Proving Correctness of Constraint Logic Programs with Dynamic Scheduling*

F.S. de Boer
Universiteit Utrecht

M. Gabbrielli
Università di Pisa

C. Palamidessi
Università di Genova

Abstract

A general framework based on Hoare logic is introduced for specifying and reasoning about Constraint Logic Programs with dynamic scheduling. The framework consists of a mixed formalism of programs in a ccp-like language, on the one hand, and correctness properties of the Hoare logic, on the other hand. In this formalism delay conditions are viewed as a specific class of correctness properties. In the construction of the proof-system we follow the lines of a previous proof-system we developed for concurrent constraint programming, which was sound, but not complete (for ccp). Due to the different kind of choice used in CLP, on the contrary, the system considered here is both sound and complete.

Keywords: Dynamic scheduling, concurrent constraint programming, partial correctness, strongest postcondition.

Corresponding author:
M. Gabbrielli
Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
Tel: +39-50-887264
Fax: +39-50-887226
`gabbri@di.unipi.it`

*This work was supported by the HCM Project "EXPRESS".

1 Introduction

Most of the existing (constraint) logic programming systems use a flexible selection rule which allows to dynamically delay (or suspend) the selection of atoms until the arguments are sufficiently instantiated. This dynamic scheduling is obtained by adding so-called *delay declarations* to program clauses. Delay declarations (introduced by Naish in [19]) allow to run the calls more efficiently, to prevent run-time errors and to enforce termination. In Constraint Logic Programming (CLP for short) 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, they provide the programmer with a better control over computations and, similarly to guards in concurrent logic languages, allow one to express some degree of synchronization among the different processes (i.e. atoms) in a program. The practical relevance of delay declarations is demonstrated by the fact that they are implemented in such languages as Sicstus Prolog [6], Eclipse [26], Gödel [10] and in many CLP systems [13].

From the semantic viewpoint, the fact that an atom can be delayed until certain conditions are satisfied introduces some complications similar to those arising in concurrent languages. The operational semantics of logic programs with dynamic scheduling has been discussed by Naish [20] while their denotational semantics has been investigated by Marriott et al. [18, 17]. Recently also the issue of correctness has received some attention. Apt and Luitjes [2] have investigated deadlock freedom, occur check freedom, termination and absence of builtins' errors in presence of delay declarations. Termination has also been studied by Marchiori and Teusink [16] and by Lüttringhaus [15]. In particular, the latter paper describes an algorithm which, given a program, generates automatically delay declarations which ensure termination. However, as far as we know, the issue of partial correctness, namely the fact that the computed constraints correspond with the intended meaning of the program, has not been considered yet.

In particular, while the declarative meaning of (constraints) logic programs is well understood in terms of first-order logic theories, the logical meaning of delay declarations has not been addressed yet. In fact, if we have a program `P` defining a predicate `p(X)`, then (the constraints computed by the query) `p(X)` will satisfy certain properties which can be derived logically from the theory of `P`. Thus, considering for instance logic programs, one can reason about program correctness by simply using the least Herbrand model [1]. If we now add the delay declaration

```
DELAY p(X) UNTIL Condition(X)
```

than what can we say about the properties satisfied by `p(x)` in the new program? In this paper we show that, if `p(x)` in `P` satisfies a property ϕ, then under the delay declaration it will satisfy the property `Condition(x)` $\rightarrow \phi$. In other words, the logical meaning of delay declarations can be understood as logical implication.

More generally, in this paper we introduce a general framework based on Hoare logic for specifying and reasoning about the partial correctness of constraint logic programs with delay declarations. The framework consists of a mixed formalism of programs on the one hand, and correctness properties of the Hoare logic, on the other hand. In order to prove the soundness and completeness of the system it is convenient to deal with a more "algebraically structured" language. To this end we translate CLP into a sort of concurrent constraint programming language (ccp for short), which we call δ-ccp, and which is basically an extension of ccp with local choice [14]. The extension consists in allowing the presence of delay conditions in the "ask" construct. Such an extension enhances the expressive power, but preserves the nice properties of local ccp, in particular the denotational semantics can still be defined in a simple way by using closure operators, following [14].

We use the denotational semantics as a guideline for the development of a compositional proof-system, where "compositional" means that the correctness (or specification) of a composite program can be derived directly from the correctness (or specifications) of its immediate constituents. This abstraction from the internal structure of the components of a program is crucial in mastering the complexity of large programs and supports the methodology of top-down design.

The proof-system is parametric wrt a theory of properties based on the elements of the constraint system, a class of user-defined properties (including the delay conditions), and the connectives of the propositional calculus. Differently from the system developed in [4], here we can use classical logic to reason about properties. This is mainly due to the fact that we deal with the local variables of CLP just by variable-renaming, hence we don't need to impose a cylindrification operator on properties. Thus, a specific theory is constituted by a set of proper axioms, which represent the entailment relations among the constraints and the user-defined properties, plus the laws of predicate calculus.

We prove that our proof-system is sound and complete by using the denotational semantics. Thanks to its simplicity, this proof turns out to be (relatively) easy. It is worth noticing that, on the contrary, the proof-system defined in [4] for ccp is sound but incomplete. This is because ccp has global choice, whereas the kind of nonderminism of CLP can be modeled by local choice, even if we add the delay declarations.

The plan of the paper is as follows: In the following section we discuss some preliminaries and introduce formally constraint logic programs with delay declarations. Then in Section 3 we introduce the language δ-ccp and we discuss a translation of constraint logic programs with delay declarations into δ-ccp. In Section 4 the notion of (partial) correctness is formally introduced and in Section 5 the proof-system is presented. Section 6 concludes.

2 Preliminaries

Both CLP and ccp frameworks are defined parametrically wrt to some notion of constraint system. In the standard CLP setting constraints are essentially first order formulas built up by using predicate symbols which have a predefined interpretation (e.g. equations on the term algebra or inequalities over the reals). A more abstract view is followed in [22] by formalizing the notion of constraint system along the guidelines of Scott's treatment of information systems [24]. Here we follow this second approach and assume that both CLP and ccp use the abstract notion of constraint system defined below. In this way we are more general since the standard CLP view can be obtained as a particular instance of the abstract formalization. In the following we introduce the basic definition of cylindric constraint system. We refer to [22, 23, 3] for more details concerning its construction and properties.

Definition 2.1 A constraint system is a complete algebraic lattice $\langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false} \rangle$ where $\sqcup$ is the lub operation, and *true*, *false* are the least and the greatest elements of $\mathcal{C}$, respectively.

Following the standard terminology and notation, instead of $\leq$ we will refer to its inverse relation, denoted by $\vdash$ and called *entailment*. Formally, $\forall c, d \in C.\ \ c \vdash d \ \Leftrightarrow\ d \leq c$. When considering first order formulas as constraints (e.g. in the examples) we will use $\wedge$ instead of $\sqcup$, since in this case the least upper bound corresponds to logical conjunction.

In order to treat the hiding operator of the language a general notion of existential quantifier is introduced which is formalized in terms of cylindric algebras [11]. Moreover, in order to model renaming of variables, *diagonal elements* [11] are added to the primitive constraints. This leads to the concept of *cylindric constraint system*. In the following, we assume given a (denumerable) set of variables *Var* with typical elements $x, y, z, \ldots$.

Definition 2.2 Let $\langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false} \rangle$ be a constraint system. Assume that for each $x \in \mathit{Var}$ a function $\exists_x : \mathcal{C} \to \mathcal{C}$ is defined such that for any $c, d \in \mathcal{C}$:

$$
\begin{array}{ll}
\text{(i) } c \vdash \exists_x(c), & \text{(ii) if } \ c \vdash d \ \text{ then } \ \exists_x(c) \vdash \exists_x(d), \\
\text{(iii) } \exists_x(c \sqcup \exists_x(d)) = \exists_x(c) \sqcup \exists_x(d), & \text{(iv) } \exists_x(\exists_y(c)) = \exists_y(\exists_x(c)).
\end{array}
$$

Moreover assume that for x, y ranging in *Var*, $\mathcal{C}$ contains the constraints d_{xy} (so-called diagonal elements) which satisfy the following axioms:

$$
\begin{array}{ll}
\text{(v) } \mathit{true} \vdash d_{xx}, & \text{(vi) if } z \neq x, y \text{ then } d_{xy} = \exists_z(d_{xz} \sqcup d_{zy}), \\
\text{(vii) if } x \neq y \text{ then } d_{xy} \sqcup \exists_x(c \sqcup d_{xy}) \vdash c. &
\end{array}
$$

Then $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}, \mathit{Var}, \exists_x, d_{xy} \rangle$ is a *cylindric constraint system*.

Note that if $\mathbf{C}$ models the equality theory, then the elements d_{xy} can be thought of as the formulas $x = y$. In the sequel we will identify a system $\mathbf{C}$ with its underlying set of constraints $\mathcal{C}$ and we will denote $\exists_x(c)$ by $\exists_x c$ with the convention that, in case of ambiguity, the scope of $\exists_x$ is limited to the first constraint subexpression (so, for instance, $\exists_x c \sqcup d$ stands for $\exists_x(c) \sqcup d$). We define the set $fv(c)$ of the free variables which appear in the constraint c as $fv(c) = \{x \mid \exists_x c \neq x\}$. Then, given a constraint c which contains finitely many free variables, we denote by $\exists_{-x} c$ the existential closure of c with the exception of the variable x which remains unquantified, i.e. $\exists_{-x} c$ is a shorthand for $\exists_{x_1} \ldots \exists_{x_n} c$ where $\{x_1, \ldots, x_n\} = fv(c) \setminus \{x\}$. Moreover we denote by $c[x/y]$ the constraint $\exists_y(c \sqcup d_{xy})$ which represents abstractly the constraint obtained from c by renaming the variable x for y. Previous notations are extended to tuples of distinct variables in the obvious way.

2.1 CLP programs with dynamic scheduling

As previously mentioned, many CLP languages use delay declarations to dynamically delay (or suspend) the selection of atoms until the arguments are sufficiently instantiated. In the following we briefly introduce the terminology and the basic notions for these languages. For CLP we refer to the original paper [12] by Jaffar and Lassez and to the survey [13] by Jaffar and Maher.

We let $vars(e)$ to denote the set of variables which appear in the syntactic expression e and we denote by $\overline{x}$ a tuple of distinct variables. An atom of the form $p(\overline{x})$ is called pure and a conjunction of pure atoms which do not share variables with each other will be often denoted by Q.

CLP queries and clauses have the form c, Q and $p(\overline{x}) \leftarrow c, Q$ respectively, where c is a constraint different from $false$ and $vars(Q) \cap \overline{x} = \emptyset$. We use this syntax since it simplifies the parameter passing mechanism. Note that this is not a restriction, since the presence of generic terms as arguments to predicate symbols can be expressed via the constraints.

A program, as usual, is a set a clauses. Given a program P and a pure atom A, we denote by $defn_P(A)$ the set of variant of clauses in P which have A as head[1].

A derivation step for a query c, Q in a program P consists in replacing an atom A in Q with the body of a clause $cl \in defn_P(A)$ provided that the constraint in cl is consistent with c and no variable clash occurs. We denote by $\leadsto_P$ the relation on queries corresponding to derivation steps in the program P. So, more precisely, given a query $c, p_1(\overline{x}_1), \ldots, p_n(\overline{x}_n)$ we have a derivation step $c, p_1(\overline{x}_1), \ldots, p_n(\overline{x}_n) \leadsto_P$ $c \sqcup d, p_1(\overline{x}_1), \ldots, p_{i-1}(\overline{x}_i), Q, p_{i+1}(\overline{x}_{i+1}), \ldots, p_n(\overline{x}_n)$ if there exists a clause $p_i(\overline{x}_i) \leftarrow d, Q$ in $defn_P(p_i(\overline{x}_i))$ such that $vars(d, Q) \cap vars(c, p_1(\overline{x}_1), \ldots, p_n(\overline{x}_n)) \subseteq \overline{x}_i$ and $c \sqcup d \neq false$.

We denote by $\leadsto_P^*$ the reflexive and transitive closure of $\leadsto_P$, i.e. the relation corresponding to derivations. We do not make any assumption on the selection rule which is used to select the atom to be rewritten, since this allows us to model also parallel executions.

Naish [19] first introduced delay declarations in logic languages by using `when` constructs in MU-Prolog. Here we follow loosely the syntax used in the language Gödel and consider declarations of the form

$$\texttt{DELAY p}(\overline{\texttt{X}})\ \texttt{UNTIL Condition}(\overline{\texttt{X}})$$

where $\texttt{Condition}(\overline{\texttt{X}})$ is a formula in some assertion language. The meaning of such a declaration is that the atom $\texttt{p}(\overline{\texttt{Y}})$ can be selected in a resolvent c, Q only if the constraint c which has been produced so far in the derivation satisfies the condition $\texttt{Condition}(\overline{\texttt{Y}})$. For example, considering such a numerical constraint system as the one used in CLP($\Re$), the declaration

$$\texttt{DELAY p(X) UNTIL X} > 0$$

means that `p(X)` can be selected in a resolvent c, Q only if c forces `X` to assume values greater than 0.

In our setting delay conditions can be formalized as follows. We assume given a set of conditions Δ, with typical elements $\delta(\overline{x})$, $\delta_1(\overline{x}) \ldots$, whose syntax we do not further specify since this is not relevant to our results. Then we assume that an interpretation for conditions is given in the form of a function

$$I : \Delta \to \wp(\mathcal{C})$$

which, for each condition, returns the set of constraints which satisfy it. In other words, we say that c satisfies $\delta(\overline{x})$ iff $c \in I(\delta(\overline{x}))$.

[1]A variant of a clause cl is obtained by replacing the tuple $\overline{x}$ of all the variables appearing in cl for another tuple $\overline{y}$ of (distinct) variables.

Most of the existing programming systems assume that if an atom is not delayed then adding more information will never cause it to delay, i.e. if c satisfies $\delta(\overline{x})$ and $d \vdash c$ then d satisfies $\delta(\overline{x})$. Thus, we also make the same assumption and require that, for each $\delta(\overline{x}) \in \Delta$, $I(\delta(\overline{x}))$ is an upward closed set, i.e. $I(\delta(\overline{x})) = \uparrow I(\delta(\overline{x}))$ where $\uparrow$ is the obvious extension to set of constraints of the operator defined on $\mathcal{C}$ by $\uparrow c = \{d \mid c \leq d\}$. Many natural delay conditions satisfy this monotonicity requirement. For example, given the usual Herbrand constraint system $\mathcal{H}$, the declaration

```
DELAY p(X) UNTIL ground(X)
```

can be used to delay `p(X)` until `X` has a ground value. The interpretation of the condition `ground(X)` is given by the upward closed set

$$I(\mathtt{ground(X)}) = \{c \in \mathcal{H} \mid \text{ for any pair of solutions } \theta \text{ and } \gamma \text{ of c, } X\theta \equiv X\gamma\}$$

i.e. the set of all those constraints which force `X` to have a unique value[2].

We assume that for each predicate symbol exactly one delay declaration is given (multiple declarations can be obtained by using logical connectives in the syntax of conditions). When the condition holds vacuously (i.e. when $I(\delta(\overline{x})) = \mathcal{C}$) we simply omit the declaration. Now, given a declaration `DELAY p (`$\overline{\mathtt{X}}$`) UNTIL` $\delta(\overline{\mathtt{X}})$ for the predicate symbol `p`, an atom `p(`$\overline{\mathtt{Y}}$`)` can be selected in the resolvent c, Q only if $c \in I(\delta(\overline{\mathtt{Y}}))$. In the following we will implicitly consider only CLP derivations in which the selection of atoms in resolvents respects the above rule.

The result of a finite (maximal) derivation for a query, usually called answer constraint, consists of the constraint obtained from the one accumulated in the derivation by disregarding the information on the local variables, i.e. by existentially quantifying the variables which do not appear in the query. Thus we will consider the following notion of observables. Given a program P and a query c, Q we define

$$\mathcal{O}_{clp}(P, Q, c) = \{\exists_{-vars(c,Q)} d \mid \text{ there exists a derivation } c, Q \rightsquigarrow_P^* d, Q' \not\rightsquigarrow\}.$$

Note that we consider here also results of deadlocked computations, i.e. of those computation which end up in a resolvent d, Q' where Q' is non-empty and cannot be further reduced because the delay conditions are not satisfied.

3 CLP with dynamic scheduling as a ccp-like language

In this section we show how constraint logic programs with dynamic scheduling can be translated into a ccp-like language in a quite natural way.

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 (tell) a constraint to the store. Dually, agents can also check (ask) whether a constraint is entailed by the store, thus allowing synchronization among different agents. Differently from the standard ccp setting, here we assume that delay conditions are arguments to ask actions. As we show later, this allows us to formulate in a natural way the delay mechanism by using the ask evaluation. Tell actions instead also in our case have standard constraints as arguments, since we use tell's evaluation to simulate the accumulation of constraints in CLP derivations.

A notion of locality is obtained in ccp by introducing the agent $\exists x A$ 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 $p(\overline{x}) \leftarrow A$ and a variable y in A does not appear in $\overline{x}$, then y has to be considered local to A. This will simplify the parameter passing mechanism and therefore our proof-system for partial correctness.

The $\|$ operator, both in ccp and in our case, allows one to express parallel composition of two agents and it is usually described in terms of interleaving. Finally (local) non-determinism arises by introducing an (associative and commutative) *choice operator* $+$: the agent $A + B$ nondeterministically selects either A or B and then behaves as the selected agent. Usually ccp languages adopt *global non-determinism*

[2] We assume here that the signature contains at least two constant symbols.

which is more expressive than the local one. In fact in the global case the external environment can affect the choice, since only those branches whose guards are enabled can be selected. On the other hand, as shown in [14], local ccp has a very simple denotational model based on closure operators. Since CLP with dynamic scheduling has only local non-determinism, we use here this restricted form of non-determinism. In this way we have a simple semantic basis for our calculus.

Thus we end up with the following syntax for δ-ccp.

Definition 3.1 Let $\mathcal{C}$ be a constraint system and let Δ be a set of delay conditions. The syntax of *declarations* and *agents* is given by the following grammar:

$$\begin{array}{lll} \text{Declarations} & D ::= & \epsilon \mid p(\overline{x}) : -A \mid D, D \\ \text{Agents} & A ::= & stop \mid tell(c) \mid ask(\delta(\overline{x})) \rightarrow A \mid A + A \mid A \parallel A \mid p(\overline{x}) \\ \text{Processes} & P ::= & D.A \end{array}$$

where c is a finite element in $\mathcal{C}$ and $\delta(\overline{x}) \in \Delta$. A δ-ccp *process* $D.A$ is supposed to be *closed*, i.e. we assume that each predicate symbol occurring in D or in A is defined in D. Furthermore, we assume that D contains at most one declaration for each predicate symbol.

The operational model of δ-ccp can be described by a transition system $T = (Conf, \longrightarrow)$ as follows. The configurations (in) $Conf$ are pairs consisting of a process, and a constraint in $\mathcal{C}$ representing the common *store*. The transition relation $\longrightarrow \subseteq Conf \times Conf$ is described by the (least relation satisfying the) rules **R1-R5** of Table 1 where we assume given a set D of declarations and an interpretation $I : \Delta \rightarrow \wp(\mathcal{C})$ for delay conditions.

The agent *Stop* represents successful termination. The basic actions are given by $tell(c)$ and $ask(\delta(\overline{x}))$ constructs which act on the common store. Given a store d, as shown by rule **R1**, the execution of $tell(c)$ updates the store to $c \sqcup d$.

The action $ask(\delta(\overline{x}))$ represents a guard, i.e. a test on the current store c, whose execution does not modify c. We say that $ask(\delta(\overline{x}))$ is *enabled* in c iff $c \in I(\delta(\overline{x}))$, i.e. if c satisfies the delay condition $\delta(\overline{x})$. Thus, as shown in rule **R2**, if $ask(\delta(\overline{x}))$ is enabled in the current store then the agent $ask(\delta(\overline{x})) \rightarrow A$ can proceed in the computation and behaves as A. Otherwise the agent suspends, waiting for other (parallel) agents to add information to the store. Clearly this mechanism allows us to simulate the delay evaluation previously described for CLP.

The operator $\parallel$ represents parallel composition which is described by rule **R3** as interleaving.

According to rule **R4** the *choice* operator gives rise to local (or internal) non-determinism: the agent $A + B$ nondeterministically selects one of the two components and the environment cannot control the choice.

Rule **R5** treats the case of a procedure call. Here, analogously to the case of CLP, we assume that all the variables which appear in the pure atoms contained in $p(\overline{x}) : -A$ are distinct and we denote by $defn_D(p(\overline{x}))$ the set of variants of declarations in D which have $p(\overline{x})$ as head. We assume also the presence of a renaming mechanism that takes care of using fresh variables each time a clause is considered (in **R5**). For the sake of simplicity we do not describe this renaming mechanism in the transition system. The interested reader can find in [22] and [5] various formal approaches to this problem.

Using the transition system in Table 1 we can now define the standard notion of observables as the results of finite computations. As in the case of CLP observables, we do not consider the values of local variables in the results. We denote by $\longrightarrow^*$ the reflexive and transitive closure of the relation $\longrightarrow$.

Definition 3.2 Let $D.A$ be a δ-ccp process and $\mathcal{C}$ be a constraint system. The observables of $D.A$, wrt an initial store $c \in \mathcal{C}$, are

$$\mathcal{O}_{ccp}(D.A, c) = \{\exists_{-vars(A,c)} d \mid \text{ there exists } \langle D.A, c\rangle \longrightarrow^* \langle D.B, d\rangle \not\longrightarrow\}.$$

3.1 Mapping CLP programs with dynamic scheduling to δ-ccp programs

A CLP program can be seen as a δ-ccp program essentially by identifying atoms with agents, constraints with tell agents, conjunction with parallel composition, and disjunction (of clauses) with nondeterministic choice. Clearly there remain some differences, mainly due to the different underlying computational model. In fact, in CLP a consistency test is done at each derivation step in order to prevent (finitely) failing computation, while this is not the case for δ-ccp and for the so-called "eventual tell" ccp languages. However these differences are not relevant to the issue of partial correctness.

R1	$\langle D.tell(c), d\rangle \longrightarrow \langle D.Stop, c \sqcup d\rangle$	
R2	$\langle D.ask(\delta(\overline{x})) \rightarrow A, c\rangle \longrightarrow \langle D.A, c\rangle$	$c \in I(\delta(\overline{x}))$
R3	$\dfrac{\langle D.A, c\rangle \longrightarrow \langle D.A', c'\rangle}{\begin{array}{c}\langle D.A \parallel B, c\rangle \longrightarrow \langle D.A' \parallel B, c'\rangle \\ \langle D.B \parallel A, c\rangle \longrightarrow \langle D.B \parallel A', c'\rangle\end{array}}$	
R4	$\begin{array}{l}\langle D.A + B, c\rangle \longrightarrow \langle D.A, c\rangle \\ \langle D.A + B, c\rangle \longrightarrow \langle D.B, c\rangle\end{array}$	
R5	$\langle D.p(\overline{x}), c\rangle \longrightarrow \langle D.A, c\rangle$	$p(\overline{x}) : -A \in defn_D(p(\overline{x}))$

Table 1: The transition system for δ-ccp.

As for delay declarations it is quite natural to simulate them by using the *ask* mechanism previously mentioned.

So we can formally define the translation τ from CLP with dynamic scheduling to δ-ccp as follows. Given a CLP query $c, A_1, \ldots, A_n$ we obtain an δ-ccp agent defined by

$$\tau(c, A_1, \ldots, A_n) = tell(c) \parallel A_1 \parallel \ldots \parallel A_n$$

(here and in the following we omit parenthesis since the operators $\parallel$ and $+$ are associative).

As for programs, if the CLP program P consists of the following clauses, defining a single predicate $\mathtt{p}$

$$\begin{array}{l}\mathtt{p}(\overline{\mathtt{x}}) \leftarrow \mathtt{c}_1, \mathtt{Q}_1 \\ \vdots \\ \mathtt{p}(\overline{\mathtt{x}}) \leftarrow \mathtt{c}_n, \mathtt{Q}_n\end{array}$$

and of the delay declaration

$$\mathtt{DELAY\ p}(\overline{\mathtt{x}})\ \mathtt{UNTIL}\ \delta(\overline{\mathtt{x}})$$

then $\tau(P)$ is the δ-ccp declaration

$$p(\overline{x}) : -ask(\delta(\overline{x})) \rightarrow (\tau(c_1, Q_1) + \ldots + \tau(c_n, Q_n)).$$

This can be extended to programs defining several predicates in the obvious way.

The equivalence of the original CLP program with the translated one is expressed by the following.

Proposition 3.3 *Let P be a CLP program with delay declarations and c, Q be a query where $c \in \mathcal{C}$ ($c \neq false$) is the input constraint. Then*

$$\mathcal{O}_{clp}(P, Q, c) = \mathcal{O}_{ccp}(\tau(P).\tau(Q), c) \setminus \{false\}.$$

Given the previous proposition, any partial correctness result which holds for δ-ccp can be immediately applied to CLP programs with delay declarations. Therefore in the following we will focus on δ-ccp programs.

Example 3.4 We give now an example of translation of a CLP program into the ccp-like language above introduced. We consider a not-so-trivial program which we will use again later to illustrate an application of our verification method. This program takes two lists `Li` and `Lp` representing the nodes obtained from the in-order and pre-order traversal of a binary tree `T`, respectively, and returns `T`.

A tree with root a, left subtree Tl, and right subtree Tr, will be represented by the term (a,Tl,Tr); the empty tree will be represented by ().

The idea is the following: If the lists are empty, then T will be the empty tree. Otherwise, we pick the first element X of Lp, which will become the root of T, and we split Li in two parts: the sublist Lil of the elements to the left of X, and the sublist Lir of the elements to the right of X. By comparing recursively Lil and the tail of Lp we will obtain the left subtree Tl of T and a residual list Lpr of the rightmost elements of Lp which have not been used for the construction of Tl. By comparing recursively Lir and Lpr we will obtain the right subtree Tr of T. Generally this second recursive call will also produce a residual list L of the rightmost elements of Lpr which have not been used for the construction of Tr.

```
tree(Li,Lp,T) ←
   L=[], tree&residual(Li,Lp,T,L).

tree&residual(Li,Lp,T,L) ←
   Li=[], T=(), L=Lp.
tree&residual(Li,Lp,T,L) ←
   Li=[Y|Lit], Lp=[X|Lpt], T=(X,Tl,Tr),
   split(X,Li,Lil,Lir),
   tree&residual(Lil,Lpt,Tl,Lpr),
   tree&residual(Lir,Lpr,Tr,L).

split(X,L,Ll,Lr) ←
   L =[X|Lr], Ll=[].
split(X,L,Ll,Lr) ←
   L=[Y|Lt], X ≠ Y, Ll=[Y|Llt], split(X,Lt,Llt,Lr).
```

A sequence of constraints, in a clause or in a goal, is to be interpreted as their logical conjunction (join). For instance Li=[], T=(), L=Lp corresponds to Li=[] ∧ T=() ∧ L=Lp.

The delay declarations for this program are:

```
DELAY tree&residual(Li,Lp,_,_) UNTIL nonvar(Li) ∧ nonvar(Lp).
DELAY split(X,L,_,_) UNTIL ground(X) ∧ ∃ Y: ∃ Lt:  L=[Y|Lt] ∧ ground(Y).
```

The translation of this program into δ-ccp gives the following:

$$
\begin{array}{l}
\mathit{tree}(Li, Lp, T) \text{ :-} \\
\quad \mathit{tell}(L = [\,]) \parallel \mathit{tree\&residual}(Li, Lp, T, L) \\
\\
\mathit{tree\&residual}(Li, Lp, T, L) \text{ :-} \\
\quad \mathit{ask}(\mathit{nonvar}(Li) \wedge \mathit{nonvar}(Lp)) \rightarrow \\
\qquad \mathit{tell}(Li = [\,] \wedge T = () \wedge \; L = Lp) \\
\qquad + \\
\qquad (\;\mathit{tell}(Li = [Y|Lit] \wedge Lp = [X|Lpt] \wedge T = (X, Tl, Tr)) \parallel \\
\qquad\quad \mathit{split}(X, Li, Lil, Lir) \parallel \\
\qquad\quad \mathit{tree\&residual}(Lil, Lpt, Tl, Lpr) \parallel \\
\qquad\quad \mathit{tree\&residual}(Lir, Lpr, Tr, L)\;) \\
\\
\mathit{split}(X, L, Ll, Lr) \text{ :-} \\
\quad \mathit{ask}(\mathit{ground}(X) \wedge \exists Y : \exists Lt : L = [Y|Lt] \wedge \mathit{ground}(Y)) \rightarrow \\
\qquad \mathit{tell}(L = [X|Lr] \wedge Ll = [\,]) \\
\qquad + \\
\qquad (\;\mathit{tell}(L = [Y|Lt] \wedge X \neq Y \wedge Ll = [Y|Llt]) \parallel \\
\qquad\quad \mathit{split}(X, Lt, Llt, Lr)\;)
\end{array}
$$

3.2 Optimal delay declarations versus guarded choice

The translated program in Example 3.4 is not as efficient as the original CLP program: It has more failing computations, due to the fact that we could select a branch whose constraints (in the tell processes) are inconsistent with the present store. The problem is actually made relevant by the fact that the execution of a tell process does not necessarily have priority in a parallel context. Hence the execution might go on for several steps before detecting the inconsistency. To solve this problem, we could impose a computational rule ensuring that the tell processes at the outmost level in a parallel context are executed first. These considerations are out of the scope of this paper, since here we only consider ccp only as a formalism to prove properties on the original CLP program. However it is interesting to analyze the reasons of this discrepancy. In pure CLP there is only one kind of test on the store: the consistency check, which is done before choosing a clause for the resolution step. The delay declarations provide additionally a sort of entailment check. These two mechanisms together allow to simulate a restricted form of the ccp global choice: the *mutually exclusive choice* [9]. For instance, the constraints `Li=[]` and `Li=[Y|Lit]` in the clauses for `tree&residual`, together with the delay condition `nonvar(Li)`, will ensure that we select the first clause only if the current store entails that `Li` is the empty list, and we select the second clause only if the current store entails that `Li` has at least an element. In ccp, the most natural (and efficient) way to define `tree&residual` would be:

$$
\begin{array}{l}
tree\&residual(Li, Lp, T, L) \text{ :-} \\
\quad ask(Li = [\,]) \rightarrow tell(Li = [\,] \wedge T = () \wedge\ L = Lp) \\
\quad + \\
\quad ask(\exists_Y \exists_{Lit} \exists_X \exists_{Lpt} Li = [Y|Lit] \wedge Lp = [X|Lpt]) \rightarrow \\
\qquad (\ tell(Li = [Y|Lit] \wedge Lp = [X|Lpt] \wedge T = (X, Tl, Tr)) \parallel \\
\qquad\ \ split(X, Li, Lil, Lir) \parallel \\
\qquad\ \ tree\&residual(Lil, Lpt, Tl, Lpr) \parallel \\
\qquad\ \ tree\&residual(Lir, Lpr, Tr, L)\ )
\end{array}
$$

The form of choice used in this program is now the standard ccp global choice, i.e. a branch can be selected only if the guard (*ask*) is satisfied. In this particular case it is mutually exclusive, in the sense that the conjunction of the constraints used in the guards is inconsistent. Mutually exclusive choice doesn't add any nondeterminism. In fact it can be eliminated by replacing the choice operator $+$ with the parallel operator $\parallel$: It is easy to see that (if we abstract from suspension) the semantics doesn't change.

Given a CLP program, we say that a delay declaration for a predicate `p` is *optimal* if, for each goal, the selection of the atom $\mathtt{p}(\overline{\mathtt{x}})$ will be delayed until it is "safe" to choose a clause in its definition and to execute its body, in the sense that the future information computed by the other atoms (in non-failing computations) will be consistent with that choice.

An optimal delay declaration for $\mathtt{p}(\overline{\mathtt{x}})$ has the property that, for each store `s` (the accumulated constraint at a certain point of the computation) which satisfies it, for each clause in the definition of $\mathtt{p}(\overline{\mathtt{x}})$, if `s` is consistent with the constraint `c` of such clause, then every store greater than `s` (apart from *false*) will also be consistent with `c`. Therefore, if the constraints of two clauses are not mutually exclusive, it will never be the case that a store `s` satisfies the delay condition and is consistent with both of them, while a store greater than `s` is consistent only with one of them.

The most natural way to translate a CLP program with optimal delay declarations into ccp would be to use the mutually exclusive choice for the clauses with mutually exclusive constraints, and local choice (like in Example 3.4) for the others. For the reason explained above, this translation would not induce additional failures, although the problem that the detection of an inconsistency might be inefficient is still there.

4 Partial correctness

In this section we formalize the concept of partial correctness for δ-ccp programs. To this aim we first define the correctness properties to be used in the specification logic. Then we define validity of assertions in terms of a strongest postcondition semantics.

4.1 Properties as set of constraints

Intuitively, a property expresses some common feature of the results of (finite) computations. Since in our case results are constraints it is natural to consider a property as a set of basic constraints, namely the set of all those constraints which satisfy the property itself.

In order to define a specification logic we need some syntactic representation of properties. As far as tell operations are concerned, we can simply consider a language consisting of (a syntactic representation of) constraints themselves, together with the usual connectives of the predicate calculus.

However, in the specification logic we need to represent also delay conditions, plus other user-defined predicates which have an intentional meaning, defined by the user, to specify properties of the computation. For example, given a numerical constraint system such as $\Re$, one could need the property $fact(x, y)$ which is satisfied by all the constraints forcing x to be the factorial of y. These properties usually correspond to an infinite set of constraints, hence they cannot be represented in the language above as we would need an infinite disjunction.

Therefore we consider as basic properties also *user defined predicates*. These are predicates introduced by the user, together with an interpretation which identify them with a specific set of basic constraints. We denote by U the set of user defined properties, with typical elements $u(\overline{x}), u_1(\overline{x}) \ldots$ and we assume that $\Delta \subseteq U$. Thus the language of properties is defined as follows.

Definition 4.1 Given a constraint system $\mathcal{C}$ and a set of user defined predicates U, the language of properties is defined by the following grammar:

$$\phi ::= c \mid u(\overline{x}) \mid \phi \wedge \psi \mid \neg\phi$$

where $c \in \mathcal{C}$ and $u(\overline{x}) \in U$.

Logical disjunction ($\vee$) and implication ($\rightarrow$) are defined in the usual way: $\phi \vee \psi =_{\mathrm{df}} \neg(\neg\phi \wedge \neg\psi)$ and $\phi \rightarrow \psi =_{\mathrm{df}} \neg\phi \vee \psi$.

A constraint c viewed as a property will be interpreted as the set of constraints d that entail c, i.e. as the upward closure $\uparrow c$ of c in $\mathcal{C}$ (wrt the $\leq$ ordering). Thus a constraint d satisfies the basic property c iff $d \in \uparrow c$. For user defined predicates, we assume given an interpretation

$$I : U \rightarrow \wp(\mathcal{C})$$

which defines their meaning. Thus a user defined property $u(\overline{x})$ is satisfied by a constraint c iff $c \in I(u(\overline{x}))$. We assume that I is consistent wrt variable renaming, i.e. $c \in I(u(\overline{x}))$ iff $c[\overline{y}/\overline{x}] \in I(u(\overline{y}))$. Moreover, as previously mentioned, for $\delta(\overline{x}) \in \Delta$ we assume that $I(\delta(\overline{x})) = \uparrow I(\delta(\overline{x}))$.

The logical operations of conjunction and negation are interpreted in the classical way by their corresponding set theoretic operations of intersection and complement: Thus a constraint c satisfies a property $\phi \wedge \psi$ iff it satisfies both ϕ and ψ, and a constraint c satisfies a property $\neg\phi$ iff c does not satisfy ϕ. Formally, the semantics of properties is defined as follows:

Definition 4.2 The semantics of properties is given by a function $[\![\cdot]\!]$ which maps properties into non-empty sets of constraints and it is defined as follows:

$$\begin{array}{lcl}
[\![c]\!] & = & \uparrow c \\
[\![u(\overline{x})]\!] & = & I(u(\overline{x})) \\
[\![\neg\phi]\!] & = & (\mathcal{C} \setminus [\![\phi]\!]) \cup \{false\} \\
[\![\phi \wedge \psi]\!] & = & [\![\phi]\!] \cap [\![\psi]\!].
\end{array}$$

Note that the semantics of a property is parametric wrt $\mathcal{C}$ and I. We omitted them from the notation just for the sake of simplicity.

Note that the semantic domain is a boolean algebra of non-empty sets of constraints, where $\{false\}$ represents the least element (this justifies the above definition of $[\![\neg\phi]\!]$ as $(\mathcal{C} \setminus [\![\phi]\!]) \cup \{false\}$ instead than simply $\mathcal{C} \setminus [\![\phi]\!]$). The reason why we consider non-empty sets only is merely technical: it allows for a more direct correspondence with the semantics of δ-ccp.

Definition 4.3 A property ϕ is called valid, notation $\models_{\mathcal{C},I} \phi$, iff every constraint satisfies the property, i.e. $[\![\phi]\!] = \mathcal{C}$, where $\mathcal{C}$ is the given constraint system and I is the given interpretation for user defined properties.

Since we are in a boolean algebra, the validity of a property can be proved simply by the inference system of the predicate logic, on the set of proper axioms specifying the relations among the basic elements of the algebra. These are the sets corresponding to user-defined predicates and constraints, hence we assume an axiom of the form

$$c \to d$$

for each pair of constraints c, d such that $d \leq c$, and an axiom of the form

$$c \to u(\bar{x})$$

for each constraint c, user-defined prdicate $u(\bar{x})$, such that $c \in I(u(\bar{x}))$.

If we indicate by $Ax(\mathcal{C}, I)$ the collection of all these axioms, then we have

Proposition 4.4 $\models_{\mathcal{C},I} \phi$ *iff* $Ax(\mathcal{C}, I) \vdash \phi$, *where* $\vdash$ *is the derivability relation in the predicate calculus.*

We conclude this section by discussing the relation between the logic of the properties and the logic implicit in the underlying constraint system. It should be clear that the former is a richer logic. The only connective which is present also in the constraint system is $\wedge$, in fact $c \sqcup d$ and $c \wedge d$ are equivalent. But for the other connectives this is not true in general. The $\vee$ does not correspond to the greatest lower bound $\sqcap$ in $\mathcal{C}$. In fact $(c \vee d) \to (c \sqcap d)$ is valid, but the reverse is not, since every constraint which satisfy $(c \vee d)$ should entail c or d, while this is not the case for $(c \sqcap d)$. Analogously, if we have a form of negation in $\mathcal{C}$, this is in general a weaker form of negation than $\neg$. For instance, if we have the constraints $x = 0$ and $x \neq 0$, this only means that $x = 0 \sqcup x \neq 0$ is *false*, but doesn't generally mean that $x = 0 \sqcap x \neq 0$ is *true* (it could be, for instance, a constraint saying that x is ground). Thus $x \neq 0 \to \neg x = 0$ is valid, but the reverse is not, and $x = 0 \vee \neg x = 0$ is valid, but $x = 0 \vee x \neq 0$ is not. In particular, the latter is satisfied by a constraint c only if c entails $x = 0$ or c entails $x \neq 0$.

4.2 Partial correctness assertions

We introduce now our language for specifying correctness assertions.

Definition 4.5 A *partial correctness assertion* is a construct of the form $D.A\ sat\ \phi$ where $D.A$ is a δ-ccp process and ϕ is a property. The semantic of an assertion $D.A\ sat\ \phi$ is given as follows:

$$\models D.A\ sat\ \phi \quad \text{iff} \quad SP(D.A) \subseteq [\![\phi]\!]$$

where

$$SP(D.A) =_{def} \{d \mid \quad \text{there exists } c \in \mathcal{C} \text{ s.t. } d \in \mathcal{O}_{ccp}(D, A, c) \quad \}.$$

Intuitively $\models D.A\ sat\ \phi$, i.e. $D.A\ sat\ \phi$ is valid, iff every terminating computation of $D.A$ (for any input) results in a constraint which satisfies the property ϕ. The set $SP(D.A)$ actually describes the *strongest postcondition* of the process $D.A$ with respect to the precondition *true*, since every constraint satisfies the property *true*.

5 A proof-system for δ-ccp

In this section we develop a proof-system for δ-ccp following the lines of the one defined in [4] for ccp. To this aim, it is convenient to give first a compositional characterization of the semantics $SP(D.A)$. It comes out that we can use nearly the same denotational construction as the one introduced by Jagadeesan et al. in [14] for local ccp. This is worth noticing, since δ-ccp has more expressive power than local ccp, for it allows tests (ask guards) on (upward closed) properties.

D0	$[\![D.stop]\!](e) = \mathcal{C}$
D1	$[\![D.tell(c)]\!](e) = \uparrow c$
D2	$[\![D.ask(\delta(\overline{x})) \rightarrow A]\!](e) = (\mathcal{C} \setminus I(\delta(\overline{x}))) \cup (I(\delta(\overline{x})) \cap [\![D.A]\!](e))$
D3	$[\![D.A \parallel B]\!](e) = [\![D.A]\!](e) \cap [\![D.B]\!](e)$
D4	$[\![D.A + B]\!](e) = [\![D.A]\!](e) \cup [\![D.B]\!](e)$
D5	$[\![D.p(\overline{x})]\!](e) = e(p)(\overline{x}) \qquad p \notin D$
D6	$[\![D.p(\overline{x})]\!](e) = \mu\Psi$ where $\Psi(f) = \exists\!\!\!\exists_{-\overline{x}}[\![D \setminus \{p\}.A]\!](e[p/f](\overline{x}))$, $p(\overline{x}) : -A \in defn_D(p(\overline{x}))$

Table 2: Strongest postcondition semantics of δ-ccp

The point here is that even though δ-ccp can be seen as a "lifted ccp", i.e. a ccp language defined on a lifted constraint system, yet the results of computations can be considered as constraints in $\mathcal{C}$. More precisely, given a constraint system $\mathcal{C}$, consider the structure

$$\mathbf{C}^l = \langle \wp(\mathcal{C})^u, \supseteq, \cap, \mathcal{C}, \{false\}, Var, \uparrow\exists_x, \uparrow d_{xy}\rangle$$

where $\wp(\mathcal{C})^u$ is the set of non-empty upward-closed subsets of $\mathcal{C}$. It can be proved ([3]) that $\mathbf{C}^l$ is a constraint system. Now if we consider $ask(\delta(\overline{x}))$ and $tell(c)$ as representatives of $ask(I(\delta(\overline{x}))$ and $tell(\uparrow c)$, respectively, we obtain a local ccp language defined on the lifted constraint system $\mathbf{C}^l$. In fact, the condition $c \in I(\delta(\overline{x}))$ used in the transition system in Table 1 can equivalently be expressed as $\uparrow c \subseteq I(\delta(\overline{x}))$ (since $I(\delta(\overline{x}))$ is upward closed) and $\subseteq$ is indeed the entailment relation in $\mathbf{C}^l$.

The ask mechanism in this lifted language is more powerful than the one defined on $\mathcal{C}$, since it allows us to use generic upward closed sets as arguments. For example, we could test that the store contains the maximal information on x (a generalization of the notion of groundness) by using a guard $ask(g(x))$ where $g(x)$ is defined as:

$$g(x) = \{c \mid \text{for every } d.\ \exists_{-x}d \vdash \exists_{-x}c \text{ implies } \exists_{-x}c \vdash \exists_{-x}d\}.$$

However, (assuming inputs of the form $\uparrow c$) the results of lifted computations can still be considered as elements in $\mathcal{C}$, since $\uparrow c \cap \uparrow d = \uparrow(c \sqcup d)$ for any $c, d \in \mathcal{C}$ and the set $\uparrow(c \sqcup d)$ can be simply represented by $c \sqcup d$ (note that $\cap$ is the lub in $\mathbf{C}^l$).

Thus we can define the following denotational semantics.

Definition 5.1 Let $\wp_f(\mathcal{C})$ denote the collection of the subsets of $\mathcal{C}$ containing *false*, and let *Env* denote the set of environments, namely functions e which associate to each procedure name p and to each tuple $\bar{x}$ an element of $\wp_f(\mathcal{C})$, with the property that $e(p)(\bar{y}) = e(p)(\bar{x})[\bar{x}/\bar{y}]$. The denotational semantics of δ-ccp is expressed by the function $[\![\cdot]\!]$: $Declarations \times Agents \rightarrow Env \rightarrow \wp_f(\mathcal{C})$ defined by the equations in Table 2, where μ denotes the least fixpoint wrt subset inclusion and $\exists\!\!\!\exists$ is the cylindrification operator on sets of constraints, namely $\exists\!\!\!\exists_x(C) = \{c \mid \exists_x c = \exists_x d \text{ for some } d \in C\}$.

In equations **D5** and **D6** $p \notin D$ means that D does not contain any declaration for p, while $D \setminus \{p\}$ denotes the program obtained from D by removing the declaration for p.

Note that the environment is used only in the fixpoint construction, but it is not relevant in the semantics of a process, since a process is closed (wrt procedure names) by definition. Namely we have:

C0	$D.stop\ sat\ true$	
C1	$D.tell(c)\ sat\ c$	
C2	$\dfrac{D.A\ sat\ \phi}{D.ask(\delta(\overline{x})) \rightarrow A\ sat\ \delta(\overline{x}) \rightarrow \phi}$	
C3	$\dfrac{D.A\ sat\ \phi \quad D.B\ sat\ \psi}{D.A \parallel B\ sat\ \phi \wedge \psi}$	
C4	$\dfrac{D.A\ sat\ \phi \quad D.B\ sat\ \psi}{D.A + B\ sat\ \phi \vee \psi}$	
C5	$\dfrac{D.p(\overline{x})\ sat\ \phi}{D.p(\overline{y})\ sat\ \phi[\overline{x}/\overline{y}]}$	
C6	$\dfrac{D \setminus \{p\}.p(\overline{x})\ sat\ \phi \vdash D \setminus \{p\}.A\ sat\ \phi}{D.p(\overline{x})\ sat\ \phi}$	$p(x) : -A \in defn_D(p(\overline{x}))$
C7	$\dfrac{D.A\ sat\ \phi \quad Ax(\mathcal{C}, I) \vdash \phi \rightarrow \psi}{D.A\ sat\ \psi}$	

Table 3: A calculus for δ-ccp

Remark 5.2 *For any process $D.A$, for every pair of environments e and e', $[\![D.A]\!](e) = [\![D.A]\!](e')$.*

Therefore we can denote the semantics of a process $D.A$ simply by $[\![D.A]\!]$.

Using the lifting described above we now obtain the following result.

Proposition 5.3 *For any process $D.A$ we have $SP(D.A) = [\![D.A]\!]$.*

As discussed in [4], the denotational semantics allows us to obtain the proof-system by simply "mirroring" the equations. In fact the operators of the language are modeled in these equations by standard set theoretic notions which in the specification logic get replaced by the corresponding logical notions.

Table 3 contains the rules of the calculus for assertions of the form $D.A\ sat\ \phi$. Rules **C0-C6** are given in the usual natural deduction style and are self explaining since they are obtained essentially by a logical reading of equations **D0-D6**, i.e. by turning $\cup$ into $\vee$ and $\cap$ into $\wedge$. It is also worth noticing that they provide a logical meaning for the operators of the language: action prefixing ($\rightarrow$) corresponds to implication (still denoted by $\rightarrow$), parallel composition to conjunction and choice to disjunction.

Reasoning about recursion is formalized in terms of the meta-rule **C6** (Scott-induction [25]) which allows to conclude that the agent $p(\overline{x})$ satisfies a property ϕ whenever the body of $p(\overline{x})$ satisfies the same property assuming the conclusion of the rule. Also here we assume to choose an instance of the definition of p such that all local variables are fresh wrt $\overline{x}$ and ϕ.

Finally, we have the consequence rule **C7** for weakening a property. This rule allows to include in the reasoning process information about the underlying constraint system and the user-defined predicates.

We illustrate now the formal justification of the above calculus, namely its soundness and completeness.

Theorem 5.4 (Soundness) *If $D.A\ sat\ \phi$ can be derived by the inference system in Table 3, then $\models D.A\ sat\ \phi$ holds.*

Completeness of the system is proved in the sense of Cook ([7]): We assume the expressibility of the strongest postcondition of a process $D.A$, i.e. that there exists a property ϕ such that $SP(D.A) = [\![\phi]\!]$, and that we are able to prove all the valid relations of the formulas involving ϕ. In the case of CLP, in general ϕ can be expressed with a predicate defined recursively via clauses.

Theorem 5.5 (Completeness) *If $\models D.A$ sat ϕ, then $D.A$ sat ϕ can be derived by the inference system in Table 3.*

The above results can be applied in a direct way to CLP programs with delay declarations by using Proposition 3.3. Thus we have the following result where, for the if part, we make the same assumptions made above for the completeness theorem.

Corollary 5.6 *Let P be a CLP program with delay declarations, c, Q be a query and ϕ a property. Then $\tau(P).\tau(Q)$ sat ϕ can be derived by the inference system in Table 3 iff $\bigcup_{c\in\mathcal{C}} \mathcal{O}(P,Q,c) \cup \{false\} \subseteq [\![\phi]\!]$.*

So we can prove that $\tau(P).\tau(Q)$ satisfies the property ϕ iff any finite non-failing computation for the query c, Q, for any constraint c, in the CLP program P, produces an answer constraint which satisfies ϕ.

Example 5.7 We give now an example of the use of our proof system to prove properties of programs. Consider again the program in Example 3.4. We want to prove the following property:

$$tree(Li, Lp, T) \;\; sat \;\; (ground(Li) \wedge ground(Lp)) \rightarrow (samenodes(Li,T) \wedge samenodes(Lp,T)),$$

where $somenodes(L,T)$ holds iff L and T have the same elements. Let's denote by ϕ the property $(ground(Li) \wedge ground(Lp)) \rightarrow (samenodes(Li,T) \wedge samenodes(Lp,T))$. By the rule for the procedure call (**C6**) it is sufficient to prove that

$$tell(L = [\,]) \parallel tree\&residual(Li, Lp, T, L) \;\; sat \;\; \phi.$$

By the rule for the weakening (**C7**) it is sufficient to consider

$$tell(L = [\,]) \parallel tree\&residual(Li, Lp, T, L) \;\; sat \;\; L = [\,] \wedge \psi,$$

where ψ is the property $(ground(Li) \wedge ground(Lp)) \rightarrow (samenodes(Li,T) \wedge samenodes(Lp \setminus L,T))$, with $Lp \setminus L$ denoting the nodes of Lp minus the nodes of Li. By the parallel rule (**C3**) and the tell rule (**C1**) we reduce the proof to

$$tree\&residual(Li, Lp, T, L) \;\; sat \;\; \psi.$$

We can now apply the rule for the procedure call, thus reducing the proof to

$$ask(nonvar(Li) \wedge nonvar(Lp)) \rightarrow A_1 + A_2 \;\; sat \;\; \psi$$

where A_1 is the process $tell(Li = [\,] \wedge T = () \wedge \; L = Lp)$ and A_2 is the process

$$\begin{array}{l} tell(Li = [Y|Lit] \wedge Lp = [X|Lpt] \wedge T = (X, Tl, Tr)) \parallel \\ split(X, Li, Lil, Lir) \parallel \\ tree\&residual(Lil, Lpt, Tl, Lpr) \parallel \\ tree\&residual(Lir, Lpr, Tr, L). \end{array}$$

Remember that to make this last proof we can use the assumption

$$tree\&residual(Li, Lp, T, L) \quad sat \quad \psi, \tag{1}$$

as it is allowed by the Rule **C6**. Observe now that $\psi \equiv (nonvar(Li) \wedge nonvar(Lp)) \rightarrow \psi$, thus using the ask rule (**C2**) we reduce the proof to

$$A_1 + A_2 \;\; sat \;\; \psi$$

Consider that $\psi \equiv (\psi \wedge Li = [\,]) \vee (\psi \wedge Li \neq [\,])$, and that $\psi \wedge Li = [\,] \equiv Li = [\,] \wedge T = () \wedge Lp \subseteq L$ (where $Lp \subseteq L$ means that the elements of Lp are contained in the elements of L). From Rule **C1** we have

$$A_1 \equiv tell(Li = [\,] \wedge T = () \wedge \; L = Lp) \;\; sat \;\; Li = [\,] \wedge T = () \wedge Lp \subseteq L,$$

hence for the parallel rule it is sufficient to prove that

$$A_2 \;\; sat \;\; \psi \wedge Li \neq [\,].$$

Observe that $\psi \wedge Li \neq [\,]$ is implied by the condition

$$\begin{array}{ll}\eta \equiv & (Li = [Y|Lit] \wedge Lp = [X|Lpt] \wedge T = (X, Tl, Tr)) \\ & \wedge \\ & ((ground(X) \wedge ground(Li)) \rightarrow Li = Lil * X * Lir \\ & \wedge \\ & ((ground(Lil) \wedge ground(Lpt)) \rightarrow (samenodes(Lil, Tl) \wedge samenodes(Lpt \setminus Lpr, Tl))) \\ & \wedge \\ & ((ground(Lir) \wedge ground(Lpr)) \rightarrow (samenodes(Lir, Tr) \wedge samenodes(Lpr \setminus L, Tr)))\end{array}$$

(here $Lil * X * Lir$ represents the concatenation of Lil, X, and Lir), hence by the weakening rule it is sufficient to prove that $A_2 \;\; sat \;\; \eta$. This can be done by using the parallel rule, and then proving:

- $tell(Li = [Y|Lit] \wedge Lp = [X|Lpt] \wedge T = (X, Tl, Tr)) \;\; sat \;\; (Li = [Y|Lit] \wedge Lp = [X|Lpt] \wedge T = (X, Tl, Tr))$. By the tell rule.

- $tree\&residual(Lil, Lpt, Tl, Lpr) \;\; sat \;\; \begin{array}{l}(ground(Lil) \wedge ground(Lpt)) \\ \rightarrow \\ (samenodes(Lil, Tl) \wedge samenodes(Lpt \setminus Lpr, Tl)).\end{array}$

  By the assumption (1), and the renaming rule (**C5**).

- $tree\&residual(Lir, Lpr, Tr, L) \;\; sat \;\; \begin{array}{l}(ground(Lir) \wedge ground(Lpr)) \\ \rightarrow \\ (samenodes(Lir, Tr) \wedge samenodes(Lpr \setminus L, Tr)).\end{array}$

  Like before.

- $split(X, Li, Lil, Lir) \;\; sat \;\; (ground(X) \wedge \exists Y : \exists Lt : L = [Y|Lt] \wedge ground(Y)) \rightarrow Li = Lil * X * Lir$. We leave this for exercise to the reader.

6 Conclusion and future work

We presented a correct and complete compositional proof-system for proving partial correctness of CLP programs with delay declarations.

Several extensions and different applications are possible. Firstly, since we have presented a calculus for the strongest postcondition wrt the precondition $true$, a natural extension would be to consider arbitrary preconditions, i.e. to give a calculus for triples of the form $\{\psi\}\ D.A\ \{\phi\}$ in the classical Hoare-logic style. This would have the intuitive meaning that if the computation of $D.A$ starts in a store which satisfies the property ψ and terminates in a state d, then d satisfies the property ϕ. As observed in [4], if preconditions are described by upward closed properties (i.e. they are expressed by formulas involving constraints, conjunctions and disjunctions only) then triples are not more expressive than the assertions considered in previous sections. In fact it is easy to see that $\{\psi\}\ D.A\ \{\phi\}$ iff $D.A \;\; sat \;\; \psi \rightarrow \phi$. Introducing preconditions containing negation is not straightforward, since the stronger postcondition semantics in this case is not any more compositional.

Secondly, since properties could be expressed in a CLP language itself, our calculus could be used to prove equivalence of programs and "internal" properties of programs. For example, given the usual `REVERSE` program defining the predicate `reverse(s,t)` which reverses a list, one could easily prove the symmetricity of the predicate, i.e. that

$$\texttt{REVERSE.reverse(s,t)} \;\; sat \;\; \texttt{REVERSE.reverse(t,s)}.$$

Thirdly, we can use our calculus for the transformational design of programs by means of refinement techniques. The main idea underlying the transformational approach to the design of programs is the stepwise transformation of specifications into programs. To express the stepwise construction of a program usually the formalism of *mixed terms* ([21]) is introduced, i.e. terms that are constructed out of programs and specifications. In the case of δ-ccp programs, as previously mentioned, we can easily have a unified formalism both for programs and properties. In such a formalism an assertion of the from $P \;\; sat \;\; \phi$

corresponds to the implication $P \rightarrow \phi$ and thus models the *refinement (or satisfaction or implementation)*: A (mixed) term Φ *refines* a mixed term Ψ iff $\Phi \rightarrow \Psi$ holds. So, a derivation of a process P from a specification ϕ in this approach corresponds with proving (using our calculus) the sequence $P \equiv \Phi_n \rightarrow \ldots \rightarrow \Phi_1 \equiv \phi$ of implications between mixed terms $\Phi_1, \ldots \Phi_n$ where Φ_1 is the given specification ϕ and Φ_n denotes the derived program.

References

[1] K. R. Apt. Declarative Programming in Prolog. In D. Miller, editor, *Proceedings of the Int'l Symposium on Logic Programming*, The MIT Press, pages 11–35x, 1993.

[2] 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, pages 1–19, Berlin, 1995. Springer-Verlag.

[3] F.S. de Boer, A. Di Pierro, and C. Palamidessi. An algebraic perspective of constraint logic programming. *Journal of Logic and Computation*. To appear.

[4] F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving Concurrent Constraint Programs Correct. In *Proc. of Twentyfirst POPL*, ACM Press, 1994.

[5] F.S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT/CAAP, LNCS 493*, pages 296–319. Springer-Verlag, 1991.

[6] M. Carlsson. *SICStus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[7] S.A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Computation*, 7(1):70–90, 1978.

[8] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional Analysis for Concurrent Constraint Programming. In *Proc. Eighth IEEE Symp. on Logic In Computer Science*, pages 210–221. IEEE Computer Society Press, Los Alamitos, California, 1993.

[9] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence and Concurrent Constraint Programming. *Proc. of the Fourth International Conference on Algebraic Methodology and Software Technology (AMAST)*. Volume 936 of *Lecture Notes in Computer Science*, pages 531–545. Springer-Verlag, 1995.

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

[11] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I).* North-Holland, 1971.

[12] 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.

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

[14] R. Jagadeesan, V.A. Saraswat, and V. Shanbhogue. Angelic non-determinism in concurrent constraint programming. Technical report, Xerox Park, 1991.

[15] S. Lüttringhaus-Kappel. Control generation for logic programs. In D.S. Warren, editor, *Proc. Tenth Int'l Conf. on Logic Programming*, pages 478–495. MIT Press, 1993.

[16] 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.

[17] K. Marriott, M. Falaschi, M. Gabbrielli and C. Palamidessi. A simple semantics for logic programming languages with delay. In R. Kotagiri, editor, Proc. Eighteenth Australian Computer Science Conf., *Australian Computer Science Comm.* **17** (1): 356–363, 1995.

[18] K. Marriott, M. Garcia de la Banda and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. *Proc. 21st ACM Symp. Principles of Programming Languages*, pages 240–253. ACM Press, 1994.

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

[20] L. Naish. *Negation and Control in Prolog*, LNCS 238, Springer-Verlag, 1985.

[21] E.-R. Olderog. Nets, Terms and Formulas. Cambridge Tracts in Theoretical Computer Science 23, Cambridge Univ. Press, 1991.

[22] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of POPL*, pages 232–245, 1990.

[23] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of Concurrent Constraint Programming. In *Proc. of POPL*, 1991.

[24] D. Scott. Domains for denotational semantics. In *Proc. of ICALP*, 1982.

[25] D. Scott and J.W. de Bakker. A theory of programs. Technical Report Unpublished, Notes of the IBM Vienna Seminar, 1969.

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