A Transformation System for CLP with Dynamic Scheduling and ccp

Sandro Etalle[‡] Maurizio Gabbrielli[§] Elena Marchiori[¶]

Abstract

In this paper we study unfold/fold transformations for constraint logic programs (CLP) with dynamic scheduling and for concurrent constraint programming (ccp). We define suitable applicability conditions for this transformations which ensure us that the original and the transformed program have the same results of successful computations and have the same deadlocked derivations.

The possible applications of these results are twofold. On one hand we can use the unfold/fold system to optimize CLP and ccp programs while preserving their intended meaning and in particular without the risk of introducing deadlocks. On the other hand, unfold/fold transformations can be used for proving deadlock freedom for a class of queries in a given program: to this aim it is sufficient to specialize the program with respect to the given queries in such a way that the resulting program is trivially deadlock free. As shown by several interesting examples, this yields a methodology for proving deadlock freedom which is simple and powerful at the same time. All our results extend smoothly from CLP with dynamic scheduling to ccp.

Keywords: Program's Transformation, Constraint Logic Programming, Concurrent Constraint Programming, Coroutining, Deadlock.

1 Introduction

Constraint Logic Programming (CLP for short) is a powerful declarative programming paradigm which has been successfully applied to several diverse fields such as financial analysis [21], circuit synthesis [15] and combinatorial search problems [31]. The main reason for the interest in this paradigm is that it combines in a clean and effective way the power of constraint solving with logical deduction. In this way complex practical problems can be tackled by using (reasonably) simple and concise programs.

In nearly all the practical CLP systems [18] the flexibility of CLP computation is further enhanced by allowing a dynamic selection rule which allows to dynamically delay (or suspend) the selection of an atom until its arguments are sufficiently instantiated. This *dynamic scheduling* is obtained by adding the so-called *delay declarations* next to program clauses. Delay declarations, advocated by van Emden and de Lucena [30] and introduced explicitly in logic programming by Naish in [25], allow to improve the efficiency of programs, to prevent run-time errors and to enforce termination. In many CLP systems they are used also to postpone the evaluation of constraints which are "too hard" for the solver. For example, in CLP($\Re$) non-linear arithmetic constraints are delayed until they become linear. More generally, delay declarations provide the programmer with a better control over the computation and, similarly to guards in concurrent logic languages, allow one to express some degree of synchronization, usually called coroutining, among the different processes (i.e. atoms) in a program. In this sense, even though the underlying computational model uses a different kind of non-determinism, CLP with dynamic scheduling can be considered as an intermediate language between the purely sequential CLP and the concurrent language ccp.

The increased expressiveness of the language mentioned above comes with a price: in presence of dynamic scheduling the computation can end up into a state in which no atom can be selected for reduction.

[‡]D.I.S.I, Università di Genova, Via Dodecaneso, 35, 16146 Genova, Italy. sandro@disi.unige.it. Phone: ++39 (0)10 3536732, Fax ++39 (0)10 3536699.

[§]Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. gabbri@di.unipi.it.

[¶]Dipartimento di Matematica Applicata e Informatica, Università di Venezia, Italy. elena@dsi.unive.it.

This situation, analogous to that one which can arise when considering concurrent (constraint) languages, is called *deadlock* and is clearly undesirable, since it corresponds to a programming error. Checking whether the computation for a generic query in a program will end up in a deadlock is an undecidable yet crucial problem. In order to give it a (partial) solution, several techniques have been employed. Among them we should mention those based on abstract interpretation (in [6, 7]), mode and type analysis (in [1, 12]) and assertions (in [23, 4]).

A central issue for the development of large, correct and efficient applications is the study of optimization techniques. When considering (constraint) logic programs, the literature on this subject can be divided into two main branches. On one hand, we find several methods for compile-time optimization based on abstract interpretation. These methods have been recently applied also to CLP with dynamic scheduling [10, 24, 9] with promising results. On the other hand, there are techniques based on transformation of programs such as unfold/fold and replacement. As shown by several applications, these techniques provide a powerful methodology for program development, synthesis and optimization.

Historically, the unfold/fold transformation rules, were first introduced for functional programs in [3], and then adapted to logic programs both for program synthesis [5, 16], and for program specialization and optimization [19]. Soon later, Tamaki and Sato [29] proposed an elegant framework for the unfold/fold transformation of logic programs which has recently been extended to CLP in [2, 22, 13] (for an overview of the subject, see the survey by Pettorossi and Proietti [26]). However, these systems cannot be applied as they are to languages with dynamic scheduling. This is due to the following reasons. First, as we have already mentioned, a peculiar feature of such languages is that the computation might deadlock. It is therefore crucial that the transformation system does not introduce new deadlocking computation in the resulting program. For this new applicability conditions and new correctness results are needed. Secondly, languages with dynamic scheduling (often) use special constructs (similar to the ask of ccp) for providing simple yet powerful suspension tools. In order to handle these tools one needs new, specific transformations.

In this paper we handle these problems as follows. First, we concentrate on CLP with dynamic scheduling, enhanced with an if then else construct, and we extend the transformation system defined in [13]. The extension is twofold: on one hand we provide new applicability conditions which guarantee that the initial and the transformed program have exactly the same answer constraints and the same deadlocking derivations. On the other hand we define a new transformations for handling the if then else parts. Thanks to these new operations we can now perform optimization which were not possible in [13]. Afterward, we show that the resulting transformation system can be smoothly extended to ccp. Actually, in this case, due to the more expressive language, we can weaken the applicability conditions and still have a correctness result which ensure us that results of successful computations and deadlocks are preserved.

These results for CLP and ccp have applications to both the practical issues mentioned above.

Firstly, they can be used for optimizing programs while preserving their intended meaning and, in particular, without the risk of introducing deadlocked derivations. We will show examples of such optimizations.

Secondly, Unfold/Fold transformations can be used for proving deadlock freedom of a class of (intended) queries for a given program. To this aim, it is sufficient to specialize the program with respect to the considered queries and then to prove deadlock freedom of the resulting program. This latter proof is in many cases trivial, either because the resulting CLP program has no delay declarations at all or because the guards of the resulting ccp program are trivially satisfied. In this way we obtain a method for proving deadlock freedom that is simple and powerful at the same time. Its simplicity stems from the fact that it only uses transformation operations, thus it does not have to introduce other formalisms, like modes, types or assertions, as in the above mentioned proof methods, or like abstract domains, as in the methods based on abstract interpretation. Its power will be demonstrated in the paper by applying it to several non trivial programs whose execution uses full coroutine.

This paper is organized as follows. The next section contains some preliminaries on CLP with dynamic scheduling and on the if − then − else construct (some more details are deferred to the Appendix). Section 3 contains two examples which illustrate the use of transformations for optimizing programs and proving deadlock freedom. In Section 4 we define formally the transformation system and Section 5 shows its application to the proof of deadlock freedom. Finally in Section 6 we extend previous system and results to the ccp languages.

2 Preliminaries

We recall here some basic notions on CLP with dynamic scheduling, deferring to the Appendix some more details concerning the computational model of this language. For a thorough treatment of the theory of constraint logic programming, the reader is referred to the original paper [17] by Jaffar and Lassez and to the survey [18] by Jaffar and Maher.

A *constraint* c is a (first-order) formula built using predefined predicates (called primitive constraints) over a computational domain $\mathcal{D}$. Formally, $\mathcal{D}$ is a *structure* which determines the interpretation of the constraints, and the formula $\mathcal{D} \models \forall c$ denotes that c is true under the interpretation provided by $\mathcal{D}$, for every binding of its free variables. The empty conjunction of primitive constraints will be identified with true. A CLP rule is a formula of the form

$\mathsf{H} \leftarrow \mathsf{c}, \mathsf{B}_1, \ldots, \mathsf{B}_n.$

where c is a constraint, H (the head) is an atom and $\mathsf{B}_1, \ldots, \mathsf{B}_n$ (the body) is a sequence of atoms[1] and the connective "," denotes conjunction. Analogously, a *query* is denoted by $\mathsf{c}, \mathsf{B}_1, \ldots, \mathsf{B}_n$. In the sequel, $\tilde{\mathsf{t}}$ and $\tilde{\mathsf{X}}$ denote a tuple of terms and a tuple of distinct variables, respectively, while $\tilde{\mathsf{B}}$ denotes a (finite, possibly empty) conjunction of atoms. For a formula ϕ, its existential closure is denoted by $\exists\phi$, while $\exists_{-\tilde{x}}\ \phi$ stands for the existential closure of ϕ *except* for the variables in $\tilde{x}$ which remain unquantified. Moreover, for two atoms A and H, the expression $\mathsf{A} = \mathsf{H}$ is used as shorthand for:

- $\mathsf{s}_1 = \mathsf{t}_1 \wedge \ldots \wedge \mathsf{s}_n = \mathsf{t}_n$, if, for some predicate symbol p, $\mathsf{A} \equiv \mathsf{p}(\mathsf{s}_1, \ldots, \mathsf{s}_n)$ and $\mathsf{H} \equiv \mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ (where $\equiv$ denotes syntactic equality)

- false, otherwise. This notation extends to conjunctions of atoms in the expected way.

The standard operational model of CLP is obtained from SLD resolution by simply substituting $\mathcal{D}$-solvability for unifiability. As previously mentioned, in many CLP systems the programmer can dynamically control the selection of the atoms in a derivation by augmenting a program with delay declarations. These allow to specify that the selection of an atom can be delayed, or suspended, until its arguments become sufficiently instantiated. So we consider declarations of the form

$\mathsf{delay}\ \mathsf{p}(\tilde{\mathsf{X}})\ \mathsf{until}\ \mathsf{Condition}$

where $\mathsf{Condition}$ is a formula (in some assertion language) whose free variables are contained in $\tilde{\mathsf{X}}$. Intuitively, the meaning of such a declaration is that the atom $\mathsf{p}(\tilde{\mathsf{X}})$ can be selected in a query c, Q only if the constraint c satisfies $\mathsf{Condition}$ (this can be generalized to an atom $\mathsf{p}(\tilde{\mathsf{t}})$ in the obvious way, as shown in the Appendix).

For example, consider a numerical constraint system like the one used in CLP($\Re$): the declaration $\mathsf{delay}\ \mathsf{p}(\mathsf{X})\ \mathsf{until}\ \mathsf{X} > 0$ means that $\mathsf{p}(\mathsf{X})$ can be selected in a query c, Q only if c forces X to assume values greater than 0, while the declaration $\mathsf{delay}\ \mathsf{q}(\mathsf{X})\ \mathsf{until}\ \mathsf{ground}(\mathsf{X})$ means that $\mathsf{q}(\mathsf{X})$ can be selected in a query c, Q only if c forces X to assume a unique value.

We consider as delay conditions formulas consisting of conjunctions and disjunctions of primitive constraints and of "delay predicates" ($\mathsf{ground}(\mathsf{X})$ for example is such a predicate). The resulting assertion language, as well as its interpretation, is formalized in the Appendix. The interpretation simply defines for each delay condition ϕ the set $[\![\phi]\!]$ of constraints which satisfy ϕ, where $[\![\phi]\!]$ is assumed being closed under entailment. This assumption corresponds to the fact that most of the existing programming systems assume that if an atom is not delayed then adding more information will never cause it to delay. Furthermore, delay conditions are usually assumed to be consistent wrt renaming, so we make the same assumption here. The Appendix contains a more precise formalization of these conditions.

We also assume that for each predicate symbol exactly one delay declaration is given. This is not restrictive, since multiple declarations can be obtained by using logical connectives in the syntax of conditions. When the condition holds vacuously (i.e., when it is satisfied by all the constraints) we simply omit the corresponding declaration.

[1] It is assumed that atoms use predicate symbols different from those of the primitive constraints.

We say that an atom $\mathsf{p}(\tilde{\mathsf{t}})$ satisfies its delay declaration in the context of a constraint c, if, assuming that delay $\mathsf{p}(\tilde{\mathsf{X}})$ until ϕ is the delay declaration for p we have that $(\mathsf{c} \wedge \tilde{\mathsf{t}} = \tilde{\mathsf{Y}})$ satisfies $\phi[\tilde{\mathsf{X}}/\tilde{\mathsf{Y}}]$[2], where $\tilde{\mathsf{Y}}$ are new distinct variables which do not appear in c and in $\tilde{\mathsf{t}}$.

A derivation step for a query c, Q in a program P consists in replacing an atom A selected in Q with the body of a (suitably renamed version of a) clause cl, provided that the constraint in cl is consistent with c, no variable clash occurs and A in the context of c satisfies its delay declaration in P. A *derivation* for a query Q in the program P is a finite or infinite sequence of queries, starting in Q, such that every next query is obtained from the previous one by means of a derivation step. In the following a derivation $\mathsf{Q}, \mathsf{Q}_1, \ldots, \mathsf{Q}_i$ in P will be denoted by $\mathsf{Q} \overset{\mathsf{P}}{\rightsquigarrow} \mathsf{Q}_i$.

A *successful* derivation (or *refutation*) for the query Q is a finite derivation whose last element is of the form c, i.e. consists only of a constraint. In this case, $\exists_{-\mathsf{vars}(\mathsf{Q})}\ \mathsf{c}$ is called the *answer constraint of* Q. Since the answer constraint represents the result of a successful computation, it is considered the standard observable property for CLP programs. Therefore we will consider it as the notion of observable to be preserved by our transformation system.

A *deadlocked derivation* is a finite derivation $\mathsf{Q} \overset{\mathsf{P}}{\rightsquigarrow} \mathsf{c}, \tilde{\mathsf{B}}$ such that $\tilde{\mathsf{B}}$ is a non-empty sequence of atoms, and each atom in $\tilde{\mathsf{B}}$ does not satisfy its delay declaration[3]. A query Q *deadlocks in* P if there exists a deadlocked derivation for Q in P, while Q *is deadlock free* in P if it does not deadlock in P .

2.1 Adding the *if - then - else*

The *if - then - else* is present in logic programming languages since their appearance, but it has always been rather overlooked by theoreticians and considered only as "syntactic sugar" for programs fragments involving cuts or negation-as-failure. The use of delay declarations in a CLP language allows us to usefully employ a restricted form of *if - then - else* construct, which still retains a simple declarative meaning. In this context the presence of this construct is crucial for two reasons: (i) it allows transformations which otherwise would not be possible; (ii) it allows us to introduces in the queries new suspension points without having to create new artificial predicates. Thus in the following we shall allow in a query constructs of the form

$$\mathsf{if}\ \mathsf{c}\ \ \mathsf{then}\ \ \tilde{\mathsf{A}}\ \ \mathsf{else}\ \tilde{\mathsf{B}}$$

where c is a constraint and $\tilde{\mathsf{A}}$ and $\tilde{\mathsf{B}}$ are queries. Operationally, the intended meaning of such a construct is that the execution of the queries $\tilde{\mathsf{A}}$ and $\tilde{\mathsf{B}}$ is delayed until the current store either entails c or entails $\neg\mathsf{c}$, i.e., until it is known which branch has to be selected. As soon as one of the two conditions is satisfied, the corresponding branch is selected and the other is discarded.

From a declarative viewpoint, the meaning of previous construct is simply obtained by considering an occurrence of if c then $\tilde{\mathsf{A}}$ else $\tilde{\mathsf{B}}$ as a shorthand for an occurrence of the predicate $\mathsf{if\_then\_else}(\tilde{\mathsf{X}})$ defined by the following program

$$\mathsf{if\_then\_else}(\tilde{\mathsf{X}}) \leftarrow \mathsf{c}, \tilde{\mathsf{A}}.$$
$$\mathsf{if\_then\_else}(\tilde{\mathsf{X}}) \leftarrow \neg\mathsf{c}, \tilde{\mathsf{B}}.$$

$$\mathsf{delay}\ \mathsf{if\_then\_else}(\tilde{\mathsf{X}})\ \mathsf{until}\ \mathsf{c} \vee \neg\mathsf{c}.$$

where, $\mathsf{if\_then\_else}$ is a new predicate symbol, $\tilde{\mathsf{X}} = \mathsf{vars}(\tilde{\mathsf{A}}, \tilde{\mathsf{B}})$ and the free variables appearing in c are contained in $\tilde{\mathsf{X}}$[4]. Notice that the above delay declaration forces the if - then - else to suspend itself until it is known which branch is to be selected. We assume here that for each occurrence of an if c then $\tilde{\mathsf{A}}$ else $\tilde{\mathsf{B}}$ construct a different if_then_else predicate symbol is introduced. It is immediate to check that the computational behavior of program is equivalent to that one previously described for the if-then-else. The reader

[2] $\phi[\tilde{\mathsf{X}}/\tilde{\mathsf{Y}}]$ denotes the formula obtained from ϕ by replacing the variables $\tilde{\mathsf{X}}$ for the variables $\tilde{\mathsf{Y}}$.

[3] Note that $\tilde{\mathsf{B}}$ does not contain constraints: In fact, usually the case in which a derivation ends in a query containing only suspended constraints is considered successful.

[4] Note that in the above clauses $\neg\mathsf{c}$ is a basic constraint: in fact negation is allowed at the "constraint level" and not at the level of delay declarations. Thus $\mathsf{c} \vee \neg\mathsf{c}$ is a legal delay condition and according to the interpretation formally defined in the Appendix it is satisfied by any d such that either d entails c or d entails $\neg\mathsf{c}$. Given this interpretation, clearly the delay condition $\mathsf{c} \vee \neg\mathsf{c}$ in general is not equivalent to true.

familiar with concurrent constraint programming has probably recognized in the above constructs a form of ask operator. We will discuss the relations existing between the two paradigms in Section 6.

In the following we will often replace a clause by an equivalent one in order to *clean up* the constraints, and, in general, to present a clause in a more readable form. Formally, such an equivalence $\simeq$ is defined as follows on clauses which do not contain if − then − else: $\mathsf{H}_1 \leftarrow \mathsf{c}_1, \tilde{\mathsf{B}}_1 \simeq \mathsf{H}_2 \leftarrow \mathsf{c}_2, \tilde{\mathsf{B}}_2$ iff for any $\mathcal{D}$-solution[5] ϑ of c_i there exists a $\mathcal{D}$-solution γ of c_j such that $\mathsf{H}_i\vartheta \equiv \mathsf{H}_j\gamma$ and $\tilde{\mathsf{B}}_i\vartheta$ and $\tilde{\mathsf{B}}_j\gamma$ are equal as multisets[6].

This notion of equivalence is extended to clauses containing the if − then − else by considering, in the obvious way, the $\simeq$ equivalence of the clauses obtained by their translation in standard clauses shown above. It is worth noticing that replacing a clause by a $\simeq$-equivalent one does not affect the applicability and the results of the transformations, and that all the observable properties we refer to are invariant under $\simeq$.

3 Motivating Examples

In this section we want to show what are the possibilities offered by program's transformation in the context of CLP with dynamic scheduling. We'll do this by giving two simple yet significant examples. In order to avoid making the reading too heavy, we postpone the formal definitions of the single operations (and their applicability conditions) until the next section.

To simplify the notation, here and in the sequel, when the constraint in a query or in a clause is true we omit it. So the notation $\mathsf{H} \leftarrow \tilde{\mathsf{B}}$ actually denotes the CLP clause $\mathsf{H} \leftarrow \mathsf{true}, \tilde{\mathsf{B}}$.

Example 3.1 The following program DELMAX allows one to delete all the occurrences of the maximum from a given list of integers.

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

d1: del_max(Xs, Zs) ← find_max(Xs, Max), del_el(Xs, Max, Zs).

%%%% find_max(InList, Max) ← Max is the maximum element of the list InList

    find_max([], 0).
    find_max([X|Xs], Max) ← find_max(Xs, Max'),
        if Max' < X then Max = X else Max = Max'.

%%%% del_el(InList, El, OutList) ← OutList is obtained from Inlist by deleting all the occurrences of El from it

    del_el([], _, []).
    del_el([X|Xs], El, OutList) ← del_el(Xs, El, OutList'),
        if El = X then OutList = OutList' else OutList = [X|OutList'].
```

Given the list Xs of integer values, del_max(Xs,Zs) produces the list Zs by deleting all the occurrences of its maximum element and in order to do this it scans the input list Xs twice. Now, via an unfold/fold transformation we can obtain an equivalent program which scans the list only once. For this we apply a transformation strategy known as *tupling* (cf. [26]) or as *procedural join* (cf. [20]). First, we add to the program a *new* predicate.

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

d2: find_max_and_del(InList,Max,El,OutList) ← find_max(InList, Max), del_el(InList, El, OutList).
```

Now, we *unfold* find_max(InList, Max) in the body of d2. This basic operation corresponds to applying a resolution step (in all possible ways). In this cases it originates the following two clauses.

```
    find_max_and_del([], 0, El,OutList) ← del_el([], El, OutList).
    find_max_and_del([X|Xs], Max, El, OutList) ← find_max(Xs, Max'),
        if Max' < X then Max = X else Max = Max'.
        del_el([X|Xs], El, OutList).
```

[5]If ϑ is a valuation (i.e., a function mapping variables into elements of $\mathcal{D}$), and $\mathcal{D} \models c\vartheta$ holds, then ϑ is called a $\mathcal{D}$*-solution* of c. Moreover, $c\vartheta$ denotes the application of ϑ to the free variables of c.

[6]Notice that considering sets here would not be correct.

By further unfolding del_el in both the above clauses, we obtain,

```
    find_max_and_del([], 0, _, []).
d3: find_max_and_del([X|Xs], Max, El, OutList) ← find_max(Xs, Max'),
        if Max' < X then Max = X else Max = Max',
        del_el(Xs, El, OutList'),
        if El = X then OutList = OutList' else OutList = [X|OutList'].
```

Now, we can *fold* find_max(Xs, Max'), del_el(Xs, El, OutList') in the body of d3, using d2 as folding clause. Intuitively, the folding operation corresponds to the inverse of the unfolding one. In this case, first notice that part of the body of d3, corresponds to a renaming of the body of d2, thus the folding operation will replace this part with the correspondent renaming of the head of d2. The result is a recursive definition:

```
    find_max_and_del([], 0, _, []).
    find_max_and_del([X|Xs], Max, El, OutList) ← find_max_and_del(Xs, Max', El, OutList'),
        if Max' < X then Max = X else Max = Max',
        if El = X then OutList = OutList' else OutList = [X|OutList'].
```

This definition traverses the input list only once. Now, in order to let del_max exploit the efficiency achieved by find_max, we have to fold find_max(Xs, Max), del_el(Xs, Max, Zs) in the body of the d1, thus obtaining the following final program DELMAX_EFF

```
d8: del_max(Xs, Zs) ← find_max_and_del(Xs, Max, Max, Zs).

    find_max_and_del([], 0, _, []).
    find_max_and_del([X|Xs], Max, El, OutList) ← find_max_and_del(Xs, Max', El, OutList'),
        if Max' < X then Max = X else Max = Max',
        if El = X then OutList = OutList' else OutList = [X|OutList'].
```

plus other clauses, which are no longer useful.

As we have already mentioned, this program has the advantage of needing to traverse the list only once. At the same time, its dynamic behavior is now much more complex than at the beginning. For instance, the Max variable of clause d8 is now used as an asynchronous communication channel between processes, in fact the atom find_max_and_del(Xs, Max, Max, Zs) in the body of d8 uses Max as input value that it has to produce itself. This is a typical situation in which we run the risk of entering in a deadlock.

Indeed, while the initial program could be run by using a dummy left-to right selection rule, the final program certainly cannot. Further, in the final program it is not at all immediate to check that the query del_max(gl, Zs) (gl being a list of positive integers) generates a non-deadlocking computation.

Summarizing, this program shows that Unfold/Fold transformations can be used to improve program's efficiency, but that they can lead to programs whose dynamic behavior is more complicate. To cope with this problem we *need to guarantee that the transformation does not introduce deadlocked derivations.*

Next, we show an opposite example, in which the unfold/fold transformation simplifies the dynamic behavior of a program.

Example 3.2 Consider the following simple Reader-Writer program, which uses a buffer of length one:

```
d1: read-write ← reader(Xs), writer(Xs).

%   reader(Xs) reads the input stream and puts the various tokens in the list Xs

    reader([Y|Ys]) ← read(Y), reader(Ys).
    reader([]).

    delay reader(Xs) until nonvar(Xs)

%   writer(Xs) writes the elements of the list Xs to the standard output
%       and stops doing so when it encounters the token eof

    writer([Y|Ys]) ← if Y=eof then Ys=[] else (write(Y), writer(Ys)).
```

The dynamic behavior of this program is not elementary. reader(Xs) behaves as follows: (a) waits until Xs is at partially instantiated, (b1) when Xs is instantiated to [Y|Ys] then it instantiates Y with the value it reads from the input, (b2) when Xs is instantiated to [] it stops. On the other hand, the actions of writer(Xs) are the following ones: (a) it instantiates Xs to [Y|Ys] (this activates reader(Xs)), (b) it waits until Y is instantiated (via the delay declaration embedded in the if construct), (c1) if Y is the *end of file* character then it instantiates Ys to [] (this will also stop the reader), (c2) otherwise it proceeds with the recursive call (which will further activate reader and so on).

Thus, read-write actually implements a communication channel with a buffer of length one, and Xs is actually a bidirectional communication channel. We now proceed with the transformation. First we unfold writer(Xs) in the body of d1, obtaining

read-write $\leftarrow$ reader([Y|Ys]), if Y=eof then Ys=[] else (write(Y), writer(Ys))

Then, we unfold reader([Y|Ys]) in the resulting clause,

read-write $\leftarrow$ read(Y), reader(Ys), if Y=eof then Ys=[] else (write(Y), writer(Ys)).

Now, we have to apply a new operation, that we'll call *distributive property*, in order to bring reader(Ys) inside the scope of the if construct.

read-write $\leftarrow$ read(Y), if Y=eof then (Ys=[], reader(Ys)) else (write(Y), reader(Ys), writer(Ys)).

We can now *fold* reader(Ys), writer(Ys), using d1 as folding clause, and obtain,

read-write $\leftarrow$ read(Y), if Y=eof then (Ys=[], reader(Ys)) else (write(Y), read-write).

Finally, we can now *evaluate* the constraint Ys=[] in the first branch of the if construct (this operation corresponds to substituting the clause for a $\simeq$-equivalent one)

read-write $\leftarrow$ read(Y), if Y=eof then reader([]) else (write(Y), read-write).

In the end, we can unfold reader([])

read-write $\leftarrow$ read(Y), if Y=eof then (*succeed*) else (write(Y), read-write).

Which is our final program □

Three aspects are important to notice.

First, that the resulting program is more efficient than the initial one: in fact it does not need to use the heap as heavily as the initial one for passing the parameters between reader and writer. Furthermore, modern optimization techniques such as the ones implemented for the logic language MERCURY (for more informations, see `http://www.cs.mu.oz.au/mercury/papers.html`), exploit the fact that Y has no aliases in order to avoid saving it on the heap altogether, and further compiling reader-writer as a simple while program, dramatically saving memory space while speeding up the compiled code (these latter techniques are not yet implemented for programs with dynamic scheduling).

Second, that – as opposed to the initial program – the resulting one has a straightforward dynamic behavior. For instance it can be run with a simple left to right selection rule. Further, if we consider the query reader-write, one can easily see it to be deadlock-free in the latter program (to prove it formally, a straightforward extension of the tools of [1] is sufficient), while in the original program this is not at all immediate. After proving that the transformation does not introduce *nor eliminates* any deadlocking branch in the semantics of the program, we'll be able to state that "since the resulting program is deadlock-free then also the initial program is deadlock-free". Thus program's transformations can be profitably used as analysis tool: it is in fact often easier to prove deadlock freedom for a transformed version of a program than for the original one.

Third, that in order to achieve this goals, we have introduced two new operations, strictly extending the existing transformation systems.

4 The transformations

As customary for fold/unfold systems, we start with some requirements on the original (i.e., initial) program that one wants to transform. Here we say that a predicate p is *defined* in a program P, if P contains at least one clause whose head has predicate symbol p.

Definition 4.1 (Initial program) We call a CLP program P_0 an *initial program* if the following two conditions are satisfied:

(I1) P_0 is partitioned into two disjoint sets $\mathsf{P}_{\mathsf{new}}$ and $\mathsf{P}_{\mathsf{old}}$,

(I2) the predicates defined in $\mathsf{P}_{\mathsf{new}}$ do not occur neither in $\mathsf{P}_{\mathsf{old}}$ nor in the bodies of the clauses in $\mathsf{P}_{\mathsf{new}}$.

In presence of delay declarations, we also need the following:

(D1) No atom defined in $\mathsf{P}_{\mathsf{new}}$ is subject to a delay declaration. □

Following the notation above, we call *new* predicates those predicates that are defined in $\mathsf{P}_{\mathsf{new}}$. As we shall discuss in the following, previous conditions (I2) and (D1) are needed to ensure the correctness of the folding operation.

The first transformation we consider is the *unfolding*. Since all the observable properties we consider are invariant under reordering of the atoms in the bodies of clauses, the definition of unfolding, as well as those of the other operations, is given modulo reordering of the bodies atoms. To simplify the notation, in the following definition we also assume that the clauses of a program and its delay declarations have been renamed so that they do not share variables pairwise.

Definition 4.2 (Unfolding) Let $\mathsf{cl} : \mathsf{A} \leftarrow \mathsf{c}, \mathsf{H}, \tilde{\mathsf{K}}$ be a clause in the program P, and $\{\mathsf{H}_1 \leftarrow \mathsf{c}_1, \tilde{\mathsf{B}}_1, \ldots, \mathsf{H}_n \leftarrow \mathsf{c}_n, \tilde{\mathsf{B}}_n\}$ be the set of the clauses in P such that $\mathsf{c} \wedge \mathsf{c}_i \wedge (\mathsf{H} = \mathsf{H}_i)$ is $\mathcal{D}$-satisfiable. For $\mathsf{i} \in [1, \mathsf{n}]$, let cl'_i be the clause

$$\mathsf{A} \leftarrow \mathsf{c} \wedge \mathsf{c}_i \wedge (\mathsf{H} = \mathsf{H}_i), \tilde{\mathsf{B}}_i, \tilde{\mathsf{K}}$$

Then *unfolding H in cl in P* consists of replacing cl by $\{\mathsf{cl}'_1, \ldots, \mathsf{cl}'_n\}$ in P. Moreover we say that the unfolding operation is *allowed* iff

(D2) the atom H in the context c satisfies its delay declaration in P. □

Condition (D2) is clearly needed to avoid the transformation of a deadlocked derivation into a successful one, since once the atom H has been unfolded its delay declaration is not anymore visible.

We are now ready to introduce the folding operation. This operation is a fundamental one, as it allows one to introduce recursion in the new definitions. Here we use a formalization of the applicability conditions given in [13] which follows this intuitive idea. As in [29], the applicability conditions of the folding operations depend on the history of the transformation. Therefore we define a transformation sequence as follows.

Definition 4.3 A *transformation sequence* is a sequence of programs $\mathsf{P}_0, \ldots, \mathsf{P}_n$, in which P_0 is an initial program and each P_{i+1}, is obtained from P_i via a transformation operation of unfolding, clause removal, splitting, constraint replacement and folding, whose applicability conditions are satisfied.

As usual, in the following definition we assume that the folding (d) and the folded (cl) clause are renamed apart and, as a notational convenience, that the body of the folded clause has been reordered so that the atoms that are going to be folded are found in the leftmost positions. Moreover, recall that the initial program P_0 is partitioned into $\mathsf{P}_{\mathsf{new}}$ and $\mathsf{P}_{\mathsf{old}}$.

Definition 4.4 (Folding) Let $\mathsf{P}_0, \ldots, \mathsf{P}_i$, $\mathsf{i} \geq 0$, be a transformation sequence. Let also
$\mathsf{cl} : \mathsf{A} \leftarrow \mathsf{c}_\mathsf{A}, \tilde{\mathsf{K}}, \tilde{\mathsf{J}}$ be a clause in P_i,
$\mathsf{cl}_\mathsf{f} : \mathsf{D} \leftarrow \mathsf{c}_\mathsf{D}, \tilde{\mathsf{H}}$ be a clause in $\mathsf{P}_{\mathsf{new}}$.

If $\mathsf{c_A}, \tilde{\mathsf{K}}$ is an instance[7] of *true*, $\tilde{\mathsf{H}}$ and e is a constraint such that $\mathsf{vars(e)} \subseteq \mathsf{vars(D)} \cup \mathsf{vars(cl)}$, then *folding* $\tilde{\mathsf{K}}$ *in* cl *via* e consists of replacing cl by

$$\mathsf{cl}' :\ \mathsf{A} \leftarrow \mathsf{c_A} \wedge \mathsf{e}, \mathsf{D}, \tilde{\mathsf{J}}$$

provided that the following three conditions hold:

(F1) *"If we unfold* D *in* cl' *using* $\mathsf{cl_f}$ *as unfolding clause, then we obtain* cl *back"* (*modulo* $\simeq$),

(F2) *"*$\mathsf{cl_f}$ *is the only clause of* $\mathsf{P_{new}}$ *that can be used to unfold* D *in* cl'*"*, that is, there is no clause $\mathsf{b} :\ \mathsf{B} \leftarrow \mathsf{c_B}, \tilde{\mathsf{L}}$ in $\mathsf{P_{new}}$ such that $\mathsf{b} \neq \mathsf{d}$ and $\mathsf{c_A} \wedge \mathsf{e} \wedge (\mathsf{D} = \mathsf{B}) \wedge \mathsf{c_B}$ is $\mathcal{D}$-satisfiable.

(F3) *"No self-folding is allowed"*, that is

(a) either the predicate in A is an old predicate;

(b) or cl is the result of at least one unfolding in the sequence $\mathsf{P_0}, \ldots, \mathsf{P_i}$. □

The constraint e acts as a bridge between the variables of $\mathsf{cl_f}$ and cl. As shown in [13], conditions **F1** and **F2** ensure that the folding operation behaves, to some extent, as the inverse of the unfolding one; the underlying idea is that if we unfolded the atom D in cl' using only clauses from $\mathsf{P_{new}}$ as unfolding clauses, then we would obtain cl back. In this context condition **F2** ensures that in $\mathsf{P_{new}}$ there exists no clause other than $\mathsf{cl_f}$ that can be used as *unfolding clause*. Condition **F3** is from the original Tamaki-Sato's definition of folding for logic programs. Its purpose is to avoid the introduction of loops which can occur if a clause is folded by itself.

Finally, note that since the folding clause $\mathsf{cl_f}$ has to be in $\mathsf{P_{new}}$, condition (D1) in the definition of initial program implies that the head of the folding clause is not subject to any delay declaration. This condition is needed in order to avoid to transform a successful derivation into a deadlocked one, as shown by the following example. Consider the program P:

```
p(X) ← q(X), r(X).
m(X) ← q(X).
q(X).
r(X).

delay m(X) until ground(X)
```

If we fold the first clause using the second one as folding clause, we obtain the program P' which is not equivalent to P. In fact the query $\mathsf{p(X)}$ has a successful derivation in P and not in P'.

Finally, we consider a transformation consisting in distributing an atom over the $\mathsf{if} - \mathsf{then} - \mathsf{else}$ construct in the body of a a clause. This operation is often needed to be able apply the folding operation.

Definition 4.5 (Distributive Property) Let

$$\mathsf{cl} :\quad \mathsf{H} \leftarrow \mathsf{c}\ ,\ \mathsf{K},\ \tilde{\mathsf{J}},\ \mathsf{if\ d\ then}\ \tilde{\mathsf{A}}\ \mathsf{else}\ \tilde{\mathsf{B}}.$$

be a clause in a program P, then applying the *distributive property* to K in cl means replacing cl with

$$\mathsf{cl}' :\quad \mathsf{H} \leftarrow \mathsf{c},\ \tilde{\mathsf{J}},\ \mathsf{if\ d\ then}\ (\tilde{\mathsf{A}},\ \mathsf{K})\ \mathsf{else}\ (\tilde{\mathsf{B}},\ \mathsf{K}).$$

Let $\tilde{\mathsf{X}} = \mathsf{vars}(\mathsf{H}, \tilde{\mathsf{J}})$. This operation is *allowed* provided that

(D3) for each constraint $\mathsf{d}(\tilde{\mathsf{X}})$ (having $\tilde{\mathsf{X}}$ as free variables), we have that in the context $\mathsf{c} \wedge \mathsf{d}(\tilde{\mathsf{X}})$, K does *not* satisfy its delay declaration in P. □

Condition **D3** is needed to avoid the following problem. Consider the program.

```
p(X) ←q(X), if X=a then r(X) else t(X).
q(a).
r(a).
```

[7]By naturally extending the usual notion used for pure logic programs, we say that a query $c \,\Box\, \tilde{C}$ is *an instance* of the query $d \,\Box\, \tilde{D}$ iff for any solution γ of c there exists a solution δ of d such that $\tilde{C}\gamma \equiv \tilde{D}\delta$.

If we overlook condition **D3** we could transform it into

```
p(X) ← if X=a then (q(Y), r(X)) else (q(Y), t(X)).
q(a).
r(a).
```

However, in the first program the query p(X) succeeds, while in the second one it deadlocks (being X free the if construct suspends).

The correctness of the introduced transformations is stated by the following.

Theorem 4.6 (Correctness) Let $P_0, \dots, P_n$ is a transformation sequence and Q be a query. Then for any P_i, $P_j \in P_0, \dots, P_n$ we have that

(i) Q has a deadlocked derivation in P_i iff it has a deadlocked derivation in P_j.

(ii) Q has the same answer constraints in P_i and P_j, i.e., there exists a successful derivation $Q \overset{P_i}{\rightsquigarrow} d_i$ iff there exists a successful derivation $Q \overset{P_j}{\rightsquigarrow} d_j$, and $\mathcal{D} \models \exists_{-vars(Q)}\, d_i \leftrightarrow \exists_{-vars(Q)}\, d_j$.

Other Operations. Other (less prominent) operations which can nevertheless be useful in program's transformation are the *splitting* and the *constraint replacement.* Due to space limitations we are not able to provide significant examples of these operations (although, the splitting one is employed in the ccp example at the end of the paper). Therefore, we now simply report their definitions and we refer to [13] for further information on them. In the following definition, just like for the unfolding operation, it is assumed that program clauses do not share variables pairwise.

Definition 4.7 (Splitting) Let $cl : A \leftarrow c, H, \tilde{K}$ be a clause in the program P, and $\{H_1 \leftarrow c_1, \tilde{B}_1, \dots, H_n \leftarrow c_n, \tilde{B}_n\}$ be the set of the clauses in P such that $c \wedge c_i \wedge (H = H_i)$ is $\mathcal{D}$-satisfiable. For $i \in [1, n]$, let cl'_i be the clause $A \leftarrow c \wedge c_i \wedge (H = H_i), H, \tilde{K}$. If, for any $i, j \in [1, n]$, $i \neq j$, the constraint $(H_i = H_j) \wedge c_i \wedge c_j$ is unsatisfiable then *splitting H in cl in P* consists of replacing cl by $\{cl'_1, \dots, cl'_n\}$ in P. This operation is *allowed* iff

(D4) the atom H in the context c satisfies its delay declaration in P. □

Differently from previous cases, the applicability condition for *constraint replacement* relies on the semantics of the program. In fact, differently from the "cleaning up" of constraints previously discussed, this operation allows us to replace a clause for another one which is not $\simeq$-equivalent: the equivalence is ensured only with respect to to the given program.

Definition 4.8 (Constraint Replacement) Let $cl : H \leftarrow c_1, \tilde{B}$ be a clause of a program P and let c_2 be a constraint. If,

(D5) for each successful or deadlocked derivation $true, \tilde{B} \overset{P}{\rightsquigarrow} d, \tilde{D}$ we have that $H \leftarrow c_1 \wedge d, \tilde{D} \simeq H \leftarrow c_2 \wedge d, \tilde{D}$

then *replacing* c_1 *by* c_2 *in* c_l consists in substituting cl by $H \leftarrow c_2, \tilde{B}$ in P. □

The correctness of the newly introduced transformations is stated by the following.

Corollary 4.9 (Correctness) Theorem 4.6 holds also if in the transformation sequence we have additionally employed splitting and constraint replacement. □

5 Proving Deadlock Freedom via Fold/Unfold

As we have already seen in Example 3.2, unfold/fold transformations can be employed for proving deadlock-freeness of a certain query. The underlying idea is simple: we transform (specialize) the program until the query we are interested in becomes independent from those predicates which are subject to a delay declaration. If we manage, then by Theorem 4.6 we have proven that the query is deadlock-free also in the initial program. This can be extended to a class of queries in the obvious way.

More in general, program's specialization can be employed to prove absence of deadlock in combination with known methods. For instance, in the program Reader-Writer, one can easily extend the tools of Apt and Luitjes [1] to CLP programs with if then else in order to prove that the query read-write is deadlock-free in the *last* program of the sequence. Thus by Theorem 4.6 read-write is deadlock-free also in the *first* program of the sequence. Proving this latter statement with the (extended) tools of [1] is not possible. Thus, despite its simplicity, program's specialization is a rather powerful tool for proving deadlock freedom.

In order to further substantiate our assertions, we now consider a more complicated example, implementing a static network of stream transducers. This program solves the so-called Hamming problem introduced by Dijkstra: Generate an ordered stream of all numbers of the form $2^i 3^j 5^k$ without repetitions. To this end, five processes are used, that interleave their execution, thus giving rise to a number of different schedulings.

Example 5.1 In [30], van Emden and de Lucena proposed a solution for the Hamming problem based on the incorporation of coroutining in the execution of a logic program. Using our formalism with delay declarations, such a solution is implemented by the following program HAMMING

```
hammings(X) ← mult([1|X],2,X2), mult([1|X],3,X3), mult([1|X],5,X5), merge(X2,X3,X23), merge(X5,X23,X).

merge([X|Xi1],[X|Xi2],[X|Xo]) ← merge(Xi1,Xi2,Xo).
merge([X|Xi1],[Y|Xi2],[X|Xo]) ← X<Y, merge(Xi1,[Y|Xi2],Xo).
merge([X|Xi1],[Y|Xi2],[Y|Xo]) ← X>Y, merge([X|Xi1],Xi2,Xo).

mult([X|Xs],Y,[X*Y|Z]) ← mult(Xs,Y,Z).

delay mult(X,Y,Z) until nonvar(X) and ground(Y)
delay merge(X,Y,Z) until nonvar(X) and nonvar(Y)
```

We prove that hammings(X) is deadlock free in HAMMING. The problem with this program is that the execution of hammings(X) uses various schedulings of the body atoms of its first clause. For instance, initially, only the leftmost three atoms are selectable. After one resolution step, all these atoms become suspended. At this point, the leftmost merge atom is allowed to proceed. After one execution step, its execution is also suspended. However, at this point the first two arguments of the last merge atom have been instantiated in such a way that it can be selected. After one resolution step, we obtain the initial situation, with the three mult processes active and the other two processes suspended.

Our proof consists of the following steps:

1. We introduce the new clause:

```
ham1([X|Xs],[Y|Xs], [Z|Xs], X2,X3,X5,X23) ← mult([X|Xs],2,X2), mult([Y|Xs],3,X3), mult([Z|Xs],5,X5),
    merge(X2,X3,X23), merge(X5,X23,Xs).
```

2. By folding the body of the clause defining hammings via the above clause we obtain the clause:

```
hammings(Xs) ← ham1([1|Xs],[1|Xs],[1|Xs],X2,X3,X5,X23).
```

3. Unfolding the first three body atoms of the definition of ham1, we obtain:

```
ham1([X|Xs],[Y|Xs],[Z|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],X23) ← mult(Xs,2,X2), mult(Xs,3,X3), mult(Xs,5,X5),
    merge([X*2|X2],[Y*3|X3],X23), merge([Z*5|X5],X23,Xs).
```

4. Unfolding merge([X|X2],[Y|X3],X23) we get the three clauses:

```
ham1([X|Xs],[Y|Xs],[Z|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[X*2|X23]) ← X*2 < Y*3,
    mult(Xs,2,X2), mult(Xs,3,X3), mult(Xs,5,X5), merge(X2,[Y*3|X3],X23), merge([Z*5|X5],[X*2|X23],Xs).
ham1([X|Xs],[Y|Xs],[Z|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[Y*3|X23]) ← X*2 > Y*3,
    mult(Xs,2,X2), mult(Xs,3,X3), mult(Xs,5,X5), merge([X*2|X2],X3,X23), merge([Z*5|X5],[Y*3|X23],Xs).
ham1([X|Xs],[Y|Xs],[Z|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[X*2|X23]) ← X*2 = Y*3,
    mult(Xs,2,X2), mult(Xs,3,X3), mult(Xs,5,X5), merge(X2,X3,X23), merge([Z*5|X5],[X*2|X23],Xs).
```

5. Unfolding the last body atom of each of the clauses obtained in the previous step, we get nine clauses. We give here only the three obtained by unfolding the last clause in the previous step; the other ones have a similar structure.

```
ham1([X,X*2|Xs],[Y,X*2|Xs],[Z,X*2|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[X*2|X23]) ← X*2 = Y*3, Z*5 > X*2,
    mult([X*2|Xs],2,X2), mult([X*2|Xs],3,X3), mult([X*2|Xs],5,X5), merge(X2,X3,X23), merge([Z*5|X5],X23,Xs).
ham1([X,Z*5|Xs],[Y,Z*5|Xs],[Z,Z*5|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[X*2|X23]) ← X*2 = Y*3, Z*5 < X*2,
    mult([Z*5|Xs],2,X2), mult([Z*5|Xs],3,X3), mult([Z*5|Xs],5,X5), merge(X2,X3,X23), merge(X5,[X*2|X23],Xs).
ham1([X,X*2|Xs],[Y,X*2|Xs],[Z,X*2|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[X*2|X23]) ← X*2 = Y*3, Z*5 = X*2,
    mult([X*2|Xs],2,X2), mult([X*2|Xs],3,X3), mult([X*2|Xs],5,X5), merge(X2,X3,X23), merge(X5,X23,Xs).
```

6. Folding each clause obtained in the previous step, by using the clause introduced in step 1, we gets nine clauses. We give only those obtained by folding of the three clauses given in the previous step. Those for the other clauses are similar.

```
ham1([X,X*2|Xs],[Y,X*2|Xs],[Z,X*2|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[X*2|X23]) ← X*2 = Y*3, Z*5 > X*2,
    ham1([X*2|Xs],[X*2|Xs],[X*2|Xs],X2,X3,[Z*5|X5],X23).
ham1([X,Z*5|Xs],[Y,Z*5|Xs],[Z,Z*5|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[X*2|X23]) ← X*2 = Y*3, Z*5 < X*2,
    ham1([Z*5|Xs],[Z*5|Xs],[Z*5|Xs],[X*2|X2],X3,X5,X23).
ham1([X,X*2|Xs],[Y,X*2|Xs],[Z,X*2|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[X*2|X23]) ← X*2 = Y*3, Z*5 = X*2,
    ham1([X*2|Xs],[X*2|Xs],[X*2|Xs],X2,X3,X5,X23).
```

The resulting program – restricted to the definition of hammings – is the clause of step 2 plus the clauses obtained in the last step. This program is readily deadlock free (it does not have any delay declaration), hence, by Theorem 4.6 hammings([1|Xs],[1|Xs],[1|Xs],X2,X3,X5,X23) is deadlock free in HAMMING. □

It is worth noticing that the proof methods for proving deadlock freedom defined in [1, 12, 23, 4] cannot cope with this program. Thus the only available methods for proving hamming's deadlock freeness could be those based on abstract interpretation. However, these methods require the definition of suitable abstract domains which for deadlock analysis are rather complicated. We believe that our method based on transformations – possibly combined with the techniques which use mode and type information as in [1] – has the advantage of combining simplicity with usefulness, and represents therefore a valid alternative to abstract interpretation[8].

6 Extension to ccp

The CLP paradigm we have considered in the previous sections is probably the logic language which has the greatest impact on practical applications. Nevertheless, in the field of concurrent programming, there exist more expressive languages which allow for more sophisticated synchronization mechanisms. In particular concurrent constraint programming (ccp) [27] can be considered the natural extension of CLP. In fact, in [8] it has been shown that CLP with dynamic scheduling can be embedded into a strict sublanguage of ccp. Such an increased expressive power allow us to define a more flexible transformation system. As an example, consider the clause $cl:\ H \leftarrow c, \tilde{B}, A$, and assume that A is defined by the single clause $A \leftarrow \tilde{C}$ and that in the context c, A does not satisfy its delay declaration delay A until ϕ. In this situation, we cannot unfold A in cl (it is forbidden by condition **D2**). On the other hand, this would be possible in ccp, in fact the definition of A would be $A \leftarrow\ (ask(\phi) \rightarrow \tilde{C})$, and the result of the unfolding operation would be $cl' :\ H \leftarrow c, \tilde{B}, ask(\phi) \rightarrow (\tilde{C})$. In other words, in ccp the ask construct allows us to

[8]Furthermore, there are cases which cannot be handled using abstract interpretation while they can be handled with other techniques. For instance, as discussed in [6], abstract interpretation does not allow to prove absence of deadlock for the query p(a, Y), test(Y) in the following program implementing the so called *short circuit technique*:

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

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

On the other hand, such a query can be easily proved deadlock free by simply using modes. For this reason we believe that our method based on transformations, possibly combined with the techniques which use mode and type information as in [1], provides a simple and powerful tool.

R1	$\langle \mathsf{D.tell(c)}, \mathsf{d}\rangle \rightarrow \langle \mathsf{D.Stop}, \mathsf{c} \wedge \mathsf{d}\rangle$	
R2	$\langle \mathsf{D}.\sum_{i=1}^{n} \mathsf{ask}(\mathsf{c}_i) \rightarrow \mathsf{A}_i, \mathsf{d}\rangle \rightarrow \langle \mathsf{D.A}_j, \mathsf{d}\rangle$	$\mathsf{j} \in [1,\mathsf{n}]$ *and* $\mathcal{D} \models \mathsf{d} \rightarrow \mathsf{c}_i$
R3	$\dfrac{\langle \mathsf{D.A}, \mathsf{c}\rangle \rightarrow \langle \mathsf{D.A}', \mathsf{c}'\rangle}{\begin{array}{c}\langle D.A \parallel B, c\rangle \rightarrow \langle D.A' \parallel B, c'\rangle \\ \langle D.B \parallel A, c\rangle \rightarrow \langle D.B \parallel A', c'\rangle\end{array}}$	
R4	$\langle \mathsf{D.p}(\tilde{\mathsf{X}}), \mathsf{c}\rangle \rightarrow \langle \mathsf{D.A}, \mathsf{c}\rangle$	$\mathsf{p}(\tilde{\mathsf{X}}) \leftarrow \mathsf{A} \in \mathsf{defn}_\mathsf{D}(\mathsf{p}(\tilde{\mathsf{X}}))$

Table 1: The (standard) transition system.

report in the unfolded clause the delay declaration of the unfolded atom, and this enhances the flexibility of the transformation system.

In this section we'll show how our transformation system can be extended to the ccp language. Due to space limitation we'll restrict ourselves to the essential definitions and one example.

Both CLP and ccp frameworks are defined parametrically wrt to some notion of constraint system. Usually a more abstract view is followed in ccp by formalizing the notion of constraint system [28] along the guidelines of Scott's treatment of information systems. Here, for the sake of uniformity, we will assume that also in ccp constraints are first order formulae as previously defined. The basic idea underlying ccp languages is that computation progresses via monotonic accumulation of information in a global store. Information is produced by the concurrent and asynchronous activity of several agents which can add a constraint c to the store by performing the basic action $\mathsf{tell(c)}$. Dually, agents can also check whether a constraint c is entailed by the store by using an $\mathsf{ask(c)}$ action, thus allowing synchronization among different agents. A notion of locality is obtained in ccp by introducing the agent $\exists \mathsf{xA}$ which behaves like A, with x considered *local* to A. We do not use such an explicit operator: analogously to the standard CLP setting, locality is introduced implicitly by assuming that if a process is defined by $\mathsf{p}(\tilde{\mathsf{x}}) \leftarrow \mathsf{A}$ and a variable y in A does not appear in $\tilde{\mathsf{x}}$, then y has to be considered local to A. The $\parallel$ operator allows one to express parallel composition of two agents and it is usually described in terms of interleaving. Finally non-determinism arises by introducing an (global) choice operator $\sum_{i=1}^{n} \mathsf{ask}(\mathsf{c}_i) \rightarrow \mathsf{A}_i$: the agent $\sum_{i=1}^{n} \mathsf{ask}(\mathsf{c}_i) \rightarrow \mathsf{A}_i$ nondeterministically selects one $\mathsf{ask}(\mathsf{c}_i)$ which is enabled in the current store, and then behaves like A_i. Thus, the syntax of ccp *declarations* and *agents* is given by the following grammar:

$$\begin{array}{lll}
\textit{Declarations} & \mathsf{D} ::= & \epsilon \mid \mathsf{p}(\tilde{\mathsf{X}}) \leftarrow \mathsf{A} \mid \mathsf{D}, \mathsf{D} \\
\textit{Agents} & \mathsf{A} ::= & \mathsf{stop} \mid \mathsf{tell(c)} \mid \sum_{i=1}^{n} \mathsf{ask}(\mathsf{c}_i) \rightarrow \mathsf{A}_i \mid \mathsf{A} \parallel \mathsf{A} \mid \mathsf{p}(\tilde{\mathsf{X}}) \\
\textit{Processes} & \mathsf{P} ::= & \mathsf{D.A}
\end{array}$$

where c is a constraint. The operational model of ccp is described by a transition system $\mathsf{T} = (\mathsf{Conf}, \rightarrow)$ where configurations (in) Conf are pairs consisting of a process and a constraint (representing the common *store*), while the transition relation $\rightarrow \subseteq \mathsf{Conf} \times \mathsf{Conf}$ is described by the (least relation satisfying the) rules **R1-R4** of Table 1. Here and in the following we assume given a set D of declarations and we denote by $\mathsf{defn}_\mathsf{D}(\mathsf{H})$ the set of variant[9] of declarations in D which have H as head. We assume also the presence of a renaming mechanism that takes care of using fresh variables each time a clause is considered.[10].

The syntax of ccp agents allow us to define unfolding and the other transformations in a very simple way by using the notion of context. A context, denoted by $\mathsf{C}[\,]$, is simply an agent with a "hole". $\mathsf{C}[\mathsf{A}]$ denotes the agent obtained by replacing the hole in $\mathsf{C}[\,]$ for the agent A. Due to the presence of an explicit choice operator, we can assume without loss of generality that each predicate symbol is defined by exactly one declaration.

[9] A variant of a clause cl is obtained by replacing the tuple $\tilde{\mathsf{X}}$ of all the variables appearing in cl for another tuple $\tilde{\mathsf{y}}$.

[10] For the sake of simplicity we do not describe this renaming mechanism in the transition system. The interested reader can find in [28, 27] and various formal approaches to this problem.

Definition 6.1 (Unfolding) Consider a ccp process $\mathsf{D.C[p(\tilde{X})]}$ and assume that $\mathsf{p(\tilde{X}) \leftarrow B \in defn_D(p(\tilde{X}))}$[11]. Then unfolding $\mathsf{p(\tilde{X})}$ in $\mathsf{C[p(\tilde{X})]}$ consists simply in replacing $\mathsf{C[p(\tilde{X})]}$ by $\mathsf{C[B]}$. □

The notions of initial program and transformation sequence are defined as in the previous Section. Here however we do not impose the condition (D1), i.e. on the atoms defined in $\mathsf{P_{new}}$ we make only the assumption (I2). So we can define folding as follows.

Definition 6.2 Let $\mathsf{P_0}, \ldots, \mathsf{P_i}$, $i \geq 0$, be a transformation sequence. Let also $\mathsf{cl : p(\tilde{X}) \leftarrow C[A]}$ be a declaration in $\mathsf{P_i}$, and $\mathsf{cl_f : q(\tilde{Y}) \leftarrow A}$ be a (renamed version of a)[12] declaration in $\mathsf{P_{new}}$. Then folding A in cl consists of replacing cl by $\mathsf{cl' : p(\tilde{X}) \leftarrow C[q(\tilde{Y})]}$ provided that the conditions (**F1′**), and (**F3**) are satisfied, where (**F1′**) is the following modification of (**F1**):

(F1') *If we unfold* $\mathsf{q(\tilde{Y})}$ *in* $\mathsf{cl'}$ *using* $\mathsf{cl_f}$ *as unfolding clause, then we obtain cl back.* □

Analogously to the case of the if − the − else, in some cases we need to distribute a procedure call over an ask action. We define such an operation for ccp as follows.

Definition 6.3 Consider a set of declarations D containing the declaration

$$\mathsf{cl :\ p(\tilde{X}) \leftarrow C[q(\tilde{Y}) \parallel \textstyle\sum_i ask(c_i) \rightarrow A_i]}$$

and assume that, for any input constraint $\mathsf{c(\tilde{X})}$, whose variables are contained in $\tilde{\mathsf{X}}$, the configuration $\langle \mathsf{C[q(\tilde{Y})], c(\tilde{X})} \rangle$ has only deadlocked derivations in D of the form $\langle \mathsf{C[q(\tilde{Y})], c(\tilde{X})} \rangle \rightarrow^* \langle \mathsf{B, d} \rangle \not\rightarrow$ with $\mathcal{D} \models \exists_{-\tilde{z}}\mathsf{c(\tilde{X})} \leftrightarrow \exists_{-\tilde{z}}\mathsf{d}$, where $\tilde{\mathsf{z}} = \mathsf{vars(C[q(\tilde{Y})], c(\tilde{X}))}$. In this case we can transform cl into the declaration $\mathsf{cl' :\ p(\tilde{X}) \leftarrow C[\sum_i ask(c_i) \rightarrow (q(\tilde{Y}) \parallel A_i)]}$. □

Finally, in some cases we need also an operation which allow us to simplify the ask's of a programs. Consider in fact an agent of the form $\mathsf{C[ask(c) \rightarrow A + ask(d) \rightarrow B]}$. If $\mathsf{c'}$ is the conjunction of all the constraints which appear both in ask and tell actions in the "path" leading in the context $\mathsf{C[\ ]}$ from the top-level to the agent $\mathsf{ask(c) \rightarrow A + ask(d) \rightarrow B}$ and $\mathcal{D} \models c' \rightarrow c$, then clearly we can simplify previous agent to $\mathsf{C[ask(true) \rightarrow A + ask(d) \rightarrow B]}$ (note that in general it is not correct to transform previous agent into $\mathsf{C[A + ask(d) \rightarrow B]}$). This operation, which can easily be defined by structural induction on the agents, is often used to "clean up" the ccp agents, analogously to previous $\simeq$ equivalence. We omit its formal definition for space reasons.

The correctness of the transformation for ccp is expressed by the following result which, analogously to the case of CLP, shows that both answer constraints and deadlocked derivations are preserved by the transformations. Here we denote by Stop any agent which contains only stop, || and+ constructs.

Theorem 6.4 (Correctness) Let $\mathsf{P_0}, \ldots, \mathsf{P_n}$ is a transformation sequence of ccp processes and let A be a generic agent. Then for any $\mathsf{P_i}$, $\mathsf{P_j} \in \mathsf{P_0}, \ldots, \mathsf{P_n}$ we have that, for nay constraint c:

(i) if there exists a derivation $\langle \mathsf{P_i.A, c} \rangle \rightarrow^* \langle \mathsf{D.Stop, d} \rangle \not\rightarrow$ then there exists a derivation $\langle \mathsf{P_j.A, c} \rangle \rightarrow^* \langle \mathsf{D.Stop, d'} \rangle \not\rightarrow$ and $\mathcal{D} \models \exists_{\mathsf{vars(A,c)}}\mathsf{d} \leftrightarrow \exists_{\mathsf{vars(A,c)}}\mathsf{d'}$

(ii) if there exists a deadlocked derivation $\langle \mathsf{P_i.A, c} \rangle \rightarrow^* \langle \mathsf{D.B, d} \rangle \not\rightarrow$ (with $\mathsf{B} \not\equiv \mathsf{Stop}$) then there exists a deadlocked derivation $\langle \mathsf{P_j.A, c} \rangle \rightarrow^* \langle \mathsf{D.B', d'} \rangle \not\rightarrow$ (with $\mathsf{B'} \not\equiv \mathsf{Stop}$). □

We conclude with an example in order to illustrate the application of our methodology based on Unfold/Fold. We consider a stream protocol problem where two input streams are merged into an output stream. An input stream consists of lines of messages, and each line has to be passed to the output stream without interruption. Input and output streams are dynamically constructed by a reader and a monitor process, respectively. A reader communicates with the monitor by means of a buffer of length one, and is synchronized in such a way that it can read a new message only when the buffer is empty (i.e., when the previous message has been processed by the monitor). On the other hand, the monitor can access a buffer only when it is not empty (i.e., when the corresponding reader has put a message into its buffer).

[11] As before, it is assumed that $\mathsf{p(\tilde{X}) \leftarrow B}$ uses fresh variables.

[12] As usual, it is assumed here that $\mathsf{q(\tilde{Y}) \leftarrow A}$ is suitably renamed so to avoid variables names clashes.

The following program implements this stream protocol. The constants eol and eof denote the *end of line* and the *end of file* characters, respectively. Moreover, the other constants left, right and idle describe the state of the monitor, i.e., if it is processing a message from the left stream, from right stream, or if it is in an idle situation, respectively.

```
streamer(Os) ← reader(left,Ls), reader(right,Rs), monitor(Ls,Rs,idle,Os).

%%% reader(Channel,Xs) waits until Xs is instantiated

reader(Channel,Xs) ←
    ask(∃X,Xs' Xs=[X|Xs']) → (read(Channel,X), reader(Channel,Xs'))      % if Xs=[X|Xs] then reads a token
  + ask(Xs=[]) → true.                                                  % if Xs=[] then stops

% read(Channel,msg) is a primitive defined in order to read from the stream Channel

%%% monitor(Ls,Rs,State,Os) takes care of merging Ls and Rs

monitor([L|Ls],[R|RS],idle,Os) ←                                          % waiting for an input
      ask(ground(L)) → monitor([L|Ls],[R|RS],left,Os)
    + ask(ground(R)) → monitor([L|Ls],[R|RS],right,Os)

monitor([L|Ls],RS,left,[L|Os]) ←                                          % processing the left stream
  ask(ground(L)) →
      ( ask(L=eof) → (Ls=[], onestream(Rs,Os))                            % the left stream finished
      + ask(L=eol) → monitor(Ls,RS,idle,Os)                               % the left line is finished
      + ask(L=/=eol AND L=/=eof) → monitor(Ls,RS,left,Os))

% plus an analogous clause for processing the right stream

%%% onestream(Xs,Os) takes care of handling the single stream Xs

onestream([X|Xs],[X|Os]) ←
  ask(ground(X)) →
      ( ask(X = eof) → Os=[]
      + ask(X =/= eof) → onestream(Xs,Os))
```

We now going to improve the efficiency of the computation of the query streamer(Ys). We do this by transforming STREAMER via following steps.

1. We introduce the new predicate:

```
handle_two(L,R,State,Os) ← reader(left,Ls), reader(right,Rs), monitor([L|Ls],[R|RS],State,Os).
```

2. We unfold the monitor atom in the new clause. We obtain three definitions

```
handle_two(L,R,idle,Os) ← reader(left,Ls),
  reader(right, Rs),
  ( ask(ground(L)) → monitor([L|Ls],[R|Rs],left,Os)
  + ask(ground(R)) → monitor([L|Ls],[R|Rs],right,Os))

handle_two(L,R,left,[L|Os]) ← reader(left,Ls),
  reader(right, Rs),
  ask(ground(L)) →
      ( ask(L=eof) → (Ls = [], onestream([R|Rs],Os))
      + ask(L=eol) → monitor(Ls,[R|Rs],idle,Os)
      + ask(L=/=eol AND L=/=eof) → monitor(Ls,[R|Rs],left,Os))

%plus a third clause, symmetric to the second one.
```

3. Now, we can (a) apply the distributive property to the readers. (b) apply the *splitting* operation to the last two occurrences of monitor in the last definition and (c) evaluate the expression Ls=[]. We obtain the following definition.

```
handle_two(L,R,idle,Os) ←
    ask(ground(L)) → (reader(left,Ls), reader(right,Rs), monitor([L|Ls],[R|Rs],left,Os))
  + ask(ground(R)) → (reader(left,Ls), reader(right,Rs), monitor([L|Ls],[R|Rs],right,Os))
```

```
handle_two(L,R,left,[L|Os]) ←
    ask(ground(L)) →
        ( ask(L=eof) → (onestream([R|Rs],Os) reader(left,[]), reader(right,Rs))
        + ask(L=eol) → (monitor([L'|Ls'],[R|Rs],idle,Os), reader(left,[L'|Ls']), reader(right,Rs))
        + ask(L=/=eol AND L=/=eof) → (monitor([L'|Ls'],[R|Rs],left,Os), reader(left,[L'|Ls']), reader(right,Rs)))
```

% plus a third clause, symmetric to the second one

4. Now we can (a) fold handle_two in the first of these definitions and (b) unfold the various instances of reader(left,...). We obtain:

```
handle_two(L,R,idle,Outs) ←
      ask(ground(L)) → handle_two(L,R,left,Outs)
    + ask(ground(R)) → handle_two(L,R,right,Outs)

handle_two(L,R,left,[L|Outs]) ←
    ask(ground(L)) →
        ( ask(L=eof) → (onestream([R|Rs],Outs),
                ( ask(∃X,Xs []=[X|Xs']) → read(left,X),reader(left,Xs'),
                + ask([]=[]) → true ),
                reader(right,Rs))
        + ask(L=eol) → (monitor([L'|Ls'],[R|Rs],idle,Outs),
                (ask(∃X,Xs [L'|Ls']=[X|Xs']) → read(left,X),reader(left,Xs'),
                + ask([L'|Ls']=[]) → true.),
                reader(right,Rs))
        + ask(L=/=eol AND L=/=eof) → (monitor([L'|Ls'],[R|Rs],left,Outs) ,
                (ask(∃X,Xs [L'|Ls']=[X|Xs']) → read(left,X),reader(left,Xs'),
                + ask([L'|Ls']=[]) → true.),
                reader(right,Rs)))
```

5. We can now (a) evaluate the newly introduced ask constructs, and (b) fold twice handle_two in the last definition

```
handle_two(L,R,idle,Outs) ←
    ask(ground(L)) → handle_two(L,R,left,Outs)
    + ask(ground(R)) → handle_two(L,R,right,Outs)

handle_two(L,R,left,[L|Outs]) ←
    ask(ground(L)) →
        ( ask(L=eof) → (onestream([R|Rs],Outs), reader(right,Rs))          %(**)
        + ask(L=eol) → (read(left,L'), handle_two(L',R,idle,Outs))
        + ask(L=/=eol AND L=/=eof) → (read(left,L'),handle_two(L',R,left,Outs)))
```

% plus a third clause, symmetric to the second one

6. Now, the above program is *almost completely* independent from the definition of reader. The only residual only non-well-moded residual is the presence of reader(right,Rs). This atom can be further eliminated using an unfold-fold transformation similar to (but simpler than) the previous one. We start with the new predicate:

```
handle_one(X, Channel, Outs) ← reader(Channel,Xs), onestream([X|Xs],Outs).
```

7. We unfold onestream([X|Xs],Outs), and obtain

```
handle_one(X, Channel, [X|Outs]) ← reader(Channel,Xs),
    ask(ground(X)) →
        ( ask(X=eof) → (Xs=[], Outs=[])
        + ask(X=/=eof) → onestream(Xs,Outs))
```

8. We apply the distributive property to reader(Channel,Xs),

```
handle_one(X, Channel, [X|Outs]) ←
    ask(ground(X)) →
        ( ask(X=eof) → (reader(Channel,Xs), Xs=[], Outs=[])
        + ask(X=/=eof) → (reader(Channel,Xs), onestream(Xs,Outs)))
```

9. We now (a) evaluate Xs=[], (b) split onestream(Xs,Outs), (c) unfold the various instance of reader and (d) simplify the newly introduced ask construct, much like it was done when optimizing handle_two, the result is

```
handle_one(X, Channel, [X|Outs]) ←
    ask(ground(X)) →
        ( ask(X=eof) → Outs=[]
        + ask(X=/=eof) → (read(Channel,X'), reader(Channel,Xs'), onestream([X'|Xs],Outs)))
```

10. We can now fold handle_one

```
handle_one(X, Channel, [X|Outs]) ←
    ask(ground(X)) →
        ( ask(X=eof) → Outs=[]
        + ask(X=/=eof) → (read(Channel,X'), handle_one(X', Channel, Outs)))
```

11. By folding handle_one into the second definition of handle_two (the part marked with (**)), we obtain

```
handle_two(L,R,left,[L|Outs]) ←
    ask(ground(L)) →
        (ask(L=eof) → handle_one(R,right,Outs)
        + ask(L=eol) → (read(left,L'), handle_two(L',R,idle,Outs))
        + ask(L=/=eol AND L=/=eof) → (read(left,L'),handle_two(L',R,left,Outs)))
```

12. We now want to let streamer benefice of the improvements we have obtained via this transformation. We then have to transform its definition , let us start by splitting the atom monitor(Ls,Rs,idle,Os. We obtain the following program

```
streamer(Outs) ← reader(left,[L|Ls]), reader(right,[R|Rs]), monitor([L|Ls],[R|Rs],idle,Outs).
```

13. Now we can unfold the two reader atoms, and immediately eliminate the redundant asks.

```
streamer(Outs) ← read(left,L), reader(left,Ls), read(right,R), reader(right,Rs), monitor([L|Ls],[R|Rs],idle,Outs).
```

14. We can now fold handle_two in it, obtaining.

```
streamer(Outs) ← read(left,L), read(right,R), handle_two(L,R,Outs).
```

Now, the definition of streamer is more efficient than the original one. In particular it benefits from a straightforward left-to-right dataflow and streamer is now independent from the definition of reader which is the one that most negatively influences the performances of the program, having to suspend and awaken itself virtually at each input token. Moreover, it is worth noticing then by suitably extending the notion of modes to ccp languages, one could also easily obtain a deadlock freedom result for the transformed program above, while this would be more difficult for the original program.

7 Conclusions

In this paper we have presented a fold/unfold transformation system for CLP with dynamic scheduling and for ccp which can be used both for program optimization and for proving absence of deadlock.

Since CLP with dynamic scheduling can be embedded into a sublanguage of ccp, one could think that we could have better considered the latter paradigm alone. However, this is not the case for the following two reasons: first, that CLP with dynamic scheduling would in any case require a thorough restatement of the operations and their applicability conditions (leaving it as a “simple special case” would simply not work). Secondly, that CLP has a far greater practical impact, as it is much more employed as a programming language than ccp is. Therefore, from a practical perspective, it is worthwhile to focus primarily on this paradigm.

The results contained in this paper should provide the basis to develop a more general transformation system for CLP with dynamic scheduling and ccp. In particular, we are now investigating how to extend the results in [11] to these languages in order to obtain also a "replacement" transformation.

Related Papers. To the best of our knowledge transformations for these languages were not been studied yet. More generally, the only other existing work which apply unfold/fold techniques in a concurrent setting is [14], where De Francesco and Santone investigate transformations of CCS programs. The results in [14] are rather different from ours, since they define an applicability condition for the folding operation based on the notion of "guardness" which does not depend on the history of the transformation. Moreover the correctness result in [14] is given with respect to a bisimulation semantics.

References

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

[2] N. Bensaou and I. Guessarian. Transforming Constraint Logic Programs. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation*, 1994.

[3] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[4] P. Chambre, P. Deransart, and J. Małuszyński. A proof method for safety properties of clausal concurrent constraint programs. In F. de Boer and M. Gabbrielli, editors, *Proc. JICSLP'96 Post-Conference Workshop on Verification and Analysis of Logic Programs*, 1996. Technical Report TR-96-31, Dipartimento di Informatica di Pisa.

[5] K.L. Clark and S. Sickel. Predicate logic: a calculus for deriving programs. In *Proceedings of IJCAI'77*, pages 419–120, 1977.

[6] M. Codish, M. Falaschi, and K. Marriott. Suspension Analyses for Concurrent Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.

[7] C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation for Concurrent Logic Languages. In S. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 215–232. MIT Press, 1990.

[8] F.S. de Boer, M. Gabbrielli, and C. Palamidessi. Proving correctness of constraint logic programs with dynamic scheduling. In *Static Analysis, Third International Static Analysis Symposium, (SAS'96)*, LNCS. Springer-Verlag, Berlin, 1996.

[9] M. Garcia de la Banda, K. Marriott, and P. Stuckey. Efficient analysis of logic programs with dynamic scheduling. In J. Lloyd, editor, *Proc. Twelfth International Logic Programming Symposium*, pages 417–431. MIT Press, 1995.

[10] S. Debray, D. Gudemann, and P. Bigot. Detection and optimization of suspension-free logic programs. In M. Bruynooghe, editor, *Proc. Eleventh International Logic Programming Symposium*, pages 487–504. MIT Press, 1994.

[11] S. Etalle and M. Gabbrielli. The replacement operation for CLP modules. In N. Jones, editor, *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '95)*, pages 168–177. ACM press, 1995.

[12] S. Etalle and M. Gabbrielli. Layered modes. In F. de Boer and M. Gabbrielli, editors, *Proc. JICSLP'96 Post-Conference Workshop on Verification and Analysis of Logic Programs*, 1996. Tehcnical Report TR-96-31, Dipartimento di Informatica di Pisa.

[13] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166(1):101–146, 1996.

[14] N. De Francesco and A. Santone. Unfold/fold transformation of concurrent processes. In H. Kuchen and S.Doaitse Swierstra, editors, *Proc. 8th Int'l Symp. on Programming Languages: Implementations, Logics and Programs*, volume 1140, pages 167–181. Springer-Verlag, 1996.

[15] Nevin Heintze, Spiro Michaylov, and Peter J. Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. In Jean-Louis Lassez, editor, *ICLP'87: Proceedings 4th International Conference on Logic Programming*, pages 675–703, Melbourne, Victoria, Australia, May 1987. MIT Press. Also in Journal of Automated Reasoning vol. 9, pages 231–260, October 1992.

[16] C.J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, April 1981.

[17] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[18] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[19] H. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, pages 255–267. ACM, 1982.

[20] A. Lakhotia and L. Sterling. Composing recursive logic programs with clausal join. *New Generation Computing*, 6(2,3):211–225, 1988.

[21] C. Lassez, K. McAloon, and R. Yap. Constraint Logic Programming and Option Trading. *IEEE Expert*, 2(3), 1987.

[22] M.J. Maher. A transformation system for deductive databases with perfect model semantics. *Theoretical Computer Science*, 110(2):377–403, March 1993.

[23] E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. Lloyd, editor, *Proc. Twelfth International Logic Programming Symposium.* MIT Press, 1995.

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

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

[26] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[27] V.A. Saraswat, M. Rinard, , and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages.* ACM Press, 1991.

[28] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, New York, 1990.

[29] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.

[30] M.H. van Emden and G.J. de Lucena. Predicate logic as a language for parallel programming. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, London, 1982. Academic Press.

[31] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

A CLP with dynamic scheduling

In the following we give a more precise formalization of the computational model of CLP with dynamic scheduling. To this aim we first define the assertion language for delay conditions as follows. We assume a given set Δ of delay predicates (e.g. $\mathsf{ground}(\mathsf{X})$) a given set $\mathcal{C}$ of constraints and a given structure $\mathcal{D}$.

Definition A.1 The language of delay conditions is defined by the following grammar, where $\mathsf{c} \in \mathcal{C}$ and $\delta \in \Delta$:

$$\phi ::= \mathsf{c} \mid \delta \mid \phi \wedge \phi \mid \phi \vee \phi$$

Note that we do not allow negation, since we require that delay conditions are interpreted as sets of constraints closed under entailment.

An interpretation for delay conditions is given by a function which, for each condition, returns the set of constraints which satisfy it. A constraint c viewed as a delay condition is satisfied by all the constraints d that entail c. For delay predicates, we assume given an interpretation $\mathsf{I} : \Delta \rightarrow \wp(\mathcal{C})$ which defines their meaning. Thus a user defined predicate δ is satisfied by a constraint c iff $\mathsf{c} \in \mathsf{I}(\delta)$. For example, the interpretation of $\mathsf{ground}(\mathsf{X})$ is the set of all the constraints which force X to assume a unique value. The logical operations of conjunction and disjunction are interpreted in the classical way by their corresponding set theoretic operations of intersection and union. Formally, the interpretation of delay conditions is then defined as follows.

Definition A.2 An *interpretation* $[\![\cdot]\!]$ is a function from delay conditions into non-empty sets of constraints such that :

$$\begin{array}{lcllcl} [\![\mathsf{c}]\!] & = & \{d \mid \mathcal{D} \models d \rightarrow c\} & [\![\delta]\!] & = & I(\delta) \\ [\![\phi \wedge \psi]\!] & = & [\![\phi]\!] \cap [\![\psi]\!]. & [\![\phi \vee \psi]\!] & = & [\![\phi]\!] \cup [\![\psi]\!]. \end{array}$$

As previously mentioned, we impose three assumptions on delay declarations. The first one describes a property which is satisfied by the majority of the (C)LP systems with delay declarations: if an atom is not delayed then adding more information will never cause it to delay. This amounts to say that the interpretation of any delay predicate δ is closed under entailment, i.e.,

(i) for each $\delta \in \Delta$, if $\mathsf{c} \in \mathsf{I}(\delta)$ and $\mathcal{D} \models \mathsf{d} \rightarrow \mathsf{c}$ then $\mathsf{d} \in \mathsf{I}(\delta)$.

It is straightforward to check that this requirement ensures that the interpretation $[\![\phi]\!]$ of a delay condition ϕ is also closed under entailment. Generic negated conditions are not allowed because they could violate previous requirement For example, this is case for the assertion $\neg\mathsf{ground}(\mathsf{X})$, which is satisfied by all the constraints which do not force X to assume a unique value.

Furthermore, delay conditions are usually assumed to be consistent wrt renaming, so we require that

(ii) for each delay condition ϕ whose free variables are $\tilde{\mathsf{X}}$, if $\mathsf{c} \in [\![\phi]\!]$ then $\mathsf{c} \wedge \tilde{\mathsf{Y}} = \tilde{\mathsf{X}} \in [\![\phi[\tilde{\mathsf{X}}/\tilde{\mathsf{Y}}]]\!]$.

where $\tilde{\mathsf{Y}}$ contains new fresh variables ($\phi[\tilde{\mathsf{X}}/\tilde{\mathsf{Y}}]$ denotes the formula obtained from ϕ by replacing the variables $\tilde{\mathsf{X}}$ for $\tilde{\mathsf{Y}}$).

Finally, we assume that for each predicate symbol exactly one delay declaration is given. This is not restrictive, since multiple declarations can be obtained by using logical connectives in the syntax of conditions.

Now, we can specify more precisely a derivation step for CLP with dynamic scheduling as follows.

Recall that we say that an atom $\mathsf{p}(\tilde{\mathsf{t}})$ in the context of a constraint c satisfies its delay declaration in the program P if, assuming that $\mathsf{DELAY}\ \mathsf{p}(\tilde{\mathsf{X}})\ \mathsf{UNTIL}\ \phi$ is the delay declaration for p in P, we have that $(\mathsf{c} \wedge \tilde{\mathsf{t}} = \tilde{\mathsf{Y}}) \in [\![\phi[\tilde{\mathsf{X}}/\tilde{\mathsf{Y}}]]\!]$ holds, where $\tilde{\mathsf{Y}}$ are new distinct variables which do not appear in c and in $\tilde{\mathsf{t}}$.

Then, a goal $\mathsf{c}, \mathsf{p}(\tilde{\mathsf{t}}_1), \ldots, \mathsf{p}(\tilde{\mathsf{t}}_n)$ in the program P results in the goal

$$\mathsf{c} \wedge (\tilde{\mathsf{t}}_i = \tilde{\mathsf{s}}_i) \wedge \mathsf{d}, \mathsf{p}_1(\tilde{\mathsf{t}}_1), \ldots, \mathsf{p}_{i-1}(\tilde{\mathsf{t}}_{i-1}), \tilde{\mathsf{B}}, \mathsf{p}_{i+1}(\tilde{\mathsf{t}}_{i+1}), \ldots, \mathsf{p}(\tilde{\mathsf{t}}_n)$$

provided that

1. the atom $\mathsf{p_i}(\tilde{\mathsf{t}}_\mathsf{i})$ in the context of the constraint c satisfy its delay declaration in P,
2. there exists a (renamed version of a) clause $\mathsf{p_i}(\tilde{\mathsf{s}}_\mathsf{i}) \leftarrow \mathsf{d}, \tilde{\mathsf{B}}$ in P which does not share variables with the given goal,
3. $\mathcal{D} \models \exists\ (\mathsf{c} \wedge (\tilde{\mathsf{t}}_\mathsf{i} = \tilde{\mathsf{s}}_\mathsf{i}) \wedge \mathsf{d})$.

A derivation for the query Q in the program P is therefore a finite or infinite sequence of goals, starting in Q, such that every next goal is obtained from the previous one by means of a derivation step defined as above.