Partial Deduction for Structured Logic Programming

Michele Bugliesi Evelina Lamma Paola Mello

D.S. Logics s.r.l.
Viale Silvani, 1
40100 Bologna, Italy

DEIS Università di Bologna
Viale Risorgimento, 2
40136 Bologna, Italy

Abstract

In this paper we discuss an extension of Partial Deduction in the framework of structured logic programs. The class of programs we consider includes statically configured systems such as block- and inheritance-based systems, as well as more dynamic configurations which support hypothetical reasoning and viewpoints.

We show that the basic Partial Deduction definition can be extended to deal with a richer class of programs while maintaining, under appropriate *closedness* conditions, the properties of soundness and completeness of the transformation which hold in the case of logic programming.

1 Introduction

The need for a structured approach to logic programming has been addressed by a number of researches in the area during the last decade. The various proposals for extending the basic logic paradigm to include a consistent set of structuring mechanisms rely on different motivations and pursue different goals.

Some of them ([23, 24, 8, 12, 19]) focus on the specific issue of program design and aim at devising a structured programming paradigm based on the notions of modules and blocks, thus enhancing program modularity, ease of maintenance and modifiability. Others ([20, 13, 11, 9]) address the issue of integrating the object-oriented notions of object and inheritance in the logic framework, thus coupling the deductive power of resolution with the more structured approach to knowledge representation espoused by the object-oriented paradigm. Still others ([10, 21]) discuss the use of separate knowledge theories to support powerful forms of hypothetical reasoning based on the notion of *viewpoints*.

The highly heterogeneous nature of these approaches has lead to a wide spectrum of different solutions. A first attempt to achieve deeper understanding of the connections among them is contained in [22, 3], where the authors develop an extensive analysis of the current proposals and define a unifying framework for their integration.

Their approach, inspired by [23] and [25], is based on the idea of structuring a program as a collection of elementary components, where each component is a separate logic theory called *unit*. Units can be (possibly dynamically) connected into linear hierarchies called *contexts*, and contexts, in turn, provide the (possibly dynamic) data-base of clauses used for evaluating the queries. The evaluation of a query is then performed as a two-step process in which one first identifies, in the current context, the set of clauses for the current atom in the query, and then evaluates each atom with respect to that set.

The flexibility of these simple notions allows the definition of a fairly large set of structuring mechanisms to be restated in terms of operations on contexts and of different policies for establishing the bindings between predicate calls and predicate definitions. Thus, blocks, modules and inheritance-based systems can be easily realized as statically configured hierarchies of units, whereas dynamic configurations provide a natural support for those artificial intelligence techniques such as viewpoints and hypothetical reasoning we have mentioned earlier.

As a further step, in [3], the authors provide a formal definition of the declarative and operational semantics for the unifying framework, which is given on the basis of a simple extension to the standard semantics of Horn Clause Logic.

Finally, in [15, 16], the authors focus on the implementation and devised a compilation scheme for their framework. The underlying abstract machine is an extension of the Warren Abstract Machine [27], where new instructions and data structures have been introduced to deal with units and contexts. The analysis of the performance of the compiled code shows the effectiveness of the approach. Yet, it also shows that a major source of run-time overhead is caused by the resolution of the bindings between predicate calls and definitions. Roughly, the cost of binding resolution amounts at a look-up access in the current context to determine the appropriate set of clauses defining a predicate call whenever such a set is not found in the same unit in which the predicate itself is called.

In this paper we show how the overhead due to context look-up may be substantially reduced by means of a source-to-source transformation of structured programs based on Partial Deduction. From this perspective, the use of Partial Deduction is to be viewed as a further enhancement to the compilative approach proposed in [15] and [16]. A structured program is transformed into a (possibly) more efficient one which can be then compiled to produce a more efficient object code.

The extension of Partial Deduction to the framework of structured logic programming is a non trivial one. First, one has to consider that the evaluation of queries occurs in a context and therefore that the set of clauses for a given predicate may dynamically vary depending on the context in which the predicate is being evaluated. This implies that the result of the transformation is not only a function of the goal but also of the structural configuration of the source program.

Furthermore, in a structured programming environment, it is highly desirable that the structural properties of programs be invariant of the transformation. In fact, this allows for an incremental transformation process to take place, in which sub-components of a given configuration can be replaced by their specialized versions without affecting the original behaviour. This also provides the basis for a smooth integration with the compilation technique described in [15] and [16].

The approach we follow originates from the formal definition of Partial Deduc-

tion in logic programming given in [18]. We extend the definition to capture the idea of specialization of a structured program with respect to a given goal and a given context. The result of the transformation is a new program in which some of (possibly all) the units occurring in the initial context are replaced by their specialized versions.

The scheme we devise here is an extension of the one we described in [6]. In that paper we presented a similar technique for the restricted case of statically configured structured systems. Now we extend those ideas to deal with dynamic configurations, and provide a more general and formal scenario. We prove the soundness and completeness of the extended scheme with respect to the operational semantics of structured programs. Following the approach introduced in [18], the soundness and completeness results are established under appropriate *closedness* conditions on the transformed program and the goal. The conditions depend on the different structuring mechanism under consideration. In the case of static configurations they amount at a syntactic check on the transformed program and the goal. As a matter of fact, in the case of block- and module-based systems, they just correspond to that introduced in [18] for definite logic programs, while some further checks on the static structure of the transformed program are required for inheritance-based systems. Conversely, for dynamic configurations, the transformation can be proved sound and complete only under a further condition on the dynamic structure of the contexts for the transformed program.

The paper is structured as follows. In section 2, we first briefly sketch the basic notions of unit and context and we show how they can support different policies for structuring logic programs. In section 3, we introduce the extended definition of Partial Deduction and develop its underlying formal framework. In section 4, we establish the main results for soundness and completeness. Finally, in section 5 we discuss the application of the general scheme to the different settings of static configurations, in section 5, and of dynamic configurations, in section 6.

Part of the results for statically configured systems described in this paper can also be found in [6] although in a less formal and detailed fashion. We restate them here to provide a self-contained description of the various techniques and applications.

2 Structured Logic Programs

Our characterization of structured logic programs originates from the Contextual Logic Programming paradigm introduced in [25]. The key idea is that a program can be conceived as a collection of independent modules called *units*. A unit is simply identified by the set of clauses it defines and by a unique atomic name used to denote it. Units can be (possibly dynamically) connected into *contexts* and contexts, in turn, provide the set of definitions for the evaluation of the queries. Contexts are represented as ordered lists of units of the form $\lfloor u_N, \ldots, u_i, \ldots, u_1 \rfloor$, and denote the union of the clauses of the component units.

Example 2.1 Let P be the program composed by the following units:

$\boldsymbol{unit}(u_1):$	$\boldsymbol{unit}(u_2):$	$\boldsymbol{unit}(u_3):$
$a(X) : -b(X).$	$b(1).$	$c(X) : -a(X).$

The proof of the goal $c(X)$ in the context $\lfloor u_3, u_2, u_1 \rfloor$ corresponds, in logic programming, to a proof for the same goal with respect to the set of clauses:

$$c(X) : -a(X).$$
$$b(1).$$
$$a(X) : -b(X).$$

Different policies for composing units into contexts can be adopted. Accordingly, different classes of structuring mechanisms can be identified as discussed in [22]. A detailed discussion of these issues and a formal definition of the semantics of the various mechanisms can be found in [3]. In the next section we briefly introduce these ideas in a more informal and intuitive fashion.

2.1 Static and Dynamic Systems

A first relevant issue concerns the distinction between static and dynamic compositions.

Statically configured systems are systems in which each unit has a *fixed* associated context. This means that, if $\lfloor u_N, \ldots, u_1 \rfloor$ is the context associated with a unit u, then, whenever u is asked for the proof of a goal g, the evaluation of g actually takes place in the context $\lfloor u, u_N, \ldots, u_1 \rfloor$. The association of a unit with a context is obtained by establishing explicit inheritance links between units. Therefore, when a unit gets defined, its ***parent*** unit must be also specified and its associated context is obtained by recursively computing the ordered list of its ancestors. Formally, if $U(P)$ and $C(P)$ respectively denote the set of units of a program P and the set of contexts formable over $U(P)$, the context associated with a unit u is computed by the function $hierarchy : U(P) \mapsto C(P)$ defined as follows:

$$hierarchy(u) = \begin{cases} u.hierarchy(u') & \text{if } u' \text{ is the parent unit of } u. \\ \lfloor\ \rfloor & \text{if } u \text{ is the top unit of the hierarchy} \end{cases}$$

where "." is the list constructor.

The empty unit, *top*, is assumed to be the root for all the hierarchies. The switch operator ":" allows us to switch from one hierarchy to another. The invocation of a goal of the form $u : g$ in the current hierarchy causes a switch to the hierarchy associated with the unit u.

Example 2.2 Let P be the following structured program:

$\boldsymbol{unit}(u_1, \boldsymbol{parent}(top)) :$	$\boldsymbol{unit}(u_2, \boldsymbol{parent}(u_1)) :$	$\boldsymbol{unit}(u_3, \boldsymbol{parent}(u_2)) :$
$a(X) : -b(X).$	$b(1).$	$c(X) : -a(X).$

The unit specified as parameter of the ***parent*** structure represents the inheritance link for the unit itself. Then, $\lfloor u_2, u_1 \rfloor$ is the context associated with the unit u_3, and, accordingly, the call $u_3 : c(X)$ enforces $\lfloor u_3, u_2, u_1 \rfloor$ as the context for evaluating $c(X)$.

Examples of statically configured systems are blocks and modules [12], Meta-Prolog [2, 1], McCabe's Class Template Language [20], Multilog [13], and many others including [11] and [9].

Dynamically configured systems, on the other side, provide a more flexible framework in which, beside switching to a new context, one can also dynamically extend the current one to include new units. Each unit can have therefore a dynamic associated context. This is actually the idea behind the proposal in [25]. The meaning of the *context extension* operator $\gg$ can be understood as follows: the evaluation of the goal $u \gg G$ succeeds in a context $\lfloor u_N, \ldots, u_1 \rfloor$ if and only if the evaluation of G succeeds in the context $\lfloor u, u_N, \ldots, u_1 \rfloor$. Then in example 2.1, the context $\lfloor u_3, u_2, u_1 \rfloor$ for the goal $c(X)$ can be built through the sequence of context extensions $u_1 \gg u_2 \gg u_3 \gg c(X)$.

Examples of dynamically configured systems are Miller's one [23], Contextual Logic Programming [25], and N-Prolog [10].

2.2 Conservative/Evolving Systems

The evaluation of a goal in a context involves two issues: first we have to identify, in that context, the appropriate set of clauses which provide the definition for the goal, and then we solve the goal with respect to that definition.

An important remark here concerns the distinction between conservative and evolving policies. Let us suppose that $C = \lfloor u_N, \ldots, u_i, \ldots, u_1 \rfloor$is the current context. An *evolving system* is a system in which the definition associated with each predicate call is given by the clauses for g contained in the whole context C, regardless of the unit where the call occurs. Examples of evolving systems are Miller's one [23, 24], Multi-Prolog [7] and N-Prolog [10].

A *conservative system* is a system in which the definition associated with predicate call g occurring in the unit u_i, is given by the clauses for g contained in the sub-context $\lfloor u_i, \ldots, u_1 \rfloor$. The definitions *visible* from a unit u are therefore those found in u and its ancestors in the current context. Examples of conservative systems are blocks and modules [12], Meta-Prolog [1] and Contextual Logic Programming [25].

Example 2.3 Let us consider the statically configured program of example 2.2 and the top goal: $u_3 : c(X)$. We use the notation $Ctx \vdash G$, introduced in [25], and defined formally in the appendix, to explicitly represent the context associated with each derivation step. In an evolving system, we get the derivation steps:

$$\begin{array}{lcl} \lfloor\ \rfloor & \vdash & u_3 : c(X) \\ \lfloor u_3, u_2, u_1 \rfloor & \vdash & c(X) \\ \lfloor u_3, u_2, u_1 \rfloor & \vdash & a(X) \\ \lfloor u_3, u_2, u_1 \rfloor & \vdash & b(X) \\ success & : & \{X \leftarrow 1\} \end{array}$$

Notice that the formula $b(X)$ is proved in the whole context $\lfloor u_3, u_2, u_1 \rfloor$ even though it is called in the sub-context $\lfloor u_1 \rfloor$ where the definition $a(X)$:-$b(X)$ has been found.

Conversely, in a conservative system, the context for the evaluation of $b(X)$ is

$\lfloor u_1 \rfloor$ and, since $\lfloor u_1 \rfloor$ contains no definition for $b(X)$, the query $u_3 : c(X)$ fails.

$$\begin{array}{ll} \lfloor\ \rfloor & \vdash\ u_3 : c(X) \\ \lfloor u_3, u_2, u_1 \rfloor & \vdash\ c(X) \\ \lfloor u_3, u_2, u_1 \rfloor & \vdash\ a(X) \\ \lfloor u_1 \rfloor & \vdash\ b(X) \\ failure & \end{array}$$

2.3 Examples of Structuring Policies

Most of the proposals for structuring logic programs found in the literature can be included in the above classification. A detailed discussion about this topic is reported in [3]. Here we briefly sketch some clarifying examples.

2.3.1 Blocks and Modules

In block-based systems, static scope rules determine the visibility of a predicate definition on the basis of the nesting of blocks in the program. To prove an atomic goal occurring in a clause of a given block, only the clauses defined in that block or in enclosing blocks can be used. In terms of our classification, such a behaviour corresponds to a statically configured system with a conservative policy, where each block is encapsulated into a separate unit having the enclosing block as its ***parent***.

Example 2.4 Let us consider the following program P inspired by [12]:

$$\begin{array}{lll} member(X, [Y|\_]) & :- & \{eq(X_1, X_1). \\ & & eq([X_1|A], [Y_1|B]) :- \\ & & \qquad perm([X_1|A], [Y_1|B])\} \Rightarrow eq(X, Y)\} \\ member(X, [\_|Y]) & :- & member(X, Y). \\ perm([], []). & & \\ perm(L, [X|P]) & :- & delete(X, L, L_1), \\ & & perm(L_1, P). \\ delete(X, L, L_1) & :- & ... \end{array}$$

The program is structured into two nested blocks: b_1, the enclosing one, which contains the definitions for *member* and *perm*, and b_2, the inner one, which contains the definitions for *eq*. Block b_2 is nested into the first clause of block b_1. P can be mapped into the following conservative, statically configured program:

$$\begin{array}{ll} \boldsymbol{unit}(b_1, \boldsymbol{parent}(top)) : & \boldsymbol{unit}(b_2, \boldsymbol{parent}(b_1)) : \\ member(X, [Y|\_]) : -b_2 : eq(X, Y). & eq(X, X). \\ member(X, [\_|Y]) : -member(X, Y). & eq([X_1|A], [Y_1|B]) : - \\ & \qquad perm([X_1|A], [Y_1|B]). \\ perm([], []). & \\ perm(L, [X|P]) : -delete(X, L, L_1), & \\ \qquad perm(L_1, P). & \\ delete(X, Y, Z) : -... & \end{array}$$

Notice that, being the system conservative, the scope rules are now expressed in terms of the visibility rules for the context $\lfloor b_2, b_1 \rfloor$. In fact, the definition for *perm* is now visible in both b_1 and b_2, while that for *eq* is only visible in b_2.

The above scope rules become more strict in the case of closed modules ([2, 12, 19]): proving a goal in a module M, means considering only the predicate definitions which are local to M. As such, module-based systems are actually a special case of statically configured conservative systems, in which units always have an empty associated context and therefore they only use their local definitions. In practice, modules are defined as units which are siblings rather than parents or children. Since we do not provide *import* declarations, external definitions are accessible only through an explicit reference to the module containing them. Import declarations in a multi-theory framework are discussed in [23, 4].

Example 2.5 In a module-based system, the example 2.4 can be structured as follows:

$\boldsymbol{unit}(m_1, \boldsymbol{parent}(top)) :$
$member(X, [Y|\_]) : -m_2 : eq(X, Y).$
$member(X, [\_|Y]) : -member(X, Y).$

$perm([], []).$
$perm(L, [X|P]) : -delete(X, L, L_1),$
$\quad perm(L_1, P).$
$delete(X, Y, Z) : -...$

$\boldsymbol{unit}(m_2, \boldsymbol{parent}(top)) :$
$eq(X, X).$
$eq([X_1|A], [Y_1|B]) : -$
$m_1 : perm([X_1|A], [Y_1|B]).$

The definition for *perm* is no longer implicitly visible from m_2, and then the corresponding definition in m_1 can only be accessed by means of an explicit context switch operation.

2.3.2 Inheritance-based systems

Inheritance- and object-based systems can be classified as statically configured evolving systems. We interpret contexts as the explicit representation of a branch in an inheritance tree (for the sake of simplicity we will not consider multiple inheritance). The first unit in the context is the tip node, while the last one is the top of the hierarchy. Inheritance-based systems are intrinsically evolving since, as stated in [29], "a self-reference in a type or class is bound to the object on whose behalf an operation (proof) is being executed, rather than to the class where the self-reference occurs".

Example 2.6 Let us consider the class template language described in [20]. When we say that a *bird* is a special case of *animal* we are stating that whatever holds for animals also holds for birds: the theory *bird* inherits from the theory *animal*. In the class template language we express this kind of relationship between classes

by means of class rules:

$$\begin{array}{lcl} bird & \Leftarrow & animal \\ tweety & \Leftarrow & bird \\ horse & \Leftarrow & animal \\ human & \Leftarrow & animal \end{array}$$

where:

$$animal: \left[\begin{array}{l} mode(walk). \\ mode(run) : -self : no\_of\_legs(2). \\ mode(gallop) : -self : no\_of\_legs(4). \end{array}\right.$$

$$bird: \left[\begin{array}{l} mode(fly). \\ no\_of\_legs(2) \\ covering(feather). \end{array}\right.$$

$$horse: \left[\begin{array}{l} no\_of\_legs(4). \end{array}\right.$$

$$human: \left[\begin{array}{l} no\_of\_legs(2). \end{array}\right.$$

$$tweety: [no\_of\_wings(2).$$

The call *self:g* causes the proof of g to be performed in the tip class of the current hierarchy, no matter what the current class is. The use of *self* allows therefore the expected behaviour of inheritance to be modeled.

In our framework, this program can be translated into the following evolving one:

$$\boldsymbol{unit}(animal, \boldsymbol{parent}(top)) : \left[\begin{array}{l} mode(walk). \\ mode(run) : -no\_of\_legs(2). \\ mode(gallop) : -no\_of\_legs(4). \end{array}\right.$$

$$\boldsymbol{unit}(bird, \boldsymbol{parent}(animal)) : \left[\begin{array}{l} mode(fly). \\ no\_of\_legs(2) \\ covering(feather). \end{array}\right.$$

$$\boldsymbol{unit}(horse, \boldsymbol{parent}(animal)) : \left[\begin{array}{l} no\_of\_legs(4). \end{array}\right.$$

$$\boldsymbol{unit}(human, \boldsymbol{parent}(animal)) : \left[\begin{array}{l} no\_of\_legs(2). \end{array}\right.$$

$$\boldsymbol{unit}(tweety, \boldsymbol{parent}(bird)) : [no\_of\_wings(2).$$

Now the taxonomy is embedded in the parent declaration, and the *self* behaviour is automatically expressed by the evolving policy.

2.4 Viewpoints

Dynamic configurations, together with the context extension operator, provide a natural support for a limited form of hypothetical reasoning based on viewpoints.

In fact, extending the current context with a new unit can be understood as adding a new hypothesis to the current line of reasoning. Then, if a unit u embodies some hypotheses, evaluating the goal $u \gg G$ means evaluating G after assuming the hypotheses in u. From this point of view, the extension operator is very similar to the *assume* predicate introduced in [28] and to a restricted form of *embedded implication* ([23, 21, 10]).

Systems for hypothetical reasoning and viewpoints can be classified as a particular form of dynamically configured evolving systems, where hypotheses or viewpoints are collected into units. These systems are intrinsically evolving since the newly added knowledge always updates the old knowledge. Similarly, they have to be dynamically configured, in order that new hypotheses or viewpoints can be added to the dynamic state of the computation.

Example 2.7 The following dynamically configured evolving program provides a structured representation of the well-known example of the block world.

$\boldsymbol{unit}(u_0):$
$on(a,b).$
$on(b,c).$
$next(X,Y) :- on(X,Y).$
$next(X,Y) :- on(Y,X).$
$color(b,black).$
$next\_white(B) :- next(B,X), color(X,white).$
$\boldsymbol{unit}(main):$
$next\_w(B) :- v_1 \gg next\_white(B), v_2 \gg next\_white(B).$

$\boldsymbol{unit}(v_1):$
$color(a,white).$
$\boldsymbol{unit}(v_2):$
$color(c,white).$

The unit u_0 represents a given *state* of the world. Units v_1 and v_2, in turn, represent two possible viewpoints expressing the additional knowledge:

$$color(a,white) \vee color(c,white).$$

In order to prove that there exists a block next to a white block, we use the definition for *next_w* in unit *main* which takes into account both the viewpoints. Accordingly, the goal $u_0 \gg main \gg next\_w(B)$ succeeds with substitution $\{B \leftarrow b\}$.

2.5 Expressive Power vs Efficiency

The expressive power and the flexibility of the various context-based mechanisms described so far are achieved at the expense of a high computational cost. As already mentioned in the introduction and as shown in the examples of this section, the major source of overhead is the resolution of the binding for each predicate call. In fact, each step in the evaluation involves a preliminary search in the current context for the clauses defining the current goal. In [15] and [16], the authors show how, for static configurations, binding resolution for the conservative policy can be actually performed at compile time. Conversely, in the case of dynamic configurations, it must be delayed until run-time, thus lowering the performance

of the compiled code. The worst situation occurs in the case of failure branches since, when there exists no definition for the current goal, the cost of the access to the context is linear with the length of the context itself.

A solution to the problem of failure branches can be achieved by resorting to well-known techniques for compile-time failure detection. A further optimization can be achieved by *specializing the configuration* of the program with respect to the given query. Intuitively, this amounts to *sliding* the definition for each predicate call towards the top of the context in which the call is being evaluated, thus reducing the number of look-up steps needed to compute the appropriate binding.

Static detection of failure branches is a standard application of Partial Deduction. Our approach to structured program specialization is the subject of the next section.

3 Partial Deduction for Structured Programs

Partial Deduction (henceforth called PD) has been devoted great attention during the last few years since it has been introduced in logic programming [14]. PD is a source-to-source transformation technique which, given a program P and a goal G, produces a new program P', which is more efficient than P and has the same set of answer substitutions for the goal G and its instances. As stated in [18] "the basic technique for obtaining P' from P is to construct a partial search tree for P and suitably chosen atoms as goals, and then extract the definitions — the *resultants* — associated with the leaves of the tree".

The extension of Partial Deduction to the framework of structured logic programs involves two issues. First, it has to be considered that the evaluation of the initial goal occurs in a context and that, therefore, the result of PD depends on the initial goal as well as on the initial context in which the goal is evaluated.

The second issue concerns the structure of the transformed program. A first possible approach to structured program specialization could be based on the following observation. Given a query and a context, the set of program clauses defining each predicate call can be determined during the symbolic evaluation which PD is based on. Then, one could compute all the bindings during PD and then produce a *flattened* configuration of the program in which the bindings are established in terms of a naming convention.

Example 3.1 Let P be the following conservative program:

$$\begin{array}{ll} \boldsymbol{unit}(u_1): & \boldsymbol{unit}(u_2): \\ a(X) :- b(X). & b(1). \\ b(2). & \end{array}$$

We observe that evaluating the goal $a(X)$ in the initial context $\lfloor u_2, u_1 \rfloor$ simply corresponds to evaluating the goal $a_{u_1}(X)$ with respect to the program P' defined as follows:

$$\begin{array}{l} a_{u_1}(X) :- b_{u_1}(X). \\ b_{u_1}(1). \\ b_{u_2}(2). \end{array}$$

Notice that the call $b(X)$ in unit u_1 has been bound to the definition for b found in unit u_1, being $\lfloor u_1 \rfloor$ the context in which the evaluation of $b(X)$ actually takes place. Then, P' could be chosen as the specialized version of P with respect to the goal $a(X)$ and the context $\lfloor u_2, u_1 \rfloor$.

The first problem with this approach is that, by flattening the structure of the source program, the configuration which results from the transformation looses the flexibility of the original modular configuration. Furthermore, if we consider u_1 and u_2 of example 3.1 as sub-components of a more general system of unit, then this approach raises the question of how to define the interactions between the new module P' and other components of the system.

Preserving the structural properties of the source programs will be therefore one of the crucial requirements for our transformation scheme. Obviously, we will also impose that the new configuration be sound and complete with respect to the original one. Soundness and completeness are indeed the main foundational questions about Partial Deduction in logic programming [18]. In this latter framework, we say that a partially evaluated program P' is *sound* with respect to the original program P and the goal G, if each computed answer for G and P' is a computed answer for G and P. The new program P' is *complete* with respect to P and G, if the converse of the above implication holds true. According to this definition, the soundness and completeness of Partial Deduction in Logic Programming can proved, as done in [18], under a closedness condition required for the transformed program and the goal.

Our construction re-establishes these properties in the extended framework of structured logic programs.

3.1 Preliminaries

We restrict our discussion to the case of an *atomic* initial goal. This assumption simplifies the development of our framework, which relies on the following definition of *resultant* ([18]).

Definition 3.1 *Let G_0 be an atomic formula, :-$G_0, \ldots,$:-G_j a sequence of goals associated with an SLD-derivation S for G_0 and $\sigma_1, \ldots, \sigma_j$ the corresponding sequence of substitutions. Let also σ denote the restriction of the composition of $\sigma_1, \ldots, \sigma_j$ to the variables of G_0. Then $G_0\sigma : -G_j$ is the resultant associated to S and G_j is called the* residual *associated with the resultant.*

As pointed out in [18], the requirement that the initial goal be atomic guarantees that at each step of a derivation, the associated resultant is indeed a program clause.

In this section, we consider a restricted class of structured programs in which no context switch (nor extension) occurs during the evaluation of a query. The treatment of context switch and context extension is thus postponed until sections 5 and 6. We first introduce an extended definition of derivation to take into account the notion of context. Let:

- $\langle G, C \rangle$ denote a *c-atom*, where G is an atomic formula and C a context,

- a *c-goal* be a conjunction of c-atoms,
- $\langle (g_1, \ldots, g_n), c\rangle$ be a shorthand for $\langle g_1, c\rangle, \ldots, \langle g_n, c\rangle$.

Definition 3.2 (Derivation) *Let P be a program, CG be the c-goal*

$$\langle g_1, c_1\rangle, \ldots, \langle g_i, c_i\rangle, \ldots, \langle g_n, c_n\rangle$$

and let $\langle g_i, c_i\rangle$ be the selected c-atom, where c_i is the context $\lfloor u_i, \ldots, u_j, \ldots, u_1\rfloor$. We say that the new c-goal CG' is derived from CG and P via the substitution σ, the clause Cl and the context L_C, if the following conditions hold:

$$\begin{array}{ll} Cl & = h : -b_1, \ldots, b_m \text{ is the selected clause in the unit } u_j \text{ of } c_i \\ \sigma & = mgu(h, g_i) \\ CG' & = [\langle g_1, c_1\rangle, .., \langle g_{i-1}, c_{i-1}\rangle, \langle (b_1, \ldots, b_m), c'\rangle, \langle g_{i+1}, c_{i+1}\rangle, \ldots, \langle g_n, c_n\rangle]\sigma \\ L_C & = \lfloor u_j, \ldots, u_1\rfloor \text{ is the context whose top unit contains the clause } Cl. \end{array}$$

L_C *the* label context *associated with the derivation step. The new context c' is computed according to the operational semantics for conservative and evolving systems (see Appendix). Namely:*

$$c' = \begin{cases} L_C & \text{if the system is conservative} \\ c_i & \text{if the system is evolving} \end{cases}$$

We will henceforth denote a derivation step from a c-goal CG to a c-goal CG', by $CG \vdash_{L_C, \sigma_j} CG'$ where the substitution σ_j will be omitted unless explicitly required. According to definition 3.2, we then define a *C-SLD derivation* for a c-goal $CG_0 = \langle G_0, C_0\rangle$ to be any (finite or infinite) sequence of derivation steps $CG_0 \vdash_{c_1, \sigma_1} \ldots \vdash_{c_j, \sigma_j} CG_j \ldots$

Finally, we define the notions of *successful C-SLD derivation* (*C-SLD refutation*), of *C-SLD tree* and of *c-resultant* by generalizing the corresponding definitions given in logic programming [17].

Definition 3.3 (C-SLD Refutation) *Let P be a program and $CG_0 = \langle G_0, C_0\rangle$ be a c-goal. A C-SLD refutation of $P \cup \{\langle G_0, C_0\rangle\}$ is a finite C-SLD derivation for $\langle G_0, C_0\rangle$ in P which has the empty formula as the last formula in the derivation.*

Example 3.2 Let P be the statically configured evolving program:

$$\begin{array}{ll} \boldsymbol{unit}(m_1, \boldsymbol{parent}(top)) : & \boldsymbol{unit}(m_2, \boldsymbol{parent}(m_1)) : \\ p : -q. & q. \end{array}$$

Starting from the query p in the context $\lfloor m_2, m_1\rfloor$ we have the following C-SLD refutation, where each edge is marked by the label context used in the corresponding

derivation step.

$$\begin{array}{c} \langle p, \lfloor m_2, m_1 \rfloor \rangle \\ | \\ \lfloor m_1 \rfloor \\ | \\ \langle q, \lfloor m_2, m_1 \rfloor \rangle \\ | \\ \lfloor m_2, m_1 \rfloor \\ | \\ \Box \end{array}$$

Definition 3.4 (c-resultant) *Let G_0 be an atomic formula and C_0 a context. Then the c-resultant associated with the C-SLD derivation:*

$$\langle G_0, C_0 \rangle \vdash_{c1,\sigma_1} \ldots \vdash_{cj,\sigma_j} \langle g_1, C_1 \rangle, \ldots, \langle g_n, C_n \rangle$$

is given by:

$$G_0\sigma : -\langle g_1, C_1 \rangle, \ldots, \langle g_n, C_n \rangle.$$

where σ be the restriction of the composition of $\sigma_1, \ldots, \sigma_j$ to the variables of G_0 and $\langle g_1, C_1 \rangle, \ldots, \langle g_n, C_n \rangle$ is the residual associated with the c-resultant.

Notice that, if the residual of a c-resultant is empty, then the c-resultant is actually a resultant. The definition of c-resultant is the first important brick of our construction. Remember that the final goal of the transformation is to produce a new configuration for the program that supports a more efficient computation of the bindings. This will be achieved by generalizing the technique introduced in [18]. Namely, given the initial goal G and the context C, we will first compute the set of c-resultants associated with the partial C-SLD tree for $\langle G, C \rangle$. Then we will define the new configuration for the program by assigning each c-resultant to the appropriate unit by means of a semantic-preserving *assignment function*. Finally, we will derive from each c-resultant the corresponding resultant. The newly generated resultants will then replace the old definitions for G contained in C.

3.2 Configuring the Transformed Program

The next step in the development of our PD scheme consists of defining a semantic-preserving assignment function to determine which units the c-resultants should be assigned to. Notice that different choices for the assignment function determine different configurations for the transformed program and, correspondingly, different properties for the transformation.

We finally come to the question of what we mean by *preserving the semantics* of structured programs. In logic programming, Partial Deduction provides a specialization of the program with respect to a goal. This means that, if G is the goal chosen to specialize the source program, the transformed program will produce a complete set of computed answers only for queries that are *less general* (more instantiated) than G [18].

In the same way, in structured logic programming, since we specialize the program with respect to a goal in a context C, the transformed program will produce

a complete set of computed answers only for that goal in contexts *less general* than C. In this case, *less general* means *logically subsumed*. We say that a context C' is logically subsumed by a context C if for each C-SLD refutation in C' computing a substitution σ there exists a corresponding C-SLD refutation in C computing the same substitution. This notion can be actually stated in terms of a structural relation over contexts. We say that a context C' *is a sub-context of* C ($C' \sqsubseteq C$) if C' is an initial sub-list of C. Notice that, being the composition of units in contexts a monotonic operation, it is easy to show that C' is logically subsumed by C if $C' \sqsubseteq C$. Then what we can expect from the transformation is that it preserves the equivalence between the source and the transformed programs with respect to instances of G (as in [18]) and *sub-contexts* of C, i.e. with respect to all those computations in which instances of G are evaluated in sub-contexts of C.

Choosing a semantic-preserving assignment function is not straightforward. In fact, replacing each clause with the corresponding set of resultants in the same unit, which might appear the obvious choice, may lead to unsoundness even in sub-contexts of the initial one.

Example 3.3 Let P be the statically configured evolving program:

$$\begin{array}{ll} \boldsymbol{unit}(m_1, \boldsymbol{parent}(top)) : & \boldsymbol{unit}(m_2, \boldsymbol{parent}(m_1)) : \\ p :- q. & q. \end{array}$$

By partially evaluating P wrt the goal p in the context $\lfloor m_2, m_1 \rfloor$ we obtained the c-resultant $CR = p$. Notice that, since the residual for p is empty, p is also a resultant. Therefore, by applying the assignment function defined above, we achieve the final configuration by simply assigning p to the unit m_1.

$$\begin{array}{ll} \boldsymbol{unit}(m_1, \boldsymbol{parent}(top)) : & \boldsymbol{unit}(m_2, \boldsymbol{parent}(m_1))) : \\ p. & q. \end{array}$$

The transformation is unsound: in fact, in the context $\lfloor m_1 \rfloor$, which is a sub-context of the initial one, the goal p fails in P and succeeds in P'.

The problem comes from the fact that, while in P the body of the clause $p :- q.$ needs the definition of q in m_2 to succeed, the corresponding c-resultant in P' succeeds in the sub-context $\lfloor m_1 \rfloor$ without even calling q.

In other words, while in P the *deduction context* for $p :- q.$ is $\lfloor m_2, m_1 \rfloor$, in P' the *deduction context* for the corresponding c-resultant $p.$ is $\lfloor m_1 \rfloor$. In order to make the transformation sound, we should instead ensure that the deduction context for the clause and the corresponding c-resultant be preserved.

The notion of deduction context can be given formally as follows.

Definition 3.5 (Deduction context) *Let CR be the c-resultant associated with a C-SLD derivation from the c-goal* $\langle G, C \rangle$. *The deduction context* $\mathcal{D}c(CR)$, *of CR is the minimal sub-context of* C, *needed to derive CR.*

The deduction context can be obtained from a C-SLD derivation by considering the label contexts associated to the derivation steps.

Proposition 1 (Deduction context) *Given a finite C-SLD derivation*

$$\langle G_0, C_0\rangle \vdash_{c_1,\sigma_1} \ldots \vdash_{c_j,\sigma_j} \langle g_1, C_1\rangle, \ldots, \langle g_n, C_n\rangle$$

the deduction context $\mathcal{D}c(CR)$ for the c-resultant

$$CR = G_0\sigma : -\langle g_1, C_1\rangle, \ldots, \langle g_n, C_n\rangle$$

is the maximum element of the set $\{c_1, \ldots, c_j\}$.
Proof. We notice that, since we assume that no context switch or context extension occurs during the C-SLD derivation, the computation of the maximum is well-defined. The proof follows immediately from the definition of C-SLD derivation.

The deduction context for a c-resultant is preserved through the transformation by simply assigning the c-resultant to the *top* unit of its deduction context. This brings us to the following definition of *assignment function*.

Definition 3.6 (Assignment function: $\mathcal{F}_{ass}$) *Let T be a finite C-SLD tree. Let CR be a c-resultant in T and $\mathcal{D}c(CR) = \lfloor u_k, \ldots, u_1\rfloor$ be the deduction context for CR. Then*

$$\mathcal{F}_{ass}(CR) = u_k.$$

3.3 From c-resultants to resultants

Once the new configuration of the program has been defined, the last step consists of establishing the association between c-resultants and the corresponding resultants. Namely, given the c-resultant $G_0\sigma : -\langle g_1, C_1\rangle, \ldots, \langle g_n, C_n\rangle$ we's like to transform it into a *clause* of the form $G_0\sigma : -g'_1, \ldots, g'_n$. The problem is obviously how to derive g'_i in terms of g_i and C_i. It is easy to see that the simple choice $g'_i = g_i$ is not correct.

Example 3.4 In a conservative statically configured system, let P be the following program:

$\boldsymbol{unit}(u_1, \boldsymbol{parent}(top)) :$	$\boldsymbol{unit}(u_2, \boldsymbol{parent}(u_1)) :$	$\boldsymbol{unit}(u_3, \boldsymbol{parent}(u_2)) :$
$q : -r.$	$r.$	$p : -q.$

Let's the consider the c-resultant $p : -\langle r, \lfloor u_1\rfloor\rangle$ obtained from the following partial C-SLD tree for $\langle p, \lfloor u_3, u_2, u_1\rfloor\rangle$.

$$\begin{array}{c}\langle p, \lfloor u_3, u_2, u_1\rfloor\rangle \\ | \\ \lfloor u_3, u_2, u_1\rfloor \\ | \\ \langle q, \lfloor u_3, u_2, u_1\rfloor\rangle \\ | \\ \lfloor u_1\rfloor \\ | \\ \langle r, \lfloor u_1\rfloor\rangle\end{array}$$

Then, u_3 is the unit associated with $p : -\langle r, \lfloor u_1 \rfloor \rangle$ and with the corresponding resultant $p : -r$ since the deduction context is $\lfloor u_3, u_2, u_1 \rfloor$. Accordingly, the new program P' is:

$\boldsymbol{unit}(u_1, \boldsymbol{parent}(top)) :$	$\boldsymbol{unit}(u_2, \boldsymbol{parent}(u_1)) :$	$\boldsymbol{unit}(u_3, \boldsymbol{parent}(u_2)) :$
$q : -r.$	$r.$	$p : -r.$

The transformation is unsound, since the successful derivation for $u_3 : p$ in P' corresponds to a failure in P. The problem is that the context for r is $\lfloor u_3, u_2, u_1 \rfloor$ in P' whereas it is $\lfloor u_1 \rfloor$ in P, and the corresponding definitions are obviously not equivalent.

We will then require that the transformation from c-resultants to resultants preserve the contexts associated with each atom in the residuals. Finding the correct context to be associated with each residual in the body of the resultants is indeed straightforward in our scheme, since it can be determined by a direct inspection of the partial C-SLD tree. The switch operator can be then explicitly used to enforce the appropriate context both in dynamically and statically configured systems.

Definition 3.7 (Resultants) *Let $CR = G_0\sigma : -\langle g_1, C_1\rangle, \ldots, \langle g_n, C_n\rangle$ be a c-resultant for the c-goal $\langle G_0, C_0\rangle$. Then, the corresponding resultant R is computed by the function $\psi : \{c\text{-}resultants\} \mapsto \{resultants\}$ defined as follows:*

$$\psi(G_0\sigma : -\langle g_1, C_1\rangle \ldots, \langle g_n, C_n\rangle) = G_0\sigma : -g'_1, \ldots, g'_n$$

where for each i:

(i) *if $C_i \sqsupseteq \mathcal{D}c(CR)$, then $g'_i = g_i$;*

(ii) *if $C_i \sqsubset \mathcal{D}c(CR)$, i.e. $\mathcal{D}c(CR) = \lfloor u_N, \ldots, u_j, \ldots, u_1 \rfloor$ and $C_i = \lfloor u_j, \ldots, u_1 \rfloor$, then:*

$$\begin{array}{ll} g'_i = u_j : g_i & \textit{for statically configured systems} \\ g'_i = u_1 : (u_2 \gg \ldots \gg u_j \gg g_i) & \textit{for dynamically configured system} \end{array}$$

Notice that in definition 3.7 we consider only cases in which C_i and $\mathcal{D}c(CR)$ are related according to the *sub-context* ($\subseteq$) relation. This is consistent with our initial assumption that the source programs we consider don't contain any occurrence of context extension and/or context switch. Yet, the definition of the mapping ψ explicitly introduces the use of context switch and context extension in the transformed configuration. It appears then that the class of source programs is strictly included in the class of transformed programs. This inconsistency will be resolved in section 5 and section 6 where we finally extend the technique described here to deal with context switch and context extension.

We can now introduce a formal definition of the Partial Deduction of a structured program P with respect to an atomic goal G and a context C in terms of the following notions of *program representation* and of *Partial Deduction of a goal in a context*.

Definition 3.8 (Program Representation) *Let U(P) be the set of units of a program P. The representation of P is given by:*

$$\mathcal{PR}(P) = \{\langle d, u\rangle : \ u \in U(P), d \text{ is a clause of } u\}$$

Let now ψ be the mapping from c-resultants to resultants introduced in definition 3.7 and $\mathcal{F}_{ass} : \ CR_T \mapsto U(P)$ be the assignment function of definition 3.6.

Definition 3.9 (Partial Deduction of a goal in a context) *Let P be a program, C a context, G an atomic goal, and CR_T the set of c-resultants of a partial C-SLD tree T for P and $\langle G, C\rangle$. The Partial Deduction of G in C is given by:*

$$PD(G, C) = \{\langle \psi(d), u\rangle | d \in CR_T \text{ and } u = \mathcal{F}_{ass}(d)\}$$

Finally, let us denote with $PD(P, G, C)$ the Partial Deduction of a structured program P with respect to an atomic goal G and a context C. $PD(P, G, C)$ is obtained by replacing, in the source program, all the clauses for G in the units of C with the clauses obtained by the Partial Deduction of G in C.

Definition 3.10 (Partial Deduction of P wrt G and C) *Let P be a program, C a context, G an atomic goal. Then:*

$$PD(P, G, C) = P' \ \ \text{such that} : \ \mathcal{PR}(P') = (\mathcal{PR}(P) \setminus \mathcal{D}(C, G)) \cup PD(G, C)$$

where $\mathcal{D}(C, G)$ is the representation of the set of definitions for the goal G (i.e. the representation of the clauses which have the same predicate symbol and arity of G) contained in the context C.

Example 3.5 Let's consider again the program of example 3.4. Using the construction of the resultants given in definition 3.7, the transformation produces the following program:

$\boldsymbol{unit}(u_1, \boldsymbol{parent}(top)) :$	$\boldsymbol{unit}(u_2, \boldsymbol{parent}(u_1)) :$	$\boldsymbol{unit}(u_3, \boldsymbol{parent}(u_2)) :$
$q : -r.$	$r.$	$p : -u_1 : r.$

The resultant $p : -u_1 : r$ is obtained by the corresponding c-resultant $p : -\langle r, \lfloor u_1 \rfloor\rangle$ by applying case **(ii)** of definition 3.7. Notice that now, if $: -u_3 : p$ is the top goal, during the C-SLD derivation the call to the predicate r is evaluated in $\lfloor u_1 \rfloor$ in the source as well as in the transformed program.

4 Soundness and Completeness

In this section, we give a formal characterisation of the class of structuring policies for which our Partial Deduction scheme is sound and complete. We first extend the notions of soundness and completeness of PD to the framework of structured logic programming. Then, following the approach proposed in [18], we isolate a

closedness condition under which our transformation is sound and complete with respect to the operational semantics defined in [3] and reported in the appendix.

We first summarize here the basic results for PD in Logic Programming and give an informal outline of the proof in that case. The proof in the extended case of Structured Logic Programming will be then carried on along the same line and will make use of some of those results.

4.1 Results in Logic Programming

Definition 4.1 *Let P be a program, G_0 an atomic goal, P' a partial deduction of P wrt G_0.*

Soundness *If $P' \cup \{G\}$ has an SLD refutation with computed answer θ, then $P \cup \{G\}$ has an SLD refutation with computed answer θ.*

Completeness *If $P \cup \{G\}$ has an SLD refutation with computed answer θ, then $P' \cup \{G\}$ has an SLD refutation with computed answer θ.*

The soundness and the completeness of PD in logic programming are stated in [18] under the additional requirement that a *closedness* condition holds on the transformed program and the goal. Let's briefly recall the $A-closedness$ condition introduced in [18]. Given an atomic formula A, we say that a program P is $A-closed$ if each occurrence in P of an atom with the same predicate name as A is an instance of A. Then, in [18], the authors prove that, if A is the goal used for the PD of P, then for any goal G such that $P' \cup \{G\}$ is A-closed, $P' \cup \{G\}$ computes a substitution θ if and only if so does $P \cup \{G\}$.

The proof uses the following version of the lifting lemma which generalizes the standard lifting lemma to the case of arbitrary SLD-derivations. The two properties stated in the lemma 4.2 provide then a formal justification for the use of a derived clause (a resultant). Let's assume that A and A' are atoms.

Lemma 4.1 (Lifting) *Let R be the resultant of an SLD derivation D from a goal :-A. Let ϕ be a substitution. If there is a corresponding derivation D' from :-$A\phi$, then its resultant R' is an instance of R.*

Lemma 4.2 *Let P be a program and :-A be a goal. Let R be the resultant associated to a derivation D for A. Let also :-A' be a goal such that A' unifies with the head of the resultant R. Let θ be the mgu and let $R'' = R\theta$. Then:*

(i) there exists a derivation D' for A' corresponding to D which selects at each step a corresponding literal and uses a (variant of) the same clauses as D. Furthermore, if R' is the resultant associated to D' then R' is an instance of R''.

(ii) if the atom A' in point (i) is an instance of A ($A' = A\phi$), then there is a corresponding SLD derivation and its resultant R' is equal to R''.

Based on the above results, the proof that Partial Deduction is sound follows almost trivially. In fact, we can consider an SLD refutation as an SLD derivation whose associated resultant is a unit clause. Let's consider the program P' obtained as a PD of P wrt a goal G_0. Let also D' be a refutation of $P' \cup \{G\}$ that uses a resultant R_i. If $P' \cup \{G\}$ is G_0-closed, then, from lemma 4.2(ii), any use of R_i in D', can be replaced by a corresponding derivation D in P with the same associated resultant. Hence, if the unit clause '$G\theta$.' is the resultant associated to D', there is a corresponding refutation in P whose resultant is also '$G\theta$'.
A similar argument is used to prove the completeness. Assume that there exists a refutation D for $P \cup \{G\}$ and consider the corresponding refutation D' in $P' \cup \{G\}$. As long as the selected atom in D and D' is not the predicate used during PD, D and D' coincide. Then consider the first occurrence of such predicate in D in P'. Since $P' \cup \{G\}$ is G_0 closed, such occurrence must be of the form $G_0\phi$. Furthermore, since D is a refutation for $P \cup \{G\}$, there must exist a corresponding refutation D'' for $G_0\phi$. Since a refutation ends up in the empty goal, all atoms are selected. Then, we can use the switching lemma [18] to put D'' into the form in which it starts off with the same choice of atoms and clauses as some branch B in the tree used in the PD of G_0. Consider the sub- derivation S_D of D'' corresponding to B. By the lifting lemma, the resultant associated with S_D is an instance of the resultant R_B associated to B. Let $G_0\phi\sigma$ be the head of the resultant associated to with S_D and $G_0\theta$ be the head of R_B respectively. Since $G_0\phi\sigma$ is an instance of $G_0\theta$, then clearly $G_0\phi$ unifies with the head of R_B and therefore, by lemma 4.2(ii), the resultant obtained by unifying R_B with $G_0\phi$ is also the resultant of the corresponding sub-derivation S_D. Hence, the resultant associated with S_D can be obtained using R_B. Using an inductive argument on the length of the refutation D, we conclude that the result of any sub-derivation of D in P for an instance of the goal used during PD can be obtained by using the corresponding resultant in P'. Hence we conclude that PD is complete.

4.2 Results for Structured Logic Programming

We first introduce a definition of soundness and completeness for structured logic programming that corresponds to the one stated for logic programming.

Definition 4.2 *Let P be a program, $\langle G_0, C_0\rangle$ a c-atom, G a goal and P' a partial deduction of P wrt $\langle G_0, C_0\rangle$.*

Soundness *If $P' \cup \{\langle G, C\rangle\}$ has a C-SLD refutation with computed answer θ, then $P \cup \{\langle G, C\rangle\}$ has a C-SLD refutation with computed answer θ.*

Completeness *If $P \cup \{\langle G, C\rangle\}$ has a C-SLD refutation with computed answer θ, then $P' \cup \{\langle G, C\rangle\}$ has a C-SLD refutation with computed answer θ.*

We then introduce the notion of $\langle G, C\rangle$-*closedness* which extends the one given in [18] by requiring a corresponding property for the transformed program, the goal and the context in which the goal is to be evaluated.

Definition 4.3 ($\langle G, C\rangle$-closedness) *Let $\langle G, C\rangle$ be a c-atom and S be a set of clauses of a structured program. We say that S is $\langle G, C\rangle$-*closed *iff it is G-closed*

and each occurrence in S of an atom which has the same predicate name as G is evaluated in a context $C' \sqsubseteq C$ or in a context that has no unit in common with C.

As already mentioned, the proof of soundness and completeness follows the same argument used in [18] for the case of Logic Programming. This is justified on the basis of the following definition of *the SLD derivation corresponding to a C-SLD derivation.*

Definition 4.4 *Let CD be a C-SLD derivation from the c-goal $\langle G, C\rangle$. The SLD derivation corresponding to CD, denoted by SLD(C), is an SLD derivation from :-G that at each step is presented with the set of clauses denoted by the context of the selected c-atom in CD, selects the corresponding atom and chooses the same clause as CD.*

The notion of SLD(C) derivation provides the connection between C-SLD derivations and SLD derivations needed to prove many of the following technical results. We first extend lemma 4.2 to SLD(C) derivations.

Lemma 4.3 *Let CD be a C-SLD derivation from the c-goal $\langle G, C\rangle$ and let D be the corresponding SLD(C) derivation from :-G. Let R be the resultant associated to D and let R'' be the resultant obtained by using the clause R on a goal :-$G\phi$. Then there is a corresponding SLD(C) derivation D' from :-$G\phi$ and its resultant is R''.*

Proof. By lemma 4.2(ii) there exists an SLD derivation D'' corresponding to D with the required property. Hence, at each step, D'' chooses a clause which belongs to the context associated with the selected c-atom in CD and selects the corresponding atom. Hence the SLD(C) derivation D' corresponding to D can be constructed from D''.

A useful result for the following development comes from the relation between the resultants associated with a C-SLD derivation and with the corresponding SLD(C) derivation.

Consider the c-resultant CR associated to a C-SLD derivation CD from $\langle G, C\rangle$ and let R be the corresponding resultant ($R = \psi(\mathrm{CR})$). Let also R' be the resultant associated to the SLD(C) derivation from :-G corresponding to CD. It is easy to see that R and R' have the same residuals except for possible occurrences of context extension/switch in R introduced by applying the function ψ to the c-resultant CR.

Lemma 4.4 *Let CR be the c-resultant of C-SLD derivation from the c-goal $\langle G, C\rangle$. Then, for any context C' such that $C \sqsubseteq C'$ there exists a corresponding C-SLD derivation CD' from $\langle G, C'\rangle$ that chooses (a variant of) the same clause as CD and selects a corresponding c-atom. Hence, the deduction contexts associated to the c-resultants of CD and CD' are equal.*

Proof. By saying that CD and CD' select a corresponding c-atom, we mean that they select a corresponding atom with the same associated context. If CD and CD' are C-SLD refutations, the proof follows from the definition of the relation $\sqsubseteq$

over contexts by considering the corresponding SLD(C) derivations. In fact, since $C \sqsubseteq C'$, the set of clauses denoted by C is a subset of the set of clauses denoted by C'. Hence, if there exists an SLD(C) refutation from :-G, then the same sequence of choices of clauses and of selection of goals can be obviously performed by an SLD(C') refutation. The deduction contexts associated to the c-resultants of CD and CD' are obviously equal, being the label contexts associated with each step of CD and CD' equal.

A similar argument applies to the case of general C-SLD derivations.

Lemma 4.5 *Let CR be the c-resultant of a C-SLD derivation from the c-goal $\langle G, C\rangle$. Then, there exists a corresponding C-SLD derivation from $\langle G, \mathcal{D}c(CR)\rangle$.*
Proof. Since $\mathcal{D}c(CR)$ is the maximum context used in the C-SLD derivation from $\langle G, C\rangle$ to CR, the same choices of c-atoms and of clauses can be obtained from a C-SLD derivation from $\langle G, \mathcal{D}c(CR)\rangle$.

We can now extend the result of lemma 4.2 to justify the use of a resultant in the case of C-SLD derivations.

Lemma 4.6 *Let P be a structured program, $\langle G, C\rangle$ and $\langle G', C'\rangle$ be two c-atoms where $G' = G\phi$ and C' is a sub-context of C ($C' \sqsubseteq C$). Let CR be the c-resultant of a C-SLD derivation CD from $\langle G, C\rangle$, R the corresponding resultant ($R = \psi(CR)$), and $u = \mathcal{F}_{ass}(CR)$. Assume that $u \in C'$ and that $G\phi$ unifies with the head of R. Let CR'' be the c-resultant obtained by applying R to solve the c-atom $\langle G\phi, C\rangle$, and $R'' = \psi(CR'')$ the associated resultant. Then there exists a C-SLD derivation CD' for $\langle G', C'\rangle$ corresponding to CD which selects at each step a corresponding c- atom and uses a (variant of) the same clause as CD. Furthermore, if CR' is the resultant associated to CD' and $R' = \psi(CR')$, then R' is equal to R'' and $\mathcal{F}_{ass}(CR') = \mathcal{F}_{ass}(CR'')$.*
Proof. By lemma 4.5, there exists a corresponding C-SLD derivation which starts from $\langle G, \mathcal{D}c(CR)\rangle$. Since $u \in C'$ and $C' \sqsubseteq C$ it follows that $\mathcal{D}c(CR) \sqsubseteq C'$. Hence, by lemma 4.4, there exists a C-SLD derivation corresponding to CD from $\langle G, C'\rangle$. Now we proceed arguing by contradiction. Assume that the claim is false. Then $G\phi$ unifies with R and there is no C-SLD derivation from $\langle G\phi, C'\rangle$ with the required properties. Consider now the corresponding SLD(C') derivation. It follows that there is no SLD(C') derivation from :-$G\phi$ corresponding to the SLD(C) derivation from :-G. Yet, the resultant of the SLD(C) derivation from :-G corresponding to CD has the same head as R and therefore $G\phi$ unifies with it. This contradicts the result of lemma 4.3.
Thus the C-SLD derivation CD' does exist. Again, using the corresponding SLD(C) and SLD(C') derivations, it is easy to see that the residuals in R' and R'' contain the same atoms. Furthermore, since CD and CD' select always corresponding c-atoms, the contexts of the c-atoms in the residuals for CR and CR' are also equal. Hence, $R' = \psi(CR')$ is equal to $R'' = \psi(CR'')$.
Finally, since the contexts in the selected c-atoms in CD and CD' coincide, also the label context associateds with corresponding steps are equal. This implies that $\mathcal{F}_{ass}(CR) = \mathcal{F}_{ass}(CR')$. Hence, since $\mathcal{F}_{ass}(CR) = \mathcal{F}_{ass}(CR'')$, we conclude $\mathcal{F}_{ass}(CR') = \mathcal{F}_{ass}(CR'')$.

We are now ready to state the soundness and completeness properties for Partial Deduction. The result is established under the $\langle G, C\rangle$-closedness condition defined earlier in this section (definition 4.2).

Proposition 2 (Soundness and completeness) *Let P be a program, $\langle G_0, C_0\rangle$ a c-atom, and P' a Partial Deduction of P wrt $\langle G_0, C_0\rangle$. Let $\langle G, C\rangle$ be a c-goal and $P' \cup \{\langle G, C\rangle\}$ be $\langle G_0, C_0\rangle$-closed.*

(i) If $P' \cup \{\langle G, C\rangle\}$ has a C-SLD refutation with computed answer θ then so does $P \cup \{\langle G, C\rangle\}$.

(ii) If $P \cup \{\langle G, C\rangle\}$ has a C-SLD refutation with computed answer θ then so does $P' \cup \{\langle G, C\rangle\}$.

Proof.
(i). Consider a C-SLD refutation CD' of $P' \cup \{\langle G, C\rangle\}$ that uses a resultant R. Let u be the unit to which R is assigned. Since the CD' uses R, the context C' associated with the c-atom which is selected when R is chosen contains u. Furthermore, since $P' \cup \{\langle G, C\rangle\}$ is $\langle G_0, C_0\rangle$-closed, C' must be a sub-context of C_0 ($C' \sqsubseteq C_0$) and so, by lemma 4.6, any use of R in CD' can be replaced by a corresponding C-SLD derivation in P with the same associated resultant without affecting the resulting substitution.
(ii). Let CD be a C-SLD refutation for $P \cup \{\langle G, C\rangle\}$. Consider the first occurrence in CD of a c-atom corresponding to the predicate of G_0. Since $P' \cup \{\langle G, C\rangle\}$ is $\langle G_0, C_0\rangle$-closed, such c-atom must be of the form $\langle G_0\phi, C'\rangle$ with $C' \sqsubseteq C_0$.
Consider now the C-SLD refutation CD'' for $P \cup \{\langle G_0\phi, C'\rangle\}$. By using the corresponding SLD(C'') refutation, we can again appeal to the switching lemma and argue that CD'' can be put into a form in which it starts off with the same choice of c-atoms and clauses as some branch CB in the partial C-SLD tree for $\langle G_0, C_0\rangle$. Notice that such a branch does exist in the partial tree for $\langle G_0, C_0\rangle$ being $C' \sqsubseteq C_0$. Consider then the sub-derivation of CD'' corresponding to CB. By lemma 4.4, the deduction contexts associated to CB and the corresponding sub-derivation of CD'' are equal. Then let R be the resultant associated to CB and u the unit to which R is assigned. Since u belongs to the deduction context associated to CB, it follows that u is one of the units of C' ($u \in C'$). Clearly $G_0\phi$ unifies with the head of R. Hence, from lemma 4.6 we conclude that the sub-derivation of CD'' can be obtained using R.
Using an inductive argument on the length of the refutation CD, we can conclude that the result of any sub-derivation of CD for an instance of the goal used during PD can be obtained by using the corresponding resultant in P'. The completeness of PD is an immediate consequence.

The properties of Partial Deduction can also be characterised in terms of a weaker notion of soundness than the one given in definition 4.2. Namely, a soundness condition that requires that if there exists a C-SLD refutation for $P' \cup \{\langle G, C\rangle\}$ computing a substitution θ', then there be a C-SLD refutation for $P \cup \{\langle G, C\rangle\}$ that computes a substitution θ where θ is more general than θ'.

Under this definition of soundness, we can correspondingly introduce a weaker form of closedness property that imposes a restriction only on the context associated with a c-atom.

Definition 4.5 (*weak*$\langle G, C\rangle$-closedness) *Let $\langle G, C\rangle$ be a c-atom and S be a set of clauses of a structured program. We say that S is* weakly $\langle G, C\rangle$-closed *iff any occurrence in S of an atom which has the same predicate name as G is evaluated in a context $C' \sqsubseteq C$ or in a context that has no unit in common with C.*

The corresponding effect on the properties of Partial Deduction is stated in the following proposition, whose proof is omitted here. The proof for the corresponding case of Logic Programming can be found in [18].

Proposition 3 *Let P be a program, $\langle G_0, C_0\rangle$ a c-atom, and P' a Partial Deduction of P wrt $\langle G_0, C_0\rangle$. Let $\langle G, C\rangle$ be a c-goal and $P' \cup \{\langle G, C\rangle\}$ weakly $\langle G_0, C_0\rangle$-closed. If $P' \cup \{\langle G, C\rangle\}$ has an C-SLD refutation with computed answer θ' then $P \cup \{\langle G, C\rangle\}$ has a C-SLD refutation with computed answer θ such $\theta' = \theta\sigma$ for some substitution σ.*

5 PD in Statically Configured Systems

Under the strong $\langle G, C\rangle$-closedness property, the soundness and completeness of the transformation holds for all the different classes of systems presented in section 2. Less restrictive conditions can actually be imposed when considering each specific class. A first relevant result concerns statically configured systems. In this case Partial Deduction can be proved sound under the simple G-closedness condition.

Proposition 4 (Soundness for Static Systems) *Let P be a statically configured program, and P' a PD of P wrt $\langle G_0, C_0\rangle$. Let $P' \cup \{\langle G, C\rangle\}$ be G_0-closed. If $P' \cup \{\langle G, C\rangle\}$ has a C-SLD refutation with computed answer θ, then so does $P \cup \{\langle G, C\rangle\}$.*

Proof. From proposition 2, the statement trivially holds if the C-SLD refutation for $P' \cup \{\langle G, C\rangle\}$ selects only c-atoms of the form $\langle G_0\phi, C'\rangle$ with $C' \sqsubseteq C_0$. It also holds if the contexts in the selected c-atoms share no units with C. Let's then consider the only interesting case of a context $C' \not\sqsubseteq C$ which share at least one unit with C. Since the program is statically configured, C' can only be of the form $u_n \ldots u_1.C|_i$ where $C|_i$ is an arbitrary sub-context of C.
Consider a C-SLD refutation CD' of $P' \cup \{\langle G_0\phi, C'\rangle\}$ that uses a resultant R. Let CR be the corresponding c-resultant and let B_R the branch in the partial C-SLD tree for $\langle G_0, C_0\rangle$ leading to CR. Then R can only belong to a unit of $C|_i$ since no unit in $\lfloor u_n, \ldots, u_1\rfloor$ was involved during PD. But then, from lemma 4.6, there exists a C-SLD derivation corresponding to B_R with the same resultant. Hence the claim follows from the same argument used in proposition 2.

In the case of conservative systems, the previous result can be further extended. Given a goal G, Partial Deduction is also complete provided that the transformed

program and the goal G satisfy the simple G_0-*closedness* condition introduced in [18]. Therefore, for statically configured conservative systems the soundness and completeness of PD hold under the same condition introduced for logic programming.

Proposition 5 (Completeness for Static Conservative Systems) *Let P be a program, $\langle G_0, C_0\rangle$ a c-atom, and P' a PD of P wrt $\langle G_0, C_0\rangle$. Let $\langle G, C\rangle$ a c-goal. Let $P' \cup \{\langle G, C\rangle\}$ be G_0-closed. $P' \cup \{\langle G, C\rangle\}$ has an C-SLD refutation with computed answer θ if and only if so does $P \cup \{\langle G, C\rangle\}$.*

Proof. Again, from proposition 2, the statement trivially holds if the refutation for $P' \cup \{\langle G, C\rangle\}$ selects only c-atoms of the form $\langle G_0\phi, C'\rangle$ with $C' \sqsubseteq C_0$. It also holds if C' shares no units with C. Let then consider the case of a context $C' \not\sqsubseteq C$ which shares at least one unit with C. Again C' must be of the form $u_n \ldots u_1.C|_i$ where $C|_i$ is an arbitrary sub-context of C.
Consider the structure of a C-SLD refutation CD of $P \cup \{\langle G_0\phi, C'\rangle\}$. Then either all the clauses chosen by CD belong to $\lfloor u_n \ldots u_1 \rfloor$ or at least one clause is chosen from $C|i$. In the former case, the claim follows trivially. In the latter, since the system is conservative, after the first choice of a clause in $C|_i$, all further steps will choose a clause in $C|_i$. Then the claim follows by proposition 2.

Example 5.1 Let us consider the block-structured program of example 2.4. A Partial Deduction wrt the goal $b_1 : member(X, [c[b, a]])$ yields the following transformed program P':

$$
\begin{array}{ll}
\boldsymbol{unit}(b_1, \boldsymbol{parent}(top)) : & \boldsymbol{unit}(b_2, \boldsymbol{parent}(b_1)) : \\
member(c, [c, [b, a]]). & eq(X, X). \\
member([b, a], [c, [b, a]]). & eq([X_1|A], [Y_1|B]) : - \\
 & \qquad perm([X_1|A], [Y_1|B]). \\
member([a, b], [c, [b, a]]). & \\
perm([], []). & \\
perm(L, [X|P]) : -delete(X, L, L_1), & \\
\qquad perm(L_1, P). & \\
delete(X, Y, Z) : -... &
\end{array}
$$

The completeness of the new configuration is guaranteed for the evaluation of any goal G such that $P' \cup \{G\}$ is closed on *member(X,[c,[b,a]])*. Notice furthermore that, for the new program, the evaluation of the queries corresponding to the predicate *member* can now be performed without any context switch. The corresponding effect yields obviously a faster binding resolution.

Proposition 5 cannot be extended to the case of evolving systems, even though they are statically configured. For evolving systems, in fact, even if a clause belonging to $C|_i$ is eventually applied during the derivation, we are not certain that all further derivation steps will apply only clauses of $C|_i$ (see the operational semantics for evolving systems in the Appendix).

Example 5.2 Let P be the following statically configured evolving program:

$\boldsymbol{unit}(u_1, \boldsymbol{parent}(top))$:	$\boldsymbol{unit}(u_2, \boldsymbol{parent}(u_1))$:	$\boldsymbol{unit}(u_3, \boldsymbol{parent}(u_2))$:
$p(X) : -q(X).$	$r(1).$	$q(2).$
$q(X) : -r(X).$		

Let $C0 = \lfloor u_2, u_1 \rfloor$ be the initial context and $G0 = p(X)$ the initial goal. A PD of P wrt $\langle G0, C0 \rangle$ will produce the following program P':

$\boldsymbol{unit}(u_1, \boldsymbol{parent}(top))$:	$\boldsymbol{unit}(u_2, \boldsymbol{parent}(u_1))$:	$\boldsymbol{unit}(u_3, \boldsymbol{parent}(u_2))$:
$q(X) : -r(X).$	$r(1).$	$q(2).$
	$p(1).$	

Now consider the context $\mathcal{C} = \lfloor u_3, u_2, u_1 \rfloor$; the transformation is not complete since, in P, $\mathcal{C} \vdash p(2)$ but, in P', $\mathcal{C} \nvdash p(2)$.

For evolving systems, the equivalence between the source and the transformed configurations holds therefore only under the $\langle G_0, C_0 \rangle$-closedness condition. However, it is worth mentioning that, in the case of static configurations, evaluating such condition is simply accomplished by a syntactic check on the structure of the transformed program and the goal. Furthermore, the completeness of Partial Deduction is also automatically guaranteed for evolving systems, if the choice of the initial context corresponds to a whole chain in the hierarchy defined by the program. If this is the case, the transformed program is $\langle G_0, C_0 \rangle$-*closed* by construction.

This provides an effective use of Partial Deduction in inheritance-based systems (see [6]). Given the whole inheritance tree for a program P, a complete transformation is obtained by specializing P with respect to a goal G and all the inheritance chains corresponding to the branches of this tree.

If $\{C_1, \ldots, C_n\}$ are the contexts corresponding to these branches, we get the set of Partial Deductions $\{PD(G, C_1), \ldots, PD(G, C_n)\}$. The new program is then obtained by deleting all the original definitions for G, and adding the new ones according to definition 3.10. This yields to the extended notion of Partial Deduction with respect to a goal G and a set of contexts $\{C_1, \ldots, C_n\}$, as the union of the set of Partial Deductions with respect to each C_is. Namely:

$$PD(P, G, \{C_1, \ldots, C_n\}) = P' \text{ such that :}$$
$$\mathcal{PR}(P') = (\mathcal{PR}(P) \setminus \mathcal{D}(C, G)) \cup (\cup_{i \in [1,..,n]} PD(G, C_i))$$

Example 5.3 Let us consider the inheritance scheme given in example 2.6. Suppose we are interested in specializing the program with respect to the goal *mode(X)*. We have here three inheritance chains:

$$top \rightarrow animal \rightarrow bird \rightarrow tweety$$

$$top \rightarrow animal \rightarrow horse$$

$$top \rightarrow animal \rightarrow human$$

A Partial Deduction with respect to the goal *mode(X)* and the single inheritance chain

$$\lfloor tweety, bird, animal \rfloor$$

produces the transformed program:

$$\begin{array}{ll}
\boldsymbol{unit}(animal, \boldsymbol{parent}(top)) : & \lceil\ mode(walk). \\
\boldsymbol{unit}(bird, \boldsymbol{parent}(animal)) : & \left\lceil \begin{array}{l} mode(fly). \\ mode(run). \\ no_of_legs(2). \\ covering(feather). \end{array} \right. \\
\boldsymbol{unit}(horse, \boldsymbol{parent}(animal)) : & \lceil\ no_of_legs(4). \\
\boldsymbol{unit}(human, \boldsymbol{parent}(animal)) : & \lceil\ no_of_legs(2). \\
\boldsymbol{unit}(tweety, \boldsymbol{parent}(bird)) : & \lceil no_of_wings(2).
\end{array}$$

Notice that the goal $human : mode(run)$ now fails, while it succeeded in the original program since $\lfloor human, animal \rfloor$ is not a sub-context of $\lfloor tweety, bird, animal \rfloor$. Conversely, if we take into account all the inheritance chains, the union of all the resulting partial deductions yields the following complete program:

$$\begin{array}{ll}
\boldsymbol{unit}(animal, \boldsymbol{parent}(top)) : & \lceil\ mode(walk). \\
\boldsymbol{unit}(bird, \boldsymbol{parent}(animal)) : & \left\lceil \begin{array}{l} mode(fly). \\ mode(run). \\ no_of_legs(2). \\ covering(feather). \end{array} \right. \\
\boldsymbol{unit}(horse, \boldsymbol{parent}(animal)) : & \left\lceil \begin{array}{l} mode(gallop). \\ no_of_legs(4). \end{array} \right. \\
\boldsymbol{unit}(human, \boldsymbol{parent}(animal)) : & \left\lceil \begin{array}{l} mode(run). \\ no_of_legs(2). \end{array} \right. \\
\boldsymbol{unit}(tweety, \boldsymbol{parent}(bird)) : & \lceil no_of_wings(2).
\end{array}$$

Furthermore, thanks to the choice of the assignment function, the resulting program can still take advantage from its hierarchical structure, thus avoiding unnecessary code replication. As a matter of fact, the resultant *mode(walk)* is assigned to *animal* — the top unit of the corresponding deduction context — and therefore it is shared among all its descendants in the inheritance tree.
Conversely, $mode(run)$ and $mode(gallop)$ are assigned to the more specific units, *bird* and *human* as concerns $mode(run)$, and unit $horse$ as concerns $mode(gallop)$, since they do not hold for the whole class *animal*.
The new program is clearly more efficient than the source one since the clauses for $mode(run)$ occur now in the tip nodes of the hierarchy and therefore in the top units of the corresponding contexts. As a matter of fact, observe that the evaluation of the goal $human : mode(run)$ and $horse : mode(gallop)$ is performed without any run-time look-up in the associated context.

5.1 Dealing with Context Switch During Partial Deduction

Extending the Partial Deduction scheme we have devised so far to include context switch is straightforward. Firstly, we have to extend the notion of derivation (definition 3.2) to take into account context switch. In this case, in fact, the tuple $\langle g, c\rangle$ becomes a *c-formula* where g can be a context switch of the kind $u : G$. Such a c-formula is transformed into a c-atom by applying the inference rules for context switch as reported in the Appendix.

The main problem to be solved in this case concerns the computation of the deduction context for the resultants. Firstly, notice that if a context switch occurs along a derivation path, a unique maximum element for the label contexts along that path might not exist. In fact, during the computation we can enforce a new context completely different from the previous one. Furthermore, according to the definition of deduction context given in proposition 1, a resultant may get assigned to units which do not belong to the initial context even if a unique maximum element for the label contexts along that path exists.

Example 5.4 Let us consider the following program P:

$\boldsymbol{unit}(u_1, \boldsymbol{parent}(top)) :$	$\boldsymbol{unit}(u_2, \boldsymbol{parent}(u_1)) :$
$p :- u_2 : q.$	$q.$

A Partial Deduction of p in $\lfloor u_1 \rfloor$ produces the following C-SLD tree:

$$\begin{array}{c} \langle p, \lfloor u_1 \rfloor\rangle \\ | \\ \lfloor u_1 \rfloor \\ | \\ \langle u_2 : q, \lfloor u_1 \rfloor\rangle \\ | \\ \langle q, \lfloor u_2, u_1 \rfloor\rangle \\ | \\ \lfloor u_2, u_1 \rfloor \\ | \\ \Box \end{array}$$

The resultant $p.$ is then assigned to u_2. The resulting program P' is not complete since the derivation $\lfloor u_1 \rfloor \vdash p$ succeeds in P, while it fails in P'.

We now restate the definition of the deduction context to deal with the occurrences of context switches in a derivation. In practice, we ignore the label contexts after context switch.

Proposition 6 (Deduction context revised) *Given a C-SLD derivation:*

$$\langle G_0, C_0\rangle \vdash_{c_1,\sigma_1} \ldots \vdash_{c_j,\sigma_j} \langle g_1, C_1\rangle, \ldots, \langle g_n, C_n\rangle$$

the deduction context $\mathcal{D}c(CR)$ for the c-resultant:
$CR = G_0\sigma :- \langle g_1, C_1\rangle, \ldots, \langle g_n, C_n\rangle$

is the maximum element of the set $\{c_1, \ldots, c_k \mid k \leq j\}$ *where* $\{c_1, \ldots, c_k\}$ *is the set including all label contexts in the partial C-SLD tree except those used to evaluate any context switch goal and its subgoals.*

Finally, the construction of the residuals in the body of the resultants (definition 3.7), immediately extends to the case $C_i \neq \mathcal{D}c(CR)$. Formally:

Definition 5.1 (Resultants) *Let* CR *be a c-resultant for the c-goal* $\langle G_0, C_0 \rangle$ *of the form:* $G_0\sigma : -\langle g_1, C_1 \rangle, \ldots, \langle g_i, C_i \rangle, \ldots, \langle g_n, C_n \rangle$. *Then, the corresponding resultant is defined as follows:*

$$\psi(G_0\sigma : -\langle g_1, C_1 \rangle \ldots, \langle g_i, C_i \rangle, \ldots, \langle g_n, C_n \rangle) = G_0\sigma : -g'_1, \ldots, g'_i, \ldots, g'_n$$

where for each i*:*

(i) *if* g_i *is a context switch or* g_i *is atomic and* $C_i = \mathcal{D}c(CR)$, *then* $g'_i = g_i$;

(ii) *if* g_i *is atomic and* $C_i \neq \mathcal{D}c(CR)$ *and* $C_i = \lfloor u_j, \ldots, u_1 \rfloor$, *then* $g'_i = u_j : g_i$.

With this new definition of deduction context and residual, in example 5.4 the resultant $p.$ is correctly assigned to u_1.

6 Partial Deduction in Dynamically Configured Systems

The soundness and completeness results stated for statically configured systems cannot be extended to more dynamic systems. In fact, when no limitation is imposed on the way units can be collected into contexts, there is no systematic way of statically determining the properties of the transformation. This imposes strong limitations over the range of possible applications of our technique in the case of dynamic systems. As a matter of fact, the class of queries for which the satisfiability of the closedness conditions can be statically evaluated is restricted to those queries for which context extensions occur only in the top goal. This is actually the case for *LML* [5], O'Keefe's proposal [26] and [4].

In all these cases, when the context for the top goal is a sub-context of the initial context used to specialize the program, the G_0-closedness condition is sufficient to guarantee the soundness and the completeness of the transformation. In fact, if C_0 is the initial context used during Partial Deduction and $C \sqsubseteq C_0$ is the context for the top goal G, then the whole computation for G takes place in sub-contexts of C which are also sub-contexts of C_0. This immediately follows from the operational semantics of dynamic systems reported in the appendix and guarantees that the transformation is sound and complete under the simple G_0-closedness condition, as in logic programming.

6.1 Dealing with Context Extension During Partial Deduction

Dealing with generalized occurrences of context extension during the C-SLD derivation involves several extensions to the framework we have drawn so far . First of all, we have to extend the notion of derivation (definition 3.2) to take into account context extension. In this case, in fact, the tuple $\langle g, c\rangle$ becomes a *c-formula* where g can be a context extension of the kind $u \gg G$. Such a c-formula is transformed into a c-atom by applying the inference rules for context extension as reported in the Appendix.

The solution adopted for the case of context switch to compute the deduction context for the resultants cannot be applied to the case of context extension. In fact, if we ignore all the label contexts during the evaluation of a context extension goal and its subgoals, the transformation may be unsound even though the closedness conditions are satisfied.

Example 6.1 Let P be the following evolving, dynamically configured program:

$\boldsymbol{unit}(u_1):$	$\boldsymbol{unit}(u_2):$	$\boldsymbol{unit}(u_3):$
$p :- u_3 \gg q.$	$f.$	$q :- f.$

A Partial Deduction wrt $\langle p, \lfloor u_2, u_1 \rfloor\rangle$ produces the following C-SLD tree:

$$\langle p, \lfloor u_2, u_1 \rfloor\rangle$$
$$|$$
$$\lfloor u_1 \rfloor$$
$$|$$
$$\langle u_3 \gg q, \lfloor u_2, u_1 \rfloor\rangle$$
$$|$$
$$\langle q, \lfloor u_3, u_2, u_1 \rfloor\rangle$$
$$|$$
$$\lfloor u_3, u_2, u_1 \rfloor$$
$$|$$
$$\langle f, \lfloor u_3, u_2, u_1 \rfloor\rangle$$
$$|$$
$$\lfloor u_2, u_1 \rfloor$$
$$|$$
$$\square$$

If the label contexts for the derivation steps following the evaluation of extension goal $u_3 \gg q$ are ignored, the resultant $p.$ gets assigned to the unit u_1 and the final effect of the transformation is the program P':

$\boldsymbol{unit}(u_1):$	$\boldsymbol{unit}(u_2):$	$\boldsymbol{unit}(u_3):$
$p.$	$f.$	$q :- f.$

Then, the derivation $\lfloor u_1 \rfloor \vdash p$ succeeds in P' while it fails in P. The problem derives from the fact we have ignored that also u_2 is actually part of the deduction context for the resultant $p.$ Conversely, taking into account all the label contexts

along the derivation we would assign the resultant $p.$ to the unit u_3 which doesn't belong to the initial context. The resulting configuration would then be obviously incomplete.

The solution to both these problems is to associate an additional field to each c-atom to hold the *reference context* for the corresponding c-atom. Each node in a C-SLD tree is then associated with a sequence of triples of the form

$$\langle g_1, C_1, Ref_{C_1}\rangle, \ldots, \langle g_n, C_n, Ref_{C_n}\rangle$$

where the field Ref_{C_i} represents the reference context associated with the atom g_i.

The purpose of the reference context is twofold. Firstly, it is used to bound the size of the label context associated with the derivation steps generated from the evaluation of extension goals, thus ensuring that all the resultants are indeed assigned to units which belong to the initial context. Analogously, it determines the structure of the resultants associated with the c-resultants by the mapping ψ.

The reference context associated with the initial c-goal $\langle G_0, C_0, Ref_{C_0}\rangle$ of a derivation is simply the initial context (i.e.$Ref_{C_0} = C_0$). The new definition of derivation is then given as follows:

Definition 6.1 (Derivation revised) *Let P be a program, CG be the c-goal*

$$\langle g_1, C_1, Ref_{C_1}\rangle, \ldots, \langle g_i, C_i, Ref_{C_i}\rangle, \ldots, \langle g_n, C_n, Ref_{C_n}\rangle$$

and let $\langle g_i, C_i, Ref_{C_i}\rangle$ the selected c-atom. We say that the new c-goal CG' is derived from CG and P via the substitution σ, the clause Cl and the context L_C if the following conditions hold:

$$\begin{array}{ll} Cl & = h : -b_1, \ldots, b_m \text{ is the selected clause in } u_j \in C_i = \lfloor u_N, \ldots, u_1\rfloor \\ \sigma & = mgu(h, g_i) \\ CG' & = [\langle g_1, C_1, Ref_{C_1}\rangle, \ldots, \langle (b_1, \ldots, b_m), C', Ref_{C'}\rangle, \ldots, \langle g_n, C_n, Ref_{C_n}\rangle]\sigma \\ Ref_{C'} & = min\{C_i, Ref_{C_i}\} \end{array}$$

The new context C' is computed according to the operational semantics for conservative and evolving systems (see Appendix). Namely:

$$C' = \begin{cases} L_C & \text{if the system is conservative} \\ C_i & \text{if the system is evolving} \end{cases}$$

The label context L_C is determined as follows:

$$L_C : \begin{cases} = \lfloor\ \rfloor & \text{if } g_i \text{ is an extension goal} \\ = \lfloor u_j, \ldots, u_1\rfloor & \text{if } g_i \text{ is an atomic goal and } \lfloor u_j, \ldots, u_1\rfloor \sqsubseteq Ref_{C_i} \\ = Ref_{C_i} & \text{otherwise} \end{cases}$$

The introduction of the reference context in the definition of derivation guarantees a correct computation of the label contexts associated with the derivation steps. This, in turn, ensures the well-foundedness of the construction for the deduction context given in proposition 1.

Example 6.2 Let P be the following conservative, dynamically configured program:

$\boldsymbol{unit}(u_1):$	$\boldsymbol{unit}(u_2):$	$\boldsymbol{unit}(u_3):$
$p :- u_2 \gg u_3 \gg q.$	$f.$	$q :- f.$

According to the new definition for derivation, the evaluation of the c-goal

$$\langle p, \lfloor u_3, u_2, u_1 \rfloor \rangle$$

produces the following derivation:

$$\begin{array}{c}
\langle p, \lfloor u_3, u_2, u_1 \rfloor, \lfloor u_3, u_2, u_1 \rfloor \rangle \\
| \\
\lfloor u_1 \rfloor \\
| \\
\langle u_2 \gg u_3 \gg q, \lfloor u_1 \rfloor, \lfloor u_3, u_2, u_1 \rfloor \rangle \\
| \\
\lfloor \ \rfloor \\
| \\
\langle u_3 \gg q, \lfloor u_2, u_1 \rfloor, \lfloor u_1 \rfloor \rangle \\
| \\
\lfloor \ \rfloor \\
| \\
\langle q, \lfloor u_3, u_2, u_1 \rfloor, \lfloor u_1 \rfloor \rangle \\
| \\
\lfloor u_1 \rfloor \\
| \\
\langle f, \lfloor u_3, u_2, u_1 \rfloor, \lfloor u_1 \rfloor \rangle \\
| \\
\lfloor u_1 \rfloor \\
| \\
\Box
\end{array}$$

Accordingly, applying the original definition of the function $\mathcal{F}_{ass}$, we obtain the expected assignment of the resultant $p.$ to the unit u_1.

As last step, we define the new mapping ψ for the computation of the resultants.

Definition 6.2 (Resultants) *Let*

$$CR = G_0\sigma :- \langle g_1, C_1, Ref_{C_1} \rangle, \ldots, \langle g_i, C_i, Ref_{C_i} \rangle, \ldots, \langle g_n, C_n, Ref_{C_n} \rangle$$

be a c-resultant for the c-goal $\langle G_0, C_0, Ref_{C_0} \rangle$. For the sake of simplicity let all the g_i be atomic. Then, the corresponding resultant R is:

$$\psi(CR) = G_0\sigma :- g'_1, \ldots, g'_i, \ldots, g'_n$$

where for each i:

(i) *if $C_i \sqsupseteq \mathcal{D}c(CR)$, then $g'_i = g_i$;*

(ii) *if* $\mathcal{D}c(CR) \sqsubset C_i$*, i.e.* $C_i = u_n \ldots u_j.\mathcal{D}c(CR)$*, then* $g'_i = u_j \gg \ldots \gg u_n \gg g_i$

(iii) *if* $\mathcal{D}c(CR) \not\sqsubseteq C_i$ *and* $C_i = \lfloor u_j, \ldots, u_1 \rfloor$*, then* $g'_i = u_1 : u_2 \gg \ldots \gg u_j \gg g_i$

The hypothesis that all the g_i in the c-resultant are atomic does not imply lack of generality. In fact, each extension formula can be reduced to an atomic one by suitably applying the inference rules reported in the Appendix.

Example 6.3 The definition of the mapping ψ supports a consistent construction of the resultants from the c-resultants associated to any partial C-SLD tree. This is exemplified in the following partial derivation for the c-goal $\langle p, \lfloor u_3, u_2, u_1 \rfloor \rangle$ in the program of example 6.2.

$$\begin{array}{c}
\langle p, \lfloor u_3, u_2, u_1 \rfloor, \lfloor u_3, u_2, u_1 \rfloor \rangle \\
| \\
\lfloor u_1 \rfloor \\
| \\
\langle u_2 \gg u_3 \gg q, \lfloor u_1 \rfloor, \lfloor u_3, u_2, u_1 \rfloor \rangle \\
| \\
\lfloor \ \rfloor \\
| \\
\langle u_3 \gg q, \lfloor u_2, u_1 \rfloor, \lfloor u_1 \rfloor \rangle \\
| \\
\lfloor \ \rfloor \\
| \\
\langle q, \lfloor u_3, u_2, u_1 \rfloor, \lfloor u_1 \rfloor \rangle \\
| \\
\lfloor u_1 \rfloor \\
| \\
\langle f, \lfloor u_3, u_2, u_1 \rfloor, \lfloor u_1 \rfloor \rangle
\end{array}$$

The application of the mapping ψ produces the resultant $p : -u_2 \gg u_3 \gg f.$ which is then assigned to the unit u_1. The final configuration is:

$\boldsymbol{unit}(u_1)$:	$\boldsymbol{unit}(u_2)$:	$\boldsymbol{unit}(u_3)$:
$p : -u_2 \gg u_3 \gg f.$	$f.$	$q : -f.$

7 Conclusions

In this paper, we have discussed an extension of Partial Deduction for the framework of structured logic programs. The technique we have described applies to a wide class of structuring mechanisms supporting static as well as dynamic modular configurations.

The key point of our approach to structured program specialisation is that we choose the structural properties of programs as invariants of the transformation. This choice has important consequences from both the implementation and foundational point of view. Preserving the structure of the programs allows an incremental transformation process to take place, in which sub-components of a given configuration can be replaced by their specialised versions without affecting the semantics

of the original one. Furthermore, it makes our scheme fully compatible with the compilation techniques presented in [15, 16].

We have then established the well-foundedness of the extended Partial Deduction scheme in terms of the standard notions of soundness and completeness. Following the approach proposed in [18], we have characterised the soundness and the completeness of the transformation in terms of a single closedness property to be checked on the transformed program and the goal. The nature of the various results varies on the account of the class of structured programs under consideration.

For statically configured systems, the transformation can be proved sound under the simple A-closedness condition. The same hypothesis is sufficient to establish a completeness result for conservative systems such as module- and block-based systems. Conversely, for statically configured evolving systems, which represent inheritance-based schemes, a stronger condition which also involves a check on the context in which the goals are to be evaluated must be imposed to prove the same completeness result.

Weaker results hold for dynamically configured systems. In fact, for this class of programs, there is apparently no systematic way of statically checking the satisfiability of the closedness property over the programs produced by the transformation.

An effective improvement of the results presented in this paper for dynamic systems of units could be achieved through the application of data-flow and abstract interpretation techniques to support the static satisfiability evaluation of the closedness property.

Acknowledgements

We are in debt with Antonio Natali who contributed to the definition of the structuring framework, with Antonio Brogi who contributed to define the language semantics, and with the referees for their helpful suggestions. This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grants n.90.00.695.PP65, n.91.00921.PF69 and n.90.00757.69.

References

[1] K. Bowen. Meta-level programming and knowledge representation. *New Generation Computing*, 3:359–383, 1985.

[2] K. Bowen and R. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 153–173. Academic Press, 1982.

[3] A. Brogi, E. Lamma, and P. Mello. A general framework for structuring logic programs. Technical Report 4/1, CNR - Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, Sottoprogetto 4, 1990.

[4] A. Brogi, E. Lamma, and P. Mello. Composing open logic theories. Technical Report 4/28, CNR - Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, Sottoprogetto 4, Submitted, 1991.

[5] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Hierarchies through Basic Meta-Level Operators. In *Proceedings of Second Workshop on Meta-Programming in Logic*, pages 381–396, 1990.

[6] M. Bugliesi, E. Lamma, and P. Mello. Partial evaluation for hierarchies of logic theories. In S. Debray and M. Hermenegildo, editors, *Proc. NACLP*. The MIT Press, 1990.

[7] M. Cavalieri, E. Lamma, and P. Mello. An extended prolog machine for dynamic context handling. In *Proceedings of ECAI 88*, pages 284–289. Pitman Publishing, 1988.

[8] W. Chen. A Theory of Modules Based on Second Order Logic. In *Proceedings of IEEE Symposium on Logic Programming*, pages 24–33. IEEE Computer Society Press, 1987.

[9] K. Fukunaga and S. Hirose. An experience with a Prolog-based object-oriented language. In *Proceedings of OOPSLA-86*. ACM Press, Portland (Oregon), 1986.

[10] D. Gabbay and N. Reyle. N-Prolog: an extension of Prolog with hypothetical implications. *Journal of Logic Programming*, 4:319–355, 1984.

[11] H. Gallaire. Merging objects and logic programming: Relational semantics. In *AAAI-86 Conference Proceedings*, 1986.

[12] L. Giordano, A. Martelli, and G. Rossi. Local definitions with static scope rules in logic languages. In *Proceedings of International Conference FGCS*, pages 389–396, 1988.

[13] H. Kaufman and A. Grumbach. Multilog: Multiple Worlds in Logic Programming. In *Proceedings of ECAI86*. North-Holland, 1986.

[14] H. J. Komorowski. A specification of an abstract Prolog machine and its application to Partial Evaluation. Technical Report Dissertation, Linkoping University, 1981.

[15] E. Lamma, P. Mello, and A. Natali. The design of an abstract machine for efficient implementation of contexts in logic programming. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming*, pages 303–317. The MIT Press, 1989.

[16] E. Lamma, P. Mello, and A. Natali. An Extended Warren Abstract Machine for the Execution of Structured Logic Programs. *Journal of Logic Programming*, Forthcoming, 1992.

[17] J. Lloyd. *Foundations of logic programming*. Springer-Verlag, second edition, 1987.

[18] J. Lloyd and J. Shepherdson. Partial evaluation in logic programming. Technical Report CS-87-09, Department of Computer Science, University of Bristol, England, 1987.

[19] Logicware Inc., Toronto, Canada. *M-Prolog Language Reference*, 1985.

[20] F. McCabe. *Logic and Objects*. PhD thesis, University of London, November 1988.

[21] L. McCarthy. Clausal intuitionistic logic 1: fixed point semantics. *Journal of Logic Programming*, 5:1–31, 1988.

[22] P. Mello, A. Natali, and C. Ruggieri. Logic programming in a software engineering perspective. In L. Lusk and R. Overbeek, editors, *Proc. NACLP*, pages 451–458. The MIT Press, 1989.

[23] D. Miller. A theory of modules in logic programming. In *Proceedings of Symposium on Logic Programming*, pages 106–114, 1986.

[24] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[25] L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming*, pages 284–302. The MIT Press, 1989.

[26] R. O'Keefe. Towards an algebra for constructing logic programs. In J. Cohen and J. Conery, editors, *Proceedings of IEEE Symposium on Logic Programming*, pages 152–160. IEEE Computer Society Press, 1985.

[27] D. Warren. An abstract Prolog instruction set. Technical Report TR 309, SRI International, 1983.

[28] D. Warren. Database updates in Prolog. In *Proceedings of International Conference FGCS*, pages 244–253, 1984.

[29] P. Wegner and S. Szodink. Inheritance as an incremental modification mechanismor what like is ans isnt like. In *Proceedings of ECOOP 88*, number 322 in LNCS. Springer-Verlag, 1988.

Appendix: The Operational Semantics

Let:

- A, A' be atomic goal formulae;
- g be an atomic goal or a context extension/switch goal;
- G a conjunction of atomic goals;
- ϵ, θ, σ δ be substitutions; ϵ be the empty substitution;
- $(\theta\sigma)$ be the composition of the substitutions θ and σ;
- $G\theta$ be the application of the substitution θ to the formula G;
- $mgu(A, A')$ be the most general unifier of the atomic formulas A and A';

- the atomic clauses have the conventional body "true", which always holds;
- $U(P)$ be the set $\{u \mid u \text{ is a unit name}\}$;
- $|u| = \{c \mid c \text{ is a clause in u}\}$;
- $C(P) = \{ctx \mid ctx \text{ is a list of unit names}\}$
- $hierarchy : U(P) \mapsto C(P)$ be the following function:

$$hierarchy(u) \quad = \begin{cases} u.hierarchy(u') & \text{if } u' \text{ is the parent unit of } u. \\ \lfloor\ \rfloor & \text{if } u \text{ is the top unit of the hierarchy} \end{cases}$$

A $G\vartheta$ is derivable starting from the context C if there exists a proof for $C \vdash_\vartheta G$. A proof for $C \vdash_\vartheta G$ is a tree such that:

- The root node is labelled by $C \vdash_\vartheta G$
- The internal nodes are derived by using the following inference rules
- All the leaves are labelled by the empty formula ($true$)

TRUE:

$$\frac{}{\lfloor u_N, \ldots, u_1 \rfloor \ \vdash_\epsilon true}$$

CONJUNCTION:

$$\frac{\lfloor u_N, \ldots, u_1 \rfloor \ \vdash_\theta g; \lfloor u_N, \ldots, u_1 \rfloor \ \vdash_\sigma G\theta}{\lfloor u_N, \ldots, u_1 \rfloor \ \vdash_{\theta\sigma} (g, G)}$$

ATOMIC GOAL **(Evolving Systems)**:

$$\frac{A' : -G \in |u_i|; \theta = mgu(A, A'); \lfloor u_N, \ldots, u_1 \rfloor \ \vdash_\sigma G\theta}{\lfloor u_N, \ldots, u_1 \rfloor \ \vdash_{\theta\sigma} A}$$

ATOMIC GOAL **(Conservative Systems)**:

$$\frac{A' : -G \in |u_i|; \theta = mgu(A, A'); \lfloor u_i, \ldots, u_1 \rfloor \ \vdash_\sigma G\theta}{\lfloor u_N, \ldots, u_1 \rfloor \ \vdash_{\theta\sigma} A}$$

CONTEXT SWITCH **(Static Systems)**:

$$\frac{u \in U(P); ctx = hierarchy(u); ctx \vdash_\theta g}{ctx' \vdash_\theta u : g}$$

CONTEXT SWITCH **(Dynamic Systems)**:

$$\frac{u \in U(P); \lfloor u \rfloor \vdash_\theta g}{ctx \vdash_\theta u : g}$$

CONTEXT EXTENSION **(Dynamic Systems)**:

$$\frac{u \in U(P); u.ctx \vdash_\theta g}{ctx \vdash_\theta u \gg g}$$