UNIVERSITÀ DI PISA
DIPARTIMENTO DI INFORMATICA

TECHNICAL REPORT : TR-97-21

On logic programs that do not fail

Dino Pedreschi, Salvatore Ruggieri

October 08, 1997

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

On logic programs that do not fail

Dino Pedreschi and Salvatore Ruggieri

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
e-mail: {pedre, ruggieri}@di.unipi.it

Abstract

This paper investigates the advantages of reasoning on logic programs and queries that have only successful derivations. We consider an extension of the logic programming paradigm that combines guarded clauses, in the style of concurrent logic languages, and dynamic selection rules. Some general conditions for a class of programs and queries are stated, which characterize when successful derivations only are present. A few practical instances of the general conditions are studied, and their applicability to real programs demonstrated. The main contributions of the proposed method are: (i) *don't care* non determinism can be safely adopted for programs that do not fail, (ii) termination of process networks expressed as logic programs can be proved, by means of a simple proof method developed for a fixed selection rule, and (iii) a strategy for parallelizing terminating Prolog programs is identified.

Keywords: Guarded logic programs, Dynamic selection rules, Don't care non determinism, Termination, Parallel Programming.

1 Introduction

Both in the imperative and logic programming paradigm, parallelism is usually interpreted in terms of *don't care* or *committed-choice* non determinism. In this form of non determinism, the operated choices are never retracted, i.e., backtracking is not permitted. This is the case of, for instance, Djikstra's guarded commands [11], and of concurrent logic languages [16]. From a study of the bibliographic sources, it is apparent that all (correct) parallel programs, in either paradigms, *do not fail*. The reason for this phenomenon is that a *don't care* interpretation of non deterministic choices implies that all possible choices are admissible, i.e., either all choice lead to some solution. This is particularly striking in the logic paradigm, where *don't know* non deterministic programming is also supported, and is implemented by means of backtracking. As a matter of fact, the parallel logic programs retrievable in the literature, are logic programs that do not fail.

Let us assume to have a characterization of logic programs that do not fail. There are two main advantages of reasoning on programs of that class.

On the one hand, *don't care* non determinism can be safely adopted in parallel executions of logic programs, without incurring in speculative parallelism.

On the other hand, it is possible to simplify the verification of such programs, and in particular the termination proofs. To see this point, consider, as an example, the following program, and the query `p`.

```
p :- q, p.
q :- fail.
p.
```

A left-to-right selection strategy incurs in a failure when `q` is selected, and, after backtracking, finds a successful derivation. On the contrary, a right-to-left strategy produces an infinite derivation if the first clause is always used. As a consequence, in presence of failures, the termination behaviour of a program depends on the particular search strategy adopted.

One of the main contributions of this article is to show that, in the case that no failure occurs, then termination w.r.t. a fixed selection rule implies termination w.r.t. every, possibly dynamic, selection rule. In particular, we show how a termination proof method developed for Prolog programs can be used, in the case of programs that do not fail, to prove termination w.r.t. dynamic selection rules, which give raise to a parallel interpretation. The advantage of this approach is twofold:

- a simple method, developed for fixed selection rules, can be used to reason about dynamic rules, which exhibit a highly complex behaviour;
- as a by-product, we obtain a strategy for parallelizing terminating Prolog programs.

Another contribution of this paper is to provide a characterization of logic programs that do not fail. Moreover, an instance of the general characterization is given in a purely syntactic way.

Our object of study is an extension of the logic programming paradigm obtained by combining *guarded clauses* and *delay declarations*. The former are taken from parallel logic programming languages [16, 18], whilst the latter were introduced in MU-Prolog [15] and further adopted by many logic programming systems, such as Gödel [12]. The resulting operational interpretation is called SLDG-resolution, and it is an extension of SLD-resolution.

SLDG-derivations have a natural parallel interpretation in terms of dynamic networks of asynchronous processes, communicating by means of multiparty channels. Since the early studies (see [19]), logic programs have been given a concurrent interpretation, where processes are identified by atoms, and communication is modeled by shared logical variables. In this context, delay declarations model synchronization among processes.

We introduce the notion of *persistent relations* μ over programs P and queries Q, such that if $\mu(P, Q)$ holds then $\mu(P, Q')$ holds for every resolvent Q' of P and Q. In addition, $\mu(P, A)$ is required to hold for the leftmost atom A of Q. This requirement reflects the fact that the considered relations are taken from frameworks for the analysis of Prolog programs. In particular, we extend the notions of well-typedness [4], acceptability [5] and simple-modedness [2].

The method for proving that program P and query Q have only successful SLDG-derivations is expressed for generic persistent relations μ such that P is $\mu-$failure-free. $\mu-$failure-freedom requires that every atom A such that $\mu(P, A)$ holds has a justification in the program.

We discuss some persistent relations which fulfill the proof obligations of the method. In particular, a weakest persistent relation μ_0 is characterized, which gives us a precise understanding of strengths and limitations of the proposed approach. The kind of parallelism addressed by the method is, intuitively, that of dynamic (nested) pipeline networks.

Another, syntactic, relation μ_2 is proposed, for which an automatic check method has been recently proposed in [9].

In conclusion, we discuss some examples describing applicability and limitations of the approach, that open natural research directions. We also discuss how, in its present form, the proposed method represents a strict improvement on the existing approaches for termination analysis of guarded programs with delay declarations.

2 Guarded Programs

In the paper we reason about a *conservative* extension of the logic programming paradigm obtained by combining *guarded clauses* and *dynamic selection rules*. We use the standard notation of Apt [3], when not specified otherwise. In particular, we use queries instead of goals. We denote by L the underlying language a program is defined on. $Atom_L$ denotes the set of atoms on L, B_L the Herbrand base on L.

A *guarded clause* is a construct of the form $H \leftarrow \mathbf{G} \mid \mathbf{B}$ where H is an atom and $\mathbf{G}$ and $\mathbf{B}$ are sequences of atoms. H is the clause head, the atoms in $\mathbf{G}$ are called *guards*, and $\mathbf{B}$ is called the body of the clause. A *guarded program* is a set of guarded clauses.

Logic languages that make use of guards allow to use in $\mathbf{G}$ only predefined relation symbols, such as the arithmetic built-in's `<, <=, =:=, =\=, >, >=` and the built-in $\neq$. According to Kunen [13], we assume that those symbols are declaratively defined by (possibly infinite) sets of *ground* facts, which we implicitly assume to be part of the program. For instance, the built-in `<` is declaratively defined by the set

$$M_{<} = \{\ x < y \mid x, y \in Gae \wedge val(x) < val(y)\ \}$$

where Gae is the set of ground arithmetic expressions, and $val(x)$ is the integer represented by the Gae x.

The union of all the sets M_{op}, where op is a built-in occurring in program P is denoted by M_P^{Ar}.

Consider a query $\mathbf{A}, B, \mathbf{C}$ and a guarded clause $c : H \leftarrow \mathbf{G} \mid \mathbf{B}$ variable disjoint with it, called *input clause*. We say that B *guardedly unifies* with H if for some mgu θ of B and H the query $\mathbf{G}\theta$ is ground and $M_P^{Ar} \models \mathbf{G}\theta$. The query $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ is called the SLDG-resolvent of $\mathbf{A}, B, \mathbf{C}$ and c. We define $gmgu(B, c) = \theta$. $mgu(B, c)$ denotes, instead, an mgu of B and c.

Let A be an atom and $c : H \leftarrow \mathbf{G} \mid \mathbf{B}$ be a renamed apart clause of P. We define:

$$try(A, c) = \begin{cases} \theta & \text{if } gmgu(A, c) = \theta, \\ fail & \text{if } mgu(A, H) = fail \vee \\ & \quad mgu(A, H) = \theta \wedge \mathbf{G}\theta \text{ ground } \wedge M_P^{Ar} \not\models \mathbf{G}\theta \\ suspend & \text{otherwise} \end{cases}$$

A SLDG-derivation is a maximal sequence $Q_1, \ldots, Q_n, \ldots$ of queries such that for $i \in [1, n-1]$, Q_{i+1} is a SLDG-resolvent of Q_i. An SLDG-derivation may be infinite.

A *SLDG-selection rule* associates to a prefix $Q_1, \ldots, Q_n$ of a SLDG-derivation an atom A in Q_n such that

$$(\exists\ c \in ren(P).\ try(A, c) = \theta) \vee (\forall\ c \in ren(P).\ try(A, c) = fail),$$

if any such atom exists. $ren(P)$ denotes the set of renamed apart clauses of P. A SLDG-derivation *deadlocks due to guards* if for every atom A in Q_n,

$$(\forall\ c \in ren(P).\ try(A, c) \neq \theta) \wedge (\exists\ c \in ren(P).\ try(A, c) = suspend).$$

For a finite SLDG-derivation $Q_1, \ldots, Q_n$, we say that it is *successful* (also a *refutation*) if Q_n is the empty query. It *fails* if for the selected atom A in Q_n and every renamed apart clause c of P, $try(A, c) = fail$.

When $\mathbf{G}$ is empty we drop the vertical bar "|" and SLDG-resolvents coincide with SLD-resolvents. So SLDG-resolution is indeed a conservative generalization of SLD-resolution.

LDG-resolution is SLDG-resolution together with a left-to-right selection rule.

Dynamic selection rules are characterized by constructs called *delay declarations*. A delay declaration for a predicate p is a construct of the form

`DELAY` $p(\mathbf{X})$ `UNTIL` $Condition(\mathbf{X})$

where $Condition$ is a formula in some type system. We say that an atom $p(\mathbf{t})$ *satisfies its delay declarations* if $Condition(\mathbf{t})$ is satisfied.

We say that a SLDG-selection rule is *safe* if every selected atom satisfies its delay declarations. We assume that a SLDG-derivation *deadlocks due to delays* if a query is reached such that no atom in it satisfies its delay declarations.

For simplicity, we make the hypothesis that the built-in's `<`, `<=`, `=:=`, `=\=`, `>`, `>=` and $\neq$ can only appear in guards.

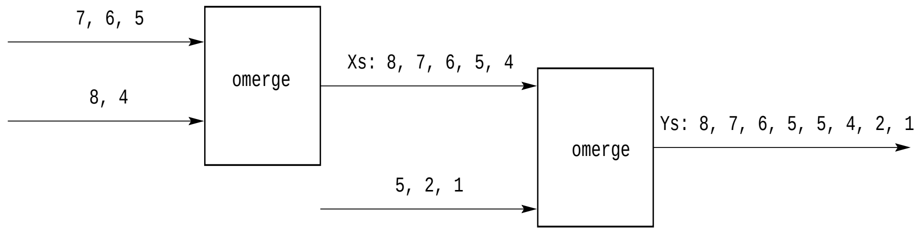

Figure 1: A pipeline network of processes

The following program `Merge` is often used as an example:

omerge(In_1, In_2, Out) :- *If* In_1 *and* In_2 *are*
ordered list of numbers, then Out is an ordered
merge of In_1 and In_2

```
omerge([], In2, In2).
omerge(In1, [], In1).
omerge([X|In1], [Y|In2], [X|Out]) :- X ≤ Y | omerge(In1, [Y|In2], Out).
omerge([X|In1], [Y|In2], [Y|Out]) :- X ≥ Y | omerge([X|In1], In2, Out).
```

If we consider the query `merge([5,6,7],[4,8],Xs),merge(Xs,[1,2,5],Ys)` together with the delay declaration:

```
DELAY omerge( In1 , In2 , Out ) UNTIL    (nonvar(In1) AND nonvar(In2) ) .
```

we obtain a network of asynchronously communicating processes, represented in Figure 1. The operational behaviour assigned by SLDG-resolution reflects in this example a form of pipeline parallelism. Clearly, different selection rules correspond to different behaviours of the process network. In general, process networks are dynamic.

3 Persistent relations

Reasoning on programs and queries is usually done by abstracting the interesting properties (termination, correctness, call-patterns, etc.) by means of some intended relation $\mu(P, Q)$ over the considered program P and query Q. Persistency of the intended relation along SLD(G)-derivation yields a method for conducting inductive proofs of correctness, termination, etc., and has been shown for many analysis frameworks [4] of logic programs.

This section formally defines a general notion of persistent relation, and shows how several persistent relations can be lifted from pure logic programming to guarded programs, thus generalizing existing results. Later in this paper, persistent relations will be the basis to characterize failure-free guarded programs.

Definition 1 A relation μ over programs and queries is a *persistent* relation if when $\mu(P, Q)$ holds then

(i) $\mu(P, Q')$ holds for every Q' SLDG-resolvent of P and Q via a safe selection rule,

(ii) and $\mu(P, A)$ holds, where A is the leftmost atom in Q. □

We *arbitrarily* assume that the leftmost atom of a query is in the relation if the query is. This choice is due to the approach of extending to guarded programs some methods originally developed for Prolog programs. A weaker requirement that $\mu(P, A)$ holds for *some* atom A in Q is clearly preferable. However, our limited notion of persistency has two advantages. First, it yields a simple method, developed with reference to a fixed selection rule, which generalizes to dynamic selection rules. Second, this notion yields a method for parallelizing Prolog programs.

Another interesting property of fixing the atom A in Definition 1*(ii)* is that $\mu_1 \cap \mu_2$ is a persistent relation if μ_1 and μ_2 are.

We introduce next several persistent relations for guarded program, by lifting notions existing in the literature of logic programs.

3.1 Well-typing

Logic programming languages are usually untyped. Several proposals have been introduced in the literature to provide type systems for logic programs. Here, we report one of them, that merges simplicity and expressiveness.

Definition 2 A *type* is an non-empty set of terms closed under substitution. □

We now fix a specific set of types, denoted by *Types*, which includes:

U - the set of all terms,

Ground - the set of ground terms,

List - the set of lists. Formally:

$$List ::= \texttt{[]} \mid \texttt{[}U \ \texttt{|} \ List\texttt{]}$$

Gae - the set of all ground arithmetic expressions (gae's, in short). Formally:

$$Gae ::= \texttt{n} \mid Gae \ \texttt{bop} \ Gae \mid \texttt{uop} \ Gae$$

where `n` is a numeric constant, `bop` is any binary arithmetic operator (including $+, -, *$), and `uop` is any unary arithmetic operator (including $-$),

List(*Gae*) - the set of lists of gae's. Formally:

$$List(Gae) ::= \texttt{[]} \mid \texttt{[}Gae \ \texttt{|} \ List\texttt{]}$$

We call a construct of the form $s : S$, where s is a term and S is a type, a *typed term*. Given a sequence $s_1 : S_1, \ldots, s_n : S_n$ of typed terms, also denoted by $\mathbf{s} : \mathbf{S}$, we write $\mathbf{s} \in \mathbf{S}$ if for $i \in [1, n]$ we have $s_i \in S_i$. Further, we abbreviate the sequence $s_1\theta, \ldots, s_n\theta$ to $\mathbf{s}\theta$.

Definition 3 A *type* for an n-ary relation symbol p is a function t_p from $[1, n]$ to the set *Types*.

A *mode* for p is a function m_p from $[1, n]$ to $\{+, -\}$. If $m_p(i) =' -'$, we call i an *output position* of p. If $m_p(i) =' +'$, we call i an *input position* of p. By a *moding* we mean a collection of modes, each for a different relation symbol.

We assume that every considered relation has a fixed mode and a fixed type associated with it. We denote the mode and the type of p by

$$p(m_p(1) : t_p(1), \ldots, m_p(n) : t_p(n)).$$

Finally, for the built-in predicate `<` we fix the following mode and type:

$$<(- : Gae, - : Gae)$$

and analogously for `<, <=, =:=, =\=, >, >=`. □

For example, the following are modes and types for program `Merge`:

$$\texttt{omerge(} \ + : List(Gae)\texttt{,} \ + : List(Gae)\texttt{,} \ - : List(Gae) \ \texttt{).}$$

Here, the first and second positions are moded as input and typed as lists of *Gae*, and the third position is moded as output and typed as *List*(*Gae*).

Observe that the choice of moding all positions of built-in's as output is a weak requirement, made possible by the fact that SLDG-resolutions already guarantees that the argument of atoms in guards are gae's when a resolution step takes place.

To simplify the notation, when writing an atom as $p(\mathbf{i} : \mathbf{I}, \mathbf{o} : \mathbf{O})$ we assume that $\mathbf{i} : \mathbf{I}$ is a sequence of typed terms filling in the input positions of p and $\mathbf{o} : \mathbf{O}$ is a sequence of typed terms filling in the output positions of p. The following notion is an adaptation to guarded programs of the one due to Bronsard et al. [7].

Definition 4

- A query Q is called *well-typed* if for every $p_1(\mathbf{i}_1 : \mathbf{I}_1, \mathbf{o}_1 : \mathbf{O}_1), \ldots, p_n(\mathbf{i}_n : \mathbf{I}_n, \mathbf{o}_n : \mathbf{O}_n)$ instance of Q, for $i \in [1, n]$

$$\mathbf{o}_1 \in \mathbf{O}_1 \wedge \ \ldots \ \wedge \mathbf{o}_{i-1} \in \mathbf{O}_{i-1} \quad implies \quad \mathbf{i}_i \in \mathbf{I}_i$$

- A guarded clause C is called *well-typed* if for every

$$\begin{array}{lcl} p_0(\mathbf{o}_0 : \mathbf{O}_0, \mathbf{i}_{n+1} : \mathbf{I}_{n+1})) & \leftarrow & \mathbf{G}(\_ : \_, \mathbf{o_1} : \mathbf{O_1}) \mid \\ & & p_1(\mathbf{i}_2 : \mathbf{I}_2, \mathbf{o}_2 : \mathbf{O}_2), \ldots, p_n(\mathbf{i}_n : \mathbf{I}_n, \mathbf{o}_n : \mathbf{O}_n) \end{array}$$

  instance of C, for $i \in [2, n+1]$

$$\mathbf{o}_0 \in \mathbf{O}_0 \wedge \ \ldots \ \wedge \mathbf{o}_{i-1} \in \mathbf{O}_{i-1} \quad implies \quad \mathbf{i}_i \in \mathbf{I}_i$$

- A program is called *well-typed* if every clause of it is. □

By noting that if a query is well-typed then also its leftmost atom is well-typed, and by the following lemma (which extends one due to Apt and Luijtes [1]) we have that well-typedness is a persistent relation.

Lemma 5 *Let P and Q be a well-typed program and query. Then any SLDG-resolvent of P and Q is well-typed.*

Proof. See Appendix A. □

For instance, the `Merge` program is well-typed with the specified modes and types.

3.2 Acceptability

Acceptable logic programs are introduced in [5] as a correct and complete characterization of the class of logic programs such that every ground query has a finite LD-tree. Here, we extend the notion of [5] to guarded programs. We recall that a level mapping $| \ | : B_L \rightarrow N$ is a function from B_L to natural numbers. We denote by *ground*(P) the set of ground instances of clauses from P.

Definition 6 Let P be a guarded program, $I \subseteq B_L$ a Herbrand interpretation, and $| \ | : B_L \rightarrow N$ a level mapping.

- P is *acceptable by* $| \ |$ *and* I iff I is a model of P and for every $A \leftarrow \mathbf{G} \mid B_1, \ldots, B_n$ in *ground*(P) : for $i \in [1, n]$

$$I \models \mathbf{G}, B_1, \ldots, B_{i-1} \quad implies \quad | \, A \, | \ > | \, B_i \, | \, .$$

- A query Q is *acceptable by* $| \ |$ *and* I iff there exists $k \in N$ such that for every ground instance $A_1, \ldots, A_n$ of it: for $i \in [1, n]$

$$I \models A_1, \ldots, A_{i-1} \quad implies \quad k > | \, A_i \, |$$

□

No proof obligation is required on the guards, since they cannot generate infinite derivations.

Lemma 7 *Let P and Q be a program and query both acceptable by $|\ |$ and I. Then, every SLDG-resolvent of P and Q is acceptable by $|\ |$ and I. Moreover, every LDG-derivation for P and Q is finite.*

Proof. Immediate by Lemma 21 reported in Appendix A, and the fact that the multiset ordering on naturals is well-founded. □

As a consequence of the above Lemma and Definition 6, acceptability is a persistent relation.

Differently from the case of pure logic programs, acceptability is not a complete termination method. In fact, the following program and the query `q(X)` have only finite SLDG-derivations, but they are not acceptable.

```
q(X) :- X > 0 | q(X-1).
q(0).
```

The deficiency of the method lies in the fact that the lifting lemma does not extend to SLDG-resolution. As a consequence, termination of a query cannot be deduced from termination of its ground instances.

As an example, the `Merge` program is acceptable by B_L and $|\ |$, where

$$|\texttt{omerge(}\ in_1\texttt{,}\ \ in_2\texttt{,}\ \ out\ \ \texttt{)}| = ll(in_1) + ll(in_2)$$

where the *list-length* function ll is defined as follows: $ll(f(t_1\ ,\ldots,t_n\ ))$ is 0 if $f \neq [.|.]$ and $ll(t_2) + 1$ if $f(t_1\ ,\ldots,t_n\ ) = [t_1|t_2]$.

3.3 Simple-modedness

Simple-modedness was first introduced by Apt and Pellegrini [2] as part of a sufficient method for safe omission of the *occur-check* test in the unification algorithm.

Definition 8

- A query $p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is called *simply moded* if $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a tuple of distinct variables and for $i \in [1, n]$

$$Var(\mathbf{s}_i) \cap (\bigcup_{j=i}^{n} Var(\mathbf{t}_j) = \emptyset$$

- A guarded clause

$$p_0(\mathbf{s}_0, \mathbf{t}_0) \leftarrow \mathbf{G} \mid p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$$

  is called *simply moded* if $p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is *simply moded* and

$$Var(\mathbf{s}_0) \cap (\bigcup_{j=1}^{n} Var(\mathbf{t}_j) = \emptyset$$

  In particular, every unit clause is simply moded.

- A program is called *simply moded* if every clause of it is. □

Intuitively, simple-modedness reflects the existence of a left-to-right ordering of atoms in a query in such a way that the input of a process does not depend from the output of a later process, i.e. a process lying at its right. In other words, only pipeline parallelism can be modeled by simple-modedness.

Definition 9 We say that the delay declarations *imply matching* if for every atom $p(\mathbf{u}, \mathbf{v})$ which satisfies its delay declaration and for every head $p(\mathbf{u}', \mathbf{v}')$ of a clause disjoint with it, if $p(\mathbf{u}, \mathbf{v})$ and $p(\mathbf{u}', \mathbf{v}')$ unify, then $\mathbf{u}$ is an instance of $\mathbf{u}'$. □

Lemma 10 *Let P and Q be a simply moded program and query, and assume that the delay declarations* imply matching*. Then every SLDG-resolvent of P and Q via safe selection rules is simply moded.*

Proof. See Appendix A. □

Directly from the above Lemma and Definition 8, we conclude that *simple-modedness* is a persistent relations under the assumption that delay declarations imply matching.

The assumption that delay declaration imply matching may seem restrictive. However, two observation can clarify that it is not the case. First, matching is required only for input arguments, which – being input – should be ever sufficiently instantiated before computation takes place. Second, in a concurrent logic programming language [16] the hypothesis that input arguments of the selected atom are instances of input arguments of the input clause head, is always satisfied. In fact, consider for instance the simple language FCP(|) of [16]. There unification is replaced by matching, i.e. a resolution step is executed if the selected atom is an instance of the input clause, with the only exception of the predicate `=`. In the FCP(|) system, if an atom A and an input clause have a resolvent then the atom is an instance of the clause head, and then the input arguments of the atom are instances of the input arguments of the clause head. The same conclusion holds also for `=`, by considering the moding `=( -, -)`, or `=(-, +)`, or `=(+, -)`.

As an example, the `Merge` program is simply moded by assuming the moding:

$$\texttt{omerge}(+,+,-)).$$

Moreover, the delay declaration of `Merge` implies matching.

4 Failure-free programs

SLDG-derivations can be infinite, successful, failed or deadlocked. In the last case, deadlocks can be caused either by guards (that are not sufficiently instantiated to allow any resolution steps) or by delay declarations (that do not allow any atom selection.) In this section, we provide some sufficient conditions on programs, which allows us to prove absence of infinite derivations, absence of deadlock due to either delays or guards, and absence of failing derivations. In other words, only successful derivations are admitted. Even if apparently highly restrictive, the proposed conditions are sufficiently general to reason about a significant class of guarded programs and queries.

First of all, we note that deadlock due to delays is prevented by assuming that a persistent relation implies delay declarations.

Definition 11 We say that μ *implies the delay declarations* of P if for every atom A such that $\mu(P, A)$ holds then A satisfies its delay declarations. □

Since μ is a persistent relation, in any query Q such that $\mu(P, Q)$ holds, there exists A (namely, the leftmost atom) such that $\mu(P, A)$ holds.

Lemma 12 *Let μ be a persistent relation that implies the delay declarations and suppose that $\mu(P, Q)$ holds. Then no SLDG-derivation of P and Q via a safe selection rule deadlocks due to delays.*

Proof. By Definition 1 *(ii)*, $\mu(P, A)$ holds for every leftmost atom of a resolvent of P and Q. By Definition 11, A satisfies its delay declarations. □

We introduce next the key notion of this paper.

Definition 13 We say that P is *$\mu-$failure-free* iff for every non-built-in atom A such that $\mu(P, A)$ holds, there exists a clause that guardedly unifies with A. □

Intuitively, P is $\mu-$failure-free if every atom in the intended relation μ has a justification in the program. On the one hand, μ-failure-freedom prevents deadlock due to guards.

Lemma 14 *Let μ be a persistent relation and suppose that $\mu(P, Q)$ holds. If P is $\mu-$failure-free, then no SLDG-derivation of P and Q via a safe selection rule deadlocks due to guards.*

Proof. By Definition 1 *(ii)*, $\mu(P, A)$ holds for the leftmost atom of a resolvent of P and Q. By Definition 13, A guardedly unifies with some clause c. Therefore, $try(A, c) \neq suspend$, and then no SLDG-derivation deadlocks due to guards. □

On the other hand, under certain conditions μ-failure-freedom prevents failures of selected atoms.

Lemma 15 *Let μ be a persistent relation and P a $\mu-$failure free program. Let A be an atom selected in a SLDG-derivation for P and any query Q via a safe selection rule. If there is an instance A' of A such that $\mu(P, A')$ holds, then A guardedly unifies with some clause.*

Proof. Since P is *μ-failure-free*, there exists a clause c in P that guardedly unifies with A'. Since A' is an instance of A, then the head of c and A unify. As a consequence, $try(A, c) \neq fail$, otherwise $try(A', c)$ would be $fail$.

Since A is selected, then

$$(\exists\ c \in P.\ try(A, c) = \theta) \vee (\forall\ c \in P.\ try(A, c) = fail).$$

Therefore, this and $try(A, c) \neq fail$ imply

$$(\exists\ c \in P.\ try(A, c) = \theta),$$

i.e. A guardedly unifies with some clause. □

The following Theorem introduces a general method for proving that a given program and query have only successful derivations.

Theorem 16 *Let P be a guarded program, Q a query and μ a persistent relation such that:*

(i) $\mu(P, Q)$ *holds,*

(ii) P *is* μ*−failure-free,*

(iii) P *and* Q *are acceptable by* $|\ |$ *and* I*,*

(iv) μ *implies the delay declarations.*

Then every SLDG-derivation of P *and* Q *via a safe selection rule is successful.*

Proof. See Appendix A. □

Persistency of μ and of acceptability, and *(i)* guarantee that the hypotheses hold for every resolvent of P and Q. *(ii)* implies absence of failures and deadlocks due to guards. *(iii)* implies termination, and *(iv)* implies absence of deadlocks due to delays.

As pointed out in the introduction, the main weakness of the method is that it relies on notions that reflect Prolog's leftmost selection rule. A precise characterization of the class of relations, programs and queries we can reason about is described later. On the other hand, the method allows us to parallelize Prolog programs and is applicable to a surprisingly large class of parallel logic programs.

5 Some practical persistent relations

The problem we are now faced is to find out some persistent relation μ such that the hypotheses of Theorem 16 are satisfied.

On one extreme, assuming that $\mu(P, Q)$ *never* holds, every program is μ−failure-free, although hypothesis *(i)* of the Theorem never holds.

On the opposite extreme, assuming that $\mu(P, Q)$ holds for every P, Q, we have that P is μ−failure-free only if *every* query does not fail.

We therefore need to identify some practical condition for a sufficiently large class of interesting programs.

5.1 The weakest persistent relation: μ_0

We introduce now a relation μ_0, which turns out to be the weakest persistent relation satisfying the hypotheses of Theorem 16.

Definition 17 $\mu_0(P, Q)$ holds iff

- every SLDG-derivation of P and Q via a safe selection rule is successful, and
- for every query Q' in a SLDG-derivation of P and Q via a safe selection rule, every LDG-derivation of P and Q' is successful. □

The next result shows that for a given program P, there exists at least one persistent relation, namely μ_0, such that P is μ_0-failure free. In addition, μ_0 is the weakest persistent relation such that hypotheses *(i-iv)* of Theorem 16 hold.

Theorem 18 *Given a program* P*,* μ_0 *is a persistent relation,* μ_0 *implies the delay declarations, and* P *is* μ_0*−failure free.*

Moreover, for every persistent relation μ *and query* Q *such that the hypotheses (i-iv) of Theorem 16 hold, we have that (i-iv) hold w.r.t.* μ_0 *as well.*

Proof. See Appendix A. □

Relation μ_0 exactly characterizes the limitations of the proposed method, in that it shows that every program and query we can reason about have a natural interpretation in terms of LDG-resolution.

5.2 Well-typing: μ_1

As a first approximation, consider μ_1 such that $\mu_1(P, Q)$ holds iff P and Q are a well-typed program and query respectively. Suppose now that hypotheses *(i-iv)* of Theorem 16 are all satisfied. We can then conclude that all SLDG-derivations of Q via a safe selection rule are successful. Since hypotheses *(i-iv)* hold for every instance Q' of Q, we obtain the same conclusion for every such Q'. This implies that the query Q directly holds in the least Herbrand model of P, and therefore the chosen types must be accurate enough to describe the intended meaning of the program. This is not the usual meaning of types, which are instead used to circumscribe the set of intended queries, i.e., a rough approximation of the least Herbrand model.

As an example, consider the `Merge` program, which is well-typed with:

$$\texttt{omerge(}\ +: List(Gae)\texttt{,}\ \ +: List(Gae)\texttt{,}\ \ -: List(Gae)\ \texttt{).}$$

To show that `Merge` is μ_1-failure-free, we should prove that for every In_1, In_2, Out lists of Gae's, `omerge(`In_1`,` In_2`,` Out`)` guardedly unifies with some clause. Clearly, this is not true, as Out is required to be an ordered merge of In_1 and In_2.

5.3 Adding simple modedness: μ_2

Consider a well-typed atom $p(\mathbf{i} : \mathbf{I}, \mathbf{o} : \mathbf{O})$. A natural use of it, is to calculate values for output positions starting from given values in input positions. In general, we cannot assume that for every input and output values there exists a refutation. A more reasonable assumption is, instead, to require that for every input value there exists some output value computed by SLDG-resolution. This idea is made clear by the notion of simple modedness, where $\mathbf{o}$ is assumed to be a tuple of distinct variables.

We say that $\mu_2(P, Q)$ holds iff P and Q are well-typed, simply moded, and the delay declarations of P imply matching. By the various results of Section 3, μ_2 is a persistent relation.

As an example, consider the `Merge` program. To show that it is μ_2-failure-free, we have to prove that for every pair In_1, In_2 of lists of Gae's and variable Out, `omerge(`In_1`,` In_2`,` Out`)` guardedly unifies with some clause. This is established by distinguishing four (not mutually exclusive) cases:

- $In_1 =$ `[]`, is covered by the first clause,
- $In_2 =$ `[]`, is covered by the second clause,
- $In_1 =$ `[`$X \mid In_1'$`]`, $In_2 =$ `[`$Y \mid In_2'$`]`, and $X \leq Y$, is covered by the third clause,
- $In_1 =$ `[`$X \mid In_1'$`]`, $In_2 =$ `[`$Y \mid In_2'$`]`, and $X \geq Y$, is covered by the fourth clause.

In addition, it is readily checked that μ_2 implies the delay declaration of `Merge`, and that the delay declaration implies matching.
Moreover, `Merge` and `omerge(`In_1`,` In_2`,` Out`)`, such that In_1 and In_2 are lists of Gae's and Out is a variable, are an acceptable program and query. Thus, we are in

the hypotheses of Theorem 16, which implies that every SLDG-derivation for `Merge` and `omerge(`In_1`, `In_2`, `Out`)` via any safe selection rule is successful.

6 Discussion

6.1 Covering algorithm

The recent work by Debray et al. [9] helps in proving that a program is μ-failure free for a given μ. They provide a method that, given mode and type information, can detect whether the clauses defining a predicate *cover* the type of the predicate. In particular, let us consider relation μ_2.

The non-failure analysis of [9] is based on regular types [8], which are specified by *regular term grammars* in which each type symbol has exactly one defining *type rule*. They assume that programs are *moded*, i.e. for each unification operation we know whether the operation acts as a test or creates output bindings.

Consider a predicate $p(\mathbf{i} : \mathbf{I}, \mathbf{o} : \mathbf{O})$ and a guarded clause

$$\mathrm{p}(\mathbf{t},\ \mathrm{s}) \coloneq \mathbf{G} \ \mid \ \mathbf{B}$$

Assume that the input types $\mathbf{I}$ of p are regular types and the guards $\mathbf{G}$ of the clauses defining p are tests over the *Herbrand domain* or over *linear arithmetic over integers* not involving $var(\mathbf{s})$.

Debray et al. provide an algorithm that decides whether for every $\mathbf{i} : \mathbf{I}$ there is a clause such that $\mathbf{t} = \mathbf{i}, \mathbf{G}$ holds.

In practice, they provide us with a decision procedure for μ_2-failure-freedom, under the mentioned hypotheses. In fact, when $\mu_2(P, p(\mathbf{i}, \mathbf{o}))$ holds, then $\mathbf{o}$ is a tuple of distinct variables. Therefore, if $\mathbf{t} = \mathbf{i}, \mathbf{G}$ holds and $var(\mathbf{s})$ do not occur in $\mathbf{G}$, then also

$$\mathbf{t} = \mathbf{i}, \mathbf{s} = \mathbf{o}, \mathbf{G}$$

holds, i.e. $p(\mathbf{i}, \mathbf{o})$ unifies with at least one clause head.

Comparing the approach in [9] with that of this paper, we concentrate on designing sufficient condition for failure-freedom, rather than on developing specific algorithms for checking *covering*.

6.2 Applications and limitations of the approach

QuickSort

Consider the following guarded version of the `QuickSort` program.

```
qs([],[]).
qs([X|Xs],Ys) ←
   part(X,Xs,Littles,Bigs),
   qs(Littles,Ls),
   qs(Bigs,Bs),
   append(Ls,[X|Bs],Ys).

part(_,[],[],[]).
part(X,[Y|Xs],[Y|Ls],Bs) ← X≥Y | part(X,Xs,Ls,Bs).
part(X,[Y|Xs],Ls,[Y|Bs]) ← X≤Y | part(X,Xs,Ls,Bs).
```

```
append([], Xs, Xs).
append([X|Xs], Ys, [X|Zs]) ← append(Xs, Ys, Zs).
```

It is readily checked that `QuickSort` is well-typed, simply moded and failure-free with the typing:

$$\texttt{qs}(+ : List(Gae), - : List(Gae))$$

$$\texttt{part}(+ : Gae, + : List(Gae), - : List(Gae), - : List(Gae))$$

$$\texttt{append}(+ : List(Gae), + : List(Gae), - : List(Gae))$$

Moreover, `QuickSort` is acceptable by an interpretation and a level mapping $|\ |$ such that $|\texttt{qs}(xs, ys)| = ll(xs) + 2$.

Consider now a query $Q = \texttt{qs}(xs,\texttt{Ys})$ where xs is a list of gae's. Q is well-typed, simply moded and acceptable. Therefore, to apply Theorem 16 we only have to derive delay declarations that imply matching, and that are implied by μ_2. The following choice of delay declarations allows for a high degree of parallelism.

```
DELAY qs(X,Y) UNTIL nonvar(X)
DELAY part(X, Xs, Ls, Bs) UNTIL nonvar(Xs)
DELAY append(X,Y,Z) UNTIL nonvar(X) ∧ nonvar(Y)
```

Producer & Consumer

As another example, consider the following *producer & consumer* program.

```
DELAY read(H) UNTIL H = [M | _] AND ground(M)
DELAY produce(_, In) UNTIL nonvar(In)

    prod&cons :-
       H = [message(_) | H1],
       produce(m, H),
       read(H).

    read([message(X) | H]) :-
       H = [message(_) | H1],
       read(H).
    read([]).

    produce(s(N), [message(s(0))|In]) :-
       produce(N, In).
    produce(0, []).
```

Intuitively, `produce` is the producer of a sequence of naturals (for brevity, here all the numbers are 1). `read` is the consumer, which once received `message`(X) consumes it. `produce` and `read` communicate via a one-position buffer represented by the partial list `H`: `produce` writes in the head of the list if the position is available, i.e. if it is not a variable; `read` reads from the head when a message is available and write an incomplete message denoting that the position is become available. Termination is decided by the producer by instantiating `H` to `[]` after having produced m messages.

The program *producer & consumer* activates a tight co-routining behaviour between the two processes, which prohibits a Prolog-like execution strategy. In fact, any Prolog-like execution ends in a deadlock due to delays, because of the delay declaration of `produce`. As a consequence, Theorem 16 is not applicable.

Generate & Test

Another limitation of the proposed method is in dealing with programs following the *generate & test* scheme:

```
gtsolve(Input,Output) ← generate(Input,Output),test(Output).
```

The problem with *generate & test* programs is that relation `test` can fail, and control passes back to `generate` through backtracking. However, it should be noted that these programs are not parallelizable without incurring in speculative parallelism. Moreover, we are still in the position of applying the method of Theorem 16 to the `generate` relation.

7 Conclusions

In this article we presented a method for reasoning on guarded programs and queries that do not fail. We provide sufficient conditions that enable us to turn Prolog programs into guarded programs, whose parallel execution gives raise to dynamic networks of communicating processes. In particular, the proposed approach fully addresses nested pipeline networks. The main limitations of the approach, which we are currently addressing, are:

- full co-routining, as in the *producer & consumer* example, based on two-way synchronization;
- amalgamation of *don't care* and *don't know* non determinism, as in the *generate & test* scheme;
- characterization of other, more expressive persistent relations, which improve over the proposed combination of well-typedness and simple-modedness.

In any case, the proposed method represents a strict improvement on the existing approaches for reasoning about termination of guarded programs with delays.

- The approach of Apt and Luitjes [1] deals with determinate programs and queries, i.e., those admitting only one derivation which must also be successful, whereas our method applies to non deterministic programs which do not fail.
- The approach of Marchiori and Teusink [14] deals only with so-called *local* selection rules, i.e., depth-first execution strategies. Their method is not appropriate for dealing with parallel programs, since local selection rules prevent parallel execution. Therefore, their method and ours are uncomparable.

Recently, Bossi and Cocco [6] proposed a notion of *programs without failures* in the context of Prolog program verification and transformation. Such a notion turns out to coincide with well-typedness together simple modedness. However, their interest is restricted to LD-resolution: delay declarations and arbitrary selection rules are not taken into account. On the other hand, we provide a general framework for proving failure freedom and *termination* with respect to any safe SLDG-selection rule. Special cases of guarded programs with delay declarations include definite logic programs, parallel logic programs, and `Gödel` programs.

Acknowledgements

Thanks are owing to Barbara Frigerio who helped us in clarifying the contents of this article.

References

[1] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In V.S. Alagar and M. Nivat, editors, *Proc. of AMAST'95*, volume 936 of *Lecture Notes in Computer Science*, pages 66–90, Berlin, 1995. Springer-Verlag.

[2] K. R. Apt and A. Pellegrini. Why the occur-check is not a problem. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of PLILP 92*, Lecture Notes in Computer Science 631, pages 69–86, Berlin, 1992. Springer-Verlag.

[3] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 493–574. Elsevier, 1990.

[4] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6A:743–764, 1994.

[5] K.R. Apt and D. Pedreschi. Studies in Pure Prolog: Termination. In J. W. Lloyd, editor, *Symposium on Computational Logic*, pages 150–176. Springer-Verlag, Berlin, 1990.

[6] A. Bossi and N. Cocco. Programs without Failures. In *LOPSTR '97*, Lecture Notes in Computer Science, 1997. To appear.

[7] F. Bronsard, T.K. Lakshman, and U.S. Reddy. A framework of directionality for proving termination of logic programs. In K. R. Apt, editor, *Proceedings of the 1992 Joint Int. Conference and Symposium on Logic Programming*, pages 321–335. MIT Press, 1992.

[8] P. W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In F. Pfenning, editor, *Types in Logic Programming*, Logic Programming Series. The MIT Press, 1992.

[9] S. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *Proceedings of the 1997 International Conference on Logic Programming*. The MIT Press, 1997.

[10] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 8:69–116, 1987.

[11] E.W. Dijkstra. *A discipline of programming*. Prentice Hall, 1976.

[12] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.

[13] K. Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7:978–992, 1989.

[14] E. Marchiori and F. Teusink. On Termination of Logic Programs with Delay Declarations. In *Proc. of the 1995 International Symposium on Logic Programming*, pages 447–461. The MIT Press, 1995.

[15] L. Naish. An introduction to MU-PROLOG. Technical Report 82/2, Department of Computer Science, University of Melbourne, 1982.

[16] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[17] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, second edition, 1994.

[18] E. Tick. The Deevolution of Concurrent Logic Programming Languages. *Journal of Logic Programming*, 23(2):89–123, 1995.

[19] M. H. van Emden and G. J. de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 189–198. Academic Press, 1982.

A Proofs

A.1 Guarded Programs and Logic Programs

Next lemma relates guarded programs and logic programs with arithmetic built-in's from an operational point of view. For a guarded program P, we define its logic programming version $P^\star$ as follows:

$$P^\star = \{A \leftarrow \mathbf{G}, \mathbf{H} \mid A \leftarrow \mathbf{G} \mid \mathbf{H} \in P\}.$$

Accordingly to [17], we assume that if an arithmetic built-in is selected during a SLD-derivation, then the derivation *fails* if its arguments are not ground arithmetic expressions (gae's for short). If $x < y$ is selected and x, y are gae's, then the SLD-derivation fails if $M_< \not\models x < y$. Otherwise, the SLD-resolvent is the rest of the query in which $x < y$ appears.

Lemma 19 *Let P be a guarded program and Q a query. There is a bijection between SLDG-refutations w.r.t. a fixed selection rule for P and Q and SLD-refutations w.r.t. some selection rule for $P^\star$ and Q.*

Proof. We associate to a selection rule for SLDG-resolution the selection rule for SLD-resolution that for a program $P^\star$ and a query Q selects the atom selected by the corresponding SLDG-rule for P and Q first, and then the guards atoms of the used input clause.

Any SLDG-refutation for P and Q can be immediately viewed as a SLD-refutation for $P^\star$ and Q by noting that if an atom guardedly unifies with an input clause, then the built-in's in the guard succeed.

Conversely, consider a SLD-refutation for $P^\star$ and Q. The built-in's corresponding to guards are ground and succeed when selected. Therefore, moving those atoms from explicit solving to guard unification we obtain a SLDG-refutation for P and Q. $\Box$

A.2 Well-typedness

Lemma 5

Proof. Suppose that the i^{th} atom in Q is the selected atom. Let θ be the mgu of the i^{th} atom in Q and the input clause c. We observe that any instance of a well-typed clause is well-typed, and analogously for query. Then $Q\theta$ and $c\theta$ are well-typed. Moreover, suppose that:

$$Q\theta = p_1(\mathbf{i}_1 : \mathbf{I}_1, \mathbf{o}_1 : \mathbf{O}_1), \ldots, p_n(\mathbf{i}_n : \mathbf{I}_n, \mathbf{o}_n : \mathbf{O}_n)$$

and that:

$$\begin{aligned} c\theta \quad = \quad & p_i(\mathbf{i}_i : \mathbf{I}_i, \mathbf{o}_i : \mathbf{O}_i)) \leftarrow \mathbf{G}(\_ : \_, \mathbf{o}' : \mathbf{O}') \mid \\ & q_1(\mathbf{i}'_1 : \mathbf{I}'_1, \mathbf{o}'_1 : \mathbf{O}'_1), \ldots, q_m(\mathbf{i}'_m : \mathbf{I}'_m, \mathbf{o}'_m : \mathbf{O}'_m). \end{aligned}$$

We observe that $\mathbf{o}' \in \mathbf{O}'$ since the selected atom and c guardedly unify, hence the arguments of the guard atoms are gae's. As a consequence, $\mathbf{o}'\gamma \in \mathbf{O}'$ for every substitution γ. Then the SLDG-resolvent of Q and c is:

$$\begin{aligned} Q' \quad = \quad & p_1(\mathbf{i}_1 : \mathbf{I}_1, \mathbf{o}_1 : \mathbf{O}_1), \ldots, p_{i-1}(\mathbf{i}_{i-1} : \mathbf{I}_{i-1}, \mathbf{o}_{i-1} : \mathbf{O}_{i-1}), \\ & q_1(\mathbf{i}'_1 : \mathbf{I}'_1, \mathbf{o}'_1 : \mathbf{O}'_1), \ldots, q_m(\mathbf{i}'_m : \mathbf{I}'_m, \mathbf{o}'_m : \mathbf{O}'_m), \\ & p_{i+1}(\mathbf{i}_{i+1} : \mathbf{I}_{i+1}, \mathbf{o}_{i+1} : \mathbf{O}_{i+1}), \ldots p_n(\mathbf{i}_n : \mathbf{I}_n, \mathbf{o}_n : \mathbf{O}_n). \end{aligned}$$

Let us show that Q' is well-typed. Consider an instance $Q'\gamma$:

- For $k \in [1, i-1]$ then
$$\mathbf{o}_1\gamma \in \mathbf{O}_1 \wedge \ldots \wedge \mathbf{o}_{i-1}\gamma \in \mathbf{O}_{k-1} \quad \textit{implies} \quad \mathbf{i}_k\gamma \in \mathbf{I}_k$$
since $Q\theta$ is well-typed.

- For $k \in [1, m]$ then if
$$\mathbf{o}_1\gamma \in \mathbf{O}_1 \wedge \ldots \wedge \mathbf{o}_{i-1}\gamma \in \mathbf{O}_{i-1} \wedge \mathbf{o}'_1\gamma \in \mathbf{O}'_1 \wedge \ldots \wedge \mathbf{o}'_{k-1}\gamma \in \mathbf{O}'_{i-1}$$
then, since $c\theta$ is well-typed and $\mathbf{o}'\gamma \in \mathbf{O}'$, then
$$\mathbf{i}'_k\gamma \in \mathbf{I}'_k.$$

- For $k \in [i+1, n]$ then if
$$\mathbf{o}_1\gamma \in \mathbf{O}_1 \wedge \ldots \wedge \mathbf{o}_{i-1}\gamma \in \mathbf{O}_{i-1} \wedge \mathbf{o}'_1\gamma \in \mathbf{O}'_1 \wedge \ldots \wedge \mathbf{o}'_m\gamma \in \mathbf{O}'_m \wedge$$
$$\wedge\, \mathbf{o}_{i+1}\gamma \in \mathbf{O}_{i+1} \wedge \ldots \wedge \mathbf{o}_{k-1}\gamma \in \mathbf{O}_{k-1}$$
then, since $c\theta$ is well-typed and $\mathbf{o}'\gamma \in \mathbf{O}'$, then $\mathbf{o}_i\gamma \in \mathbf{O}_i$. Since $Q\theta$ is well-typed, this fact and the hypothesis above imply $\mathbf{i}_k\gamma \in \mathbf{I}_k$., .

□

A.3 Acceptability

Definition 20 We associate to a query $Q = A_1, \ldots, A_n$ which is acceptable by $|\;|$ and I, a multiset of naturals defined as follows:
$$|Q|_I \;=\; bag\,(max\,|Q|^I_1, \ldots,\; max\,|Q|^I_n).$$
where
$$|Q|^I_i = \{|A'_i| \mid A'_1, \ldots, A'_n \text{ is a ground instance of } Q \text{ and } I \models A'_1 \wedge \ldots \wedge A'_{i-1}\}.$$
□

In the following, we denote by $\prec_m$ the finite multiset partial ordering, and by $\preceq_m$ its reflexive closure. It is well-known (see e.g. [10]) that multiset ordering over a well-founded ordering is again well-founded. In particular, the multiset ordering over the set of natural numbers with their usual ordering is well-founded.

Theorem 21 *Consider P and Q both acceptable by $|\;|$ and I. Let Q' be a SLDG-resolvent of P and Q. Then*

1. *Q' is acceptable by $|\;|$ and I,*
2. *$|Q'|_I \preceq_m |Q|_I$,*
3. *if Q' is a LDG-resolvent, then $|Q'|_I \prec_m |Q|_I$.*

Proof. We observe that every instance of a clause acceptable by $|\;|$ and I is acceptable by the same $|\;|$ and I, and analogously for queries. Therefore, by reasoning as in the proof of Lemma 5, it is readily checked that Q' is acceptable by $|\;|$ and I, and
$$bag(max|Q'|^I_1, \ldots, max|Q'|^I_{n_{Q'}}) \preceq_m bag(max|Q|^I_1, \ldots, max|Q|^I_{n_Q})$$
where n_Q and $n_{Q'}$ are the number of atoms in Q and Q' respectively. This establishes *(i, ii)*. The $\prec_m$ relation does not hold in general, since for the selected atom the corresponding set $|Q|^I_i$ may be empty, and then $max|Q|^I_i = 0$. Finally, *(iii)* follows by observing that $|Q|^I_1$ is non-empty and $max|Q|^I_1 > max|Q'|^I_k$, where k is the number of atoms in the body of the input clause. □

A.4 Simple-modedness

Lemma 10

Proof. First, we observe that if a query Q and a renamed apart guarded clause $c : A \leftarrow \mathbf{G} \mid \mathbf{B}$ guardedly unify, then the SLDG-resolvent of Q and c coincide with a SLD-resolvent of Q and $c' : A \leftarrow \mathbf{B}$. Moreover, by Definition 8, c' is simply moded. By [1, Lemma 30], we have that every SLD-resolvent of Q and c' is simply moded if the input part of the selected atom is an instance of the input part of A. Since delay declarations imply matching, then such a condition is satisfied. A *fortiori* the SLDG-resolvent of P and Q via a safe selection rule is simply moded. □

A.5 Failure-free Programs

Theorem 16

Proof. *The hypothesis of the Theorem hold for every SLDG-resolvent of P and Q via a safe selection rule*, since μ and acceptability are persistent relations.

No derivation deadlocks due to guards or delays. By *(iv,v)* and Lemmata 12 and 14.

The following result is useful in the rest of the proof.

Lemma 22 *In the hypothesis (i-iv), every LDG-derivation of P and Q is successfull.*

Proof. No LDG-derivation deadlocks due to guards or delays, by *(iv,v)* and Lemmata 12 and 14. As a consequence, the leftmost selection rule is *safe*.

Termination is an immediate consequence of Lemma 21 *(iii)*, since the multiset ordering is well-founded.

An LDG-derivation cannot fail, by Lemma 15 and the fact that $\mu(P, A)$ holds for the leftmost atom of every query in a LDG-derivation of P and Q.

Therefore, termination and absence of failures imply that every LDG-derivation is successfull. □

No derivation fails. Assume that Q is $A_1, \ldots, A_n$, and that A_i is the selected atom. By Definition of SLDG-resolution, we have that

$$(\exists\ c \in ren(P).\ try(A_i, c) = \theta) \vee (\forall\ c \in ren(P).\ try(A_i, c) = fail).$$

We show that $try(A_i, c) = \theta$ for some input clause c. By Lemma 22 there exists a successfull LDG-derivation for P and A_1. Analogously, we can proceed further by resolving (instances of) the atoms $A_2, \ldots A_{i-1}$ via the leftmost selection rule until an instance A' of A_i is selected. By Definition 1 $\mu(P, A')$ holds. By Lemma 15, we have that there exists a clause that guardedly unifies with A_i. Therefore, no SLDG-derivation via a safe selection rule can fail.

No derivation is infinite. Consider again $Q = A_1, \ldots, A_n$ where A_i is the selected atom. By Lemma 22, we there exists a successfull SLDG-derivation of Q. By Lemma 19, there exists a ground instance Q' of Q such that $I \models Q'$. As a consequence, the set

$$|Q|_i^I = \{|A_i'| :\ A_1', \ldots, A_n' \text{ ground instance of } Q \text{ and } I \models A_1', \ldots, A_{i-1}'\}$$

is non-empty. Consider now a SLDG-resolvent of Q

$$Q' =\ (A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n)\theta.$$

By Definition 6, for $l \in [1,k]$, for every $n \in |Q'|^I_{i-1+l}$ there exists $m \in |Q|^I_i$ such that $m > n$. This implies $max|Q|^I_i > max|Q'|^I_{i-1+l}$. Moreover, by Lemma 21 *(ii)*, for $l \in [1, i-1]$, we have that

$$|Q'|^I_l \preceq_m |Q|^I_l$$

and for $l \in [i+1, n]$,

$$|Q'|^I_{l+k-1} \preceq_m |Q|^I_l.$$

Summarizing, we have that

$$|Q'| \prec_m |Q|.$$

Since the multiset ordering is well-founded, we cannot have an infinite descending chain, and *a fortiori* an infinite SLDG-derivation.

In conclusion, we showed that no SLDG-derivation via a safe selection rule cannot deadlock or fail or be infinite. Then, every SLDG-derivation via a safe selection rule is successful. □

Theorem 18

Proof. First, we show that μ_0 is a persistent relation. Assume that $\mu_0(P,Q)$ holds.

Let Q' be a SLDG-resolvent of P and Q via a safe selection rule. Then every SLDG-derivation ξ of Q' is successful via a safe selection rule is successful. Otherwise, Q, ξ is a non-successfull SLDG-derivation of P and Q. Moreover, since every query Q'' in a SLDG-derivation ξ of Q' is also in a SLDG-derivation ξ of Q, then $\mu_0(P,Q)$ implies that every LDG-derivation of P and Q' is successful. Summarizing, $\mu_0(P,Q')$ holds.

Let $Q = A, Q'$. We show that $\mu_0(P,A)$ holds. Let Q'' be a query in a SLDG-derivation of P and A via a safe selection rule. Then $Q'', Q'\theta$ appears in a SLDG-derivation of P and Q via a safe selection rule, for some θ. Since $\mu_0(P,Q)$ holds, every LDG-derivation of P and $Q'', Q'\theta$ is successfull, hence every LDG-derivation of P and Q'' is successfull. As a consequence every SLDG-derivation of P and A via a safe selection rule does not deadlock due to guards or delays and does not fail. Moreover, no SLDG-derivation ξ via a safe selection rule is infinite. Otherwise by selecting clauses according to ξ, we get an infinite SLDG-derivation for Q via a safe selection rule. Summarizing, every SLDG-derivation of P and A via a safe selection rule is successful, and for every Q'' in a SLDG-derivation of P and A via a safe selection rule we have that every LDG-derivation of P and Q'' is successfull. By Definition 17, we conclude that $\mu_0(P,A)$ holds, and then that μ_0 is a persistent relation.

Assume now that $\mu_0(P,A)$ holds. Then every LDG-derivation is successfull. Therefore, A satisfies its delay declarations and there exists a clause that guardedly unifies with it, i.e. P is μ_0-failure free.

Assume now that hypotheses *(i-iv)* of Theorem 16 hold w.r.t. a persistent relation μ. Therefore, *(iii)* holds also w.r.t. μ_0. By the first part of the theorem, we know that *(iii-iv)* holds w.r.t. μ. Thus, we have only to show that $\mu_0(P,Q)$ holds. By the conclusion of Theorem 16, every SLDG-derivation of P and Q via a safe selection rule is successfull. Moreover, by persistency of acceptability every query Q' in a SLDG-derivation of P and Q via a safe selection rule, is acceptable. Finally, by Lemma 22, every LDG-derivation of P and Q' is successfull. Therefore, by Definition 17, we have that $\mu_0(P,Q)$ holds. □