Proving Concurrent Constraint Programs Correct

FRANK S. DE BOER
Universiteit Utrecht
MAURIZIO GABBRIELLI
Università di Pisa
ELENA MARCHIORI
Università di Venezia
and
CATUSCIA PALAMIDESSI
Università di Genova

We address the problem of proving (partial) correctness of concurrent constraint programs (ccp). We develop a compositional proof theory on the basis of a denotational approximation of the strongest postcondition semantics. This denotational semantics allows to view programs as theories in a suitable specification logic thus providing a justification of the (claimed) declarative nature of ccp. Concerning completeness, we prove that the system is (relatively) complete for the class of programs in which the denotational semantics characterizes exactly the strongest postcondition. This class includes the so-called confluent ccp, a special case of which is constraint logic programming with dynamic scheduling.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent programming; D.3.1 [**Programming languages**]: Formal definitions and Theory—*Semantics*; F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*; *Pre- and post-conditions*; F.3.2 [**Logics and meanings of programs**]: Semantics of Programming Languages—*Denotational semantics*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Constraint programming, dynamic scheduling, strongest postcondition, proof theory

1. INTRODUCTION

Concurrent constraint programming ([Saraswat 1989; Saraswat and Rinard 1990; Saraswat et al. 1991]) (ccp, for short) is a concurrent programming paradigm which derives from replacing the *store-as-valuation* concept of von Neumann computing by

Author's address: F. de Boer, Dept. of Computer Science, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands.
M. Gabbrielli, Dip. di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy.
E. Marchiori. Dipartimento di Matematica Applicata e Informatica, Università di Venezia, Italy.
C. Palamidessi, DISI, Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy.

the *store-as-constraint* model. The computational model of ccp is based on a global *store*, represented by a constraint, which expresses some partial information on the values of the variables involved in the computation. The concurrent execution of different processes, which interact through the common store, refines the partial information of the values of the variables by adding (*telling*) constraints to the store. Communication and synchronization are achieved by allowing processes to test (*ask*) if the store entails a constraint before proceeding in the computation.

One of the most characteristic features of the ccp paradigm is a formalization of the basic operations which allow to update and to query the common store, in terms of the logical notions of conjunction and entailment supported by a given underlying constraint system. An intriguing question however is to what extent the incorporation of synchronization mechanisms that are intended to describe *flow of control* features, still allows a declarative interpretation of ccp programs, i.e. to view a program as a logical theory and the output of its computations as its logical consequences.

From a purely semantical point of view, there is no clear evidence of a declarative interpretation. Indeed the semantic structures needed to give fully abstract models for the standard input/output behavior of full ccp programs ([de Boer and Palamidessi 1990]) are similar to those used for imperative languages based on asynchronous communication mechanisms ([de Boer et al. 1991]) and are essentially more complicated than those used for pure (constraint) logic languages ([Apt 1990; Jaffar and Lassez 1987]). It should be noted that for the deterministic fragment of ccp there exists an elegant denotational semantics based on closure operators defined by Saraswat et al. [1991], which Panangaden et al. [1992] show to be intimately related to the logic of constraints.

The main contribution of this paper is the introduction of a proof theory for ccp, i.e. a calculus for proving correctness of ccp programs. The issue of the design of proof systems appropriate to proving correctness of ccp programs has received little attention so far. For logic languages like Prolog the proof techniques of Hoare-Logic ([Hoare 1969]) have been applied by Colussi and Marchiori [1991] to reason about properties of the flow of control. Correctness of CLP programs has also been investigated by Colussi et al. [1995], while a process algebra for ccp has been developed by de Boer and Palamidessi [1992] along the lines of ACP ([Bergstra and Klop 1986]). The focus of this paper concerns more generally the development of calculi for the correctness of ccp programs with respect to a first-order specification of *what* the program is supposed to compute.

A proof theory for a concurrent imperative language in general relates the 'how', that is, the flow of control described by a program, to the 'what', a specification of the program in some (usually first-order) logic. To relate these two different worlds, the store-as-valuation semantics (or state-based semantics, for short) of an imperative program is lifted to a predicate transformer semantics based on the notion of the *weakest precondition* or, equivalently, the *strongest postcondition* ([Dijkstra 1976]).

In the case of ccp one may try to use the logic of the underlying constraint system, in terms of which the computational model is described, also for reasoning about the correctness of programs. However this logic is in general not appropriate for this purpose: Consider for example a constraint system which contains constraints such as $t > u$, $t = u$, $t \geq u$, where t, u are terms denoting natural numbers, and consider the following ccp agent:

$$\begin{array}{l}(ask(x = 0) \rightarrow tell(y = 1)) \\ + \\ (ask(x > 0) \rightarrow tell(y = 1))\end{array}$$

This agent adds $y = 1$ to the store in case the initial store either entails $x = 0$ or $x > 0$ (otherwise it suspends). Thus one would like to state that the above program satisfies the specification $(x = 0 \vee x > 0) \rightarrow y = 1$, i.e. every computation results in a store such that whenever $x = 0$ holds or $x > 0$ holds, then $y = 1$ is guaranteed to hold as well. However if we try to express this correctness statement at the level of the constraint system, i.e. as a transformation on constraints, then we run into the problem that the "disjunction" provided by the constraint system, $x \geq 0$, does not really represent the disjunction $x = 0 \vee x > 0$ in the way we intend it. If we add in parallel to the above agent the construct $tell(x \geq 0)$, for instance, then we can expect the resulting agent to satisfy the specification

$$(x \geq 0) \wedge ((x = 0 \vee x > 0) \rightarrow y = 1)$$

but we cannot infer that $y = 1$. In fact, the execution of $tell(x \geq 0)$ establishes the constraint $x \geq 0$, but this neither entails $x = 0$ nor $x > 0$, so the agent will suspend afterwards and $tell(y = 1)$ will not be executed. This simple example shows that one cannot reason about the non-deterministic choice in terms of the disjunction of the underlying constraint system.

To deal with the correctness of ccp programs, we have therefore to enrich the logic of the constraint system. We introduce a specification logic which describes *properties* of constraints, in particular combinations of constraints themselves via the classical logical operators. A property represented by a constraint is interpreted as the set of constraints that entail it. The logical operations of disjunction, negation etc, are then interpreted in terms of the corresponding set-theoretic operations. This definition of the specification logic allows for a correspondence between the programming constructs of ccp and logical expressions. For example, parallel composition corresponds to conjunction, action prefixing to implication, and guarded non-deterministic choice to a combination of implication and disjunction.

From a semantical point of view this correspondence derives from a notion of observables which associates to a ccp program the set of all its possible outputs (of terminating computations). This notion of observables is generally known in the imperative tradition as the *strongest postcondition* of a program given the precondition *true* (the set of all initial states). The strongest postcondition semantics of a program supports the concept of *partial correctness*: A program P is said to be partially correct with respect to a (first-order) specification ϕ iff all terminating computations of P result in a state (or constraint) satisfying ϕ, i.e. the strongest postcondition of P is contained in the meaning of ϕ.

In contrast with imperative programming, the strongest postcondition semantics is compositional wrt the parallel operator: It is just set-intersection. No need arises for additional tests for *interference freedom* ([Owicki and Gries 1976]). However, the exact denotational characterization of full ccp (with such simple semantic domain) is not possible because of the hiding operator, for which compositionality doesn't hold. More precisely, the problem arises because of the interaction between the local variables and the non-deterministic choice construct. At most we can *approximate* the the strongest postcondition semantics, in the sense of defining a denotational semantics which is (in general) a superset of it.

From this denotational approximation of the strongest postcondition semantics we derive a correct and compositional proof theory for partial correctness of ccp programs. Since in general the denotational semantic counterpart of our proof theory does not coincide with the strongest postcondition semantics the axiomatization is incomplete. However completeness holds for the class of programs for which the denotational semantics does coincide with the strongest postcondition semantics.

An interesting condition which ensures such an equivalence is that of confluence ([Falaschi et al. 1996a]). Intuitively, confluent programs are those for which the scheduling of parallel processes does not affect the observed behaviour. That is, for every process scheduling the set of input/output pairs is exactly the same. It is worth noticing that confluence does not imply determinism, in fact several non-deterministic confluent subclasses of ccp have been identified (see [Falaschi et al. 1996a]), including ccp with local non-determinism ([Jagadeesan et al. 1991]). For these sublanguages of ccp we do obtain a complete proof system. Thus we may conclude that the given axiomatization provides a declarative interpretation of the strongest postcondition semantics of confluent ccp programs. However confluence is certainly not a necessary condition for completeness: In Section 4.1 we discuss an application of our proof system to the angelic-merge program, which is a typical case of a non confluent agent.

More generally, it is worth noticing that many "real-life" programs either are confluent of they do not give rise to the problematic interaction between choice and hiding mentioned before, so for them our proof system is complete. For example, this is the case for all the programmming examples appearing the thesis of Saraswat [1989]: the only two programs which have the problematic choice are `Nqueens` and `Gcd` which, however, are confluent.

Confluent ccp subsumes deterministic ccp, ccp with local non-determinism, and, therefore, also constraint logic programming (CLP). A yet more interesting sublanguage is CLP with dynamic scheduling. In this language, the standard left to right selection rule of Prolog is replaced by a flexible selection rule which allows to dynamically delay (or suspend) the selection of atoms until the arguments are sufficiently instantiated. This dynamic scheduling mechanism has several advantages with respect to the actual practice of programming. Indeed, nearly all the existing logic programming systems admit dynamic scheduling by allowing the addition of so-called *delay declarations* to a program. These declarations are of the form

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

The intuition of such a declaration is that the selection of the atom $p(x)$ has to be delayed until the condition expressed by the formula $Condition(X)$ is verified.

The declarative meaning of CLP is well understood in terms of first-order logic theories: 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

([Apt 1993]). The logical meaning of delay declarations, on the contrary, has not been clarified yet: If we add to `P` a delay declaration such as the previous one, than what can we say about the properties satisfied by `p(x)` in the new program? 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, the proof theory defined for ccp can be applied to CLP with dynamic scheduling by translating the latter into a concurrent constraint programming language which we call δ-ccp and which is basically an extension of ccp with local choice. The extension consists of the introduction of delay conditions in the "ask" construct and enhances the expressive power while preserving the nice semantic properties of local ccp. Since δ-ccp is confluent, we obtain a complete proof system.

The remaining of this paper is organized as follows. The next Section contains some basic notions concerning concurrent constraint programming and its operational semantics. In Section 3 we formalize the notion of partial correctness by defining the language of properties in terms of constraints and by introducing partial correctness assertions of the form P *sat* ϕ, where P is a program and ϕ is a property. Validity of assertions is defined in terms of the strongest postcondition $SP(P)$ of a program P wrt the precondition *true*. In Section 4 we define the compositional proof system for proving partial correctness assertions and we prove soundness and completeness. Section 6 contains the application of the proof theory to constraint logic programs with delay declarations. Section 7 concludes with a discussion of possible extensions and further applications of our results.

Preliminary versions of parts of this paper, without proofs, appeared in the Proceedings of the Symposium on Principles of Programming Languages (POPL), Portland, USA, 1994, and in the Proceedings of the Static Analysis Symposium (SAS), Aachen, Germany, 1996.

2. PRELIMINARIES

In this section we give the basic definitions of ccp languages following Saraswat and Rinard [1990]. We refer to that paper for more details. The ccp languages are defined parametrically wrt to a given *constraint system*. The notion of constraint system has been formalized by Saraswat and Rinard [1990] following Scott's treatment of information systems ([Scott 1982]). The basic ingredients are a set of *primitive constraints* D, each expressing some partial information, and a compact entailment relation $\vdash$ defined on D. This gives basically an information system with the consistency requirement removed. Then, following the usual construction, a constraint system is obtained by considering sets of primitive constraints and by extending the entailment relation on it in such a way that the resulting structure is a complete *algebraic* lattice. This ensures the effectiveness of the extended entailment relation. Here we only consider the resulting structure.

DEFINITION 2.1. A constraint system is a complete algebraic lattice

$$\langle \mathcal{C}, \leq, \sqcup, true, false \rangle$$

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 usually refer to its inverse relation, denoted by $\vdash$ and called *entailment*. Formally, $\forall c, d \in C.\ c \vdash d \Leftrightarrow d \leq c$. 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 ([Henkin et al. 1971]). 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 $\exists_x : \mathcal{C} \rightarrow \mathcal{C}$ is defined such that for any $c, d \in \mathcal{C}$:

(i) $c \vdash \exists_x(c)$,
(ii) if $c \vdash d$ then $\exists_x(c) \vdash \exists_x(d)$,
(iii) $\exists_x(c \sqcup \exists_x(d)) = \exists_x(c) \sqcup \exists_x(d)$,
(iv) $\exists_x(\exists_y(c)) = \exists_y(\exists_x(c))$.

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

In the sequel we will identify a system $\mathbf{C}$ with its underlying set of constraints $\mathcal{C}$. In order to model parameter passing, *diagonal elements* ([Henkin et al. 1971]) are added to the primitive constraints: We assume that, for x, y ranging in *Var*, D contains the constraints d_{xy} which satisfy the following axioms.

(i) $d_{xx} = \mathit{true}$,
(ii) if $z \neq x, y$ then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
(iii) if $x \neq y$ then $d_{xy} \sqcup \exists_x(c \sqcup d_{xy}) \vdash c$.

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 following $\exists_x(c)$ is denoted 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 $fv(c) = \{x \mid \exists_x c \neq c\}$. 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.

Definition 2.3. Assuming a given cylindric constraint system $\mathbf{C}$ the syntax of the ccp *Agents* is given by the following grammar:

$$A ::= \mathit{stop} \mid \mathit{tell}(c) \mid \sum_{i=1}^{n} \mathit{ask}(c_i) \rightarrow A_i \mid A \parallel B \mid \exists x A \mid p(x)$$

where the c, c_i are supposed to be *finite constraints* (i.e. algebraic elements) in $\mathcal{C}$. A ccp *process* P is then an object of the form $D.A$, where D is a set of procedure declarations of the form $p(x) : -A$ and A is an agent. For notational convenience only we restrict to procedure calls with one parameter.

The *deterministic* agents are obtained by imposing the restriction $n = 1$ in the previous grammar. The standard operational model of ccp can be described by a transition system $T = (Conf, \longrightarrow)$. 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-R6** of Table I.

The agent *Stop* represents successful termination. The basic actions are given by $tell(c)$ and $ask(c)$ 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 $ask(c)$ represents a guard, i.e. a test on the current store d, whose execution does not modify d. We say that $ask(c)$ is *enabled* in d iff $d \vdash c$. According to rule **R2** the *guarded choice* operator gives rise to global non-determinism: the agent $\sum_{i=1}^{n} ask(c_i) \rightarrow A_i$ nondeterministically selects one $ask(c_i)$ which is enabled in the current store, and then behaves like A_i. The external environment can then affect the choice since $ask(c)$ is enabled iff the current store d entails c, and d can be modified by other agents (rule **R1**). If no guard is enabled, then the guarded choice agent *suspends*, waiting for other (parallel) agents to add information to the store. The situation in which all the components of a system of parallel agents suspend is called *global suspension* or *deadlock*. The operator $\|$ represents parallel composition which is described by rule **R3** as interleaving. The agent $\exists x A$ behaves like A, with x considered *local* to A. To describe locality in rule **R4** the syntax has been extended by an agent $\exists^d x A$ where d is a local store of A containing information on x which is hidden in the external store. Initially the local store is empty, i.e. $\exists x A = \exists^{true} x A$. Rule **R5** treats the case of a procedure call when the actual parameter differs from the formal parameter: It identifies the formal parameter as a local alias of the actual parameter. Note that the parameter-passing mechanism is call by name. For a call involving the formal parameter a simple body replacement suffices (rule **R6**).

It is worthwhile to remark here on the two different kind of existaetial operators we have introduced sofar. The first one, denoted by $\exists_x$, is the operator formalized by the axioms given in Definition 2.2. This formalization derives from the theory of cylindric algebras and it is used to define an abstract notion of constraints system. As a specific instance one can consider as constraints standard first order formulas on some suitable domain. These constraints can be used as arguments to ask and tell actions. On the other hand the second form of existential quantification is introduced in Definition 2.3 (which gives the syntax of ccp agents) and is denoted by $\exists x A$. The operational behaviour of this operator is described by rule **R4** in Table I and consists of hiding the information on x in the agent A. The difference between the two kinds of existantial operators is best explained by the following simple example.

EXAMPLE 2.4. Consider the following agents which use as constraints first-order formulas on some domain.

$$A_1 = ask(\exists_x x = a) \rightarrow tell(true)$$

and

$$A_2 = \exists x(ask(x = a) \rightarrow tell(true)$$

R1	$\langle D.tell(c), d\rangle \longrightarrow \langle D.Stop, c \sqcup d\rangle$	
R2	$\langle D.\sum_{i=1}^{n} ask(c_i) \to A_i, d\rangle \longrightarrow \langle D.A_j, d\rangle$	$j \in [1, n]$ *and* $d \vdash c_j$
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	$\dfrac{\langle D.A, d \sqcup \exists_x c\rangle \longrightarrow \langle D.B, d'\rangle}{\langle D.\exists^d xA, c\rangle \longrightarrow \langle D.\exists^{d'} xB, c \sqcup \exists_x d'\rangle}$	
R5	$\langle D.p(y), c\rangle \longrightarrow \langle D.\exists^{d_{xy}} xA, c\rangle$	$p(x) : -A \in D,\ x \neq y$
R6	$\langle D.p(x), c\rangle \longrightarrow \langle D.A, c\rangle$	$p(x) : -A \in D$

Table I. The (standard) transition system.

(Thus a acts here as a individual constant.) Since $\exists_x x = a$ equals *true* the agent A_1 will always terminate successfully whereas the agent A_2 will deadlock on the query $x = a$ on the local variable x.

3. PROPERTIES AS (SETS OF) CONSTRAINTS

In this section we formalize the concept of partial correctness of ccp programs. To this end we introduce first the notion of a property. 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 constraints, namely the set of all those constraints which satisfy the property itself. Basically properties are represented by logical combinations of constraints. However, in certain applications we need also other user-defined predicates which have an intentional meaning, defined by the user, to specify properties of the computation. Therefore we consider as basic properties also *user-defined predicates*. These are predicates introduced and specified by the user by means of a theory. We denote by U the set of user-defined properties, with typical elements $u(\overline{x}), u_1(\overline{x}) \ldots$ ($u(\bar{x})$ indicates that the predicate u involves the variables $\bar{x}$). Thus the language of properties is defined as follows.

DEFINITION 3.1. Given a constraint system $\mathcal{C}$ and a set of user-defined predicates U, the language of properties is defined by the following grammar:

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

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 $\phi \rightarrow \psi =_{\mathrm{df}} \neg\phi \vee \psi$. Given a constraint $c \in \mathcal{C}$ we denote by

$\uparrow c$ its upward closure (in $\mathcal{C}$) defined by

$$\uparrow c = \{d \mid d \in \mathcal{C} \text{ and } c \leq d\}.$$

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

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

which defines their meaning. Thus a user-defined predicate $u(\overline{x})$ 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 $c[\bar{y}/\bar{x}] \in I(u(\overline{y}))$. The logical operations of conjunction and negation are interpreted in the classical way: a constraint c satisfies a property $\phi \wedge \psi$ iff it satisfies both ϕ and ψ, a constraint c satisfies a property $\neg\phi$ iff c does not satisfy ϕ. Furthermore, a constraint c satisfies a property $\exists x \phi$ iff there exists a constraint d satisfying ϕ such that $\exists_x c = \exists_x d$. Formally, the semantics of a property is defined as follows:

Definition 3.2. Given a constraint system $\mathcal{C}$ and an interpretation I, a property ϕ will be interpreted as the set of constraints $[\![\phi]\!]_I$ which satisfy it:

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

where for a set of constraints f, we define $\boldsymbol{\exists} x(f) = \{d \mid$ there exists $c \in f.\ \exists_x c = \exists_x d\}$.

The semantics of $\neg\phi$ is justified by observing that in the lattice of properties, ordered by $\supseteq$, the meaning of the property *false* is the set $\{\mathit{false}\}$ which acts as the greatest element.

Given a cylindric constraint system $\mathcal{C}$, the set of properties on $\mathcal{C}$ can be embedded into a (lifted) constraint system as follows. Let us denote the set $\{C \cup \{false\} \mid C \in \wp(\mathcal{C})\}$ by $\wp_f(\mathcal{C})$ and the set $\uparrow d_{xy}$ by D_{xy}, for any diagonal element d_{xy} in $\mathcal{C}$. Then we have the following.

Lemma 3.3. *Given a constraint system $\mathcal{C}$, for each $c \in \mathcal{C}$, $\uparrow \exists_x c = \boldsymbol{\exists} x \uparrow c$ holds.*

Proof. Immediate. we $\square$

Theorem 3.4. *If $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, true, false, Var, \exists \rangle$ is a cylindrical constraint system, then the structure $\langle \wp_f(\mathcal{C}), \supseteq, \cap, \mathcal{C}, \{false\}, Var, \boldsymbol{\exists} \rangle$ is a cylindrical constraint system.*

Proof. Clearly $(\wp_f(\mathcal{C}), \supseteq)$ is a complete algebraic lattice. We have now to show that $\boldsymbol{\exists}$ and D_{xy} satisfy the axioms (i)-(vii) given in Definition 2.2. The cases (i), (ii), (iii) and (v) are immediate. As for the others, let C, D be generic elements in $\wp_f(\mathcal{C})$. We have:

(iv) $\exists x(\exists y C) = \exists y(\exists x C)$: Assume that $d \in \exists x(\exists y C)$. Then there exists $c \in C$ and there exists $e \in \mathcal{C}$ such that $\exists_y c = \exists_y e$ and $\exists_x e = \exists_x d$. From these equalities and Axiom (iv) it follows that

$$\exists_x(\exists_y d) = \exists_y(\exists_x c). \tag{1}$$

Consider now the constraint $d' = \exists_y d \sqcup \exists_x c$. From Equality (1) and the Axioms (iii) and (i) of the underlying constraint system, it follows that

$$\exists_x d' = \exists_x(\exists_y d) \sqcup \exists_x c = \exists_x c$$

so $d' \in \exists x C$. Moreover, analogously we have that

$$\exists_y d' = \exists_y d \sqcup \exists_y \exists_x c = \exists_y d,$$

which implies that $d \in \exists y(\exists x C)$. The reverse inclusion is analogous.

(vi) If $z \neq x, y$ then $D_{xy} = \exists z(D_{xz} \cap D_{zy})$. We have the following equalities

$$\begin{array}{ll} D_{xy} & = \text{(definition)} \\ \uparrow d_{xy} & = \text{(Axiom (vi))} \\ \uparrow \exists z(d_{xz} \sqcup d_{zy}) & = \text{(Lemma 3.3)} \\ \exists z \uparrow (d_{xz} \sqcup d_{zy}) & = \text{(set theory)} \\ \exists z(\uparrow d_{xz} \cap \uparrow d_{zy}) & = \text{(definition)} \\ \exists z(D_{xz} \cap D_{zy}). & \end{array}$$

(vii) If $x \neq y$ then $D_{xy} \cap \exists x(C \cap D_{xy}) \subseteq C$. Consider a generic $d \in D_{xy} \cap \exists x(C \cap D_{xy})$. From the definitions of D_{xy} and $\exists$ it follows that $d \vdash d_{xy}$ and there exists $c \in C$ such that $c \vdash d_{xy}$ and $\exists_x c = \exists_x d$. Axiom (vii) implies that $d_{xy} \sqcup \exists_x(c \sqcup d_{xy}) \vdash c$. Since $c \vdash d_{xy}$, also $c \vdash d_{xy} \sqcup \exists_x(c \sqcup d_{xy})$ holds and therefore we have

$$c = d_{xy} \sqcup \exists_x c.$$

Analogously we have

$$d = d_{xy} \sqcup \exists_x d.$$

From these equalities and the assumption $\exists_x c = \exists_x d$ it follows that $c = d$, thus $d \in C$, which completes the proof.

□

Thus, the previous theorem shows that any property can be seen as an element of the lifted constraint system $\wp_f(\mathcal{C})$ (as usual, we identify the lifted constraint system with its underlying set of constraints $\wp_f(\mathcal{C})$). This fact justifies the use of the system of properties as a specification *logic* of ccp programs. Moreover it will be used later on when considering a ccp language where properties are arguments to ask agents.

We define a property to be valid iff every constraint satisfies it:

Definition 3.5. Given a constraint system $\mathcal{C}$ and an interpretation I, a property ϕ is called valid, notation $I \models \phi$, iff every constraint satisfies the property, i.e. $[\![\phi]\!]_I = \mathcal{C}$.

Let us analyze the relation between the logic of 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 $(c \vee d) \rightarrow (c \sqcap d)$ 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 $x \neq 0 \rightarrow \neg x = 0$ 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 $x \neq 0$. Observe that by Lemma 3.3 we do have that $\exists x c \leftrightarrow \exists_x c$.

The notion of logical consequence is defined as follows.

Definition 3.6. Given a constraint system $\mathcal{C}$, a *theory* T is a set of properties. A *model* of T is any interpretation I which satisfies all the properties of T, i.e. $I \models \phi$ for each $\phi \in T$. We say that ϕ is a *logical consequence* of T, notation $T \models \phi$, if every model of T is also a model of $\{\phi\}$.

We can now introduce partial correctness assertions and their semantics.

Definition 3.7. Given some underlying constraint system $\mathcal{C}$ and a theory T, partial correctness *assertions* are of the form $P\ sat\ \phi$ where P is a process and ϕ is a property. Given a theory T of the user-defined properties, the semantics of an assertion $P\ sat\ \phi$, with P closed (namely, every procedure occurring in P is declared), is given as follows:

$$T \models P\ sat\ \phi \quad \text{iff} \quad \text{for every } I \text{ model of } T, SP(P) \subseteq [\![\phi]\!]_I \text{ holds.}$$

where

$$SP(P) =_{def} \{d \in \mathcal{C} \mid \text{there exist } c \in \mathcal{C} \text{ and } Q \in Agents \text{ s.t. } \langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow\}$$

Intuitively $T \models P\ sat\ \phi$ holds iff every terminating computation of P (for any input c) results in a constraint d which satisfies the property ϕ under any model of T. The set $SP(P)$ actually describes the *strongest postcondition* of the program P with respect to the precondition *true* (every constraint satisfies the property *true*). The final store of a terminating computation is often called a *resting point* because, due to the monotonicity of the tell operation and the fact that once an ask operation is enabled it cannot be disabled, we have: $\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow$ iff there exists a derivation $\langle P, d\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow$. So $SP(P)$ can equivalently be characterized as follows:

Proposition 3.8. *For any closed process P, $SP(P) = \{d \in \mathcal{C} \mid$ there exists $Q \in Agents$ s.t. $\langle P, d\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow\}$.*

Proof. Let us denote $\{d \in \mathcal{C} \mid \text{there exists } Q \in Agents \text{ s.t. } \langle P, d\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow\}$ by $Re(P)$. The inclusion $SP(P) \supseteq Re(P)$ is obvious. For the other one, consider $d \in SP(P)$. Then there exists a derivation $\xi : \langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow$. Since the store in a computation increases monotonically, we have $d \vdash c_i$ for each c_i

which appears in ξ. Moreover, according to rule R2 in Table I, if a guard $ask(c)$ is enabled in the store c_i and $d \vdash c_i$ then $ask(c)$ is enabled also in the store d. Therefore, given the initial configuration $\langle P, d\rangle$, exactly the same steps of the derivation ξ can be repeated and we have a derivation $\langle P, d\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow$. Thus $d \in Re(P)$, which completes the proof. $\square$

4. A CALCULUS FOR CCP

In order to obtain a calculus for partial correctness assertions, we first introduce a compositional semantics for ccp which approximates the operational semantics $SP(P)$. Technically such a denotational semantics is used to prove soundness and completeness of the calculus. More interestingly, it turns out that we can obtain the rules for the proof system by simply "mirroring" the equations of the denotational semantics. This is due to the fact that the operators of the language are modeled in these equations by simple set theoretic notions (e.g. parallel composition as intersection) which in the specification logic can be replaced by the corresponding logical notions (e.g. intersection by conjunction).

4.1 Denotational approximation of the strongest postcondition semantics

Let $Pvar$ denote the set of procedure identifiers and let $Env = Pvar \rightarrow \wp_f(\mathcal{C})$, with typical element e, be the set of *environments*. Given a program P, its denotational semantics $[\![P]\!] : Env \rightarrow \wp_f(\mathcal{C})$ is defined by the equations in Table II where μ denotes the least fixpoint wrt subset inclusion of elements of $\wp_f(\mathcal{C})$. In equations **D5** and **D6** we implicitly assume that with each procedure identifier p there is associated a unique variable x, its formal parameter (so any declaration of p is of the form $p(x) : -A$, for some A).

The first two equations of Table II state that the agents $stop$ and $tell(c)$ have as resting points all the constraints and all the constraints which entail c, respectively. Equation **D2** states that d is a resting point of a guarded choice agent $\sum_{i=1}^{n} ask(c_i) \rightarrow A_i$ if either it enables the guard $ask(c_i)$ and additionally it is a resting point of A_i, or it does not enable any guard (and hence the agent suspends). Equation **D3** is based on a simple semantic property: d is a resting point of $A \parallel B$ iff d is a resting point of both A and B. According to equation **D4**, the resting points of the agent $\exists x A$ are all those constraints which are equal to resting points of A up to the information on x. Finally recursion is modeled, as usual, by a least fixpoint construction. In rule **D5** the variable y is assumed to be different from the formal parameter and in rule **D6** on the other hand x is assumed to be the formal parameter. If P is a closed program, i.e. a program where all procedures names are defined, then $[\![P]\!](e)$ does not depend on e and will be indicated as $[\![P]\!]$.

The following theorem, whose proof can be found in [de Boer et al. 1995], shows that this denotational semantics approximates the strongest postcondition semantics, in the sense that it provides a superset of it.

THEOREM 4.1. *If P is a closed program then $SP(P) \subseteq [\![P]\!]$.*

The reverse inclusion does not hold in general. The problem is related to the hiding operator, as shown by the following Example 4.2.

Here and in the following examples we use the (usual) Herbrand constraint system $\mathcal{H}$ defined as follows. Consider the terms $t_1, t_2 \ldots$ on a given signature (using a set

D0	$[\![D.stop]\!](e) = \mathcal{C}$
D1	$[\![D.tell(c)]\!](e) = \uparrow c$
D2	$[\![D.\sum_i ask(c_i) \rightarrow A_i]\!](e) = \bigcap_i(\mathcal{C} \setminus \uparrow c_i) \cup \bigcup_i(\uparrow c_i \cap [\![D.A_i]\!](e))$
D3	$[\![D.A \parallel B]\!](e) = [\![D.A]\!](e) \cap [\![D.B]\!](e)$
D4	$[\![D.\exists x A]\!](e) = \exists\!\!\exists x[\![A]\!](e)$
D5	$[\![D.p(y)]\!](e) = [\![D.\exists x(p(x) \parallel tell(d_{xy}))]\!](e)$
D6	$[\![D.p(x)]\!](e) = e(p),\ p \notin D$
D7	$[\![D.p(x)]\!](e) = \mu\Psi$ where $\Psi(f) = [\![D \setminus \{p\}.A]\!](e\{f/p\}),\ p(x) : -A \in D$

Table II. Strongest postcondition semantics of CCP

Var of variables). Let $\mathcal{E}$ be the least set such that

—for any pair of terms t, t', $t = t' \in \mathcal{E}$,

—if $e \in \mathcal{E}$ then $\exists x.e \in \mathcal{E}$, $x \in Var$,

—if $e, e' \in \mathcal{E}$ then $e \sqcup e' \in \mathcal{E}$.

$\mathcal{H}$ is then the (cylindric) constraint system whose elements are those in $\mathcal{E}$ modulo logical equivalence with ordering $[e] \leq [e']$ iff $e' \models e$. The operations are the obvious ones, i.e. $\sqcup$ is the logical conjunction and $\exists_x$ is the existential quantifier while the diagonal element d_{xy} corresponds to the equation $x = y$.

Example 4.2. Consider the following agent A

$$\begin{array}{l}(ask(x = a) \rightarrow tell(true))\\ +\\ (ask(true) \rightarrow tell(y = b))\end{array}$$

defined on the constraint system $\mathcal{H}$. It is easy to see that $SP(A) = [\![A]\!] = \uparrow(x = a) \cup \uparrow(y = b)$, but $[\![\exists x A]\!] = \exists\!\!\exists x(\uparrow(x = a) \cup \uparrow(y = b)) = \mathcal{C} \neq SP(\exists x A) = \uparrow(y = b)$.

The above example illustrates that $\exists\!\!\exists$ is not the denotational correspondent of $\exists$ wrt the strongest postcondition observables. In fact it is not possible to express denotationally the $\exists$ operator in this semantic domain. In other words, the strongest postcondition semantics is not compositional wrt the hiding operator, as shown by the following example.

EXAMPLE 4.3. Consider the agent B

$$\begin{array}{l}(ask(true) \rightarrow tell(x=a))\\ +\\ (ask(true) \rightarrow tell(y=b))\end{array}$$

defined on $\mathcal{H}$. It is easy to see that $SP(B) = \uparrow(x=a) \cup \uparrow(y=b)$. Hence $SP(B) = SP(A)$ where A is the agent in Example 4.2. However, $SP(\exists_x B) = \mathcal{C} \neq SP(\exists x A)$.

The reverse of Theorem 4.1 holds however for some sublanguages of ccp. One example is deterministic ccp ([Saraswat et al. 1991]): when we restrict to this sublanguage, in fact, it is easy to see that $[\![P]\!]$ becomes the denotational semantics defined by Saraswat et al. [1991], which associates with each deterministic agent its set of resting points and which is a fully abstract characterization of its input/output behavior.

A more general example is (non-deterministic) confluent ccp ([Falaschi et al. 1996a]). As mentioned in the introduction, confluent ccp is a subclass of ccp for which independence from the scheduling of parallel process holds. More precisely, an agent A is called (structurally) confluent if, for each sub-agent A' of A, the choice of different schedulings for parallel processes does not affect the results of computations of the agent $A' \parallel tell(c)$, for all the constraints c. For confluent ccp we have then following result proved by Falaschi et al. [1996a].

PROPOSITION 4.4. *For any confluent (closed) process $D.A$ we have $SP(D.A) = [\![D.A]\!]$.*

Intuitively, the main cause of non-confluency, in a ccp program, is the presence of a global-choice operator. Consider for example the agent $tell(x=a) \parallel C$ defined on the Herbrand constraint system $\mathcal{H}$, where C is the agent

$$\begin{array}{l}(ask(true) \rightarrow tell(true))\\ +\\ (ask(x=a) \rightarrow tell(y=b))\end{array}$$

Given the input constraint $true$, clearly if one schedules the agent $tell(x=a)$ first then the result $x=a \wedge y=b$ can be obtained for the agent $tell(x=a) \parallel C$. This is not the case if one schedules the agent C first, since in this case case the input $true$ forces the computation to select the first branch of C. Note that the above problem disappears if one considers $+$ as a local choice, since in this case the second branch of the agent C could be selected also in presence of the input constraint $true$. In fact, ccp with local choice is easily seen to be a confluent language ([Falaschi et al. 1996a]).

4.2 The proof system

Given a theory T of user-defined properties, we define a calculus for assertions $P\ sat\ \phi$ using the standard natural deduction style. The calculus, illustrated in Table III, is obtained essentially by a description in the logic of properties of the right-hand sides of the semantic equations of Table II.

From equation **D0** and the fact that the property $true$ describes the set $\mathcal{C}$ we derive the axiom **C0** which expresses that the agent $stop$ satisfies the weakest property $true$. From equation **D1** and the fact that the property c describes the

C0	$D.stop\ sat\ true$	
C1	$D.tell(c)\ sat\ c$	
C2	$\dfrac{D.A_i\ sat\ \phi_i \quad \forall\, i \in \{1,\ldots,n\}}{D.\sum_i ask(c_i) \rightarrow A_i\ sat\ \bigwedge_i \neg c_i \vee \bigvee_i (c_i \wedge \phi_i)}$	
C3	$\dfrac{D.A\ sat\ \phi \quad D.B\ sat\ \psi}{D.A \parallel B\ sat\ \phi \wedge \psi}$	
C4	$\dfrac{D.A\ sat\ \phi}{D.\exists x A\ sat\ \exists x \phi}$	
C5	$\dfrac{D \setminus \{p\}.p(x)\ sat\ \phi \vdash D \setminus \{p\}.A\ sat\ \phi}{D.p(x)\ sat\ \phi}$	$p(x) : -A \in D$
C6	$\dfrac{D.\exists x(p(x) \parallel tell(x = y))\ sat\ \phi}{D.p(y)\ sat\ \phi}$	
C7	$\dfrac{P\ sat\ \phi \quad T \models \phi \rightarrow \psi}{P\ sat\ \psi}$	

Table III. A calculus for CCP

set $\uparrow c$ we derive the axiom **C1** which expresses that the agent $tell(c)$ satisfies the property c.

The rule for non-deterministic choice (**C2**) can be justified by considering equation **D2**: A resting point of $\sum_i ask(c_i) \rightarrow A_i$ either does not entail any of the asked constraints c_i, in which case it satisfies the property $\bigwedge_i \neg c_i$, or it entails c_i and is a resting point of A_i, so by the premise it satisfies ϕ_i and thus it will satisfy $c_i \wedge \phi_i$. Note that for deterministic agents, rule **C2** reduces to

$$\frac{D.A\ sat\ \phi}{D.ask(c) \rightarrow A\ sat\ (c \rightarrow \phi)}$$

since $(c \wedge \phi) \vee \neg c$ is logically equivalent to $c \rightarrow \phi$.

The rules for parallel composition and hiding can be obtained in a similar way from the corresponding equations. Note that $\exists x$ both in the languages of properties and agents has the same interpretation.

Reasoning about recursion is formalized in terms of a meta-rule (Scott-induction [Scott and de Bakker 1969]) which allows to conclude that the agent $p(x)$ satisfies a property ϕ whenever the body of $p(x)$ satisfies the same property assuming the conclusion of the rule. Derivibality is denoted by $\vdash$. Given a (possibly empty) set of correctness statements Φ, $\Phi \vdash P\ sat\ \phi$ stands for the existence of a proof of the correctness statement $P\ sat\ \phi$ using the axioms and rules of the proof system with additional axioms the statements of Φ.

Finally, we have the consequence rule **C7** which states that if the program P satisfies ϕ and ϕ implies ψ in the logical theory T then P satisfies ψ. We observe that thus we include (like in all Hoare-style proof systems) as additional axioms all the logical consequences of T (which includes the set of valid properties).

We investigate now soundness and completeness of the above calculus. Soundness means that every provable correctness assertion is valid: whenever $\vdash P\ sat\ \phi$, i.e. $P\ sat\ \phi$ is derivable (given some theory T) then $T \models P\ sat\ \phi$. Completeness on the other hand consists in the derivability of every valid correctness assertion: whenever $T \models P\ sat\ \phi$ then $\vdash P\ sat\ \phi$ (in T).

In order to prove the soundness of the calculus we have to interpret also correctness assertions about arbitrary programs (that is, including programs which do contain undefined procedure variables).

DEFINITION 4.5. Given a underlying constraint system $\mathcal{C}$, a theory T and an environment e, we define $T \models_e P\ sat\ \phi$ iff $[\![P]\!](e) \subseteq [\![\phi]\!]_I$, for any I which satisfies T.

We have the following result:

LEMMA 4.6. *Given a underlying constraint system $\mathcal{C}$ and a theory T, whenever $P_1\ sat\ \phi_1, \ldots, P_n\ sat\ \phi_n \vdash P\ sat\ \phi$ and $T \models_e P_i\ sat\ \phi_i$, for $i = 1, \ldots, n$, then $T \models_e P\ sat\ \phi$.*

PROOF. Induction on the length of the derivation. We treat the case when the last rule applied is the recursion rule (the other cases follow immediately from the close correspondence between the proof rules and the semantic equations of the denotational semantics). Since the proof $D \setminus \{p\}.p(x)\ sat\ \phi \vdash D \setminus \{p\}.A\ sat\ \phi$ is shorter than the current one the induction hypothesis says that for every environment e such that $T \models_e D \setminus \{p\}.p(x)\ sat\ \phi$ we also have that $T \models_e D \setminus \{p\}.A\ sat\ \phi$. Let us take a particular e. We have to show that $T \models_e D.p(x)\ sat\ \phi$, or, in other words that $[\![D.p(x)]\!](e) \subseteq [\![\phi]\!]_I$, for any I model of T. Now $[\![D.p(x)]\!](e) = \mu\Psi$, where $\mu\Psi = \bigcup_i f_i$, with $f_0 = \{\mathit{false}\}$ and $f_{i+1} = [\![D \setminus \{p\}.A]\!](e\{f_i/p\})$. Thus it suffices to prove by induction that for all n, $f_n \subseteq [\![\phi]\!]_I$, for any I model of T. The base case is obvious. Suppose that $f_n \subseteq [\![\phi]\!]_I$, for any I model of T. So we have for $e' = e\{f_n/p\}$ that $T \models_{e'} D \setminus \{p\}.p(x)\ sat\ \phi$, and thus we infer that $T \models_{e'} D \setminus \{p\}.A\ sat\ \phi$. In other words, $f_{n+1} = [\![D \setminus \{p\}.A]\!](e\{f_n/p\}) \subseteq [\![\phi]\!]_I$, for any I model of T. □

Since for any closed program $SP(P) \subseteq [\![P]\!](e)$ holds, with e arbitrary, whenever $T \models_e P\ sat\ \phi$ we also have $T \models P\ sat\ \phi$. Hence from the above result, with $n = 0$, we can derive immediately the soundness of the system:

THEOREM SOUNDNESS. *The proof system consisting of the rules* **C0-C7** *is sound. More precisely, given a closed program P and a theory T, if we can derive the assertion $P\ sat\ \phi$, then $T \models P\ sat\ \phi$ holds.*

We consider the standard notion of completeness for Hoare-style proof systems as introduced by Cook [1978] which is also known as relative completeness: Namely, we assume the expressibility of the strongest postcondition of a process P, i.e. the existence of a property ϕ such that $SP(P) = [\![\phi]\!]_I$ for every model I of T, and moreover we assume as additional axioms all the logical consequences of the given theory T (for use in the consequence rule).

However, since $[\![P]\!]$ does not always coincide with $SP(P)$, the proof system is in general incomplete. The following is a counterexample.

EXAMPLE 4.8. Consider the agent A of Example 4.2. It is easy to see that the strongest property we can derive about A is $x = a \vee y = b$, and thus the strongest property we can derive about $\exists xA$ is the property $\exists x(x = a \vee y = b)$ which is equivalent to *true*. On the other hand, the agent $\exists xA$ satisfies the property $y = b$ which thus cannot be derived.

However we can prove completeness for the class of ccp programs P for which $[\![P]\!] = SP(P)$ does hold. In the following we will identify $SP(P)$ with its defining property. We will show afterwards how to express in the logic of properties $SP(P)$ for a arbitrary ccp program P.

To prove completeness we need the following lemma:

LEMMA 4.9. *Let D be a set of declarations and A an agent which involves only calls to procedures declared in D such that $[\![D.B]\!] = SP(D.B)$, for any subagent B of A. Then we have (assuming some theory T):*

$$\Phi_1, \ldots, \Phi_n \vdash A \ sat \ SP(D.A) \text{ (in } T).$$

where, for $i = 1, \ldots, n$, $\Phi_i = p_i(x_i) \ sat \ SP(D.p_i(x_i))$.

PROOF. The proof proceeds by induction on the complexity of A. The symbol I in the following indicates any model of T. We treat the following two cases, the remaining ones are similar.

$A = A_1 \parallel A_2$. By the induction hypothesis we have that

$$\Phi_1, \ldots, \Phi_n \vdash A_i \ sat \ SP(D.A_i),$$

for $i = 1, 2$. So by rule **C3** we obtain that

$$\Phi_1, \ldots, \Phi_n \vdash A \ sat \ SP(D.A_1) \wedge SP(D.A_2).$$

Since

$$[\![SP(D.A)]\!]_I = [\![D.A]\!] = [\![D.A_1]\!] \cap [\![D.A_2]\!] = [\![SP(D.A_1) \wedge SP(D.A_2)]\!]_I,$$

we derive by **C7** that

$$\Phi_1, \ldots, \Phi_n \vdash A \ sat \ SP(D.A).$$

$A = p(y)$. Observe that

$$\Phi_1, \ldots, \Phi_n \vdash p(x) \ sat \ SP(D.p(x))$$

and

$$\Phi_1, \ldots, \Phi_n \vdash tell(d_{xy}) \ sat \ d_{xy},$$

thus by an application of rule **C3** we obtain that

$$\Phi_1, \ldots, \Phi_n \vdash p(x) \parallel tell(d_{xy}) \ sat \ SP(D.p(x)) \wedge d_{xy}.$$

An application of **C4** gives us

$$\Phi_1, \ldots, \Phi_n \vdash \exists x(p(x) \parallel tell(d_{xy}))\ sat\ \exists x(SP(D.p(x)) \wedge d_{xy}).$$

Observe now that

$$[\![SP(D.p(y))]\!]_I = [\![D.p(y)]\!] = \exists x[\![D.p(x) \parallel tell(d_{xy})]\!] = [\![\exists x(SP(D.p(x)) \cap [\![d_{xy}]\!])]\!]_I.$$

Hence, using **C7**, we obtain

$$\Phi_1, \ldots, \Phi_n \vdash \exists x(p(x) \parallel tell(x = y))\ sat\ SP(D.p(y).$$

Finally an application of **C6** gives us the desired result

$$\Phi_1, \ldots, \Phi_n \vdash p(y)\ sat\ SP(D.p(y)).$$

□

From the above lemma we derive the following corollary:

Corollary Completeness. *Let $D = \{p_1(x_1)\ :-\ A_1, \ldots, p_n(x_n)\ :-\ A_n\}$ and A_{n+1} be an agent such that A_i, for $1 \leq i \leq n+1$, only involves calls to procedures declared in D and $[\![D.B]\!] = SP(D.B)$, for any subagent B of A_i. It follows that $T \models D.A_{n+1}\ sat\ \phi$ implies $\vdash D.A_{n+1}\ sat\ \phi$ (in T).*

Proof. By the above lemma we have for $i = 1, \ldots, n$ that

$$\Phi_1, \ldots, \Phi_n \vdash A_i\ sat\ SP(D.p(x_i))$$

where $\Phi_i = p_i(x_i)\ sat\ SP(D.p_i(x_i))$ (note that $SP(D.A_i) = SP(D.p_i(x_i))$). Now a repeated application of the recursion rule gives us $\vdash D.p_i(x_i)\ sat\ SP(D.p_i(x_i))$. Again by the above lemma we have that

$$\Phi_1, \ldots, \Phi_n \vdash A_{n+1}\ sat\ SP(D.A_{n+1})$$

It follows by a straightforward induction on the length of the derivation (which does not involve applications of the recursion rule) that

$$\Phi'_1, \ldots, \Phi'_n \vdash D.A_{n+1}\ sat\ SP(D.A_{n+1})$$

where $\Phi'_i = D.p_i(x_i)\ sat\ SP(D.p_i(x_i))$, $i = 1, \ldots, n$. So, since $\vdash \Phi'_i$, $i = 1, \ldots, n$, we thus derive that $\vdash D.A_{n+1}\ sat\ SP(D.A_{n+1})$. Let $T \models D.A_{n+1}\ sat\ \phi$, that is, $[\![SP(D.A_{n+1})]\!]_I \subseteq [\![\phi]\!]_I$, for every I model of T, which is equivalent to $T \models SP(D.A_{n+1}) \rightarrow \phi$. Thus an application of **C7** gives us $\vdash D.A_{n+1}\ sat\ \phi$ (in T). □

Note that (if there exists a ϕ such that $SP(P) = [\![\phi]\!]$) the rules **C0-C6** give a calculus for the strongest postcondition of P wrt true, and rule **C7** allows to obtain weaker properties.

The strongest postcondition has been used to prove completeness. However, in general, to prove a property of a program, it is not necessary to deal with the strongest postcondition of the components: It is often sufficient to deal with some simpler property. The situation can be compared to the problem of finding the suitable invariant when using the standard Hoare systems for imperative programming.

4.3 Expressing the strongest postcondition of a program

In general $SP(P)$ can be expressed by enriching the logic of properties with a least fixpoint operator. We extend the syntax of properties with the following rules

$$\phi ::= p(x) \mid \mu p(x).\phi$$

In $\mu p(x).\phi$ the variable x denotes the formal parameter uniquely associated with p (see also Section 4.1). The semantics of a property of this kind is defined wrt an environment:

$$\begin{array}{lcl} [\![p(x)]\!]_I(e) & = & e(p) \\ [\![p(y)]\!]_I(e) & = & [\![\exists x(p(x) \wedge d_{xy})]\!]_I(e) \\ [\![\mu p(x).\phi]\!]_I(e) & = & \mu\Phi, \text{ where } \Phi(f) = [\![\phi]\!]_I(e\{f/p\}) \end{array}$$

(Again, it is implicitly assumed that in the above first equation x denotes the unique formal parameter associated with p, whereas in the second equation y is assumed to be distinct from x.) Note that in the third equation we use the same symbol for both the syntactic recursion-operator and its semantic counterpart. In order for the third equation to be well-defined we have to impose that p occurs only positively in ϕ, as a consequence Φ can then be shown to be a monotonic operator on the lattice $\wp_f(\mathcal{C})$, where $\mathcal{C}$ denotes the given underlying constraint system, and thus we can apply the Knaster-Tarksi Theorem.

We can construct now in a straightforward manner a logical description of $[\![D.A]\!]$ inductively along the lines of the defining equations of the semantics $[\![\cdot]\!]$:

$$\begin{array}{lcl} SP(D.stop) & = & true \\ SP(D.tell(c)) & = & c \\ SP(D.\sum_i ask(c_i) \rightarrow A_i) & = & \bigwedge_i \neg c_i \vee \bigvee_i (c_i \wedge SP(D.A_i)) \\ SP(D.A \parallel B) & = & SP(D.A) \wedge SP(D.B) \\ SP(D.\exists x A) & = & \exists x SP(D.A) \\ SP(D.p(y)) & = & \exists x(SP(D.p(x)) \wedge d_{xy}) \\ SP(D.p(x)) & = & p(x),\ p \notin D \\ SP(D.p(x)) & = & \mu p(x).SP(D \setminus \{p\}.A),\ p(x) : -A \in D \end{array}$$

Thus we have identified ccp as the following subclass of positive (recursively defined) properties:

$$\phi ::= c \mid \neg c \mid p(x) \mid \phi \wedge \psi \mid \phi \vee \psi \mid \exists x \phi \mid \mu p(x).\phi$$

In this logic reasoning about the fixpoint operator is formalized in terms of the following well-known deduction rule (Scott induction rule):

$$\frac{\psi[\phi/p] \rightarrow \phi}{\mu p(x).\psi \rightarrow \phi}$$

The substitution $[\phi/p]$ applied to $p(x)$ results into ϕ, and applied to $p(y)$, y distinct from x, into $\exists x(\phi \wedge d_{xy})$.

4.4 An example

We illustrate now our proof system by showing its application to the proof of partial correctness for the following *angelic merge* ([Panangaden and Shanbhogue 1992]) program. It is worth noticing that this is a typical non-confluent program (see

[Falaschi et al. 1996a]). However, as shown below, we are able to prove a property which describes the strongest postcondition. Therefore, because of rule **C7**, the proof system allows us to derive any valid property, i.e. it is complete on this program.

Let L be the signature consisting of the list-constructors $[\,]$, the empty list, and $[\cdot \mid \cdot]$, the prefixing operator, that is, $[a \mid x]$ denotes the list composed of the element a followed by the list x. We assume here that the constraint system is $\mathcal{H}$ (the Herbrand constraint described in subsection 4.1) defined on the signature L.

The agent $m(x, y, z)$ defined below non-deterministically merges two input lists x and y into an output list z.

$$
\begin{array}{l}
m(x, y, z) : - \\
\\
ask(\exists_{x'} \exists_a (x = [a|x'])) \rightarrow \exists x', z', a(tell(x = [a|x']) \parallel tell(z = [a|z']) \parallel m(x', y, z')) \\
+ \\
ask(\exists_{y'} \exists_a (y = [a|y'])) \rightarrow \exists y', z', a(tell(y = [a|y']) \parallel tell(z = [a|z']) \parallel m(x, y', z)) \\
+ \\
ask(x = [\,] \sqcup y = [\,]) \rightarrow tell(z = [\,])
\end{array}
$$

In order to specify the correctness of the above merge program we introduce the properties $x \mid y = z$, which holds whenever the list z is a merge of the lists x and y, and $x \lfloor y = z$, which holds whenever z is a merge of the lists x and y which starts with an element of x. These properties are defined axiomatically by the theory T consisting of the following two axioms:

$$(x \lfloor y = z \vee y \lfloor x = z \vee (x = y \wedge y = z \wedge z = [\,])) \rightarrow x \mid y = z$$

and

$$\exists x', z', a(x = [a|x'] \wedge z' = x' \mid y \wedge z = [a|z']) \rightarrow x \lfloor y = z$$

We will now prove in the calculus that $T \models D.m(x, y, z)\ sat\ x \parallel y = z$, where D denotes the above declaration of the procedure m.
By the recursion rule it is sufficient to show that

$$m(x, y, z)\ sat\ x \mid y = z \vdash A\ sat\ x \mid y = z$$

where A denotes the body of m (as declared in D). We proceed as follows: By assumption we have that $m(x, y, z)\ sat\ x \parallel y = z$. Repeated use of the rules **C1** and **C3** gives:

$$m(x, y, z) \parallel tell(x = x') \parallel tell(z = z')\ sat\ x \parallel y = z \wedge x = x' \wedge z = z'.$$

Next we apply **C4**:

$$\exists x, z(m(x, y, z) \parallel tell(x = x') \parallel tell(z = z'))\ sat\ \exists x, z(x \mid y = z \wedge x = x' \wedge z = z').$$

Since $T \models \exists x, z(x \mid y = z \wedge x = x' \wedge z = z') \rightarrow x' \mid y = z'$ we derive by **C7**:

$$\exists x, z(m(x, y, z) \parallel tell(x = x') \parallel tell(z = z'))\ sat\ x' \mid y = z'.$$

Next we apply **C6**:

$$m(x', y, z')\ sat\ x' \mid y = z'.$$

Repeated use of the rules **C1** and **C3** gives:

$tell(x = [a|x']) \parallel tell(z = [a|z']) \parallel m(x', y, z)\ sat\ x = [a|x'] \wedge z = [a|z'] \wedge x' \mid y = z.$

Next we apply **C4**:

$\exists x', z', a(tell(x = [a|x']) \parallel tell(z = [a|z']) \parallel m(x', y, z))\ sat\ \exists x', z', a(x = [a|x'] \wedge z = [a|z'] \wedge x' \mid y = z).$

An application of **C7** using T gives:

$$\exists x', z', a(tell(x = [a|x']) \parallel tell(z = [a|z']) \parallel m(x', y, z))\ sat\ x\lfloor y = z.$$

Similarly we can derive

$$\exists y', z', a(tell(y = [a|y']) \parallel tell(z = [a|z']) \parallel m(x, y', z'))\ sat\ y\lfloor x = z.$$

An application of **C2** and then an application of **C7** (assuming the validity of $\forall x(x = [\,] \vee \exists x', a(x = [a|x'])))$ gives:

$A\ sat\ (\exists x', a(x = [a|x']) \wedge x\lfloor y = z) \vee (\exists y', a(y = [a|y']) \wedge y\lfloor x = z) \vee (x = [\,] \wedge y = [\,] \wedge z = [\,]).$

By an application of **C7** using T: $A\ sat\ x \mid y = z$.

4.5 The case of local non-determinism

We show now how the previous proof system can be applied to ccp with local (or internal) non-determinism (called also angelic ccp by Jagadeesan et al. [1991]). This sub-language is obtained from the one given in Definition 2.3 by replacing the global-choice agent $\sum_{i=1}^{n} ask(c_i) \rightarrow A_i$ for agents of the form

$$ask(c) \rightarrow A$$

and

$$A_1 \oplus A_2$$

where the operator $\oplus$ indicates local-choice. Operationally, the agent $ask(c) \rightarrow A$ behaves as A if the current store entails c and suspends otherwise. The agent $A_1 \oplus A_2$ nondeterministically selects either A_1 or A_2 and then behaves as the selected agent. Formally, we consider $ask(c_1) \rightarrow A_1$ and $A_1 \oplus A_2$ as shorthands for $\sum_{i=1}^{1} ask(c_i) \rightarrow A_i$ and $\sum_{i=1}^{2} ask(true) \rightarrow A_i$, respectively. Therefore the operational semantics is still obtained from the transition system defined in Table I.

Note that, as the name indicates, in ccp with local choice the external environment cannot control the choice any more, since also branches whose guards are not enabled can be selected. This implies that global choice ccp is more expressive than local ccp. On the other hand, as previously mentioned, this ensures that the strongest postcondition can be characterized precisely by the denotational semantics. In fact, since local-choice ccp is confluent ([Falaschi et al. 1996a]), we have the following corollary of Proposition 4.4. A direct proof of this result can be also found in [de Boer et al. 1995].

COROLLARY 4.11. *For any local-ccp process $D.A$ we have $SP(D.A) = [\![D.A]\!]$.*

It is worth noticing that the denotational semantics of local ccp coincides with the one introduced by Jagadeesan et al. [1991]. In fact, obviously when considering local choice the equation **D2** can be simplified into the two equations shown in Table IV, which were used by Jagadeesan et al. [1991] [1]. This semantics was also used by Falaschi et al. [1993] as a semantic basis for the static analysis of ccp programs.

The proof system defined in Table III can be used also for local ccp, being this a sublanguage of global ccp. Aanalogously to the previous case, also rule **C2** can be simplified into the rules **Cl1** and **Cl2** given in Table V[2].

Soundness and (relative) completeness of the calculus follows then from the results of the previous section. It is worth noticing that, as a consequence of the Completeness corollaries 4.11 and 4.10 (Completeness), for local ccp we have a complete proof system.

Dl1	$[\![D.ask(c) \rightarrow A]\!](e) = (\mathcal{C} \setminus \uparrow c) \cup [\![D.A]\!](e))$
Dl2	$[\![D.A \oplus B]\!](e) = [\![D.A]\!](e) \cup [\![D.B]\!](e)$

Table IV. The equations for local non-determinism

Cl1	$\dfrac{D.A\ sat\ \phi}{D.ask(c) \rightarrow A\ sat\ c \rightarrow \phi}$
Cl2	$\dfrac{D.A\ sat\ \phi \quad D.B\ sat\ \psi}{D.A \oplus B\ sat\ \phi \vee \psi}$

Table V. The rules for local non-determinism

5. THE TRANSFORMATIONAL DESIGN OF CCP PROGRAMS

In this section we show how the programming logic of ccp can be used for the design of ccp 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 it is convenient to introduce the formalism of *mixed terms* ([Olderog 1991]), i.e. terms that are constructed out of programs

[1] The equivalence of the two formulations follows immediately from set theoretic considerations and the definition of $\rightarrow$ and $\oplus$ given above.

[2] As before, the equivalence of the two formulations is immediate, since from the definition of $\models$ it follows that $\models (\neg c \vee (c \wedge \phi)) \leftrightarrow (c \rightarrow \phi)$ and $\models (\neg true \vee (true \wedge \phi)) \leftrightarrow \phi$.

and specifications. In the case of ccp programs the strongest postcondition semantics allows to view a program as a property of constraints. Formally we define the language of mixed terms of programs and properties as follows:

$$\Phi \ ::= c \mid tell(c) \mid \sum_i ask(c_i) \to \Phi_i \mid \Phi \wedge \Phi \mid \Phi \parallel \Psi \mid \neg\Phi \mid \exists x\Phi \mid p(x)$$

The semantics of a mixed term is given with respect to a set of procedure declarations D and is obtained from the semantics $[\![\ ]\!]$ as previously defined. In fact, note that $[\![\ ]\!]$ as defined in Table II for agents and $[\![\ ]\!]$ in definition 3.2 coincide if we identify $\parallel$ with $\wedge$ and $tell(c)$ with c.

In the mixed term formalism a partial correctness assertion P *sat* ϕ then corresponds to the implication $P \to \phi$. Implication itself thus models the *satisfaction* or *implementation* or *refinement* relation: A mixed term Φ *satisfies* or *implements* or *refines* a mixed term Ψ iff $\Phi \to \Psi$ holds. Note that the refinement relation, which is modeled by set-inclusion, corresponds with a *decrease* in non-determinism. An interesting example illustrating the above is the validity of the following implication (with a slight abuse of notation):

$$(tell(c) + tell(d)) \ \to \ tell(c \sqcap d)$$

(Here $\sqcap$ denote the glb of the given underlying constraint system.) So in the strongest postcondition semantics the non-determinism present in $tell(c \sqcap d)$ is reduced by $tell(c) + tell(d)$ since in the latter program we *know* that either c or d is told.

A derivation of a ccp program P from a specification ϕ in this approach corresponds with a sequence

$$P \equiv \Phi_n \to \ldots \to \Phi_1 \equiv \phi$$

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. Furthermore each of the implications $\Phi_{i+1} \to \Phi_i$ is generated by an application of a transformational rule. Such a rule either consists of some logical reasoning or a rule which allows the introduction of programming constructs. For example a rule which allows the introduction of non-deterministic choice is easily derived from the corresponding proof rule of the above proof system for partial correctness:

$$\frac{\Phi_i \to \Psi_i \quad \forall\ i \in \{1, \ldots, n\}}{(\sum_i ask(c_i) \to \Phi_i) \to \bigvee_i (c_i \wedge \Psi_i) \vee \bigwedge_i \neg c_i}$$

Note that the proof rules for parallel (i.e. $\wedge$), the hiding operator and the consequence rule reduce to purely logical rules, and that the proof rule for procedures corresponds with the logical rule for recursively defined predicates.

Example 5.1. In this example we show how one can derive the angelic merge program introduced in Example 4.4 from the logical specification $x \mid y = z$. Formally, the problem consists in finding a procedure declaration for the process $m(x, y, z)$ such that $m(x, y, z)$ is a refinement of $x \mid y = z$, that is

$$m(x, y, z) \ \to \ x \mid y = z.$$

Denote by **M** and **LM** the two axioms

$$(x\lfloor y = z \vee y\lfloor x = z \vee (x = y \wedge y = z \wedge z = [\,])) \to x \mid y = z$$

and

$$\exists x', z', a(x = [a|x'] \wedge z' = x' \mid y \wedge z = [a|z']) \to x\lfloor y = z$$

of the theory T characterizing the properties $x \mid y = z$ and $x\lfloor y = z$, respectively. Let $\Phi_1 \equiv x \mid y = z$. We obtain the following refinement steps.

(1) Let Φ_2 be the left-hand side of **M**. Then $\Phi_2 \to \Phi_1$.

(2) By replacing each occurrence of $x\lfloor y = z$ in Φ_2 with the corresponding left-hand side of **LM** we obtain

$$\begin{pmatrix} (\exists x', z', a(x = [a|x'] \wedge z' = x' \mid y \wedge z = [a|z']) \\ \vee \\ \exists y', z', a(y = [a|y'] \wedge z' = x \mid y' \wedge z = [a|z']) \\ \vee \\ x = y \wedge y = z \wedge z = [\,]) \end{pmatrix} \to \Phi_2$$

(3) We call the left-hand side of the implication in the previous step Φ_3 and the first two disjuncts Φ_{31}, Φ_{32} respectively. Moreover, let Φ_{33} be $z = [\,]$ and let c_1, c_2, c_3 be the constraints $\exists x', a(x = [a|x'])$, $\exists y', a(y = [a|y'])$ and $x = [\,] \sqcup y = [\,]$, respectively. Then it is easy to check that

$$(c_1 \wedge \Phi_{31} \ \vee \ c_2 \wedge \Phi_{32} \ \vee \ c_3 \wedge \Phi_{33}) \to \Phi_3.$$

Observe that till now the refinement steps have been logical implications. In the next steps we start to introduce mixed terms.

(4) Call the left-hand side of the previous refinement step Φ_4. By applying the introduction rules for the *tell* and the *parallel composition* we obtain

$$\exists x', z', a(\mathit{tell}(x = [a|x']) \parallel \mathit{tell}(z = [a|z']) \parallel x' \mid y = z')) \to \Phi_{31}$$

$$\exists y', z', a(\mathit{tell}(y = [a|y']) \parallel \mathit{tell}(z = [a|z']) \parallel x \mid y' = z)) \to \Phi_{32}$$

and

$$\mathit{tell}(z = [\,]) \to \Phi_{33}$$

hence by the introduction rule of the non-deterministic choice (with Φ_i, Ψ_i the left-hand side and the right-hand side respectively of the i-th step above) and from $\bigwedge_i \neg c_i$ equivalent to *false* we have

$$\begin{pmatrix} \mathit{ask}(\exists_{x'}\exists_a(x = [a|x'])) \to \exists x', z', a \\ \qquad\qquad (\mathit{tell}(x = [a|x']) \parallel \mathit{tell}(z = [a|z']) \parallel x' \mid y = z')) \\ + \\ \mathit{ask}(\exists_{y'}\exists_a(y = [a|y'])) \to \exists y', z', a \\ \qquad\qquad (\mathit{tell}(y = [a|y']) \parallel \mathit{tell}(z = [a|z']) \parallel x \mid y' = z)) \\ + \\ \mathit{ask}(x = [\,] \sqcup y = [\,]) \to \mathit{tell}(z = [\,]) \end{pmatrix} \to \Phi_4$$

(5) Finally, using the introduction rule for *recursion* we obtain the result

$$m(x, y, z) \to x \mid y = z$$

with $m(x, y, z)$ defined by the procedure declaration

$$\begin{aligned} m(x, y, z) :- \ & ask(\exists_{x'}\exists_a(x = [a|x'])) \to \exists x', z', a \\ & \qquad\qquad (tell(x = [a|x']) \parallel tell(z = [a|z']) \parallel m(x', y, z')) \\ & + \\ & ask(\exists_{y'}\exists_a(y = [a|y'])) \to \exists y', z', a \\ & \qquad\qquad (tell(y = [a|y']) \parallel tell(z = [a|z']) \parallel m(x, y', z)) \\ & + \\ & ask(x = [\,] \sqcup y = [\,]) \to tell(z = [\,]) \end{aligned}$$

6. CORRECTNESS OF CONSTRAINT LOGIC PROGRAMMING WITH DYNAMIC SCHEDULING

In this section we apply the proof theory previously developed to the proof of partial correctness for constraint logic programs with dynamic scheduling.

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 [1982]) allow to run the calls more efficiently, to prevent run-time errors and to enforce termination. In CLP they are used also to postpone the evaluation of constraints which are "too hard" for the solver. For example, in CLP($\Re$) ([Jaffar et al. 1992])[3] 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 ([Carlsson 1988]), Eclipse ([Wallace and Veron 1993]), Gödel ([Hill and Lloyd 1994]) and in many CLP systems ([Jaffar and Maher 1994]).

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 [1986] while their denotational semantics has been investigated by Marriott et al. [1994] and Falaschi et al. [1996b]. Recently also the issue of correctness has received some attention. Apt and Luitjes [1995] 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 [1995] and by Lüttringhaus [1993]. In particular, the latter paper describes an algorithm which, given a program, generates automatically

[3] CLP($\Re$) is the CLP language obtained by considering the constraint domain $\Re$ of arithmetic over the real numbers. The signature for $\Re$ contains the constant symbols 0 and 1, the binary function symbols $+$ and $*$, and the binary predicate symbols $+, <, \leq$ for constraints which are interpreted on the real numbers as usual.

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 the following we obtain a system for proving partial correctness of CLP with delay declarations by applying the proof theory previously defined for ccp. This provides also a logical interpretation of delay declaration in terms of implication.

6.1 CLP programs with dynamic scheduling: terminology and operational semantics

In this subsection we briefly introduce the terminology and the basic notions for CLP languages with dynamic scheduling. For CLP we refer to the original paper by Jaffar and Lassez [1987] and to the survey by Jaffar and Maher [1994].

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. Note also that 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). Here we follow a more abstract view and assume that CLP use the abstract notion of constraint system previously defined. In this way we are more general since the standard CLP view can be obtained as a particular instance of the abstract formalization.

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[4].

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 [1982] first introduced delay declarations in logic languages by using `when` constructs in MU-Prolog. Here we follow loosely the syntax used in the language

[4] 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.

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, consider the numerical constraint system $\Re$ previously mentioned. The declaration

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

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. Recall that, as discussed in Section 3, we assume given a set U of user-defined predicates together with an interpretation $I : U \rightarrow \wp(\mathcal{C})$ which defines their meaning.

Definition 6.1. We denote by Δ a subset of the user-defined predicates U, with typical elements $\delta(\overline{x})$, $\delta_1(\overline{x})\ldots$, such that the equality

$$I(\delta(\overline{x})) = \uparrow I(\delta(\overline{x}))$$

holds for each $\delta(\overline{x}) \in \Delta$ [5]. We say that a constraint c satisfies the condition $\delta(\overline{x})$ iff $c \in I(\delta(\overline{x})$.

Thus delay conditions are formalized as user-defined predicates whose interpretation is upper closed. This assumption is made by most of the existing programming systems and, as we discuss later, it is needed in order to apply previous results to CLP with dynamic scheduling. Intuitively, it means 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 $\delta(\overline{x})$. Many natural delay conditions satisfy this monotonicity requirement. For example, given the usual Herbrand constraint system $\mathcal{H}$, the declaration

$$\texttt{DELAY p(X) UNTIL ground(X)}$$

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

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

i.e. the set of all those constraints which force `X` to have a unique value[6].

We do not further specify the syntax of delay conditions since this is not relevant to our results. We only 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

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

[5] $\uparrow$ is the obvious extension to set of constraints of the operator $\uparrow$ previously defined on $\mathcal{C}$.
[6] We assume here that the signature contains at least two constant symbols.

for the predicate symbol $\mathtt{p}$, an atom $\mathtt{p}(\overline{\mathtt{Y}})$ 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 computations 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.

6.2 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 the following extension of local ccp, called δ-ccp.

Definition 6.2. Assuming a given cylindric constraint system $\mathbf{C}$ the syntax of δ-ccp *agents* is given by the following grammar:

$$A ::= stop \mid tell(c) \mid ask(\delta(\bar{x})) \rightarrow A \mid A \oplus B \mid A \parallel B \mid \exists x A \mid p(x)$$

where the c, c_i are supposed to be *finite constraints* (i.e. algebraic elements) in $\mathcal{C}$ and $\delta(\bar{x})$ is an user-defined predicate in Δ. A δ-ccp *process* P is then an object of the form $D.A$, where D is a set of procedure declarations of the form $p(x) : -A$ and A is an agent as defined above.

Note that, according to Definitions 6.1 and 6.2, in δ-ccp the arguments of ask actions are predicates whose interpretation is an upward closed set of constraints. It is interesting to observe that dropping this requirement, i.e. allowing also predicates interpreted by generic (non upward closed) sets would strictly augment the expressive power of the language. In fact, in this case one could express a kind of "var test", i.e. one could check whether a variable is bound to some value or not. This, for example, would allow us to implement the so called fair merge [Panangaden and Shanbhogue 1992] in the resulting language, while this cannot be done in δ-ccp (nor in standard ccp). On the other hand, such an improved expressive power complicates the semantics, since it violates the usual ccp monotonicity condition. The restriction to upward closed sets of constraints however still gives rise to a language which is more expressive than local ccp, while maintaining essentially the same denotational semantics, and hence the same proof system, shown in subsection 4.5. In fact, δ-ccp can be seen as a local ccp language defined on the lifted constraint system $\wp_f(\mathcal{C})$ (defined in Section 3) and we will use such a lifting to prove the correctness and completeness results.

Here however, in order to keep the presentation simple, we prefer to use the syntax defined above. Thus, while the operator $\oplus$ denotes local choice and operationally is interpreted as in subsection 4.5, the agent $ask(\delta(\bar{x})) \rightarrow A$ behaves as A, if the current store c *satisfies* the property $\delta(\bar{x})$, i.e. if $c \in I(\delta(\bar{x}))$. If this is not the case

$\mathbf{R}\delta$	$\langle D.ask(\delta(\overline{x})) \rightarrow A, c\rangle \longrightarrow_\delta \langle D.A, c\rangle$	$c \in I(\delta(\overline{x}))$

Table VI. The transition rule for δ-ccp

the agent suspends. Intuitively, this allows us to simulate the delay mechanism previously described for CLP by using the ask evaluation. Formally, the operational semantics of δ-ccp is then defined by the transition system T_δ obtained from the one given in Table I, provided that we add the Rule $\mathbf{R}\delta$ in Table 6.2, we assume that Rule **R2** is used only with $n = 2$ and we replace $\longrightarrow$ for $\longrightarrow_\delta$.

Using the transition system T_δ we define the standard notion of observables as the results of finite computations. We denote by $\longrightarrow_\delta^*$ the reflexive and transitive closure of the relation $\longrightarrow_\delta$ defined by T_δ.

Definition 6.3. 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

$$\mathcal{O}_\delta(D.A, c) = \{d \mid \text{ there exists } B \text{ s.t. } \langle D.A, c\rangle \longrightarrow_\delta^* \langle D.B, d\rangle \not\longrightarrow_\delta\}.$$

The strongest postcondition semantics $SP_\delta(D.A)$ is defined as in Definition 3.7, provided that we replace $\longrightarrow$ by $\longrightarrow_\delta$.

A CLP program now 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.

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 `p`

$$\begin{array}{l} \mathtt{p(\overline{x}) \leftarrow c_1, Q_1} \\ \vdots \\ \mathtt{p(\overline{x}) \leftarrow c_n, Q_n} \end{array}$$

and of the delay declaration

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

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

$$p(\overline{x}) : -\exists_{-\overline{x}}\ ask(\delta(\overline{x})) \rightarrow (\ \begin{array}{l}\tau(c_1, Q_1)\\ \oplus\\ \vdots\\ \oplus\\ \tau(c_n, Q_n)).\end{array}$$

(note that $\exists_{-\overline{x}}$ denotes the existential quantification wrt all the "local" variables, i.e. wrt those variables which do not appear in $\overline{x}$). 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 6.4. *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*

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

PROOF. Straightforward from the definitions. □

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.

6.3 A proof system for δ-ccp

In this section we show that the proof system given for local ccp can also be applied to δ-ccp. As before, we first show a denotational characterization of the strongest post-condition semantics $SP_{\delta}(D.A)$. It turns out that we can use nearly the same denotational semantics shown in subsection 4.5 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. 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:

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

The point here is that even though δ-ccp can be seen as a "lifted ccp", i.e. a ccp language defined on the lifted constraint system $\wp_f(\mathcal{C})$ defined in Section 3, yet the results of computations in this lifted language can be considered as constraints in $\mathcal{C}$.

More precisely, consider $ask(\delta(\overline{x}))$ and $tell(c)$ as shorthands for $ask(I(\delta(\overline{x})))$ and $tell(\uparrow c)$, respectively, and call lifted-ccp the resulting language. Then we have the following Lemma, where we assume that, given a lifted-ccp process A, its strongest postcondition $SP_l(D.A)$ is obtained (from Definition 3.7 and Table I) by considering the lifted constraint system $\wp_f(\mathcal{C})$ rather than $\mathcal{C}$.

LEMMA 6.5. *Let $D.A$ be a δ-ccp process and let $D'.A'$ be the corresponding lifted process defined as before. Then $c \in SP_{\delta}(D.A)$ iff $\uparrow c \in SP_l(D'.A')$.*

PROOF. First observe that the condition

$$c \in I(\delta(\overline{x}))$$

can equivalently be expressed as

$$\uparrow c \subseteq I(\delta(\overline{x})),$$

since $I(\delta(\overline{x}))$ is an upward closed set. Now, $\subseteq$ is exactly the entailment relation in the lifted constraint system $\wp_f(\mathcal{C})$. Therefore, the rule $R\delta$ in Table 6.2 is equivalent to the rule $R2$ in Table I, provided that in the latter it is used the constraint system $\wp_f(\mathcal{C})$ (clearly, as in the case of local ccp, $ask(I(\delta(\overline{x}))) \rightarrow A$ is seen as a shorthand for $\Sigma_{i=1}^{1} ask(I(\delta(\overline{x}))) \rightarrow A$).

Furthermore, observe that the constraints which are "told" in a computation are added to the store by using a lub operation and that clearly

$$\uparrow c \cap \uparrow d = \uparrow (c \sqcup d)$$

holds. Therefore, since in the lifted language the arguments of tell actions are of the form $\uparrow c$, i.e. they are principal ideals, the results of computations in the lifted language are also principal ideals. From the equivalence of the transition rules stated above it is then immediate to check (by induction on the length of the computation) that d is a result for A with input constraint c iff $\uparrow d$ is a result for A' with input constraint $\uparrow c$, which concludes the proof. $\Box$

This lemma allows us to apply the results of the previous section to the lifted language, and hence to δ-ccp. We first define the denotational semantics as follows (for the reader convenience we repeat the equation **Dl2**).

Definition 6.6. The denotational semantics of δ-ccp is expressed by the function $[\![P]\!]_\delta(e)$: *Processes* $\rightarrow \wp(\mathcal{C})$ defined by the equations in Table II modified by replacing equation **D2** for the equations **Dδ** and **Dl2** given in Table VII (and by adding the subscript $_\delta$). e and μ are defined as in Section 4.1.

Dδ	$[\![D.ask(\delta(\overline{x})) \rightarrow A]\!]_\delta(e) = (\mathcal{C} \setminus I(\delta(\overline{x}))) \cup [\![A]\!]_\delta(e)$
Dl2	$[\![D.A \oplus B]\!]_\delta(e) = [\![D.A]\!]_\delta(e) \cup [\![D.B]\!]_\delta(e)$

Table VII. Strongest postcondition semantics of δ-ccp

In order to prove the correspondence of the denotational semantics with the strongest postcondition we need also the following result which is the denotational analogous of Lemma 6.5. Given a lifted process $D.A$, define $[\![D.A]\!]_l(e)$ as the semantics obtained from Table II by replacing in the equations $\mathcal{C}$ for the set of principal ideals $\widehat{\mathcal{C}}$ (i.e. $\widehat{\mathcal{C}}$ is defined as $\{\uparrow c \mid c \in \mathcal{C}\}$) and by interpreting $\exists x$ as the obvious pointwise extension (to sets of sets) of the operator introduced in Definition 3.2. Note that $[\![D.A]\!]_l(e)$ is well defined (it is sufficient to apply Lemma 3.3 for equation D4). Then we have the following.

Lemma 6.7. *Let $D.A$ be a δ-ccp (closed) process and let $D'.A'$ be the corresponding lifted process defined as before. Then $c \in [\![D.A]\!]_\delta(e)$ iff $\uparrow c \in [\![D'.A']\!]_l(e)$.*

PROOF. Immediate from the definitions by using structural induction. We only show the case of the choice operator (the others are similar). Let $ask(\delta(\overline{x})) \rightarrow A$ be a δ-ccp agent and $ask(\delta(I(\overline{x}))) \rightarrow A'$ be the corresponding lifted process (recall that this is a shorthand for $\Sigma_{i=1}^{1} ask(I(\delta(\overline{x}))) \rightarrow A'$). First observe that we have

$$\begin{array}{l}\widehat{\mathcal{C}} \setminus \uparrow I(\delta(\overline{x})) = \\ \{\uparrow d \mid \uparrow d \not\subseteq I(\delta(\overline{x}))\} = \\ \{\uparrow d \mid d \in \mathcal{C} \setminus I(\delta(\overline{x}))\}\end{array}$$

where the last equality holds since $I(\delta(\overline{x}))$ is upward closed. Therefore, from equations $D\delta$ and $D2$ and inductive hypothesis it follows that

$$c \in [\![D.ask(\delta(\overline{x})) \rightarrow A]\!]_\delta(e) \text{ iff } \uparrow c \in [\![D'.\Sigma_{i=1}^{1} ask(I(\delta(\overline{x}))) \rightarrow A']\!]_l.$$

The other case is analogous by considering equations $Dl2$ and $D2$ and recalling that $A_1 \oplus A_2$ represents $\Sigma_{i=1}^{2} ask(true) \rightarrow A_1$. $\square$

Then we have the following correspondence result.

PROPOSITION 6.8. *For any δ-ccp process $D.A$ we have $SP_\delta(D.A) = [\![D.A]\!]_\delta$.*

PROOF. Immediate from Corollary 4.11 and Lemmata 6.5 and 6.7. $\square$

Also the proof rules for δ-ccp are similar to those for local ccp: They are obtained from those in Table III by replacing the rule **C2** for the rules **Cδ** and **Cl2** given in Table VIII.

Cδ	$\dfrac{D.A \ sat \ \phi}{D.ask(\delta(\overline{x})) \rightarrow A \ sat \ \delta(\overline{x}) \rightarrow \phi}$
Cl2	$\dfrac{D.A \ sat \ \phi \quad D.B \ sat \ \psi}{D.A \oplus B \ sat \ \phi \vee \psi}$

Table VIII. the rules for δ-ccp

The soundness and completeness for this calculus follow from Proposition 6.8 and the analogous results given for local ccp. It is worth noticing that also in this case we obtain a complete proof system.

The above results can be applied directly to CLP programs with delay declarations. In fact we have the following

COROLLARY 6.9. *Let P be a CLP program with delay declarations, c, Q be a query and ϕ a property and assume the expressibility of the strongest postcondition. Then $\tau(P).\tau(Q)$ sat ϕ can be derived by the proof system obtained from Tables III and VIII iff $\bigcup_{c \in \mathcal{C}} \mathcal{O}(P, Q, c) \cup \{false\} \subseteq [\![\phi]\!]$.*

So we can prove that the δ-ccp process $\tau(P).\tau(Q)$ 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.

6.4 An example

In this subsection we give first an example of translation of a CLP program into δ-ccp. Then we show how our proof system can be used to prove properties of programs.

EXAMPLE 6.10. 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=[]` $\wedge$ `T=()` $\wedge$ `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:

$tree(Li, Lp, T)$:-
$\quad \exists L(tell(L = [\,]) \parallel tree\&residual(Li, Lp, T, L))$

$tree\&residual(Li, Lp, T, L)$:-

$$
\begin{array}{l}
\quad ask(nonvar(Li) \sqcup nonvar(Lp)) \rightarrow \\
\qquad tell(Li = [\,] \sqcup T = () \sqcup \; L = Lp) \\
\qquad \oplus \\
\qquad \exists Y \; \exists Lit \; \exists X \; \exists Lpt \; \exists Tl \; \exists Tr \; \exists Lil \; \exists Lir \\
\qquad (\; tell(Li = [Y|Lit] \sqcup Lp = [X|Lpt] \sqcup T = (X, Tl, Tr)) \parallel \\
\qquad\; split(X, Li, Lil, Lir) \parallel \\
\qquad\; tree\&residual(Lil, Lpt, Tl, Lpr) \parallel \\
\qquad\; tree\&residual(Lir, Lpr, Tr, L) \;)
\end{array}
$$

$$
\begin{array}{l}
split(X, L, Ll, Lr) \text{ :-} \\
\quad ask(ground(X) \sqcup \exists_Y : \exists_{Lt} : L = [Y|Lt] \sqcup ground(Y)) \rightarrow \\
\qquad tell(L = [X|Lr] \sqcup Ll = [\,]) \\
\qquad \oplus \\
\qquad \exists Y \; \exists Lt \; \exists Llt \; (\; tell(L = [Y|Lt] \sqcup X \neq Y \sqcup Ll = [Y|Llt]) \parallel \\
\qquad\qquad\qquad\qquad split(X, Lt, Llt, Lr) \;)
\end{array}
$$

We want now 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 (**C5**) it is sufficient to prove that

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

Observe that, since ϕ is implied by $\exists L \; \phi$ (L does not appear in ϕ), by the rule for the weakening (**C7**) it is sufficient to prove that

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

and by the rule for the existential (**C4**) we reduce the proof to the proof of

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

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

$$ask(nonvar(Li) \sqcup nonvar(Lp)) \rightarrow A_1 \oplus \exists Locvar \; A_2 \;\; sat \;\; \psi$$

where A_1 is the process $tell(Li = [\,] \sqcup T = () \sqcup \; L = Lp)$, $\exists Locvar$ is a shorthand for $\exists Y \; \exists Lit \; \exists X \; \exists Lpt \; \exists Tl \; \exists Tr \; \exists Lil \; \exists Lir$ (the existential quantification of the local variables) and A_2 is the process

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

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

$$tree\&residual(Li, Lp, T, L) \ \ sat \ \ \psi, \tag{2}$$

as it is allowed by the Rule **C5**. Observe now that $\psi \equiv (nonvar(Li) \wedge nonvar(Lp)) \rightarrow \psi$, thus using the ask rule (**C**δ) we reduce the proof to

$$A_1 \oplus \exists Locvar \ A_2 \ \ sat \ \ \psi$$

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** and previous definition of A_1 we have

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

hence for the choice rule **Cl2** it is sufficient to prove that

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

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

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

where $Lil * X * Lir$ represents the concatenation of Lil, X, and Lir. Hence, by the weakening rule (**C7**), it is sufficient to prove that

$$\exists Locvar \ A_2 \ \ sat \ \ \exists Locvar \ \eta$$

and by the rule for the existential (**C4**) we reduce the proof to the proof of

$$A_2 \ \ sat \ \ \eta.$$

This can be done by using the parallel rule **C3**, and then proving:

— $tell(Li = [Y|Lit] \sqcup Lp = [X|Lpt] \sqcup T = (X, Tl, Tr)) \ \ sat \ \ (Li = [Y|Lit] \sqcup Lp = [X|Lpt] \wedge T = (X, Tl, Tr))$. By the tell rule (**C1**).

— $tree\&residual(Lil, Lpt, Tl, Lpr) \ \ sat \ \ \tau$ where τ is the property

$$(ground(Lil) \wedge ground(Lpt)) \rightarrow (samenodes(Lil, Tl) \wedge samenodes(Lpt \setminus Lpr, Tl)).$$

To prove previous assertion first observe that in the case of the Herbrand constraint system a diagonal element d_{xy} is simply and equation $x = y$. Then from the assumption (2), Rule **C3** and Rule **C4** it follows

$$\exists Lil \ \exists Lpt \ \exists Tl \ \exists Lpr \ B \ \ sat \ \ \exists Lil \ \exists Lpt \ \exists Tl \ \exists Lpr \ \xi$$

where B is the process

$$tree\&residual(Li, Lp, T, L) \parallel tell(Li = Lil) \parallel tell(Lp = Lpt) \parallel tell(T = Tl) \parallel tell(L = Lpr)$$

and ξ is the property

$$\psi \wedge Li = Lil \wedge Lp = Lpt \wedge T = Tl \wedge L = Lpr.$$

Then from rule **C6** it follows

$$tree\&residual(Lil, Lpt, Tl, Lpr) \;\; sat \;\; \exists Lil \; \exists Lpt \; \exists Tl \; \exists Lpr \; \xi.$$

Finally, since $\exists Lil \; \exists Lpt \; \exists Tl \; \exists Lpr \; \xi$ implies τ, from the weakening rule we have

$$tree\&residual(Lil, Lpt, Tl, Lpr) \;\; sat \;\; \tau.$$

—$tree\&residual(Lir, Lpr, Tr, L) \;\; sat \; (ground(Lir) \wedge ground(Lpr)) \rightarrow (samenodes(Lir, Tr) \wedge samenodes(Lpr \setminus L, Tr)).$

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.

6.5 Optimal delay declarations versus guarded choice

The translated program in Example 6.10 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* ([Falaschi et al. 1996a]). 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:

$tree\&residual(Li, Lp, T, L)$:-
  $ask(Li = [\,]) \rightarrow tell(Li = [\,] \wedge T = () \wedge \;\; L = Lp)$
  $+$
  $ask(\exists_Y \exists_{Lit} \exists_X \exists_{Lpt} Li = [Y|Lit] \wedge Lp = [X|Lpt]) \rightarrow$
    $(\; \exists Y \;\; \exists Lit \; \exists X \;\; \exists Lpt \; \exists Tl \; \exists Tr \; \exists Lil \; \exists Lir$
     $tell(Li = [Y|Lit] \wedge Lp = [X|Lpt] \wedge T = (X, Tl, Tr)) \;||$
     $split(X, Li, Lil, Lir) \;||$
     $tree\&residual(Lil, Lpt, Tl, Lpr) \;||$
     $tree\&residual(Lir, Lpr, Tr, L) \;)$

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 $\|$: 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 $\mathtt{p}$ 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 $\mathtt{s}$ (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 $\mathtt{s}$ is consistent with the constraint $\mathtt{c}$ of such clause, then every store greater than $\mathtt{s}$ (apart from *false*) will also be consistent with $\mathtt{c}$. Therefore, if the constraints of two clauses are not mutually exclusive, it will never be the case that a store $\mathtt{s}$ satisfies the delay condition and is consistent with both of them, while a store greater than $\mathtt{s}$ 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 6.10) 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.

7. CONCLUSIONS

We presented a compositional proof system for proving partial correctness of concurrent constraint programs. The rules of the calculus are obtained from the definition of a simple denotational semantics which approximates the set of all the final results (the so-called resting points) of a ccp process. We proved its soundness (for full ccp) and (relative) completeness (for programs for which the denotational semantics in Table II describes exactly their resting points).

From a theoretic point of view, one of our main contributions is to clarify the *declarative* nature of (confluent) ccp languages and its advantages wrt other concurrent programming paradigms. Indeed the proof theory we have defined allows one to consider a ccp program as a logical theory in the specification logic by means of a direct translation of the language operators in their classical logical counterparts. An immediate outcome of this "logical reading" of ccp programs is the simplicity of the calculus, especially if compared to proof systems for concurrent imperative languages which involve complicated notions as, for example, a test for interference freedom.

Another interesting point is the close correspondence between program denotations and logics for ccp (actually they can be viewed as different faces of the same coin). A very relevant line of research has been devoted in the past few years to establish closer links between denotational and axiomatic semantics of programming languages, via a better understanding of the relations between topology (and domain-theory) and logics ([Plotkin 1980; Scott 1982; Smyth 1983; Robinson 1987]). The significance of such a line was eventually made clear by Abramsky [1987] who exploited a seminal idea of Smyth [1983]: the classic Stone representation theorem for Boolean algebras is the key to establish a correspondence (actually a duality of categories) between denotational semantics (spaces of points which are denotations of computational processes) and program logics (lattices of properties of processes).

The simplicity of our construction is essentially due to the explicit logical interpretation of the basic programming constructs. Further investigation should clarify the relation between the logics of the constraints system and the programming logic. In particular we aim at a complete axiomatization of the logic of properties (assuming as additional axioms all the valid implications between constraints).

The advantages of obtaining a programming logic from a denotational semantics are self-evident in our case (consider for example the soundness and completeness proofs). On the other hand, proof systems for imperative concurrent languages are often designed by using an "ad hoc ingenuity" and their soundness and completeness are proved wrt an operational semantics using elaborate techniques ([Apt 1983]).

From a pragmatic point of view, for any *real* programming language the importance of formal tools to reason about the correctness of programs is evident. The concurrent constraint paradigm has already proved its usefulness in several implementations, including a commercial one ([Foster and Taylor 1989]). Techniques based on abstract interpretation have been used to analyze properties of ccp computations ([Codish et al. 1993; Falaschi et al. 1993]), but as far as we know, our is the first attempt to develop a formal calculus for (partial) correctness. Such a formal system should be viewed as a first step towards the realization of formal methods for the verification and the synthesis of ccp programs. In this practical perspective the incompleteness of our proof system for full ccp should not be a serious problem. In fact, as we mentioned in the introduction, for many real programs the denotational semantics given in Table II describes exactly the resting points, so for these programs the proof system is complete.

Hoare-logic

We have presented a calculus for assertions of the form $P \ sat \ \phi$. This allows us to describe properties of the final states of ccp computations without considering any assumption on the initial store. A natural extension would be to consider arbitrary preconditions, i.e. to give a calculus for triples of the form $\{\psi\}\ P\ \{\phi\}$ in the classical Hoare-logic style, with the intuitive meaning that if the computation of P starts in a store which satisfies ψ and terminates in a state d, then d satisfies ϕ. If preconditions are described by upward closed properties then triples are not more expressive than the assertions considered in previous sections. Indeed let us formally define the meaning of a triple as follows:

$$\{\psi\}\ P\ \{\phi\} \text{ iff } SP_{\psi}(P) \subseteq [\![\phi]\!]$$

where $SP_{\Psi}(P)$ denotes the strongest postcondition wrt ψ, i.e.

$$SP_{\Psi}(P) =_{def} \ \{d \mid \ \textit{there exist } c \in [\![\psi]\!] \textit{ and } Q \text{ s.t. } \langle P, c\rangle \longrightarrow^{*} \langle Q, d\rangle \not\longrightarrow \ \}.$$

Then we it is easy to see that the following holds

$$\{\psi\}\ P\ \{\phi\} \text{ iff } P \ sat \ \psi \rightarrow \phi.$$

Upward closed properties can be syntactically characterized as formulas constructed from constraints using conjunction and disjunction only. The above equivalence shows that for these properties the *sat* system is expressive enough.

Introducing preconditions containing negation is not straightforward since it can easily be shown that in this case the strongest postcondition semantics $SP_\Psi(P)$ is not any more compositional (counterexamples can be obtained by slight modifications of the usual ones which show that the input/output semantics is not compositional). A calculus for the strongest postcondition semantics in the general case then would require additional proof techniques similar to those used for imperative languages, e.g. test for interference freedom ([Owicki and Gries 1976]) and cooperation test ([Apt et al. 1980]). But even more basically, allowing arbitrary preconditions does not even permit an axiomatization of the tell operation; namely, it can be argued that there does not exist a logical context $C[x, y]$ such that for any property ϕ and constraint c the strongest postcondition of the tell operation $tell(c)$ wrt to ϕ is described by $C[\phi, c]$. Note, for example, that simply adding the constraint c to the precondition is incorrect: It would yield *false* as the strongest postcondition of $tell(c)$ with respect $\neg c$. Moreover, it is not clear how such a generality would really improve the expressive power of the calculus, for ccp languages, from the point of view of applications.

Future work

We have focussed in this paper on the axiomatization of the *finite* behaviours of ccp-programs as it is described by the strongest postcondition semantics. It has to be remarked, however, that often the interesting properties of a concurrent program are concerned with the infinite behaviours. For instance, the characteristic property of the angelic merge is the following:

> If some item in one input-channel does not appear in the output channel,
> then the other input channel contains infinitely many items.

Now, this property is true for the finite postcondition semantics of the merge program defined in this paper (vacuously: the output contains always all the items provided in input), but someone could object that there is no guarrantee that it would still hold if we would observe also infinite behaviours.

From a semantic point of view, it is easy to extend the denotational construction of the strongest postcondition semantics so to include the "results" of infinite runs (i.e. the least upper bounds of the intermediate stores): we have just to take the *greatest* fixpoint of the operator Ψ defined in Table II, instead of the least fixpoint. However, the rule **C5** is incorrect with respect to this new denotational semantics. Take for instance $p(x)$ be declared as $p(x)$:- $p(x)$. Then the extended postcondition semantics of $p(x)$ is $\mathcal{C}$, i.e. $[\![true]\!]$, but our system would allow to derive that $p(x)$ *sat false*, which is clearly incorrect.

In order to obtain a correct axiomatization of recursion in case of infinite observables, we think that one possibility is to introduce in the logic of properties a greatest fixpoint operator (dual to the least fixpoint operator in Section 4.3)

$$\phi ::= p(x) \mid \nu p(x).\phi$$

on which logic reasoning is formalized in terms of the co-induction rule:

$$\frac{\phi \to \psi[\phi/p]}{\phi \to \nu p(x).\psi}$$

Then we should substitute **C5** with the following rule

$$\frac{p(x)\ sat\ p(x) \vdash A\ sat\ \phi}{p(x)\ sat\ \nu p(x).\phi}$$

We conjecture that the resulting proof system is correct with respect to the strongest postcondition semantics which includes the results of infinite behaviours, and that for confluent programs also completeness holds.

It is easy to check that this system would allow to prove that our merge program satisfies the characteristic property of angelic merge. Such property can be expressed as follows:

$$\nu p(x, y, z).(a \in x \wedge a \notin x \rightarrow \exists b, y'.(y = [b|y'] \wedge p(x, y', z))$$

Further directions which we are currently working on include (possible non-compositional) extensions of the calculus which are complete for full ccp. More long term interests include the development of a refinement calculus for program synthesis along the lines suggested in section 5.

ACKNOWLEDGMENTS

We thank the referees for their useful commments. This work has been partially supported by the HCM project "EXPRESS". We thank

REFERENCES

Apt, K. 1983. Formal justification of a proof system for Communicating Sequential Processes. *Journal of the ACM 30*, 197–216.

Apt, K., Francez, N., and de Roever, W. P. 1980. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems 1,* 2, 359–385.

Apt, K. R. 1990. Introduction to Logic Programming. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 495–574.

Apt, K. R. 1993. Declarative programming in Prolog. In *Proc. Int'l Symposium on Logic Programming*, D. Miller, Ed. The MIT Press, Cambridge, Mass.

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

Arbab, B. and Berry, D. 1987. Operational and Denotational Semantics of Prolog. *Journal of Logic Programming 4*, 309–330.

Bergstra, J. and Klop, J. 1986. Conditional Rewrite Rules: confluence and termination. *J. Comput. Syst. Sci. 32*, 323–362.

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

Codish, M., Falaschi, M., Marriott, K., and Winsborough, W. 1993. Efficient Analysis of Concurrent Constraint Logic Programs. In *Proc. of the 20th International Colloquium on Automata, Languages, and Programming*, A. Lingas, R. Karlsson, and S. Carlsson, Eds. Lecture Notes in Computer Science, vol. 700. Springer-Verlag, Berlin, 633–644.

Colussi, L. and Marchiori, E. 1991. Proving correctness of logic programs using axiomatic semantics. In *Proc. Eighth Int'l Conf. on Logic Programming*, K. Furukawa, Ed. The MIT Press, Cambridge, Mass., 629–644.

Colussi, L., Marchiori, E., and Marchiori, M. 1995. A dataflow semantics for constraint logic programs. In *Proc. Seventh Int'l Symp. on Programming Languages, Implementations, Logics and Programs.*

COOK, S. 1978. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computation 7,* 1, 70–90.

DE BOER, F. AND PALAMIDESSI, C. 1992. A process algebra for concurrent constraint programming. In *Proc. Joint Int'l Conf. and Symposium on Logic Programming*, K. Apt, Ed. Series in Logic Programming. The MIT Press, Cambridge, Mass., Washington, USA, 463–477.

DE BOER, F. S., DI PIERRO, A., AND PALAMIDESSI, C. 1995. Infinite computations in nondeterministic constraint programming. *Theoretical Computer Science 151*, 37–78.

DE BOER, F. S., KOK, J. N., PALAMIDESSI, C., AND RUTTEN, J. J. M. M. 1991. The failure of failures: Towards a paradigm for asynchronous communication. In *Proc. of Concur '91*. Lecture Notes in Computer Science, vol. 527. Springer-Verlag, Berlin.

DE BOER, F. S. AND PALAMIDESSI, C. 1990. A Fully Abstract Model of Concurrent Logic Languages. Tech. rep., Dipartimento di Informatica, Università di Pisa.

DIJKSTRA, E. 1976. *A discipline of programming.* Prentice-Hall.

FALASCHI, M., GABBRIELLI, M., MARRIOTT, K., AND PALAMIDESSI, C. 1993. Compositional Analysis for Concurrent Constraint Programming. In *Proceedings of the Eight Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, 210–221.

FALASCHI, M., GABBRIELLI, M., MARRIOTT, K., AND PALAMIDESSI, C. 1996a. Confluence in Concurrent Constraint Programming. Tech. Rep. 96/284, Department of Computer Science, Monash University, Australia. To appear in Theoretical Computer Science.

FALASCHI, M., GABBRIELLI, M., MARRIOTT, K., AND PALAMIDESSI, C. 1996b. Constraint logic programming with dynamic scheduling: A semantics based on closure operators. Tech. Rep. 32/96, Dipartimento di Informatica, Università di Udine. To appear in Information and Computation.

FOSTER, I. AND TAYLOR, S. 1989. *Strand: New concepts in parallel programming.* Prentice Hall.

HENKIN, L., MONK, J., AND TARSKI, A. 1971. *Cylindric Algebras. Part I and II.* North-Holland, Amsterdam.

HILL, P. M. AND LLOYD, J. W. 1994. *The Gödel programming language.* The MIT Press.

HOARE, C. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12,* 10, 576–580.

JAFFAR, J. AND LASSEZ, J.-L. 1987. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages.* ACM Press, 111–119.

JAFFAR, J. AND MAHER, M. J. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

JAFFAR, J., MICHAYOV, S., STUCKEY, P. J., AND YAP, R. H. C. 1992. The CLP($\mathcal{R}$) language and system. *TOPLAS: ACM Transactions on Programming Languages and Systems 14,* 3 (July), 339–395.

JAGADEESAN, R., SHANBHOGUE, V., AND SARASWAT, V. 1991. Angelic non-determinism in concurrent constraint programming. Tech. rep., System Science Lab., Xerox PARC.

LÜTTRINGHAUS-KAPPEL, S. 1993. Control generation for logic programs. In *Proc. Tenth Int'l Conf. on Logic Programming*, D. Warren, Ed. MIT Press, 478–495.

MARCHIORI, E. AND TEUSINK, F. 1995. Proving termination of logic programs with delay declarations. In *Proc. Twelfth International Logic Programming Symposium*, J. Lloyd, Ed. MIT Press.

MARRIOTT, K., DE LA BANDA, M. G., AND HERMENEGILDO, M. 1994. Analyzing logic programs with dynamic scheduling. In *Proc. 21st Annual ACM Symp. on Principles of Programming Languages.* ACM Press, 240–253.

NAISH, L. 1982. An introduction to mu-prolog. Tech. Rep. 82/2, The University of Melbourne.

NAISH, L. 1986. *Negation and Control in Prolog.* Lecture Notes in Computer Science, vol. 238. Springer-Verlag, Berlin.

OLDEROG, E. 1991. *Nets, Terms and Formulas.* Cambridge Tracts in Theoretical Computer Science 23. Cambridge Univ. Press.

OWICKI, S. AND GRIES, D. 1976. An axiomatic proof technique for parallel programs. *Acta Informatica 6*, 319–340.

PANANGADEN, P. AND SHANBHOGUE, V. 1992. The expressive power of Indeterminate Dataflow Primitives. *Information and Computation 98(1)*, 99–131, 1992.

PANANGADEN, P., SARASWAT, V., SCOTT, P., AND SEELY, R. 1992. A hyperdoctrinal view of concurrent constraint programming. In *In Proc. of REX Workshop*. Lecture Notes in Computer Science, vol. 666. Springer-Verlag, Berlin, 457–476.

PANANGADEN, P. AND SHANBHOGUE, V. 1992. The expressive power of indeterminate dataflow primitives. *Information and Computation 98,* 1, 99–131.

PLOTKIN, G. 1980. Dijkstra's predicate transformers and smyth's powerdomains. Number 86 in Lecture Notes in Computer Science. Springer-Verlag, Berlin.

ROBINSON, E. 1987. Logical aspects of denotational semantics. Number 283 in Lecture Notes in Computer Science. Springer-Verlag, Berlin.

SARASWAT, V. AND RINARD, M. 1990. Concurrent constraint programming. In *Proc. of the Seventeenth ACM Symposium on Principles of Programming Languages.* ACM Press, 232–245.

SARASWAT, V. A. 1989. Concurrent constraint programming languages. Ph.D. thesis, Carnegie-Mellon University. Also in ACM distinguished dissertation series. The MIT Press, 1993.

SARASWAT, V. A., RINARD, M., AND PANANGADEN, P. 1991. Semantic Foundation of Concurrent Constraint Programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages.* ACM Press, 333–353.

SCOTT, D. 1982. Domains for Denotational Semantics. In *Proc. ninth Int. Coll. on Automata, Languages and Programming*, M. Nielsen and E. M. Schmidt, Eds. Lecture Notes in Computer Science, vol. 140. Springer-Verlag, Berlin, 577–613.

SCOTT, D. AND DE BAKKER, J. 1969. A theory of programs. Technical Report Unpublished, Notes of the IBM Vienna Seminar.

SMYTH, M. 1983. Powerdomains and predicate transformers: A topological view. In *Proc. Int'l Conf. on Automata, Languages and Programming.* Number 154 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 662–675.

WALLACE, M. AND VERON, A. 1993. Two problems – two solutions: One system – ECLiPSe. In *Proceedings IEE Colloquium on Advanced Software Technologies for Scheduling.* London.