Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Proving deadlock freedom of logic programs with dynamic scheduling

E. Marchiori and F. Teusink

Computer Science/Department of Interactive Systems

CS-R9642 1996

Report CS-R9642
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Proving Deadlock Freedom of Logic Programs with Dynamic Scheduling

Elena Marchiori[1,2] and Frank Teusink[1]

[1] *CWI*
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

[2] *University of Leiden*
P.O. Box 9512, 2300 RA Leiden, The Netherlands
e-mail: {elena,frankt}@cwi.nl

Abstract

In increasingly many logic programming systems, the Prolog left to right selection rule has been replaced with dynamic selection rules, that select an atom of a query among those satisfying suitable conditions. These conditions describe the form of the arguments of every program predicate, by means of a so-called delay declaration. Dynamic selection rules introduce the possibility of deadlock, an abnormal form of termination that occurs if the query is non-empty and it contains no 'selectable' atoms. In this paper, we introduce a simple compositional assertional method for proving deadlock freedom. The method is based on the notion of suspension cover, a static description of the possible dynamic schedulings of the body atoms of a clause, according to a given delay declaration. In the method, we assume that monotonic assertions are used for specifying the conditions of the delay declaration. Apart sections are devoted to two more practical instances of the method, that use types and modes, respectively.

AMS Subject Classification (1991): 68N17, 68Q15, 68Q40, 68Q60.
CR Subject Classification (1991): D.1.6, D.2.4, F.3.1.
Keywords & Phrases: Logic programs, proof method, delay declaration, deadlock freedom.

1 Introduction

In increasingly many logic programming systems, the Prolog sequential left to right selection rule has been replaced with dynamic selection rules, where the choice of an atom in a query depends on the form of its arguments. In [Nai82], Lee Naish introduced `when` declarations, which are added to a logic program to describe the dynamic selection rule that should be used. In the language Gödel [HL94] a variant of these, called DELAY declarations, are used. A delay declaration for a predicate specifies conditions about the arguments of that predicate. Thus, a selection rule may choose an atom in a query only if that atom satisfies the conditions in its delay declaration. The advantages of using a dynamic selection rule specified by such delay declarations are various. Amongst others, it can provide a synchronization mechanism by coroutining the execution of atoms in a computation, or it can be used to avoid infinite derivations by suspending parts of the computation which would cause loops ([Nai86, Nai92]). As a consequence, the run-time behaviour of logic programs with dynamic scheduling is rather subtle, because atoms in a derivation can be *suspended*. Therefore, it is important to provide the programmer with suitable tools for studying the suspension behaviour.

The techniques most widely used to analyze logic programs with dynamic scheduling are based on abstract interpretation (e.g. [CFMW93, MdlBH94, dlBMS95, CFMW93]). Apt and Luitjes in

[AL95] have studied how proof methods originally designed for logic programs with the Prolog selection rule, can be adapted to deal also with logic programs with dynamic selection rules. However, since the verification conditions of these methods reflect the Prolog left-to-right selection rule, the obtained results are not very powerful. Recently, Etalle and Gabbrielli in [EG96] have generalized the proof method based on modes given in [AL95] by introducing a more expressive notion of mode (see e.g. [AM94]) called layered mode. Another method for proving deadlock-freedom has been given by Chambre and Deransart in [CD94]: they use the approach developed in [DM93] for proving suspension freeness of queries for a class of concurrent constraint programs. However, their method is not compositional. Moreover, the verification condition contains a strong requirement that is not satisfied by those programs where a predicate can be both a 'producer' and a 'consumer'.

In this paper, we propose a simple compositional proof method for deadlock freedom. The kernel of this method is the notion of suspension cover, which relates body-atoms of a clause with sets of sub-queries. Covers are parametric with respect to the relation between atoms and subqueries one intends to study. Roughly, a suspension cover describes, by means of a multi-producer one-consumer relation between body-atoms of a clause, the inter-relations among the atoms of a clause, which can be caused by the dynamic scheduling. We will use suspension covers to develop a method for proving deadlock freedom.

The contribution of the paper is twofold. We give a theoretical result, namely that deadlock freedom is preserved under weakening of the conditions in the delay declaration. This means amongst others, that in order to prove that a query is deadlock free, we can choose stronger conditions than those of the original delay declaration. This result is used for developing a simple, compositional assertional method for proving deadlock freedom. Conditions of a delay declaration are described by means of monotonic assertions. These assertions are fixed to be the preconditions of the program predicates. Furthermore, one has to find suitable postconditions for the program predicates, that are combined with the preconditions to describe the suspension covers of a body atom. A suspension cover of a body atom A is defined by means of a bottom-up construction, whose base is the notion of direct suspension cover. A direct suspension cover is a (minimal) set of body atoms whose postconditions together with the precondition of the head of the clause, imply the precondition of A. Since the preconditions are (equivalent to) the conditions of the delay declaration, then a direct suspension cover of A guarantees that A satisfies the condition in its delay declaration after the execution of the atoms of that direct suspension cover. We prove that a program is deadlock free if for every clause, every body atom has at least one suspension cover. We investigate two more practical instances of the method, based on modes and types, respectively.

The paper is organized as follows. The next section contains some terminology on logic programs with dynamic scheduling and our persistence theorem on deadlock freedom. In Section 3 the concept of suspension cover is introduced. In Section 4, we present a proof method for deadlock freedom, and in Section 5, we discuss two practical instances of this method. Finally, Section 6 deals with related works, and Section 7 gives some conclusions. The proofs of the results presented in the paper are contained in the Appendix.

2 On Dynamic Scheduling and Deadlock Freedom

In this section we describe dynamic scheduling in logic programming and the related subject of deadlock. First, we fix the terminology used through the paper. Next, we prove an interesting result on deadlock freedom of logic programs with dynamic scheduling.

The following notation will be used. Relation symbols are denoted by p, q, r; a sequence of atoms is denoted by $\tilde{A}$ or by Q; the letters H, A, B indicate atoms and c a clause. A *program* is a finite set of clauses, $H \leftarrow Q$ together with a set of *delay declarations*. We use the following notation, borrowed from the Gödel language, to denote a delay declaration: for a predicate p with n arguments, the delay declaration for p has the form

$$\text{DELAY } p(x_1, \ldots, x_n) \text{ UNTIL } Cond(x_1, \ldots, x_n)$$

where $x_1, \ldots, x_n$ are variables representing the arguments of p, and $Cond(x_1, \ldots, x_n)$ is a formula in some (assertion) language. The meaning of such a delay declaration is, that in a query an atom $p(t_1, \ldots, t_n)$ can only be selected if the condition $Cond(t_1, \ldots, t_n)$ is satisfied. An example of delay declaration for the predicate $append/3$ is

$$\text{DELAY } append(x_1, x_2, x_3) \text{ UNTIL } Ground(x_1, x_2)$$

which states that an atom $append(s_1, s_2, s_3)$ in a query should only be selected if s_1 and s_2 are ground terms. For the purpose of our study, we do not need to fix a particular syntax for expressing $Cond(x_1, \ldots, x_n)$. So, we suppose that $Cond(x_1, \ldots, x_n)$ is expressed in an (extension) of a first-order language. Moreover, we assume that if an atom satisfies its delay declaration, then all its instances satisfy that delay declaration too. This condition is satisfied by the majority of the logic programming systems that use delay declarations. Its importance in the study of run-time properties is crucial, and all the approaches we are aware of rely on this assumption.

The delay declarations in a program define a class of selection rules, here called *delay selection rules*. For a program P with set $\mathcal{D}$ of delay declarations, a *delay selection rule* selects at every resolution step an atom A of the query Q, among those atoms of Q which satisfy their delay declaration in $\mathcal{D}$. We call *delay SLD-derivation* an SLD-derivation obtained using a delay selection rule.

Observe that, with ordinary SLD-derivations, one has three types of derivations: *infinite* derivations, and finite derivations that are either *successful* or *failed*. However, with delay SLD-derivations, there exist also finite derivations which are neither successful nor failed, but *deadlock*. In these derivations the last query is non-empty, and none of its atoms satisfies its delay declaration.

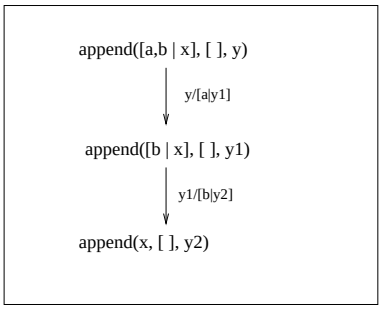

Figure 1: A deadlock derivation for $append([a, b|x], [\,], y)$

For instance, consider the 'append' program:

$$append([x|xs], ys, [x|zs]) \leftarrow append(xs, ys, zs).$$
$$append([\,], ys, ys).$$

augmented with the delay declaration

$$\text{DELAY } append(x_1, x_2, x_3) \text{ UNTIL } x_1 = [x|y] \wedge Ground(x)$$

then $append([a, b|x], [\,], y)$ has a deadlocked derivation, as is shown in Figure 1.

The aim of this paper is the study of programs having no deadlock derivations for a large class of queries, defined as follows.

Definition 2.1 (Deadlock Freedom) For a program P and a query Q, we say that:

- A delay SLD-derivation for Q is *deadlock free* if it is not a deadlock derivation.
- Q is *deadlock free (with respect to P)* if all delay SLD-derivations for Q are deadlock free.

atom	direct cover
$A1$	$\{A2, A3\}$, $\{A2, A4\}$
$A2$	$\{A3\}$
$A3$	$\emptyset$
$A4$	$\emptyset$

Figure 2: Direct covers of QS

- P is *deadlock free* if all atomic queries which satisfy their delay declaration are deadlock free. □

We prove now an expected result, namely that deadlock freedom for a query is preserved under weakening of the delay declarations.

Definition 2.2 Let $\mathcal{D}$ and $\mathcal{D}'$ be delay declarations for P. Then $\mathcal{D}'$ is *weaker* than $\mathcal{D}$ if, for every predicate p of P, if DELAY $p(x_1, \ldots, x_n)$ UNTIL $Cond'(x_1, \ldots, x_n)$ is the delay declaration for p in $\mathcal{D}'$, then the delay declaration DELAY $p(x_1, \ldots, x_n)$ UNTIL $Cond(x_1, \ldots, x_n)$ is in $\mathcal{D}$, and $\models (Cond(x_1, \ldots, x_n) \Rightarrow Cond'(x_1, \ldots, x_n))$. □

Lemma 2.3 (Weakening Lemma) *Let P be a program and let Q be a query. Let $\mathcal{D}$ and $\mathcal{D}'$ be delay declarations for P. Suppose that $\mathcal{D}'$ is weaker than $\mathcal{D}$. If Q has a deadlock delay SLD-derivation in $P \cup \mathcal{D}' \cup \{Q\}$, then Q has a deadlock delay SLD-derivation in $P \cup \mathcal{D} \cup \{Q\}$.*

The proof of this lemma uses a variant of the Switching Lemma, and is contained in the Appendix. This result is important because it allows one to use a stronger delay declaration for proving deadlock freedom of a query. It will be used in Section 4, where we shall introduce a simple method for proving deadlock freedom.

3 Suspension Covers of a Program Clause

In this section we introduce the notion of *(suspension) cover* and investigate its properties. This notion is used in the next section to define a simple compositional proof method for deadlock freedom.

In order to study deadlock freedom, we describe statically the dependences among atoms of a clause, which arise when a delay selection rule is used. Consider a generic clause of the program, say $c : H \leftarrow Q$. We relate each atom A of the body of c with a set of sets of atoms of Q, called covers. Each cover is supposed to produce suitable information which guarantees that the delay declaration for A is satisfied. In other words, the relation between atoms and subqueries of Q is a multi-producers one-consumer relation. Since the order and the multiplicity of atoms in a subquery $\tilde{C}$ of Q is here not relevant, we shall identify $\tilde{C}$ with the set of its atoms.

The construction of a cover for A is incremental: first, one has to find a minimal set of atoms of Q, say $\tilde{D}$, which are directly related with A, in the sense that after their execution A will satisfy its delay declaration. We call this set a direct cover of A. Then, for every atom B of $\tilde{D}$, a cover of B has to be added to the set so far constructed. We consider the situation in which there can be more than one direct cover. Thus, an atom can have many covers. The covers of an atom A can be graphically represented by means of an AND-OR tree, in the style of Nilsson [Nil82]. The root of the tree is A. Nodes are labeled by sets of atoms. Nodes labeled by sets containing more than one element have sets of successor nodes each labeled by one of the elements. These successor nodes are called AND nodes because in order to compute a cover of A, one cover of each of the elements of the set of atoms has to be computed. Nodes labeled by a set containing one element,

have sets of successors each labeled by one direct cover of that atom. Then, the covers of A are the sets of nodes of those paths in the tree having leave nodes equal to $\emptyset$. For instance, consider the clause:

$$H \leftarrow A1, A2, A3, A4$$

Suppose that the direct covers of this clause are as given in Figure 2. The covers of an atom, say $A1$, can be computed using the AND-OR tree for $A1$ of Figure 3. We use here Nilsson notation, and indicate an AND-node by a circular mark linking their incoming arcs. In Figure 3 there are two paths, yielding the collections of nodes consisting of the sets $\{A2, A3\}$, and $\{A2, A3, A4\}$.

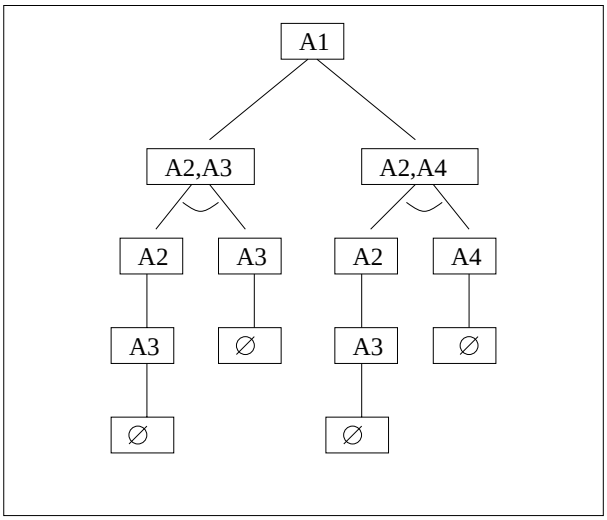

Figure 3: AND-OR tree for $A1$

In order to formalize the notion of direct cover, we use monotonic specifications, introduced by Bossi and Cocco in [BC89] for proving partial correctness of logic programs. The reader is referred to e.g. [BC89, AM94] for a thorough treatment of monotonic specifications for verification of logic programs. We recall here the main concepts.

A specification for a predicate symbol p is a pair of assertions of a suitable (extension) of a first order language, called pre- and post-condition, which describe the form of the arguments of atoms with p as predicate symbol, before and after their call. Monotonic specifications are then specifications consisting of monotonic assertions, i.e. assertions whose truth is preserved under substitution. One specification is associated with each predicate symbol of the program. We use the following notation to write down a specification of an n-ary predicate p:

$$\{Pre^p\}\, p(x_1^p, \ldots, x_n^p)\, \{Post^p\}.$$

An *asserted program* is a program augmented with one specification for every of its predicate symbols. A specification for a predicate p in P can be translated into a specification for atoms $p(t_1, \ldots, t_n)$, by linking the argument variables $x_1^p, \ldots, x_n^p$ to the terms $t_1, \ldots, t_n$ as follows. For an atom $A = p(t_1, \ldots, t_n)$ the *precondition* Pre(A) and *postcondition* Post(A) for A are:

$$\begin{aligned} Pre(A) &= Pre^p\{x_1^p/t_1, \ldots, x_n^p/t_n\} \\ Post(A) &= Post^p\{x_1^p/t_1, \ldots, x_n^p/t_n\} \end{aligned}$$

For a query $Q = A_1, \ldots, A_k$, we define $Pre(Q)$ as $Pre(A_1) \wedge \ldots \wedge Pre(A_k)$. Likewise for $Post(Q)$. We say that Q *satisfies its precondition* (resp. postcondition), if $\models Pre(Q)$ (resp. $\models Post(Q)$).

Example 3.1 One can define the following specification for the predicate *append*:

$$\{Ground(x_1, x_2) \ \vee \ Ground(x_3)\}\ append(x_1, x_2, x_3)\ \{Ground(x_1, x_2, x_3)\}$$

Here $Ground(x)$ means that x is a ground term, i.e. without variables, and $Ground(x_1, \ldots, x_k)$ is an abbreviation for $Ground(x_1) \wedge \ldots \wedge Ground(x_k)$. The atom $append([1], [2, 3], zs)$ has as pre- and post-condition $Ground([1], [2, 3]) \vee Ground(zs)$ and $Ground([1], [2, 3], zs)$, respectively. Then it satisfies its precondition, but not its postcondition. ◦

Definition 3.2 (direct suspension cover) Let c: $H \leftarrow Q$ be an (asserted) clause. A *direct (suspension) cover of A in c* is the set of pairs $(\tilde{C}, A)$, where $\tilde{C}$ is the minimal subquery of Q s.t.

$$\models Pre(H) \ \wedge \ Post(\tilde{C}) \Rightarrow Pre(A)$$

holds. □

A direct cover of an atom provides a minimal set of body atoms, which is related with that atom by means of the specified condition. Minimality is required because we want the programmer to perform as little work as possible when proving a program correct.

We formalize the notion of covers of a clause by means of the following inductive definition.

Definition 3.3 (clause covers) The set of covers of c is the least set S of pairs consisting of one set of body atoms and one body atom of c s.t.:

- $(\emptyset, A)$ is in S if $\emptyset$ is a direct cover of A in c;
- $(\tilde{C}, A)$ is in S if
  - $A \notin \tilde{C}$;
  - $\tilde{C}$ is of the form $\{C_1, \ldots, C_k\} \cup \tilde{D}_1 \cup \ldots \cup \tilde{D}_k$ s.t.
    - $\{C_1, \ldots, C_k\}$ is a direct cover of A in c, and
    - $(\tilde{D}_i, C_i)$ is in S for all i in $[1, k]$.

□

We say that *$\tilde{C}$ is a cover for A in c* if $(\tilde{C}, A)$ is in the covers of c.

Note that covers are *not* obtained by performing a kind of transitive closure of the relation *being a direct cover*, because a direct cover of A is not in general also a cover of A.

The following proposition can be immediately grasped by looking at the AND-OR tree describing the covers of an atom.

Lemma 3.4 *Let $\tilde{C}$ be a non-empty cover for B. Let A be an atom in $\tilde{C}$. Then, there exists a cover $\tilde{D}$ for A such that $\tilde{D} \subset \tilde{C}$.*

We have the following immediate consequence of this property.

Corollary 3.5 *If $\tilde{C}$ is a non-empty cover for A, then there is at least one atom of $\tilde{C}$ which has empty cover.*

This result will be used for proving the soundness of our proof method for deadlock freeness.

Observe that to construct the covers of a clause, one can use the least fixpoint construction yield by Definition 3.3. Note that every clause has a finite number of covers, because a clause consists of a finite number of atoms. Thus, Definition 3.3 provides a terminating algorithm for finding the covers of a clause (assumed that the property of being a direct cover is decidable).

4 A Method for Deadlock-Freedom

Delay declarations can be source of incompleteness when they are too restrictive; a computation can become deadlocked, when the delay declaration causes the delay of all atoms in a query. Using a weaker delay declaration, such a computation might have succeeded. In this section, we use the notion of cover to develop a simple compositional method for proving logic programs with delay declarations deadlock free.

Let us illustrate informally the operational intuition behind the method. If a query Q is deadlock free, then for every delay SLD-derivation for Q and for every query in such a derivation, every atom in that query either

- is already selectable, or
- will become selectable in some descendant of that query, unless it does fail.

Thus, in order to prove that a program is deadlock free, one has to specify statically what should be done in order to make a delayed atom selectable. If an atom in a query is delayed, it means that its delay declaration has not been satisfied (yet). This delay declaration can become satisfied by resolving some other atoms. A cover provides a set of such atoms.

Formally, we proceed as follows. First, we relate the preconditions to the delay declarations, by means of the following notion.

Definition 4.1 (good program) We say that a predicate p is *good* if its delay declaration is (equivalent to):

$$\text{DELAY } p(x_1, \ldots, x_n) \text{ UNTIL } Pre(p(x_1, \ldots, x_n))$$

A clause is *good* if every predicate in it is good. A program is *good* if every predicate in it is good. □

Next, we use the notion of goodness to define the concept of delay well-asserted program.

Definition 4.2 (delay well-asserted) A clause $H \leftarrow Q$ is *delay well-asserted* if

1. it is good, and
2. $\models Pre(H) \wedge Post(Q) \Rightarrow Post(H)$.

A program is *delay well-asserted* if every clause of it is. □

Delay well-assertedness of a program guarantees that every time an atom is called, then it satisfies its precondition, and after its execution (using a delay selection rule) it satisfies its post-condition. Thus the definition of well-assertedness allows one to prove the partial correctness of a logic program with dynamic scheduling w.r.t. a set of specifications where the preconditions are fixed to be (equivalent to) the delay declarations.

We obtain the following sufficient criterion for deadlock freedom.

Theorem 4.3 (Deadlock Freedom Theorem) *Let P be a program and let Q be a query such that*

1. *P is delay well-asserted, and*
2. *every atom occurring in Q or in the body of a clause of P has at least one cover.*

Then, every delay SLD-derivation of Q is deadlock free (with respect to P).

In order to apply Theorem 4.3, we use preconditions which are equivalent to the delay declarations. However, from Lemma 2.3, it follows that we can use preconditions that imply the conditions in the delay declarations. Therefore, in order to prove that a query in a program is deadlock free, we can apply our method to the program with a delay declaration stronger than the original one.

Observe that, from the definition of good program, it follows that every time an atom is selected, it satisfies its precondition. Therefore, from the definition of delay well-assertedness, we have that an atom satisfies its postcondition when its execution is terminated. So a program which satisfies the hypothesis of the theorem is partially correct. Thus our method allows one to prove the partial correctness of a logic program with dynamic scheduling w.r.t. a set of specifications where the preconditions are fixed to be (equivalent to) the delay declarations.

4.1 Discussion

The proof method based on Theorem 4.3 is based on the simple notion of cover for describing the possible schedulings of a program. Moreover, it is compositional, i.e. its verification condition deals with each program clause separately. As one should expect, these two nice properties affect the power of the method.

From a simple definition of cover, we have the drawback that one cannot deal directly with delay declarations consisting of a disjunction of two or more conditions. For instance, we cannot prove directly that *quicksort* is deadlock free w.r.t. the delay declaration $\mathcal{D}$ s.t.

DELAY $qs(x_1, x_2)$ UNTIL $Ground(x_1) \vee Ground(x_2)$
DELAY $part(x_1, x_2, x_3, x_4)$ UNTIL $Ground(x_1, x_2) \vee Ground(x_1, x_3, x_4)$
DELAY $app(x_1, x_2, x_3)$ UNTIL $Ground(x_1, x_2) \vee Ground(x_3)$
DELAY $> (x_1, x_2)$ UNTIL $Ground(x_1, x_2)$
DELAY $\leq (x_1, x_2)$ UNTIL $Ground(x_1, x_2)$

However, we can obtain the desired result by applying the method to one of the two *quicksort* programs obtained taking as delay declaration for a predicate the one consisting of the first and second disjunct of the original delay declaration, respectively. From the Weakening Lemma, we have for instance that queries of the form $qs(x_1, x_2)$ with x_1 ground are deadlock free w.r.t. $\mathcal{D}$. An analogous result holds for queries of the form $qs(x_1, x_2)$ with x_2 ground. So, by applying this reasoning also to *part* and *app* we obtain that *quicksort* is deadlock free w.r.t. the delay declaration $\mathcal{D}$. In order to have a method for dealing with disjunctive delay declarations in full generality, it seems that a more involved definition of cover is needed, which can deal with the case analysis caused by the disjuncts of the delay declaration.

From the compositionality, we have the drawback that the method is not applicable for proving that a query is deadlock free, when the corresponding program is not deadlock free.

The following example illustrate this situation.

Example 4.4 Consider the program *imp* (standing for *i*ncomplete *m*essage *p*rotocol).

$p([msg(y)|x]) \leftarrow$
    $read(y)$,
    $p(x)$.
$p([\,]) \leftarrow$.
$c([msg(y)|x]) \leftarrow$
    $write(y)$,
    $c(x)$.
$c([\,]) \leftarrow$.
$write(a) \leftarrow$.
$read(x) \leftarrow$.

augmented with the following delay declaration:

DELAY $p(x)$ UNTIL $true$
DELAY $c(x)$ UNTIL $x = [\,] \vee x = [y|z]$
DELAY $read(x)$ UNTIL $x = a$
DELAY $write(x)$ UNTIL $true$

It is easy to show that *imp* is not deadlock free, by considering for instance the query $p(x)$. However, the query $p(x), c(x)$ is deadlock free with respect to *imp* (see e.g. the proof given in [CD94]). However, we cannot apply our method for proving this result.

We believe that a general method for proving deadlock freedom is necessarily rather involved. Therefore, in this paper we have chosen for the simplicity and elegance, for the price of a more restrictive application range of our results. However, it seems that we can extend the applicability of our method for proving deadlock freedom of queries, by integrating it with transformational techniques. We are actually investigating a technique, where one has to find a suitable specialization of the program with respect to the considered query that allows one to apply a compositional proof method, like our one or those developed in [AL95, EG96] to the resulting program.

5 Practical Instances of the Method

In the previous section, we provided a method for proving programs deadlock free. However, there is no assumption on the assertion language to be used. In this section, we present two instances of the method, where the 'assertion' language is fixed to be the one based on modes and types, respectively.

5.1 Proving Deadlock Freedom Using Modes

In [Mel81, Red86, DM85], modes are used in verification of Prolog programs. In this section, we instantiate our method for proving deadlock freedom to the case where the assertions consist of mode declarations.

First, let us give some terminology.

Definition 5.1 (Mode) Consider an n-ary relation symbol p. A *mode* for p is a function m_p from $\{1, \ldots, n\}$ to the set $\{+, -\}$. If $m_p(i) = '+'$, then i is an *input position* of p and if $m_p(i) = '-'$,then i is an *output position* of p (both with respect to m_p).

A mode m_p for p is generally denoted as $p(m_p(1), \ldots, m_p(n))$. For an atom A, we write $Inp(A)$ (resp. $Out(A)$) to denote the set of input (resp. output) arguments of A. Also, if $\tilde{C} = A_1, \ldots, A_n$ then $Inp(\tilde{C})$ stands for $Inp(A_1), \ldots, Inp(A_n)$ and $Out(\tilde{C})$ stands for $Out(A_1), \ldots, Out(A_n)$.

A *moded program* is a logic program with one mode per predicate. □

Then, the definition of good program becomes:

Definition 5.2 Let P be a moded program. A predicate p in P is *good* if its delay declaration is the following:

$$\text{DELAY } p(x_1, \ldots, x_n) \text{ UNTIL Ground}(\text{Inp}(p(x_1, \ldots, x_n)).$$

A clause is *good* if every predicate in it is. A program is *good* if every clause in it is. □

The definition of deadlock-freedom *prod-cons* relation becomes:

Definition 5.3 Let c:$H \leftarrow Q$ be a clause. Then the deadlock-freedom *prod-cons* relation for c, denoted by DF, is s.t. $(\tilde{C}, A)$ is in DF iff

$$Var(\text{Inp}(A)) \subseteq Var(\text{Inp}(H)) \cup Var(\text{Out}(\tilde{C}))$$

□

Finally, the definition of delay well-assertedness becomes.

Definition 5.4 (delay well-moded) A clause $c : H \leftarrow Q$ is *delay well-moded* if

1. it is good, and

2. $Var(\text{Out}(H)) \subseteq Var(\text{Inp}(H)) \cup Var(\text{Out}(Q))$.

□

We conclude this section with an example.

Example 5.5 Consider the program *quicksort*:

$$
\begin{array}{l}
qs([x|xs], ys) \leftarrow \\
\qquad part(xs, x, ls, bs),\ qs(ls, sls),\ qs(bs, sbs),\ app(sls, [x|sbs], ys). \\
qs([\,], [\,]).
\end{array}
$$

$$
\begin{array}{l}
part([x|xs], y, [x|ls], bs) \leftarrow x \leq y,\ part(xs, y, ls, bs). \\
part([x|xs], y, ls, [x|bs]) \leftarrow x > y,\ part(xs, y, ls, bs). \\
part([\,], y, [\,], [\,]).
\end{array}
$$

$$
\begin{array}{l}
app([x|xs], ys, [x|zs]) \leftarrow app(xs, ys, zs). \\
app([\,], ys, ys).
\end{array}
$$

In Apt and Luitjes [AL95], they showed that the query $qs(s, y)$, with s a ground term, is deadlock free, when the moding of the program is

$$
\begin{array}{l}
qs(+, -) \\
part(+, +, -, -) \\
app(+, +, -) \\
> (+, +) \\
\leq (+, +)
\end{array}
$$

and the delay declarations are 'implied' by the moding. The same result can be proven using our method, where we choose suitable, possible stronger, delay declarations equivalent to the moding. Then, Theorem 2.3 allows us to conclude that the result holds also for weaker delay declarations. We examine here another moding for *quicksort*:

$$
\begin{array}{l}
qs(-, +) \\
part(+, -, +, +) \\
app(-, -, +) \\
> (+, +) \\
\leq (+, +)
\end{array}
$$

This moding corresponds to a non-standard use of *quicksort* to find the permutations of an ordered list of natural numbers.

As we can see, all variables that appear in the output positions of the heads of clauses, appear either in an output position in the body, or in an input position of the head. Let us now add some delay declarations in order to get a good program:

$$
\begin{array}{l}
\text{DELAY } qs(x_1, x_2) \text{ UNTIL } Ground(x_2) \\
\text{DELAY } part(x_1, x_2, x_3, x_4) \text{ UNTIL } Ground(x_1, x_3, x_4) \\
\text{DELAY } app(x_1, x_2, x_3) \text{ UNTIL } Ground(x_3) \\
\text{DELAY } > (x_1, x_2) \text{ UNTIL } Ground(x_1, x_2) \\
\text{DELAY } \leq (x_1, x_2) \text{ UNTIL } Ground(x_1, x_2)
\end{array}
$$

With these delay declarations, *quicksort* is delay well-moded. Thus, in order to prove deadlock freedom, we only need to prove that every body atom has a cover. For most clauses, this is straightforward. Therefore, we only show the (direct) covers for the first clause of qs.

atom	(direct) cover
$part(x, x_s, x_l, x_b)$	$\{qs(x_l, y_l), qs(x_b, y_b), app(y_l, [x\|y_b], y_s)\}$
$qs(x_l, y_l)$	$\{app(y_l, [x\|y_b], ys\}$
$qs(x_b, y_b)$	$\{app(y_l, [x\|y_b], ys\}$
$app(y_l, [x\|y_b], y_s)$	$\emptyset$

Note, that in this example, the covers are the same as the direct covers. Moreover, every body atom has exactly one cover.

Because all body atoms have a cover, it follows by Theorem 4.3 that *quicksort* is deadlock free. Assume that n, s and t are ground terms. Then, for instance the queries $qs(x, s)$, and $part(n, y, s, t)$ are deadlock free.

5.2 Proving Deadlock Freedom Using Types

In [DM85], types were also used for program verification of Prolog programs. In this subsection, we instantiate our approach to the case where one wants to reason using types. Such a method is more general than the method using only modes, yet simpler to implement than a method using the full power of monotonic assertions.

First, we need define the notion of a type.

Definition 5.6 (Type) A *type* is a set of terms closed under substitution. □

Note that this is a very general definition; we are not interested in the precise structure of types, or in ways to reason with types. For instance, for practical purposes, it might be advisable to restrict types to decidable sets. For our purposes, we only need the fact that a type is closed under substitution.

A *typed term* is a construct of the form $s : S$, where s is a term and S is a type. Given a sequence $\mathbf{s} : \mathbf{S} = s_1 : S_1, \ldots, s_n : S_n$ of typed terms, we write $\mathbf{s} \in \mathbf{S}$ if for $i \in [1, n]$ we have $s_i \in S_i$, and define $Var(\mathbf{s} : \mathbf{S}) = Var(\mathbf{s})$. Furthermore, we abbreviate the sequence $s_1\theta, \ldots, s_n\theta$ to $\mathbf{s}\theta$. We say that $\mathbf{s} : \mathbf{S}$ is *realizable* if $\mathbf{s}\eta \in \mathbf{S}$ for some η.

Definition 5.7 A *type judgement* is a statement of the form

$$\mathbf{s} : \mathbf{S} \;\Rightarrow\; \mathbf{t} : \mathbf{T}. \tag{1}$$

A type judgement (1) *is true*, written

$$\models \mathbf{s} : \mathbf{S} \;\Rightarrow\; \mathbf{t} : \mathbf{T}$$

if for all substitutions θ, $\mathbf{s}\theta \in \mathbf{S}$ implies $\mathbf{t}\theta \in \mathbf{T}$. □

Types for predicates are defined as follows.

Definition 5.8 (Type for p) Consider an n-ary relation symbol p. A *type* for p is a function t_p from $[1, n]$ to the set *Types*. If $t_p(i) = T$, then T is *the type associated with the position i of p*. □

In [DM85], a combination of types and modes is used. That is, one uses declarations of the form

$$member(- : Num, + : ListOfNum)$$

to denote that the predicate *member* to be used with a term of type *ListOfNum* as input in its second argument, to generate a term of type *Num* as output in its first argument. We allow only one type declaration per predicate.

We introduce some terminology and notation, that is used in the sequel. If $H = p(\mathbf{u} : \mathbf{S}, \mathbf{v} : \mathbf{T})$, then we denote $\mathbf{u} : \mathbf{S}$ by $Inp(H) : I_H$, and $\mathbf{v} : \mathbf{T}$ by $Out(H) : O_H$. Also, if $\tilde{C} = A_1, \ldots, A_n$ then $Inp(\tilde{C}) : I_{\tilde{C}}$ stands for $Inp(A_1) : I_{A_1}, \ldots, Inp(A_n) : I_{A_n}$ and $Out(\tilde{C}) : O_{\tilde{C}}$ stands for $Out(A_1) : O_{A_1}, \ldots, Out(A_n) : O_{A_n}$.

Then, the definition of good program becomes:

Definition 5.9 We say that a predicate p is good if its delay declaration is (equivalent to):

$$\text{DELAY } p(\mathbf{x} : \mathbf{I}, \mathbf{y} : \mathbf{O}) \text{ UNTIL } \mathbf{x} \text{ in } \mathbf{I}.$$

A clause is good if every predicate in it is. And a program is good if every clause in it is. □

The definition of deadlock freedom *prod-cons* relation becomes:

Definition 5.10 Let $c : H \leftarrow Q$ be a clause of P. The deadlock freedom *prod-cons* relation for c, denoted by DF, is s.t. $(\tilde{C}, A)$ is in DF iff

$$\models Inp(H) : I_H, Out(\tilde{C}) : O_{\tilde{C}} \Rightarrow Inp(A) : I_B$$

□

Finally, the definition delay well-asserted program becomes:

Definition 5.11 (delay well-typed) A clause $H \leftarrow Q$ is *delay well-typed* if

1. it is good, and
2. $\models Inp(H) : I_H, Out(Q) : O_Q \;\Rightarrow\; Out(H) : O_H$.

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

Let us see now how these results can be applied to specific programs.

Example 5.12 Consider again the program *append*. Let us type this program as follows:

$$app(+ : List, - : Top, - : List)$$

where *List* is the set of lists, and *Top* is the set of all terms. This typing corresponds to a use of the program to append a list to a generic term. Clearly *append* is delay well-asserted. Let us choose a delay declaration such that *append* is good:

$$\text{DELAY } app(x_1, x_2, x_3) \text{ UNTIL } x_1 \in List$$

It is easy to check that the atom in the body the non-unitary clause of *append* has an empty type cover. So, by Theorem 4.3 we have that *append* is deadlock free. Assume that s is a list. Then, the query $app(s, x, y)$ is deadlock free. ∘

We conclude this section by showing that the notions defined in this section and those in the previous one, are instances of the corresponding notions defined in Section 4.

Theorem 5.13

1. *The notions of Section 5.1 are instances of the corresponding notions of Section 5.2.*
2. *The notions of Section 5.2 are instances of the corresponding notions of Section 4.*

From this result, it follows that the Deadlock Freedom Theorem holds, when we replace the original definitions either with those of Section 5.2, or with those of Section 5.1.

6 Related Work

In [MT95] we have used a similar notion of cover in the verification condition of a method for proving termination of logic programs with dynamic scheduling. However, there the condition in the definition of direct cover is different, reflecting the different property we want to study, namely termination.

The notion of cover can be viewed as an alternative approach to the one based on static reordering of the atoms of a clause, as the one e.g. incorporated in the compiler of Mercury ([SHC94]). According with this latter approach, one finds a suitable reordering of the body atoms of a clause, and then applies static analysis techniques developed for Prolog programs. We think that our approach is more neat, because it allows one to reason in full generality on the

dynamic scheduling, without being committed to a specific one. Moreover, the notion of cover is constructive, and it provides an algorithm for computing all the useful reorderings. This way, we avoid to choose a specific reordering at the level of program verification. As a consequence, we leave more room to use static reorderings at a subsequent stage for other purposes, like program optimization.

In [CD94] a proof method for proving the deadlock freedom of a query in for a class of concurrent constraint programs is given. The notion of scheduling order is used to describe dependencies among sets of predicates: a partition of all the program predicates is given, and the resulting sets are arranged in a chain ordering called scheduling. Then the annotation method described in [DM93] is applied to the programs obtained by considering the clauses that define the predicates occurring in the chain prefixes. This guarantees that every derivation of the considered query where the selection of the clauses respects the scheduling order is deadlock free. Thus the result follows by the independence of the deadlock freedom from the scheduling order. There are two main differences of this method with the one we proposed. The first is that their method is not compositional, since the notion of scheduling order requires to check a condition on the program. Instead, our method works clause per clause. The second difference is that they deal with deadlock freedom of queries, while we consider (also) deadlock freedom of programs. A consequence of the first difference is that they can prove more queries to be deadlock free than by using our method; a consequence of the second difference is that their method cannot be applied directly to prove that a program is deadlock free.

Our method generalizes the two methods given by K.R. Apt and I. Luitjes in [AL95]. In essence, the difference is that we modify by means of the notion of cover, the original notions of well-modedness and well-typedness, which were introduced to deal with Prolog programs. Instead, Apt and Luitjes apply the original (stronger) notions in their methods for proving deadlock freedom. By using the notion of cover, we can prove deadlock freedom of a larger class of programs. The reason is that, well-modedness and well-typedness impose a specific order on the body atoms of a clause. Instead, our methods are independent from the order of body atoms.

An extension of the method given in [AL95] for modes has been recently introduced in [EG96]. The notion of layered mode is introduced, that is obtained by the original notion by adding some information on the order in which the arguments in an atom will be instantiated. The resulting notion of well-modedness allows one to prove deadlock freedom of a larger class of queries. It seems still less powerful than the method by [DM93], yet simpler.

In [Rao93], M.R.K. Krishna Rao defines a notion of well-moded programs, which has similarities with our condition on the existence of covers. The difference is that, he defines a producer-consumer relation on the body atoms of a clause, where the products are the variables. He then states that a clause is well-moded if this relation is acyclic and every variable has at least one producer. In the following example, we show that covers are more general than this producer consumer relation. Consider the query $p(x), q(x,y), q(y,x), p(y)$, where the modes are $p(-)$ and $q(+,-)$. Recall that a body atom A is a producer of a variable x if x occurs in its output positions, otherwise (i.e., if x does not occur in its output positions) then A is a consumer of x if it occurs in the variables of A. For the head of a clause the definition of producer and consumer are the reverse. The definition of well-modedness of a clause is based on two conditions: (a) that the producer-consumer relation is not acyclic; (b) that every variable in the clause has at least one producer.

The producer-relation for $p(x), q(x,y), q(y,x), p(y)$ is cyclic: it is the set

$$\{(p(x), q(x,y)), (q(y,x), q(x,y)), (p(y), q(y,x)), (q(x,y), q(y,x))\}.$$

On the other hand, one can compute the following (direct) covers for this query:

atom	*direct cover*	*cover*
$p(x)$	$\emptyset$	$\emptyset$
$q(x,y)$	$\{p(x)\}, \{q(y,x)\}$	$\{p(x\}, \{p(y), q(y,x)\}$
$q(y,x)$	$\{p(y)\}, \{q(x,y)\}$	$\{p(y\}, \{p(x), q(x,y)\}$
$p(y)$	$\emptyset$	$\emptyset$

The reason we can handle this query is that, in the definition of cover, we implicitly discard all cyclic paths. Then, the existence of a cover ensures that there exist acyclic paths.

7 Conclusion

In this paper we have proposed a simple compositional proof method for proving deadlock-freedom of logic programs with dynamic scheduling. The central notion used in the method is the notion of cover, which describes the possible dynamic schedulings of the body atoms of a clause, according to a given delay declaration.

The present work provides a useful theoretical tool for reasoning formally about logic programs with dynamic scheduling. In our opinion, the relevance of a simple and compositional method, is that it can be understand and used by the programmers without much effort. We are actually investigating the use of other static analysis techniques, like those based on program transformations, in order to extend the applicability of the method to queries with respect to to programs that are not deadlock free.

Acknowledgements

We would like to thank Krzysztof Apt for our useful discussions on the subject of this paper. This work was partially supported by the Netherlands Computer Science Research Foundation, with financial support from the Netherlands Organisation for Scientific Research 'NWO'.

References

[AL95] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, (AMAST'95)*, Lecture Notes in Computer Science, Berlin, 1995. Springer-Verlag. (Invited Lecture).

[AM94] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6A:743–764, 1994.

[BC89] A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Proceedings of Tapsoft*, pages 96–110, 1989.

[CD94] P. Chambre and P. Deransart. Towards a proof method of non-suspension of Concurrent Constraint Logic Programs. In F. de Boer and M. Gabbrielli, editors, *Proceedings of the W2 Post-Conference Workshop, Int. Conf. on Logic Programming*, pages 87–108. Free Univerity, 1994.

[CFMW93] M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient analysis of concurrent constraint logic programs. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proceedings of the ICALP*, pages 633–656. Springer-Verlag, 1993. LNCS 700.

[dlBMS95] M. Jose Garcia de la Banda, K. Marriott, and P. Stukey. Efficient analysis of Logic Programs with Dynamic Scheduling. In *Proceedings of the International Symposium on Logic Programming*. The MIT press, 1995. To appear.

[DM85] P. Dembinski and J. Maluszynski. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the International Symposium on Logic Programming*, pages 29–38, Boston, 1985.

[DM93] P. Deransart and J. Maluszynski. *A Grammatical View of Logic Programming*. The MIT Press, 1993.

[EG96] S. Etalle and M. Gabbrielli. *Layered Modes.* In *Proc. of the JICSLP post-conference workshop on Verification and Analysis of Logic Programs*, eds. F. de Boer and M. Gabbrielli, Bonn, Germany, 1996.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* The MIT press edition, 1994.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[MdlBH94] K. Marriott, M. Jose Garcia de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 240–253, New York, 1994. ACM Press.

[Mel81] C.S. Mellish. The automatic generation of mode declarations for Prolog programs. Technical Report DAI Research paper 166, Department of Artificial Intelligence, University of Edinburgh, 1981.

[MT95] E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 447–461. The MIT press, 1995.

[Nai82] L. Naish. An introduction to MU-PROLOG. Technical Report 82/2, Department of Computer Science, University of Melbourne, 1982.

[Nai86] Lee Naish. *Negation and Control in Prolog.* Number 238 in Lecture Notes in Computer Science. Springer-Verlag, 1986.

[Nai92] L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, Department of Computer Science, University of Melbourne, 1992.

[Nil82] N.J. Nilsson. *Principles of Artificial Intelligence.* Springer-Verlag, 1982.

[Rao93] R.K.K. Rao. *Termination Characteristics of Logic Programs.* Ph.D. Thesis, University of Bombay, 1993.

[Red86] U.S. Reddy. On the relationship between logic and functional languages. In D. DeGroot and G. Lindstrom, editors, *Functional and Logic Programming*, pages 3–36. Prentice-Hall, 1986.

[SHC94] Z. Somogyi, F.J. Henderson, and T.C. Conway. The implementation of Mercury, an efficient purely declarative logic programming language. In *Proceedings of the Post-Conference Workshop, Int. Simp. on Logic Programming*, 1994.

A Appendix

A.1 Proof of the Weakening Lemma

Theorem 2.3 (Weakening Lemma): *Let P be a program and let Q be a (definite) query. Let $\mathcal{D}$ and $\mathcal{D}'$ be delay declarations for P. Suppose that $\mathcal{D}'$ is weaker than $\mathcal{D}$. If Q has a deadlocked delay SLD-derivation in $P \cup \mathcal{D}' \cup \{Q\}$, then Q has a deadlocked delay SLD-derivation in $P \cup \mathcal{D} \cup \{Q\}$.*

To prove this theorem, we need the following lemma, essentially equivalent to Lemma 9.1 in [Llo87].

Lemma A.1 (Switching Lemma) *Let P be a program and let Q_0 a goal. Let $\eta = Q_0, Q_1, Q_2 \ldots$ be a derivation for $P \cup \{Q_0\}$, such that:*

- $Q_0 = A, B, \tilde{L}$,
- $Q_1 = (\tilde{C}, B, \tilde{L})\theta_0$, *and*
- $Q_2 = (\tilde{C}, \tilde{D}, \tilde{L})\theta_0\theta_1$.

Then there exists a derivation $\eta' = Q'_0, Q'_1, Q'_2, \ldots$ for $P \cup \{Q'_0\}$ such that

- $Q'_0 = Q_0$,
- $Q'_1 = (A, \tilde{D}, \tilde{L})\theta'_0$,
- $Q'_2 = (\tilde{C}, \tilde{D}, \tilde{L})\theta'_0\theta'_1$, *and*
- $\theta_0\theta_1$ *is a variant of* $\theta'_0\theta'_1$.

Proof: The input clauses for the first two derivation steps are $C \leftarrow \tilde{C}$ and $D \leftarrow \tilde{D}$. We have that $B\theta_0\theta_1 = D\theta_1 = D\theta_0\theta_1$. Thus we can unify B and D. Let θ'_0 be an mgu of B and D. Also, we know that, for some substitution σ, we have that $\theta_0\theta_1 = \theta'_0\sigma$.

Without loss of generality, we can assume that θ'_0 does not act on variables in $C \leftarrow \tilde{C}$. Furthermore, $C\sigma = C\theta'_0\sigma = A\theta_0\theta_1 = A\theta'_0\sigma$. Hence we can unify C and $A\theta'_0$. Let θ'_1 be a mgu. Thus, for some σ', $\sigma = \theta'_1\sigma'$. Consequently, $\theta_0\theta_1 = \theta'_0\theta'_1\sigma'$. Thus we have shown that A and B can be selected in reverse order.

We now have to show that $\theta_0\theta_1$ and $\theta'_0\theta'_1$ are variants. First, note that $A\theta'_0\theta'_1 = C\theta'_0\theta'_1$, but that θ_0 is an mgu of A and C. Thus $\theta'_0\theta'_1 = \theta_0\gamma$, for some γ. But $B\theta_0\gamma = B\theta'_0\theta'_1 = D\theta'_0\theta'_1 = D\theta'_0\gamma = D\gamma$. Thus γ unifies $B\theta_0$ and D, and therefore $\gamma = \theta_1\sigma''$ for some σ''. Consequently, $\theta'_0\theta'_1 = \theta_0\theta_1\sigma''$. This, together with the fact that $\theta_0\theta_1 = \theta'_0\theta'_1\sigma'$, implies that $\theta_0\theta_1$ and $\theta'_0\theta'_1$ are variants.

Finally, because Q'_2 is a variant of Q_2 we can complete η' by having $Q'_i = Q_i\mu$, for all $i \geq 2$, using some renaming μ. □

Proof of Theorem 2.3:
Let $\eta = Q_0, \ldots, Q_n$ be a deadlocked delay SLD-derivation in $P \cup \mathcal{D}' \cup \{Q\}$ (i.e. $Q = Q_0$). Using the Switching Lemma, we construct a delay SLD-derivation for $P \cup \mathcal{D} \cup \{Q\}$. We then show that this derivation is deadlock.

First, we construct a sequence $\eta_0, \eta_1, \ldots$ of delay SLD-derivations. We denote the j-th query in η^i as Q^i_j. We denote the selected atom in Q^i_j as A^i_j. To begin with, we set $\eta^0 = \eta$. Then, we construct η^{i+1} from η^i as follows:

Let k be the least index such that

- $k > 0$,
- A^i_k is not introduced in the resolution step $Q^i_{k-1} \rightarrow Q^i_k$,

- A_k^i is selectable in D, and
- if A_{k-1}^i is introduced in the same derivation step as A_k^i, then it is not selectable in $\mathcal{D}$.

If no such k exists, then $\eta^{i+1} = \eta^i$. Otherwise, construct η^{i+1} out of η^i by switching the selected atoms in Q_{k-1} and Q_k.

By the Switching Lemma we have that, for all i, η^i is a delay SLD-derivation of length n, for $P \cup \mathcal{D}' \cup \{Q\}$. Moreover, the queries Q_n^i (for all i) are variants of each other, and therefore all derivations η^i are deadlocked in $\mathcal{D}'$. Finally, because η is finite, we have for some finite α that $\eta^\alpha = \eta^{\alpha+1}$.

Using η^α, we construct a deadlocked delay SLD-derivation for $P \cup \mathcal{D} \cup \{Q\}$. Let m be the greatest index such that the prefix $Q_0^\alpha, \ldots, Q_m^\alpha$ of η^α is a prefix of a delay SLD-derivation for $P \cup \mathcal{D} \cup \{Q\}$. Such an m exists because the sequence consisting of only Q_0^α itself is the prefix of a delay SLD-derivation for $P \cup \mathcal{D} \cup \{Q\}$. If we have that $m = n$, then we know that Q_m has no atom selectable in $\mathcal{D}'$, thus it has no selectable atom in $\mathcal{D}$, and therefore it is deadlocked with respect to $\mathcal{D}$.

Suppose that $m < n$. We prove by contradiction that $Q_0^\alpha, \ldots, Q_m^\alpha$ is a deadlocked delay SLD-derivation for $P \cup \mathcal{D} \cup \{Q\}$. Suppose that $Q_0, \ldots, Q_m$ not a deadlocked delay SLD-derivation for $P \cup \mathcal{D} \cup \{Q\}$. As it is the prefix of a delay SLD-derivation for $P \cup \mathcal{D} \cup \{Q\}$, this implies that Q_m contains at least one atom which is selectable with respect to $\mathcal{D}$. Any atom selectable in $\mathcal{D}$, is also selectable in $\mathcal{D}'$. Because η^α is a deadlocked delay SLD-derivation for $P \cup \mathcal{D}' \cup \{Q\}$, for every atom A which is selectable in Q_m^α (with respect to $\mathcal{D}$), there exist a σ and j ($j > m$) such that $A\sigma = A_j^\alpha$. Let k be the least of these j. Then we have that $\eta^\alpha \neq \eta^{\alpha+1}$, because

- $k > 0$,
- the selected atom in Q_{m+1}^α is not selectable in $\mathcal{D}$ (otherwise m would not be maximal), which implies that $k > m+1$ and therefore, because A_k^α was introduced in Q_m or before, A_k^α is not introduced in A_{k-1}^α,
- A_k^α is selectable in $\mathcal{D}$, and
- either A_k^α is not introduced in the same derivation step as A_{k-1}^α, or A_{k-1}^α is not selectable in $\mathcal{D}$, because otherwise k would not be minimal.

But this is in contradiction with $\eta^\alpha = \eta^{\alpha+1}$. ¿From this contradiction, we can conclude that Q_m does not contain atoms which are selectable in $\mathcal{D}$, and therefore $Q_0^\alpha, \ldots, Q_m^\alpha$ is a deadlocked delay SLD-derivation for $P \cup \mathcal{D} \cup \{Q\}$. □

A.2 Proof of the Deadlock Freedom Theorem

The *prod-cons* relation DF for c is specified by means of the monotonic assertion scheme $\mathcal{A}(\tilde{C}, A) = Pre(H) \wedge Post(\tilde{C}) \Rightarrow Pre(A)$. In order to relate direct covers of an atom with direct covers of its instances, consider the *prod-cons* relation $DF\theta$ for $c\theta$ specified by means of the monotonic assertion scheme $\mathcal{A}(\tilde{C}, A)\theta = Pre(H\theta) \wedge Post(\tilde{C}\theta) \Rightarrow Pre(A\theta)$. Then we have the following result.

Proposition A.2 (monotonicity) *Let $\tilde{C}$ be a direct cover of A in c w.r.t. DF For every substitution θ, there exists a subset $\tilde{D}$ of $\tilde{C}\theta$ such that $\tilde{D}$ is a direct cover of $A\theta$ in $c\theta$ w.r.t. $DF\theta$.*

Proof: Because $\tilde{C}$ is a direct cover of A, and from the monotonicity of $\mathcal{A}(\tilde{C}, A)$, we have that either $\tilde{C}\theta$ or one of its proper subsets is a direct cover of $A\theta$ in $c\theta$ w.r.t. $DF\theta$. □

Lemma 3.4: *Let $\tilde{C}$ be a non-empty cover for B. Let A be an atom in $\tilde{C}$. Then, there exists a cover $\tilde{D}$ for A such that $\tilde{D} \subset \tilde{C}$.*

Proof: We prove the result by induction on the size n of $\tilde{C}$. Assume that the result holds for all $\tilde{C}$ of size smaller then n.

As $\tilde{C}$ is non-empty, we know by definition of cover that $\tilde{C}$ is of the form

$$\{C_1, \ldots, C_k\} \cup \tilde{D}_1 \cup \ldots \cup \tilde{D}_k$$

where $\{C_1, \ldots, C_k\}$ is a direct cover for B and, for $i \in [1..k]$, $\tilde{D}_i$ is a cover for C_i. We distinguish two cases:

- Suppose that $A = C_i$, for some $i \in [1..k]$.

  Then, $\tilde{D}_i$ is a cover for A. Moreover $\tilde{D}_i \subset \tilde{C}$, because $C_i \notin \tilde{D}_i$.

- Suppose that $A \in \tilde{D}_i$, for some $i \in [1..k]$.

  We know that $\tilde{D}_i$ is a non-empty cover for C_i, since $A \in \tilde{D}_i$. But then, because $\tilde{D}_i \subset \tilde{C}$, we know by induction hypothesis that there exists a cover $\tilde{E}$ for A such that $\tilde{E} \subset \tilde{D}_i$. Because $\tilde{D}_i \subset \tilde{C}$, it follows that $\tilde{E} \subset \tilde{C}$. □

Theorem 4.3 (Deadlock Freedom Theorem): *Let P be a program and let Q be a query such that*

1. *P is delay well-asserted, and*
2. *every atom occurring in Q or in the body of a clause of P has at least one cover.*

Then, every delay SLD-derivation of Q is deadlock free (with respect to P).

To prove this result, we proceed as follows. We consider a generic derivation η, and prove that all the queries of η satisfy the property that every of its atom has at least one cover. From this, it follows that every query of η contains at least one atom which is not delayed. Hence, η is deadlock free.

The following lemma on (direct) covers of a delay well-asserted program is used.

Lemma A.3 *Let P be a program and let Q be a query. Suppose that*

1. *P is delay well-asserted, and*
2. *every atom occurring in Q or in the body of a clause of P has at least one cover.*

Then, for every SLD-resolvent Q' of Q, every atom in Q' has at least one direct cover.

Proof: Let Q' be a SLD-resolvent of Q, using an input clause $H \leftarrow \tilde{B}$ and an mgu θ. Let C be the selected atom in Q. By hypothesis 2, there exists a direct cover $\tilde{C}$ for C in Q. By Lemma A.2, there exists a subquery $\tilde{C}'$ of $\tilde{C}\theta$ such that $\tilde{C}'$ is a direct cover for $C\theta$ in $Q\theta$.

Now, we have to prove that every atom A' in Q' has a direct cover. Let A' be of the form $A\theta$, where A is an atom in either Q or $\tilde{B}$.

- Suppose that A occurs in $\tilde{B}$.

  By hypothesis 2, there exists direct cover $\tilde{D}$ of A in $H \leftarrow \tilde{B}$. By Lemma A.2, there exists a subquery $\tilde{D}'$ of $\tilde{D}\theta$ such that $\tilde{D}'$ is a direct cover for A' in $H\theta \leftarrow \tilde{B}\theta$. Because $\models Post(\tilde{C}') \Rightarrow Pre(C\theta)$ and $C\theta = H\theta$, we have that $\models Post(\tilde{C}') \Rightarrow Pre(H\theta)$. Moreover, we have that $C\theta \notin \tilde{C}'$. It follows that $\models Post(\tilde{C}') \wedge Post(\tilde{D}') \Rightarrow Pre(A')$. Because $\tilde{C}'\tilde{D}'$ is a subquery of Q', it follows that A' has a direct cover in Q'.

- Suppose that A occurs in Q.

  By hypothesis 2, there exists a direct cover $\tilde{D}$ of A in Q. By Lemma A.2, there exists a subquery $\tilde{D}'$ of $\tilde{D}\theta$ such that $\tilde{D}'$ is a direct cover for A' in $Q\theta$. If $\tilde{D}'$ does not contain $C\theta$, we have that $\tilde{D}'$ is a subquery of Q'. Thus, A' has a direct cover in Q'.

  Suppose that $\tilde{D}'$ does contain $C\theta$. Because $C\theta$ is not in Q', $\tilde{D}'$ is not a direct cover of A' in Q'. Therefore, we need to replace $C\theta$ by some atoms that do occur in Q'. Because $C\theta = H\theta$ and $H\theta \leftarrow \tilde{B}\theta$ is delay well-asserted we know that

  $$\models Pre(C\theta) \ \wedge \ Post(\tilde{B}\theta) \Rightarrow Post(C\theta)$$

  But then, we also have that

  $$\models Post(\tilde{C}') \ \wedge \ Post(\tilde{B}\theta) \Rightarrow Post(C\theta)$$

  Finally, it follows that

  $$\models Post(\tilde{D}' - C\theta) \ \wedge \ Post(\tilde{C}') \ \wedge \ Post(\tilde{B}\theta) \Rightarrow Post(A')$$

  where $\tilde{D}' - C\theta$ is obtained from $\tilde{D}'$ by deleting $C\theta$. But then, it follows that A' has a direct groundness cover in Q'. □

Using the above lemma, we obtain the following simple proof.

Proof of Theorem 4.3: Let η be an arbitrary finite delay SLD-derivation for Q in P. By applying Lemma A.3, starting at Q, we can prove that every atom in every query in η has a suspension cover. Then, by Corollary 3.5, and by the hypothesis that P is good, it follows that every non-empty query of η contains at least one selectable atom. Hence, η is not deadlocked. □

A.3 Proof of Theorem 5.13

Theorem 5.13:

1. *The notions of Section 5.1 are instances of the corresponding notions of Section 5.2.*
2. *The notions of Section 5.2 are instances of the corresponding notions of Section 4.*

Proof:
1. Take *Ground*, consisting of all the ground terms, as the only type. Then the notions of Sections 5.1 and 5.2 coincide.
2. Translate a typed atom $p(\mathbf{x} : \mathbf{S}, \mathbf{y} : \mathbf{T})$ into the specification for p having $\{\mathbf{x} \in \mathbf{S}\}$ as precondition, and $\{\mathbf{y} \in \mathbf{T}\}$ as postcondition. Then, a program is good, according with the definition in Section 5.2 iff the corresponding asserted program obtained using the above transform is good according with the definition in Section 4. A *prod-cons* relation is a *DF* relation, according with the definition in Section 5.2 iff it is a *DF* relation, according with the definition in Section 4, where the specifications are those obtained using the above transform. Finally, a program is delay well-typed iff the corresponding asserted program obtained using the above transform is delay well asserted. □