n)$
satisfies conditions (2.1) of Definition 4 (i.e. it is a local timing), and [WM] of Definition 6. We do that by building a system S of equalities and inequalities on $\beta_1, \dots, \beta_n$ such that cl is well-moded iff S has a solution.

We start with the initialization $S := \emptyset$. The algorithm is then divided into two phases.

Phase 1. First we add to S the equalities and inequalities needed to ensure that $\beta_1, \dots, \beta_n$ form a local timing for cl. For each atom $p(t_i : \beta_i, \dots, t_{i+k} : \beta_{i+k})$ of cl (the atom can be either in the body or the head), we have to do the following. Assume that the global timing for the relation symbol p is $p(\alpha_i, \dots, \alpha_{i+k})$, where for the sake of simplicity we assume that $\alpha_i, \dots, \alpha_{i+k}$ are already in increasing order. Then for each $j \in [i, i+k-1]$ we have to consider two possibilities

– if $\alpha_j = \alpha_{j+1}$ then $S := S \cup \{\beta_j = \beta_{j+1}\}$
– if $\alpha_j < \alpha_{j+1}$ then $S := S \cup \{\beta_j < \beta_{j+1}\}$
Notice that if p has arity less than 2 then we don't have to add anything to S. The running time of this phase is $O(n)$, where n is the number of positions of cl, Moreover, after this first phase, the set S contains less than n elements.

Phase 2. This part is divided into four sub-phases. In the sequel we denote by $cl(i)$ the i-th position of cl, i.e. the position that contains t_i.

(2a) For each variable X of cl we choose a producing position $cl(i_X)$ in which it occurs and we call such a position the origin of X. If no such positions exists then the clause is not well-moded, since condition [WM] is not satisfied.

(2b) For each variable X with origin $cl(i_X)$, and for each occurrence of X in a consuming position $cl(j)$ we add to S the inequality $\beta_{i_X} < \beta_j$: so
– $S := S \cup \{\beta_{i_X} < \beta_j\}$
The running time of this sub-phase is $O(v * n)$, and S contains now less than n equalities and less than $v * n + n$ elements.

(2c) We eliminate from S all the equalities and all the duplicates: For each equality $\beta_i = \beta_j$ present in S, we eliminate it from S and we replace in S all the occurrences of β_i by β_j. The running time of this phase is $O(v * n^2)$.

(2d) We are now left with a set S of inequalities of the form $\beta_i < \beta_j$. It follows immediately from the construction of S that S has a solution iff cl is well-moded. Now, checking that S has a solution is equivalent to checking that S does not contain a subset of the form $\{\beta_{j_1} < \beta_{j_2}, \beta_{j_2} < \beta_{j_3}, \ldots, \beta_{j_r} < \beta_{j_{r+1}}\}$ where β_{j_1} coincides with $\beta_{j_{r+1}}$. This can be checked in linear time on the number of the equations: In fact S can be seen as a directed graph by considering each inequality $\beta_i < \beta_j$ as a directed edge. Therefore checking whether S has a solution corresponds to checking whether S contains no nontrivial strongly connected components. As well known [11] this problem can be solved by a linear time algorithm, more precisely by an algorithm whose running time is $O(V + E)$ where V is the number of vertices (in our case, n), and E is the number of edges (in our case, less than $n * v$) in the graph. Thus the running time of this sub-phase is $O(n * v)$.

If S does *not* have a solution, then we have to go back to step (2a), choose a different set of producer positions for the variables of cl, and go through steps (2b), (2c) and (2d) again. If all possible origins have been tried unsuccessfully, then the program is not well-moded. The number of times we have to go through this loop is given by k, as defined before, which coincides with the product of the number of possible origin for each variable. Thus the total running time of the algorithm is $O(v * n^2 * k)$. □

Theoretically k could be as high as n^v, as follows immediately from its definition. However, in practice this exponential bound does not arise, indeed k is almost always 1. In fact, for *nicely moded programs* [3], which are programs where a variable cannot occur in more than one producing position, k is always equal to 1. In [3] it is also shown that the vast majority of programs is nicely moded. Further, even for non-nicely moded programs, the case of a variable that occurs in more than one producing position is an exception to normal programming style. Thus, in practical situations, k is bound to be very small and checking well-modedness can be performed in polynomial time.

3. APPLICATIONS OF LAYERED MODES

In this Section we show some typical applications in which layered modes improve on standard modes.

3.1. Proving Absence of Deadlocks

As mentioned in the introduction, various logic programming languages use a flexible scheduling in which the selection of atoms in a goal can be dynamically delayed until its arguments are sufficiently instantiated. This is obtained by using *delay declarations* (Naish, [24]). Delay declarations provide the programmer with a powerful control mechanism, allowing to run the calls more efficiently, to prevent run-time errors (e.g. floundering) and to enforce termination. Analogously to the case of concurrent languages, the presence of delay declarations might cause *deadlocks*, i.e. the computation can reach a stage in which all the atoms are delayed. In this case no atom can be selected and the computation cannot proceed, which is an undesirable situation. Recently, standard modes have been used by Apt and Luitjes [2] to provide conditions which ensure absence of deadlocks in logic programs with delay declarations. In this section we extend their result by using layered modes.

In the sequel we follow loosely the syntax of the language Gödel [2] and consider declarations of the form

```
delay A until Condition
```

where `A` is an atom while `Condition` is a formula in some assertion language which we do not further specify, since it is not relevant to our results. The meaning of such a declaration is that an instance `A`θ of `A` can be selected in a resolvent only if `Condition`θ holds. If `Condition`θ does not hold then `A`θ cannot be selected and is suspended. For example, `delay p(X) until ground(X)` means that the atom $p(t)$ can be selected in a SLD derivation only if t is a ground term. We say that an atom `B` is *suspended* if there exists a delay declaration `delay A until Condition` and a substitution θ, $Dom(\theta) = Var($`A`$)$ such that `A`$\theta =$ `B` and `Condition`θ does not hold. In the following we will consider only SLD derivations in which only atoms which are not suspended are selected and we will call these derivations *safe*. Then we define formally the notion of deadlock as follows.

Definition 3.1. Let P be a program with delay declarations. We say that a query $\mathbf{Q}$ *deadlocks* in P iff there exists a safe SLD derivation for $\mathbf{Q}$ in P which ends in a resolvent containing only suspended atoms. □

We also need the concept of well-moded delay declaration.

Definition 3.2.
A delay declaration `delay A until Condition` is *well-moded* if, for each θ, `Condition`θ holds whenever all the input minimal positions of `A`θ are ground terms.
A program with delay declarations is well-moded iff all its clauses and its delay declarations are. □

The most commonly used delay declarations consist of conjunctions and disjunctions of the predicates `nonvar` and `ground`, and thus are likely to be well-moded (w.r.t. an appropriate mode). Now we have the following.

Theorem 3.1 Absence of Deadlocks. Let P be a program with delay declarations and $\mathbf{Q}$ be a query. If P and $\mathbf{Q}$ are well-moded then $\mathbf{Q}$ does not deadlock in P.

Proof. Let $\mathbf{R}$ be any resolvent in the derivation starting at $\mathbf{Q}$. Theorem 2.1 implies that $\mathbf{R}$ is well-moded. Then by Proposition 2.1, $\mathbf{R}$ contains an atom A whose input minimal positions are filled in by ground terms (note that this is true also if A does not contain input minimal positions). Now, by Definition 2 we have that A is not suspended. Since this applies to any resolvent $\mathbf{R}$, this completes the proof. □

Since layered modes strictly extend standard modes, using layered modes one can prove deadlock-freedom for a larger class of programs. The following example shows a program whose deadlock-freedom can be proven by using layered modes and not using standard modes.

Example 2.1 (Part 2) Let us associate with the program `DELMAX` of Example 2.1 the following delay declarations:

```
d1: delay del_if_first([X|Xs], El, _) until ground(X)  ∧  ground(El)
d2: delay sup(X, Y, _) until ground(X)  ∧  ground(Y)
```

Notice that the above delay declarations are both well-moded.

The first declaration is needed not only to avoid run-time errors of the call $\mathtt{X} \neq \mathtt{El}$ but also in order for the program to run efficiently. In fact the query `del_max(Xs, Zs)` could return the list `Zs` in linear time (scanning `Xs` only once), however, it is easy to see that if we employ any fixed search strategy (like PROLOG's search strategy) and no delay declarations, the program has to go through a remarkable amount of backtracking, which makes it run in quadratic time on the length of the input list. The purpose of the second delay declaration is to ensure that the program does not run into a run-time error. In fact, since `sup` uses the built-in's $\geq$ and $<$, we may not select an atom of the form $\mathtt{sup}(\mathtt{X},\mathtt{Y},\mathtt{Z})$ unless `X` and `Y` are already ground terms. In this case, Theorem 3.1 proves that, if `InList` is a ground list, the query `del_max(InList, OutList)` in the program `DELMAX` does not deadlock. This property cannot be proven by using standard modes (i.e. by using Theorem 19 in [2]) since, as previously mentioned, the program `DELMAX` cannot be well-moded. □

3.2. Handling Complex Data Structures

The ability of dealing with complex data structures such as difference-lists is another aspect in which layered modes improve on the classical ones. In this section we provide an example of how a program that uses difference-lists may be well-moded.

Difference-lists are a data structure for representing sequence of elements, and are used for writing efficient list-processing logic programs. Their use has become a standard advanced technique for logic programmers (see [31, 25]) and is essential in many programming situations: for instance, appending one list to another one requires linear instead of constant time if we don't use difference-lists. An example of their usefulness is given by the following solution of (a simplified version of) Dijkstra's Dutch Flag Problem, which reads as follows: given a list of objects which are either red, white or blue, we have to rearrange it in such a way that the red elements appear first, the white ones second and the blue ones third. The following program DUTCH is taken from [25, page 117].

```
dutch(InList, RWBs) ←
   distribute(InList, RWBs, WBs, WBs, Bs, Bs, []).

distribute([], Rs, Rs, Ws, Ws, Bs, Bs).
distribute([X|Xs], [X|Rs1], Rs, Ws0, Ws, Bs0, Bs) ← red(X),
   distribute(Xs, Rs1, Rs, Ws0, Ws, Bs0, Bs).
distribute([X|Xs], Rs0, Rs, [X|Ws1], Ws, Bs0, Bs) ← white(X),
   distribute(Xs, Rs0, Rs, Ws1, Ws, Bs0, Bs).
distribute([X|Xs], Rs0, Rs, Ws0, Ws, [X|Bs1], Bs) ← blue(X),
   distribute(Xs, Rs0, Rs, Ws0, Ws, Bs1,Bs).
```

where we assume that predicates red, white and blue are appropriately defined elsewhere in the program. Let us adopt the following mode and timing (m and τ):

dutch(in:1, out:2).
distribute(in:1, out:6, in:5, out:4, in:3, out:2, in:1).
red(in:1) *and the same applies to* white *and* blue.

In order to check that the above program is well-moded w.r.t. the above mode we have to find a local timing for its clauses which satisfies (2.1) in Definition 4 and [WM] in Definition 6. The task is quite simple, a possible local timing for the first three clauses is the following:

```
dutch(InList:1, RWBs:8) :-
   distribute(Inlist:2,RWBs:7,WBs:6, WBs:5,Bs:4,Bs:3,[]:2).

distribute([]:2,Rs:7,Rs:6,Ws:5,Ws:4,Bs:3,Bs:2).
distribute([X|Xs]:1,[X|Rs1]:12,Rs:9,Ws0:8,Ws:5,Bs0:4,Bs:1) ←
   red(X:1),
   distribute(Xs:2,Rs1:11,Rs:10,Ws0:7,Ws:6,Bs0:3,Bs:2).
```

Therefore, DUTCH is well-moded. This program may well be used in combination with a list-generator on a producer-consumer basis, in which we employ a goal such as

```
goal :- producer(List), dutch(List,OrderedList).
```

In situations like this one it might turn out useful to run producer and dutch in parallel, or to employ a form of coroutining in the execution of the two procedures. In these cases the variable List is regarded as an asynchronous communication channel between processes. Now, we have to force dutch to behave as a consumer i.e. to forbid it from scanning List beyond the point till where producer has instantiated it. For this we may naturally use the following delay declarations.

```
delay red(X) until ground(X)
```

(together with similar ones for white and blue). In this case (assuming that producer is well-moded w.r.t. the mode producer(out : 1), i.e. assuming that it is indeed a producer, and that it doesn't deadlock on its own) we can prove that no SLD-derivation starting in goal ends in a deadlock situation.

3.3. Proving absence of floundering

We conclude this section by showing how the previous concepts can be immediately extended to programs with negation and applied to prove absence of floundering. Here we assume that the reader is familiar with the concepts of Negation as Failure, SLDNF resolution, and floundering [20]. It is well-known that in order to maintain the correspondence between declarative and operational semantics, the SLDNF (SLD + Negation as Failure rule) rule is allowed to select solely *ground* negative literals. Consequently an SLDNF resolution can *flounder*, i.e. it can reach a state in which the current goal contains only nonground negative literals. In this case the resolution process stops without providing an answer. This situation, similar to deadlock, is also undesirable. Modes have already been employed (see Apt-Pellegrini [3] and Stroetmann [32]) in order to prove that – under suitable conditions – well-moded programs do not flounder. We now show that this applies also to programs which admit a layered mode.

First, we have to extend to programs with negation the definitions we have provided so far. This is simply done by disregarding the negation operator: thus, in order to check that a program is well-moded we have to remove from it all the negation operators, and check if the resulting definite program is well-moded. Now, we need a new concept.

Definition 3.3. We say that a relation symbol is *flat* if all its positions are input minimal positions. □

Thus, of the relations we have so far encountered, $\neq$, $\geq$, $>$, `red`, `white`, and `blue` are flat (actually, $\neq$ incorporates already a negative element: declaratively, it is a shorthand for $\neg =$). Following the same notation we say that an atom is flat if its relation symbol is flat and that the negative literal $\neg A$ is flat iff A is. We can now directly state the following.

Theorem 3.2 Absence of Floundering. Let P be a well-moded normal program and $\mathbf{Q}$ be a well-moded normal query. If P and $\mathbf{Q}$ contain only flat negative literal then no SLDNF derivation of $P \cup \mathbf{Q}$ flounders.

Proof. It is an immediate consequence of Proposition 2.1. □

4. RELATED WORK

Among the related papers, a recent one by Apt and Luitjes [2] also uses modes to prove deadlock freedom for logic programs with delay declarations. Indeed, our Theorem 3.1 is an extension of [2, Theorem 19]. Our result improves on [2, Theorem 19] in the fact that it can handle programs which use data structures such as difference-lists and – more importantly – in the fact that it can handle generic selection rules: [2] allows to prove deadlock freedom limitedly to programs and queries which generate derivations in which the leftmost atoms is never suspended. In presence of dynamic scheduling this is an important limitation, since if the leftmost atom is suspended then other atoms can be selected.

Another related paper is the one by Ueda and Morita [34] which introduces a mode system for the language Flat GHC (also called FGHC and defined by Ueda in [33]). However, the scope of the modes in [34] is disjoint from ours since they were devised in order to prove safety of unification for FGHC rather than deadlock freedom. The basic result in [34] shows that a well-moded query evaluated in an well-moded program does not cause a unification failure. This result is not suitable to prove deadlock freedom, in fact it is not difficult to find a program which is well-moded according to [34] and deadlocks.

The PhD thesis of Z. Somogy [29] (one of the coauthors of Mercury) presents an "extended mode checking algorithm" system which accepts programs with difference-lists. This system does not allow to draw the same conclusions that layered modes do; in fact practically all the results of this paper (Lemma 2.1 and Theorems 2.2 and 3.1) cannot be obtained using the mode system of [29][1]. Boye and Małuszyński [6] have also addressed the problem of proving deadlock-freedom of logic programs using as verification tool *directional types*. The main difference between our approach and the one in [6] lies in the very definition of deadlocked derivation. In [6] the key concept is provided by the definition of type-driven resolution: textually "the idea is to suspend *unification* when the arguments are not correctly typed". Here "suspension of unification" must not be confused with "suspension of resolution": the search in the search tree is in any case carried out until failure occurs or until the current goal is empty. Thus [6] follows an approach which is substantially different from [24] (and thus from ours) in the fact that the resolution of an atom is never actually suspended.

Related to the present work are also several papers which use abstract interpretation techniques. Among them we should first mention Mellish [23] and Debray [14] which investigated the automatic derivation of (standard) mode information. Techniques similar to those used in these papers could also be used to infer layered modes, possibly integrating the information provided by the programmer.

Codish et al. [9] and Codognet et al. [10] used abstract interpretation for proving deadlock freedom of concurrent logic languages. The advantage of these techniques w.r.t. systems based on modes is the fact that they are completely automatic. However, this is also the reason for their imprecision in some cases, especially when considering real, large programs. In fact, the need of ensuring termination within a reasonable time[2] forces one to use further abstractions (e.g. widening operations) which induce a further loss of precision. As an example, consider the following *ancestor* program which, in the context of concurrent logic programming, implements the so-called short circuit technique [28]:

```
p(X,Y) ← p(X,Z),p(Z,Y)
p(X,X).
test(a).

delay  test(X) until ground(X)
```

Using Theorem 3.1 it is immediate to prove the deadlock freedom of the query `p(a,X),test(X)` in the above program, since both the program and the goal are well-moded by considering the (standard) mode `p(in,out)` and `test(in)`. However, the suspension analysis of Codish et al. [9], as mentioned by the authors

[1] Consider for instance $\mathtt{P} = \{\mathtt{p} \leftarrow \mathtt{eq(X,Y)}, \mathtt{eq(Y,X)}.$ `delay eq(X,Y) until ground(X)`$\}$ together with the mode `mode eq(INBOUND, OUTBOUND)`. This program is well-moded in the sense of [29], yet clearly the query `p` deadlocks in `P`.

[2] Clearly, from a practical point of view, analyses which terminate in an exponential time are not feasible.

themselves, does not allow to obtain this result. The reason is that, in order to ensure termination, an approximation is needed which ensures that the number of atoms appearing in the abstract goal descriptions does not exceed a fixed bound[3].

Floundering analysis (similar to deadlock analysis) has been studied in [22] and [5]. The results obtained in these papers cannot be applied to logic programs with dynamic scheduling, since their correctness relies on the fact that the evaluation of negated literals cannot instantiate any variable, while this is not the case for atoms which are delayed.

Some recent papers by Marriott et al. [21], Debray et al. [13] and Garcia de la Banda et al. [12] investigate global analysis for logic programs with dynamic scheduling. Debray et al. [13] introduce an analysis for determining parts of the program which are free from "delays" and "wakeups", thus allowing optimizations in the compilation of these parts. The analyzer is not very precise since as soon as one atom may delay the information associated with the variables of that atom is lost. Both [21] and [12] are not concerned with deadlock analysis, so we cannot directly compare our results with theirs.

5. CONCLUSIONS

In this paper we have introduced *layered modes*, a generalization of the standard definition of modes for logic programs (e.g. see [15, 16, 4]). Our generalization is particularly appropriate to deal with programs which employ dynamic selection rules, since it allows us to enlarge the class of programs which can be well-moded while maintaining the simplicity of the standard mode systems. In fact, the programmer is required only to annotate the relation symbols which appear in the program by saying which output positions depend on which input positions and she/he is never required to worry about the relations existing among different atoms occurring in a conjunction (e.g. the order of instantiation of variables).

Once the relation symbols are annotated, it can be checked automatically whether there exists a consistent annotation for the program, i.e. whether the program is well-moded. This allows to verify automatically several program properties including deadlock-freedom, proof of termination and absence of run-time errors (for Prolog built-ins). Modes are useful also for low-level code optimization (e.g. detection of functionality, generation of specialized unifications instructions etc.) [14]. In Mercury, modes and – on top of them – types and determinism declarations, are heavily used for improving program's performance, for reducing program's memory requirement and as a reliable debugging tool [18].

It is also worth noticing that modes allow for a direct compositional and incremental verification and validation methodology, since the notion of well-modedness is defined locally on single clauses. So, for example, if the modules P and Q can be proven to be deadlock-free by using Theorem 3.1, then the module $P \cup Q$ resulting from their composition is deadlock-free as well. This is an important feature, since the modular design is by now an important software development technique also in the field of logic programming, and should be contrasted with the case of

[3]From a theoretic point of view this problem could be fixed by introducing a more refined notion of state description [17]. However, such a representation in general can be very large, thus making the analysis unfeasible (for the ancestor program the size of goal descriptions would be exponential in the number of (parallel) derivation steps)

modular analysis based on abstract interpretation. Such an analysis requires a suitable (compositional) semantic basis which is often very expensive to compute. Therefore, in order to obtain practical tools, further abstractions (e.g. widening) are needed which make the modular analysis rather imprecise [8, 17].

Several extensions of layered modes are possible.

Firstly, the idea of associating a timing to atoms' positions can be easily applied also to types, thus providing a "layered" extension of the existing type systems for logic programs. This could be useful to extend to logic programs with dynamic scheduling several existing proof methods for logic programs based on types.

Secondly, our results can also be applied to constraint logic programs (CLP), to this end, it is sufficient to provide a mode to each constraint symbol. In particular we can easily extend Theorem 3.1 to CLP with delay declarations. This is an interesting application, since CLP systems use often delay declarations to postpone the evaluation of "difficult" constraints (e.g. non-linear constraints when considering numerical domains) [4].

Thirdly, layered modes could be used directly to prove absence of deadlock for concurrent (constraint) logic languages. To illustrate the idea let us consider an FGHC-like concurrent language whose clauses have the form $p(X_1, \ldots, X_n) \leftarrow \mathbf{G} \mid \mathbf{B}$ where $\mathbf{G}$, the guard, consists of a conjunction of built-ins while $\mid$ represents the commit operator. A predefined interpretation for built-ins is given such that, for each substitution θ, $\mathbf{G}\theta$ either succeeds, fails or suspends[5]. We say that an atom $p(t_1, \ldots t_n)$ fails, suspends or succeeds w.r.t. a clause like the previous one if, for $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$, $\mathbf{G}\theta$ fails, suspends or succeeds, respectively. In case $p(t_1, \ldots t_n)$ succeeds it can be rewritten into the body $\mathbf{B}\theta$ (and no backtracking is allowed because of the commit operator). A deadlock arises when all the atoms in the current resolvent are either failed or suspended wrt all the clauses in a program, and there is at least one suspension. In order to apply Theorem 3.1 to such a language we have to modify the definition of well-moded clause in order to take into account also the guards' evaluation mechanism. All what we have to do is to add to Definition 6 the condition *"if the input minimal positions of* $p(X_1, \ldots X_n)\theta$ *are ground terms then* $\mathbf{G}\theta$ *does not suspend"*, which is the counterpart of the one contained in Definition 2. This is sufficient to prove that a well-moded program is deadlock free. The same kind of reasoning can easily be applied to concurrent constraint programming (ccp) [27]. It is worth noticing that proving absence of deadlocks would not ensure absence of "unification failure". This however is the case for all the systems proposed in the literature for proving deadlock freedom of concurrent (constraint) logic languages.

To conclude, let us mention that one of the crucial aspects of the implementation of logic languages with dynamic scheduling is determining *when* the interpreter should try to awaken a delayed or sleeping process. In fact the process of checking whether the delaying conditions are still (un)satisfied often requires a sensible amount of CPU time. It turns out that for well-moded programs we can always find

[4] Note that in this case a deadlock would denote also a situation in which the only delayed atoms are constraints. In such a case usually the CLP system give the computed constraints as output together with the answer "may be", since it is not sure whether the delayed constraints are satisfiable.

[5] As usual, suspension denotes absence of sufficient input: for example $3 < 5$ succeeds, $3 < 2$ fails and $3 < X$ suspends.

a search strategy that will always select atoms that do not need to be suspended. In fact, as it is explained in the proof of Theorem 3.1, if we select the atom of the current goal which contains one of the positions with the lowest local timing, we are sure that we can proceed without having to check whether its delay declaration is satisfied or not. Of course, using such a selection strategy at runtime is likely to be inconvenient, as we'd have to recalculate the input minimal positions at each resolution step. Future work includes investigating the possibility of devising a method based on layered modes and abstract interpretation for determining a selection rule specific for a target program, with the goal of minimizing atom's suspensions. This could be integrated with existing algorithms [23, 14] for the static inferences of (standard) mode information, in order to minimize the amount of information required to the programmer.

REFERENCES

1. K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.
2. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In A. Borzyszkowski and S. Sokolowski, editors, *Proc. of the Fourth Int'l Conference on Algebraic Methodology and Software Technology*, LNCS , Berlin, 1995. Springer-Verlag.
3. K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994.
4. K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from Modes through Types to Assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.
5. R. Barbuti and M. Martelli. Recognizing Non-Floundering Logic Programs and Goals. *Int'l Journal on Foundations of Computer Science*, 1(2):151–163, 1990.
6. J. Boye and J. Małuszyński. Directional types and the annotation method. *Journal of Logic Programming*, 33:179–220, 1997.
7. M. Carlsson. *SICStus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, 1988.
8. M. Codish, S. K. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *Proc. Twentieth ACM Symp. on Principles of Programming Languages*, pages 451–464. ACM Press, 1993.
9. M. Codish, M. Falaschi, and K. Marriott. Suspension Analyses for Concurrent Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.
10. C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation for Concurrent Logic Languages. In S. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 215–232. MIT Press, 1990.
11. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* The MIT Press, Cambridge, Mass., 1990.
12. M. Garcia de la Banda, K. Marriott, and P. Stuckey. Efficient analysis of logic programs with dynamic scheduling. In J. Lloyd, editor, *Proc. Twelfth Int'l Logic Programming Symp.*, pages 417–431. MIT Press, 1995.

13. S. Debray, D. Gudemann, and P. Bigot. Detection and optimization of suspension-free logic programs. In M. Bruynooghe, editor, *Proc. Eleventh Int'l Logic Programming Symp.*, pages 487–504. MIT Press, 1994.

14. S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–481, 1989.

15. P. Dembinski and J. Maluszynski. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the Int'l Symp. on Logic Programming*, pages 29–38, 1985.

16. W. Drabent. Do logic programs resemble programs in conventional languages ? In E. Wada, editor, *Proc. Int'l Symp. on Logic Programming*, pages 389–396. IEEE Computer Society, 1987.

17. M. Gabbrielli, R. Giacobazzi, and D. Montesi. Modular logic programs over finite domains. In D. Saccà, editor, *Proc. Eight Italian Conference on Logic Programming*, pages 663–678, 1993.

18. F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Australian Computer Science Conference*, pages 337–346, 1996. Available at http://www.cs.mu.oz.au/mercury/papers.html.

19. P. M. Hill and J. W. Lloyd. *The Gödel programming language*. The MIT Press, 1994.

20. J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.

21. K. Marriott, M. Garcia de la Banda, and M. Hermenegildo. Analyzing logic programs with dynamic scheduling. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pages 240–253. ACM Press, 1994.

22. K. Marriott, H. Søndergaard, and P. Dart. A Characterization of Non-Floundering Logic Programs. In S. K. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming*, pages 661–680. The MIT Press, Cambridge, Mass., 1990.

23. C. S. Mellish. The Automatic Generation of Mode Declarations for Prolog Programs. DAI Research Paper 163, Department of Artificial Intelligence, Univ. of Edinburgh, August 1981.

24. L. Naish. An introduction to mu-prolog. Technical Report 82/2, The University of Melbourne, 1982.

25. R.A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

26. U. S. Reddy. Transformation of logic programs into functional programs. In *Int'l Symp. on Logic Programming*, pages 187–198. IEEE Computer Society, 1984.

27. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the Seventeenth ACM Symp. on Principles of Programming Languages*, pages 232–245. ACM Press, 1990.

28. E. Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

29. Z. Somogy. *A Parallel Logic Programing System based on Strong and Precise Modes*. PhD thesis, University of Melbourne, 1989. Also Technical Report 89/4, University of Melbourne.

30. Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Australian Computer Science Conference*, 1995. available at http://www.cs.mu.oz.au/mercury/papers.html.

31. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

32. K. Stroetman. A completeness result for SLDNF resolution. *The Journal of Logic Programming*, 15:337–357, 1993.

33. K. Ueda. Guarded Horn Clauses. In E. Wada, editor, *Logic Programming '85*, volume 221 of *Lecture Notes in Computer Science*, pages 168–179. Springer-Verlag, Berlin, 1986.

34. K. Ueda and M. Morita. Moded flat ghc and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.

35. P. Van Roy, B. Demoen, and Y. D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection and determinism. In G. Levi H. Ehrig, R. Kowalski and A. Montanari, editors, *TAPSOFT '87*, number 250 in LNCS, pages 111–125. Springer-Verlag, 1987.

36. Mark Wallace and André Veron. Two problems – two solutions: One system – ECLiPSe. In *Proceedings IEE Colloquium on Advanced Software Technologies for Scheduling*, London, 1993.