Proving Concurrent Constraint Programs Correct

Frank S. de Boer* Free University
Maurizio Gabbrielli† CWI
Elena Marchiori‡ CWI
Catuscia Palamidessi§ DISI

Abstract

We develop a compositional proof-system for the partial correctness of concurrent constraint programs. Soundness and (relative) completeness of the system are proved with respect to a denotational semantics based on the notion of strongest postcondition. The strongest postcondition semantics provides a justification of the declarative nature of concurrent constraint programs, since it allows to view programs as theories in the specification logic.

1 Introduction

Concurrent constraint programming ([24, 25, 26]) (ccp, for short) is a concurrent programming paradigm which derives from replacing the *store-as-valuation* conception of von Neumann computing by the *store-as-constraint* model. Its computational model 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 is achieved by allowing processes also 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 consistency, 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 compositional (fully abstract) models for the standard input/output behaviour of ccp programs are similar to those used, for example, for imperative languages based on asynchronous communication mechanisms ([6]) and are essentially more complicated than those used for pure (constraint) logic languages ([3, 17]). It should be noted that for the deterministic fragment of ccp there exists an elegant denotational semantics based on closure operators [26], which in [21] are shown to be intimately related to the logic of constraints.

The main result of this paper is the introduction of a proof-theory for ccp, i.e. a calculus for proving correctness of ccp programs (or, in other words, an axiomatic semantics) which *does* provide a declarative interpretation of ccp. The issue of the design of proof systems appropriate to proving correctness of ccp programs has received no attention so far. For logic languages like Prolog the proof techniques of Hoare-Logic ([16]) have been applied in [10] to reason about properties of the flow of control and a process algebra for ccp has been developed in [8] along the lines of ACP ([5]). 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

*Free University, de Boelelaan 1081, 1081 HV Amsterdam, The Netherlands. e.mail: frankb@cs.vu.nl

†CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands. e.mail: gabbri@cwi.nl

‡CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands. e.mail: elena@cwi.nl

§Dip. di Informatica e Scienze dell'Informazione, Via Benedetto XV, 3, Genova, Italy. e.mail: catuscia@di.unipi.it

the *weakest precondition* or, equivalently, the *strongest postcondition* ([12]). Since the standard semantics of ccp can already be viewed as a predicate (i.e., store-as-constraint) transformer semantics it is rather natural to expect that the semantics itself can be used to prove correctness of programs. This then would provide a strong evidence for the declarative nature of ccp.

Unfortunately one can argue that the above suggested expectation is not justified. Consider the following simple ccp program:

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

This program adds $y = 1$ to the store in case the initial store either entails $x = 0$ or $x = 1$ (otherwise it suspends). Thus one would like to state that the above program satisfies the specification $(x = 0 \vee x = 1) \rightarrow y = 1$, i.e. every terminating computation results in a store such that whenever $x = 0$ or $x = 1$ then $y = 1$ is guaranteed to hold. However this correctness statement cannot be justified semantically: Intuitively the tell operation $tell(x = 0 \vee x = 1)$ satisfies the specification $x = 0 \vee x = 1$. Thus one would expect the parallel composition

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

to satisfy

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

which would imply $y = 1$. However the above parallel program does not satisfy $y = 1$ since $x = 0 \vee x = 1$ neither entails $x = 0$ nor $x = 1$, so the program will suspend after the tell action. This simple example shows that one cannot reason about the non-deterministic choice in terms of the disjunction of the underlying constraint system.

In this paper we introduce a specification logic to reason about the correctness of ccp programs in terms of properties of constraints. A property is described in terms of constraints themselves and the usual (classical) logical operations of negation, conjunction and existential quantification. A property described by a constraint is interpreted as the set of constraints that entail it (here the entailment relation stems from the underlying constraint system). The logical operations of disjunction, negation etc, then are interpreted in terms of the corresponding set-theoretic operations. This definition of the specification logic allows a direct correspondence between the programming constructs of ccp and their logical counterparts: For example, action prefixing corresponds to implication, non-deterministic choice to disjunction and parallel composition to conjunction.

¿From a semantical point of view this nice correspondence derives from the compositionality of 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. A compositional axiomatization of this notion of partial correctness for concurrent imperative programs requires in general the introduction of some kind of *history variables* which encode the sequence of interactions (or communications) of a process with its environment ([30]). In contrast, the monotonic computational model of ccp allows to incorporate the relevant assumptions about the parallel environment in the intial store, and to express logically the interactive behaviour of a process in terms of implication: A specification $\phi \rightarrow \psi$ can be interpreted as stating that if the environment provides ϕ then ψ is guaranteed to hold.

The strongest postcondition of a ccp program provides an abstraction of the standard input/output behaviour of ccp programs and as such it allows a simple compositional semantics (parallel composition is modelled by set-theoretic intersection and non-deterministic choice by union). Thus the strongest postcondition semantics supports a declarative interpretation of ccp programs, i.e. they can be viewed as theories in the specification logic.

The remaining of the paper is organized as follows. In the next section we introduce some basic notions on ccp languages. In section 3 we formalize the concept of partial correctness for ccp programs by introducing a specification logic. Section 4 contains the strongest postcondition semantics and the proof system for ccp. In section 5 we show how the proof system can be used for the transformational design of ccp programs along the lines of [19]. Section 6 concludes.

2 Preliminaries

In this section we give the basic definitions of ccp languages following [25]. 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 in [25] following Scott's treatment of information systems ([27]). 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 structure 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 $\langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}\rangle$ where $\sqcup$ is the lub operation, and *true*, *false* are the least and the greatest elements of $\mathcal{C}$, respectively.

Following the standard terminology and notation, instead of $\leq$ we will refer to its inverse relation, denoted by $\vdash$ and called *entailment*. Formally, $\forall c, d \in C.\ \ c \vdash d \ \Leftrightarrow\ d \leq c$. 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 ([15]). This leads to the concept of *cylindric constraint system*. In the following, we assume given a (denumerable) set of variables *Var* with typical elements $x, y, z, \ldots$.

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

(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 $\mathcal{C}$. Finally, in order to model parameter passing, *diagonal elements* ([15]) 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) $\mathit{true} \vdash d_{xx}$,

(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 $\exists_x(c) \sqcup d$.)

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

$$\begin{array}{rl} A ::= & \mathit{stop} \mid \mathit{tell}(c) \mid \sum_{i=1}^{n} \mathit{ask}(c_i) \to A_i \mid A \parallel A \\ & \exists x A \mid p(x) \end{array}$$

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.

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 $T = (\mathit{Conf}, \longrightarrow)$. The configurations (in) *Conf* are pairs consisting of a process, and a constraint in $\mathcal{C}$ representing the common *store*. The transition relation $\longrightarrow \subseteq \mathit{Conf} \times \mathit{Conf}$ is described by the (least relation satisfying the) rules **R1-R6** of table 1.

The agent *Stop* represents successful termination. The basic actions are given by $\mathit{tell}(c)$ and $\mathit{ask}(c)$ constructs which act on the common store. Given a store d, as shown by rule **R1**, the execution of $\mathit{tell}(c)$ update the store to $c \sqcup d$. The action $\mathit{ask}(c)$ represents a guard, i.e. a test on the current store d, whose execution does not modify d. We say that $\mathit{ask}(c)$ is *enabled* in d iff $d \vdash c$. According to rule **R2** the *guarded choice* operator gives rise to global non-determinism: the agent $\sum_{i=1}^{n} \mathit{ask}(c_i) \to A_i$ nondeterministically selects one $\mathit{ask}(c_i)$ which is enabled in the current store, and then behaves like A_i. The external environment can then affect the choice since $\mathit{ask}(c)$ 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 $\parallel$ 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. $\exists x A = \exists^{\mathit{true}} x A$. 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. For a call involving the formal parameter a simple body replacement suffices (rule **R6**) since we are dealing with a call by name parameter mechanism.

3 Properties as (sets of) constraints

In this section we formalize the concept of partial correctness of ccp programs.

Definition 3.1 Given a constraint system $\mathcal{C}$ the syntax of properties of constraints is given by the following grammar:

$$\phi ::= c \mid \phi \wedge \psi \mid \neg\phi \mid \exists x \phi$$

Properties are built up from constraints and the usual logical operations. Logical disjunction ($\vee$) and implication ($\rightarrow$) are defined in the usual way: $\phi \vee \psi =_{\mathrm{df}} \neg(\neg\phi \wedge \neg\psi)$ and $\phi \rightarrow \psi =_{\mathrm{df}} \neg\phi \vee \psi$. A constraint c viewed as a property will be interpreted as the set of constraints d that entail c, i.e. as the upward closure $\uparrow c$ of c in $\mathcal{C}$ (wrt the $\leq$ ordering). Thus a constraint d satisfies the (basic) property c iff $d \in \uparrow c$. 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 $\exists_x c = \exists_x d$. It is shown below that the syntactic difference between $\exists_x$ and $\exists x$ indeed corresponds to a semantical difference. Formally, the semantics of a property is defined as follows:

Definition 3.2 A property will be interpreted as the set of constraints which satisfy it:

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

where $\uparrow c = \{d \mid \ c \leq d\}$, and, for a set of constraints f, the set $\exists x(f)$ denotes $\{d \mid$ there exists $c \in f.\ \exists_x c = \exists_x d\}$.

Note that 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. We have the following theorem:

Theorem 3.3 *Given a (cylindrical) constraint system $\mathcal{C}$ the set of its properties is a complemented distributive cylindrical constraint system.*

Definition 3.4 A property ϕ is called valid, notation $\models \phi$, iff every constraint c (of the given constraint system) satisfies the property, i.e. $[\![\phi]\!] = \mathcal{C}$, where $\mathcal{C}$ denotes the underlying constraint system.

Example 3.5 A constraint e satisfies the property $c \vee d$ iff e entails c or e entails d. A constraint e satisfies the property $\neg c$ iff e does not entail c. Note that $c \sqcup d$ and $c \wedge d$ are equivalent properties, however the logical operations of negation, disjunction and quantification do not in generally correspond with the operations of complement, greatest lower bound, and quantification of the underlying constraint system. For example, if $c \sqcap d$ denotes the greatest lower bound of c and d then $(c \vee d) \rightarrow (c \sqcap d)$ is easily seen to be a valid property. But since $c \sqcap d$ does not entail c or d, the reverse implication is not valid in general. A similar observation holds for a complemented constraint system. Also it is not difficult to see that $\exists x c \rightarrow \exists_x c$ is valid, but that the reverse implication does not hold.

Definition 3.6 Partial correctness *assertions* are of the form $P\ \mathit{sat}\ \phi$ where P is a process and ϕ is a property. The semantics of an assertion $P\ \mathit{sat}\ \phi$, with P closed (namely, every procedure occurring in P is declared), is given as follows:

$$\models P\ \mathit{sat}\ \phi \text{ iff } SP(P) \subseteq [\![\phi]\!]$$

where

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

Intuitively $P\ \mathit{sat}\ \phi$ holds iff every terminating computation of P (for any input c) results in a constraint d which satisfies the property ϕ. 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 *resting point* because, essentially for 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 $\langle P, d\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow$. Then $SP(P)$ can equivalently be characterized as follows:

Proposition 3.7 *For any closed process P, $SP(P) = \{d \in \mathcal{C} \mid$ there exists Q s.t. $\langle P, d\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow\}$.*

4 A calculus for CCP

In order to obtain a calculus for partial correctness assertions, we first introduce a compositional characterization of the operational semantics $SP(P)$. Technically such a denotational semantics is used to prove

R1	$\langle D.tell(c), d\rangle \longrightarrow \langle D.Stop, c \sqcup d\rangle$	
R2	$\langle D.\sum_{i=1}^{n} ask(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle D.A_j, d\rangle$	$j \in [1, n]$ *and* $d \vdash c_i$
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 1: The (standard) transition system.

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 modelled 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). This simple compositional structure of the SP semantics gives a strong evidence of the declarative nature of this paradigm since, as shown in detail in the following, it allows to view a program as a theory of the specification logic.

Definition 4.1 Given a program P, $[\![P]\!](e) : Processes \rightarrow \wp(C)$ is defined by the equations in table 2 where μ denotes the least fixpoint wrt subset inclusion of elements of $\wp(C)$ containing *false*. Here e is an environment which assigns a set of resting points to each procedure name.

Theorem 4.2 *For any closed program* P *we have* $SP(P) = [\![P]\!](e)$*,* e *arbitrary.*

The first two equations of table 2 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 $\exists xA$ are all those constraints which are equal to a resting point of A up to the information on x. Finally recursion is modelled, 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.

For deterministic agents the semantics $[\![P]\!](e)$ coincides with the denotational semantics of [26], which associates with each deterministic agent its set of resting points and which is a fully abstract characterization of its input/output behaviour. It is worth noting that also in the case of nondeterministic processes the operational semantics SP is compositional since it is well known that the input/output behaviour of nondeterministic processes is *not* compositional ([26, 7]). As shown by the previous theorem, once we abstract from the first components of the i/o pairs we obtain compositionality at the price of a loss of information. Indeed, while for deterministic agents we can extract the i/o behaviour of P from $SP(P)$, this is not possible for non-deterministic agents. However, as previously discussed, $SP(P)$ provides the information we are interested in also for non-deterministic agents since it defines exactly the strongest postcondition wrt *true*.

The above theorem allows also an interpretation of assertions P *sat* ϕ, with P arbitrary (thus including non-closed processes). Namely, we can now define $\models P\ sat\ \phi$ iff $e \models P\ sat\ \phi$, for every environment e, where $e \models P\ sat\ \phi$ iff $[\![P]\!](e) \subseteq [\![\phi]\!]$.

In table 3 we define a calculus for assertions P *sat* ϕ using the usual natural deduction style. The rules **C0-C4** are obtained essentially by a "procedural" reading of equations **D0-D4** in table 2 and by a translation of the set-theoretic notions into the corresponding logical

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

Table 2: Strongest postcondition semantics of CCP

ones. Thus $\uparrow$ (which for properties is given by their interpretation) is deleted, $\cup$ is turned into $\vee$ and $\cap$ into $\wedge$.

More precisely, from **D0** and **D1**, which are not inductively defined, we obtain two axioms. The agent *stop* satisfies the weakest property *true* (**C0**) and the agent $tell(c)$ satisfies the property c (**C1**). The corresponding operational intuitions are clear from those given for **D0** and **D1**.

The rule for non-deterministic choice (**C2**) can be justified by considering equation **D2**: a resting point of $\sum_i ask(c_i) \to A_i$ 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

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

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

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 meaning, which is different from the one of $\exists_x$ in the constraint system.

Reasoning about recursion is formalized in terms of a meta-rule (Scott-induction [28]) 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. Finally, we have the consequence rule **C7** which states that if the program P satisfies ϕ and ϕ implies ψ in the underlying logic of properties then P satisfies ψ.

A formal justification of the above calculus consists in establishing its soundness and completeness. Soundness means that every provable correctness assertion is valid: whenever $\vdash P\ sat\ \phi$ then $\models P\ sat\ \phi$. Completeness on the other hand consists in the derivability of every valid correctness assertion: whenever $\models P\ sat\ \phi$ then $\vdash P\ sat\ \phi$.

Theorem 4.3 (Soundness) *The proof system consisting of the rules* **C0-C7** *is sound. More precisely, whenever $P_1\ sat\ \phi_1, \ldots, P_n\ sat\ \phi_n \vdash P\ sat\ \phi$ and $e \models P_i\ sat\ \phi_i$, for $i = 1, \ldots, n$, then $e \models P\ sat\ \phi$.*

Proof
Induction on the length of the derivation. We treat the case when the last rule applied is the recursion rule. 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 $e \models D \setminus \{p\}.p(x)\ sat\ \phi$ we also have that $e \models D \setminus \{p\}.A\ sat\ \phi$. Let us take a particular e. We have to show that $e \models D.p(x)\ sat\ \phi$, or, in other words that $[\![D.p(x)]\!](e) \subseteq [\![\phi]\!]$. Now $[\![D.p(x)]\!](e) = \mu\Psi$, where $\mu\Psi = \bigcup_i f_i$, with $f_0 = \{false\}$ and $f_{i+1} = [\![D \setminus \{p\}.A]\!](e\{f_i/p\})$. Thus it suffices to prove by induction that for all n $f_n \subseteq [\![\phi]\!]$. The base case is obvious. Suppose that $f_n \subseteq [\![\phi]\!]$. So we have $e\{f_n/p\} \models D \setminus \{p\}.p(x)\ sat\ \phi$, and thus we infer that $e\{f_n/p\} \models D \setminus \{p\}.A\ sat\ \phi$, that is, $f_{n+1} = [\![D \setminus \{p\}.A]\!](e\{f_n/p\}) \subseteq [\![\phi]\!]$.

□

We prove completeness of the system in the sense of Cook ([11]): we assume given as additional axioms all the valid properties and we assume the expressibility of the strongest postcondition of a process P, i.e.

C0 $D.stop \; sat \; true$

C1 $D.tell(c) \; sat \; c$

C2 $$\frac{D.A_i \; sat \; \phi_i \quad \forall\, i \in \{1, \ldots, n\}}{D.\sum_i ask(c_i) \rightarrow A_i \; sat \; \bigwedge_i \neg c_i \vee \bigvee_i (c_i \wedge \phi_i)}$$

C3 $$\frac{D.A \; sat \; \phi \quad D.B \; sat \; \psi}{D.A \parallel B \; sat \; \phi \wedge \psi}$$

C4 $$\frac{D.A \; sat \; \phi}{D.\exists x A \; sat \; \exists x \phi}$$

C5 $$\frac{D \setminus \{p\}.p(x) \; sat \; \phi \vdash D \setminus \{p\}.A \; sat \; \phi}{D.p(x) \; sat \; \phi} \qquad p(x) :: A \in D$$

C6 $$\frac{D.\exists x (p(x) \parallel tell(x = y)) \; sat \; \phi}{D.p(y) \; sat \; \phi}$$

C7 $$\frac{P \; sat \; \phi \quad \phi \rightarrow \psi}{P \; sat \; \psi}$$

Table 3: A calculus for CCP

that there exists a property ϕ such that $SP(P) = [\![\phi]\!]$. Completeness then follows from an application of the recursion rule and from the following lemma:

Lemma 4.4 *Let $D = \{p_1(x_1) :: A_1, \ldots, p_n(x_n) :: A_n\}$. For every agent A in which there occur only calls of procedures of D, if $\models D.A \; sat \; \phi$ then*

$$\Phi_1, \ldots, \Phi_n \vdash A \; sat \; \phi.$$

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

Corollary 4.5 (Completeness) *Whenever $\models P \; sat \; \phi$, with P closed, then $\vdash P \; sat \; \phi$.*

Proof
Let $P = D.A$, then by the lemma above it suffices to show that $p(x) \; sat \; SP(p(x))$, p declared in D. Again by the above lemma we have

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

where, for $i = 1, \ldots, n$, $\Phi_i = p_i(x_i) \; sat \; SP(D.p_i(x_i))$ and $p(x_i) :: A_i \in D$ (note that $SP(D.A_i) = SP(p_i(x_i))$). Now a repeated application of the recursion rule gives us $\vdash p_i(x_i) \; sat \; SP(p_i(x_i))$.

□

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.

In general $SP(P)$ can be expressed in an extension of the first-order logic of properties which includes recursively defined properties. Interpreting procedure identifiers as property variables the rules **C0-C4** of the calculus allow to translate an agent into a recursively defined property. Note that the resulting property contains only positive occurrences of property variables, thus its meaning can be defined as the least fixed point of a monotonic operator on the lattice of properties. Note also that the recursion rule for procedures corresponds with the following rule for recursively defined properties:

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

assuming the property $p(x)$ to be (recursively) defined by ψ. The substitution $[\phi/p]$ applied to $p(x)$ results into ϕ, and applied to $p(y)$, y distinct from x, into $\exists x(\phi \wedge d_{xy})$.

4.1 Local non-determinism

In order to illustrate the generality of our approach for ccp languages we consider now a modification of the standard ccp operational model where guarded choice is modelled by local (i.e. internal) non-determinism. The transition system $T_l = (Conf, \longrightarrow_l)$ is then obtained by adding rule $\mathbf{R2}_l$ to rules **R1-R6**. The agent $\sum_{i=1}^{n} ask(c_i) \rightarrow A_i$ can now non-

R2$_l$ $\quad \langle \sum_{i=1}^{n} ask(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle ask(c_j) \rightarrow A_j, d\rangle \quad j \in [1,n], n > 1$

Table 4: The transition rule for local non-determinism.

deterministically select one $j \in [1,n]$ and hence behave like $ask(c_j) \rightarrow A_j$ (which is a shorthand for $\sum_{i=1}^{1} ask(c_j) \rightarrow A_j$) even if $ask(c_j)$ is not enabled in the current store. The external environment then cannot control the choice any more.

The denotational semantics now is obtained by substituting equation **D2** in table 2 by the equation **D2**$_l$ given in table 5. Accordingly (writing $\neg c_i \vee \phi_i$ as an implication), the new proof system is obtained from table 3 by replacing rule **C2** by the rule **C2**$_l$ given in table 6. Soundness and (relative) completeness of this calculus can be proved analogously to the previous case.

Note that the semantics resulting from table 2 modified by equation **D2**$_l$ is the one introduced in [18] for angelic ccp. The same semantics was used in [13] to approximate the operational semantics obtained from rules **R1-R6** (i.e. with global choice) by observing the upward closure of the set of the resting points of a process P for a given input c.

D2$_l$ $\quad [\![D.\sum_i ask(c_i) \rightarrow A_i]\!](e) = \bigcup_i (\mathcal{C} \setminus \uparrow c_i \cup [\![A_i]\!](e))$

Table 5: The equation for local non-determinism

$$\text{C2}_l \quad \frac{D.A_i \ sat \ \phi_i \quad \forall\, i \in \{1,\ldots,n\}}{D.\sum_i ask(c_i) \rightarrow A_i \ sat \ \bigvee_i (c_i \rightarrow \phi_i)}$$

Table 6: The rule for local non-determinism

4.2 Hoare logic

We have presented a calculus for assertions of the form $P \ sat \ \phi$. This allows 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 set

$$\{d \mid \ \textit{there exist}\ c \in [\![\psi]\!]\ \textit{and}\ Q \text{ s.t. } \langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow \quad \}$$

(the strongest postcondition wrt ψ). Then, assuming that $\uparrow [\![\psi]\!] = [\![\psi]\!]$, it is easy to show that for any P, $SP_\Psi(P) = SP(P) \cap [\![\psi]\!]$ and hence, from the interpretation of $\rightarrow$,

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

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 stronger 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 [20], cooperation test [4]). 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.

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* ([19]), i.e. terms that are constructed out of programs 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:

$$\begin{array}{rcl}\Phi & ::= & c \mid tell(c) \mid \sum_i ask(c_i) \rightarrow \Phi_i \mid \Phi \wedge \Phi \mid \Phi \parallel \Phi \\ & & \exists x \Phi \mid p(x) \mid \neg\Phi\end{array}$$

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 2 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 $P \rightarrow \phi$. 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 $\Phi \rightarrow \Psi$ holds. Note that the refinement relation, which is modelled 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):

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

(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

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

of implications between mixed terms $\Phi_1, \ldots \Phi_n$ where Φ_1 is the given specification ϕ and Φ_n denotes the derived program. Furthermore each of the implications $\Phi_{i+1} \rightarrow \Phi_i$ 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:

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

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.

6 Conclusions

We presented a compositional proof system which allows to prove partial correctness of concurrent constraint programs and we proved its soundness and (relative) completeness. The rules of the calculus are obtained from the definition of a simple denotational semantics which describes the set of all the resting points of a ccp process. Indeed, such a notion of observable turns out to be compositional also for the non-deterministic case and is informative enough for partial correctness since it corresponds exactly to the strongest postcondition wrt the precondition true. Our results should be considered as the starting point of a study of ccp languages which involves both theoretical and practical aspects.

¿From a theoretic point of view, one of our main contributions is to clarify the *declarative* nature of ccp languages and its advantages wrt other concurrent programming paradigms. Indeed the proof-theory we have defined allows 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 [22, 27, 29, 23]. The significance of such a line was eventually made clear by Abramsky [1] who exploited a seminal idea in [29]: 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 relations between the logics of the constraints system and the programming logic. More precisely we aim at a complete axiomatization of the logic of properties.

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 [2].

¿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 [14]. Techniques based on abstract interpretation have been used to analyze properties of ccp computations ([9, 13]), 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.

Further directions which we are currently working on include an extension of the calculus to infinite computations and to the more general case of Hoare-triples with negative preconditions, and the development of a refinement calculus for program synthesis along the lines suggested in section 5.

References

[1] S. Abramsky. Domain Theory in Logical Form. *Proceedings, Annual Symposium on Logic in Computer Science, pp. 47-53, IEEE CS*, 1987. Extended version in Annals of Pure and Applied Logic, 51: 1-77, 1991.

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

[3] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[4] K.R. Apt, N. Francez and W. P. de Roever. A proof system for Communicating Sequential Processes. ACM Transactions on Programming Languages and Systems, 2:359-385, 1980.

[5] J.A. Bergstra and J.W. Klop. Process algebra: specification and verification in bisimulation semantics. In *Mathematics and Computer Science II*, CWI Monographs, pp. 61 – 94. North-Holland, 1986.

[6] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures: Towards a paradigm for asynchronous communication. In *Proceedings of Concur '91, Lecture Notes in Computer Science*, Vol. 527, Amsterdam, The Netherlands, August 1991.

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

[8] F.S. de Boer and C. Palamidessi. A process algebra of concurrent constraint programming. *Proceedings of the Joint International Conference and Symposium on Logic Programming, JICSLP'92.*

[9] M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient Analysis of Concurrent Constraint Logic Programs. In A. Lingas, editor, *Proc. of the 20th International Colloquium on Automata, Languages, and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 633-644. Springer-Verlag, Berlin, 1993.

[10] L. Colussi, E. Marchiori. Proving Correctness of Logic Programs Using Axiomatic Semantics. *Proceedings of the 8th International Conference on Logic Programming*, The MIT Press, pp. 629-644, 1991.

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

[12] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

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

[14] I. Foster and S. Taylor. *Strand: New concepts in parallel programming.* Prentice Hall, 1989.

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

[16] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

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

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

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

[20] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. Acta Informatica, 6:319-340, 1976.

[21] P. Panangaden, V.A. Saraswat, P.J. Scott and R.A.G. Seely. A Hyperdoctrinal View of Concurrent Constraint Programming. In *Proc. REX Workshop*, LNCS 666, pages 457–476, 1992.

[22] G. Plotkin. Dijkstra's predicate transformers and Smyth's powerdomains, LNCS 86, 1980.

[23] E. Robinson. Logical aspects of denotational semantics, LNCS 283, 1987.

[24] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989. Published by The MIT Press, U.S.A., 1991.

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

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

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

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

[29] M. Smyth. Powerdomains and Predicate Transformers: A Topological View. *Automata, Languages and Programming, Proceedings 1983*, LNCS 154, pp. 662-675, 1983.

[30] J. Zwiers. Compositionalty and Partial Correctness. LNCS 321, Springer-Verlag, 1989.

A Appendix

As an example of application of the proof system, we show a derivation of the partial correctness assertion

$$p(x,y)\ sat\ (x = 0 \vee x > 0) \rightarrow y = x!,$$

for the procedure $p(x,y)$ declared as $p(x,y) :: A$ where

$$\begin{aligned} A = \quad & ask(x=0) \rightarrow tell(y=1) \\ & + \\ & ask(x>0) \rightarrow A_2 \end{aligned}$$

and

$$A_2 = \exists u, z(tell(u = x - 1) \parallel tell(y = x * z) \parallel p(u,z)).$$

(we use $+$ as a shorthand for $\sum_{i=1}^{2}$).

According to the above specification, this procedure computes the factorial of a given integer x. Here we assume that the underlying constraint domain allows to express numerical constraint on the domain of (negative and positive) integers. In the proof we use the short notation $x \geq 0$ for $x = 0 \vee x > 0$.

1. $p(x,y)\ sat\ x \geq 0 \rightarrow y = x!$
   { assumption }
2. $p(x,y) \parallel tell(x=u) \parallel tell(y=z))\ sat$
   $(x \geq 0 \rightarrow y = x!) \wedge x = u \wedge y = z$
   { from 1, $tell(x=u)\ sat\ x = u$,
   $tell(y=z)\ sat\ y = z$ and by C3 }
3. $\exists x, y\ (p(x,y) \parallel tell(x=u) \parallel tell(y=z))\ sat$
   $u \geq 0 \rightarrow z = u!$
   { by C4 and C7: Note that
   $\exists x, y\ (\ (x \geq 0 \rightarrow y = x!) \wedge x = u \wedge y = z) \rightarrow$
   $(u \geq 0 \rightarrow z = u!)$ is valid }
4. $p(u,z)\ sat\ (u \geq 0) \rightarrow z = u!$
   { from 3 and by C6 }
5. $tell(u = x-1) \parallel tell(y = x*z) \parallel p(u,z)\ sat$
   $(u \geq 0 \rightarrow z = u!) \wedge u = x - 1 \wedge y = x * z$
   { from 4, by C1 and C3 }
6. $A_2\ sat\ x > 0 \rightarrow y = x!$
   { from 5, by C4 and C7 }
7. $tell(y=1)\ sat\ y = 1$
   { by C1 }
8. $A\ sat\ (x \geq 0) \rightarrow y = x!$
   { from 6, 7 and by C7: Note that
   $(\ (\neg(x=0) \wedge \neg(x>0)) \vee$
   $(x = 0 \wedge y = 1)\ \vee\ (x > 0 \wedge (x > 0 \rightarrow y = x!))\ )$
   $\rightarrow (x \geq 0 \rightarrow y = x!)$ is valid }
9. $p(x,y)\ sat\ x \geq 0 \rightarrow y = x!$
   { from 8 and by C5 }