A Process Algebra for Synchronous Concurrent Constraint Programming

Luboš Brim* David Gilbert†
Jean-Marie Jacquet‡ Mojmír Křetínský*

Abstract

Concurrent constraint programming is classically based on asynchronous communication via a shared store. This paper presents new version of the ask and tell primitives which features synchronicity. Our approach is based on the idea of telling new information just in the case that a concurrently running process is asking for it.

An operational and an algebraic semantics are defined. The algebraic semantics is proved to be sound and complete with respect to a compositional operational semantics which is also presented in the paper.

1 Introduction

As a consequence of being a generalisation of previous proposals for concurrent logic programming languages (Concurrent Prolog, Parlog, GHC, etc.), concurrent constraint programming has naturally inherited one of their main features: the *asynchronous* character of the communication. This is obtained by ask primitives blocking when the information on the store is not complete enough to entail the asked constraints. Following these lines, a natural way of obtaining synchronous communication in concurrent constraint programming is to force the reduction of ask and tell primitives to synchronise. Specifically, our approach considers tell primitives as lazy producers of information and views ask primitives as consumers of this information. From this point of view, a tell operation is reduced when an ask operation requires the told information. Moreover, the reduction of the two primitives is performed simultaneously. However, there is no reason to block ask and tell primitives on information which is already present. Consequently, stress is put on the novelty of the information and hence any `tell`(c) and `ask`(c) operations whose

*Dept.of Comp.Sci., Masaryk University, Brno, Czech Republic, {`brim|mojmir`}`@fi.muni.cz`
‡Dept.of Comp.Sci., University of Namur, Namur, Belgium, `jmj@info.fundp.ac.be`
†Dept.of Comp.Sci., City University, London, U.K., `drg@soi.city.ac.uk`

constraint argument c is entailed by the current store are reduced without partners. The scheme is made slightly more general by permitting the synchronisation of more than two partners.

This framework, called Scc, is presented in [3] and its expressiveness has been demonstrated through the coding of a variety of examples. It has been argued that one advantage over related work such as [12, 9, 8], which introduce synchronisation by special operators and not by altering the behaviour of tell and ask primitives, is that Scc permits the specification of on *what* information the synchronisation should be made, rather than with *whom*. Synchronisation in Scc is thus data-oriented as opposed to process-oriented.

In order to motivate its interest and to substantiate the need for novel treatments, it is worth stressing the behavioural difference of Scc with, on the one hand, traditional concurrent constraint programming, as exemplified in the cc family of languages ([12]), and, on the other hand, traditional concurrent programming models, as exemplified by CCS ([10]).

It has been argued in [6] that the main difference between ccp and CCS is that complementary actions do not synchronise in ccp. This property is due to the fact that telling a constraint never suspends in cc. In contrast, the action of telling a constraint may suspend until an ask can make use of it. A synchronisation similar to that in CCS is thus produced. However, this synchronisation does not hold in Scc in the case that the told or asked constraints are entailed by the current contents of the store. A novel kind of synchronisation is thus achieved.

Major differences appear between the three frameworks. It is to be expected that these differences call for new treatments as well. In order to formalise our reasoning somewhat, let us turn to the example given in [6]. There CCS and ccp are compared by interpreting the action a as telling the constraint $x = a$, and the co-action $\overline{a}$ as asking the constraint $x = a$. To keep our notations consistent, we shall use "+" for the non-deterministic choice operator and ";" for the sequential composition operator.

Example 1 (Differentiating ccp and CCS (from [6])) *Let $A_1 = (\overline{a}; \overline{b}) + (\overline{a}; \overline{c}) + (\overline{a}; \overline{d})$ and $A_2 = (\overline{a}; \overline{b}) + (\overline{a}; (\overline{c} + \overline{d}))$. In any compositional semantics for CCS these two processes must be distinguished. Indeed, they behave differently under the context $A = a; (b + c)$. The process A_1 can deadlock, by choosing the third alternative of the choice, while A_2 cannot. However, in* cc*, both A_1 and A_2 have the same behaviour. The process A_2 can deadlock by choosing the second alternative, because A can independently decide to produce $y = b$ (after $x = a$).*

Example 2 (Differentiating ccp and Scc) *Using the processes A, A_1, A_2 of the above example, the processes A_1 and A_2 are also distinguished by A in* Scc *for the same reason as in CCS.*

This example illustrates the difference between Scc and cc. Stated in other terms, in the ccp paradigm, since the tell operation is asynchronous, the choice guarded by *tell*(a) is a *local choice* whereas, since the tell operation is synchronous in `Scc`, this choice is *global* in `Scc`.

Nevertheless, synchronisation is only forced in `Scc` in the case that a process tries to tell information which is not already entailed by the store. Otherwise, it can proceed asynchronously. This fact is used subsequently to differentiate CCS and `Scc`.

Example 3 (Differentiating CCS and `Scc`) *Using again the above processes A, A_1, A_2, let $B_1 = \overline{b}; A_1$ and $B_2 = \overline{b}; A_2$. In CCS, these two processes can be distinguished by the process $B = b; A$ for the reasons exposed in Example 1. However, in `Scc`, both processes have the same behaviour. The process B_2 can now deadlock by choosing the second alternative because A can now independently proceed by the first alternative as $y = b$ is already entailed by the store.*

The distinction between `Scc` and CCS thus appear to be more subtle than the distinction between ccp and CCS. The choice guarded by `tell` is actually a “mixture” of global and local choice. The choice depends upon actions performed and upon the results of the past behaviour of the system, i.e. upon the constraint contained in the store.

The rest of the paper is organised as follows. Section 2 describes informally `Scc` and section 3 presents the basic operational semantics $\mathcal{O}$. Section 4 studies the algebraic semantics. To prove its soundness and completeness, we introduce a compositional model $\mathcal{M}$ in the section 5 and relate it with the algebraic semantics. Finally, section 6 draws our conclusions.

2 The Language `Scc`

This section presents the syntax and the informal semantics of the language underlying the `Scc` paradigm, also called `Scc` by abuse of language. For reasons of simplicity we consider a simplified version of `Scc` which does not contain recursion. Recursion can however be treated in the standard way described in [2].

As in [14], the constraint system underlying `Scc` consists of any system of partial information that supports the entailment relation. We assume a given cylindric constraint system $\langle C, \vdash \rangle$ over a set of variables *Svar*, defined as usual from a simple constraint system $\langle D, \vdash \rangle$. Furthermore, for the purposes of axiomatisation we will suppose that the cylindric constraint system is embedded into a *complemented and distributive* one denoted by *dc(D)*. For detailed definitions we refer to [14, 7, 4].

The language description is parametric with respect to $\langle C, \vdash \rangle$, and so are the semantic constructions presented.

In the following, we use $G, H, \ldots$ possibly subscripted to range over the set *Sgoal* of processes $c, d, \ldots$ to range over basic constraints (i.e. constraints which are equivalent to a finite set of primitive constraints), and $X, Y, \ldots$ to range over the subsets of *Svar*.

Processes $G \in \mathit{Sgoal}$ are defined by the following grammar

$$G ::= \triangle \mid \delta \mid \mathtt{ask}(c) \mid \mathtt{tell}(c) \mid G;G \mid G+G \mid G \parallel G \mid \exists_X G$$

The constant $\triangle$ denotes a process which is only capable of terminating successfully. The constant δ is used to denote deadlocking processes.

The atomic constructs `ask(c)` and `tell(c)` act on a given store in the following way: as usual, given a constraint `c`, the process `ask(c)` succeeds if `c` is entailed by the store, otherwise it is suspended until it can succeed. However, the process `tell(c)`, of a more lazy nature than the classical one, succeeds only if `c` is (already) entailed by the store and in this case it does not modify the store, and suspends otherwise. It is resumed by a concurrently suspended `ask(d)` operation provided that the conjunction of `c` and of the store entails `d`. In that case, both the `tell` and the `ask` are resumed synchronously and at the same time the store is atomically augmented with the constraint `c`.

The sequential composition $G_1; G_2$ is executed by first performing G_1 and, if G_1 terminates successfully, by performing G_2.

The nondeterministic choice $G_1 + G_2$ selects between the execution of G_1 or G_2 respectively provided that the selected component can perform at least one step of a computation (i.e. it is not immediately suspended). It is a *global nondeterministic* choice since the selection of a component can be influenced by the (global) store and by the environment of the process as well.

The parallel composition $G_1 \parallel G_2$ represents both the interleaving (merge) of the computation steps of the components involved (provided they can do these steps independently of each other) and also synchronisation: this is the case of the `tell` and `ask` described above. Note that in the general case there can be a parallel composition of a finite sets of `tell`'s and a finite set of `ask`'s such that the current store and a conjunction of some of the `tell` constraints entail the conjunction of the `ask` constraints and the other `tell`'s. In this case all the components are reduced simultaneously. This is sometimes referred to in the literature as *multi-party* synchronous communication.

The block construct $\exists_X G$ behaves like a process G with the variables in X considered as local. It hides the information about variables from X within the process G. Let us recall that full version of `Scc` contains recursive procedures too (see an example below), but for ease of reasoning they are not studied here.

To ilustrate some features of `Scc` (synchronous multi-party communication) we present the dining philosophers example. We have chosen to represent both philosophers and forks as processes; in addition communication is achieved via shared stream variables as in classical concurrent logic programming.

The inital description of the system is

:- phil(L1,R1) || fork(R1,L2) || phil(L2,R2) || ... ||fork(Rn,L1)

and the processes are described as follows:

```
fork(R,L):- ask(L=[taken|L′]); ask(L′=[free|L″]); fork(R,L″)
            +
            ask(R=[taken|R′]); ask(R′=[free|R″]); fork(R″,L)

phil(L,R):- tell(L=[taken|L′] , R=[taken|R′]);
               tell(L′=[free|L″] , R′=[free|R″]); phil(L″,R″)
```

Finally, it is worth observing that it is quite easy to recover the traditional `cc` paradigm from our framework by the introduction of an asynchronous tell by providing, for each constraint to be told, a concurrent corresponding ask operation.

3 The operational semantics $\mathcal{O}$

It turns out that it is possible to treat the sequential and parallel composition operators in a very similar way by introducing the auxiliary notion of *context*. Basically, a context consists of a partially ordered structure where place holders (subsequently referred to by $\Box$) have been inserted at a top-level place i.e. a place not constrained by the previous execution of other atoms. Viewing goals as partially ordered structures too, the ask and tell primitives to be reduced are those which can be substituted by a place holder $\Box$ in a context. Furthermore, the goals resulting from the reductions are essentially obtained by substituting the place holder by the corresponding syntax structure or the $\triangle$, depending upon whether an atom or a ask/tell primitive is considered.

The formal definition of the contexts is a very standard one and can be found e.g. in [4]. Moreover, we further state that the structure $(Sgoal, ; , \|, \triangle)$ is a bimonoid, that is, " ;" and " $\|$" are associative binary operations and have $\triangle$ as neutral element. In the following, we will also simplify the goals resulting from the application of contexts accordingly.

The operational semantics of `Scc` is defined in Plotkin's style [11] by means of a transition system, where *Sstore* denotes the set of stores.

Definition 1 *Define the transition relation* $\rightarrow$ *as the smallest relation of* $(Sgoal \times Sstore) \times (Sgoal \times Sstore)$ *satisfying the rules of Figure 1.*

Rules (H) and (C) express the classical treatment of hiding and of choice, respectively. Classical rules for the sequential and parallel composition operators are tackled by means of the notion of context.

Rule (T) defines the reduction of `tell` and `ask` primitives. The primitives to be reduced, there referred to as sp_1, ..., sp_m, are partitioned in three categories:

i) the ask primitives (the multi-set $\{ask(a_1), \cdots, ask(a_p)\}$);
ii) the tell primitives split into those which add information to the store (the multi-set $\{tell(rt_1), \cdots, tell(rt_r)\}$) and those which do not (the multi-set $\{tell(at_1), \cdots, tell(at_q)\}$);

All these primitives are then simultaneously reduced to the empty goal $\triangle$ when the information on the current store (σ) together with the new information told ($rt_1, \ldots, rt_r$) entails the information of the other primitives. The new store consists in this case of the old store enriched by the new information told. Note that this rule reflects the laziness feature of our tell primitives.

In the case a constraint c is entailed by the current store σ an `ask(c)` primitive can be reduced *alone* following rule (T) by taking the unary context $\Box$, $m = 1$, $p = 1$, $q = 0$, $r = 0$ and `tell(c)` can be reduced *alone* following rule (T) by taking the unary context $\Box$, $m = 1$, $p = 0$, $q = 1$, $r = 0$.

Other tell's and ask's need each other for reduction and reduce simultaneously. A minimality condition is required to forbid outsider tell's to be reduced by taking advantage of a concurrent reduction.

Tell and ask reduction

(T) $$<tc[sp_1, \cdots, sp_m], \sigma> \;\rightarrow\; <tc[\triangle, \cdots, \triangle], \tau>$$

$$\mathit{if} \left\{ \begin{array}{l} \{sp_1, \cdots, sp_m\} = \{ \;\; ask(a_1), \cdots, ask(a_p), \\ \qquad\qquad\qquad\qquad tell(at_1), \cdots, tell(at_q), \\ \qquad\qquad\qquad\qquad tell(rt_1), \cdots, tell(rt_r) \;\; \} \\ \sigma \cup \{rt_1, \cdots, rt_r\} \vdash \{a_1, \cdots, a_p\} \cup \{at_1, \cdots, at_q\} \\ \text{there is no strict subset } S \text{ of } \{rt_1, \cdots, rt_r\} \\ \quad \text{such that } \sigma \cup S \vdash \{a_1, \cdots, a_p\} \cup \{at_1, \cdots, at_q\} \\ \tau = \sigma \cup \{rt_1, \cdots, rt_r\} \\ m > 0 \end{array} \right\}$$

Hiding

(H) $$\frac{<tc[G[Y/X]], \sigma> \rightarrow <tc[G'], \sigma'>}{<tc[\exists_X G], \sigma> \rightarrow <tc[G'], \sigma'>} \qquad \mathit{if}\ \{Y \text{ is a fresh variable } \}$$

Choice

(C) $$\frac{<G, \sigma> \rightarrow <G'', \sigma''>}{\begin{array}{l} <G + G', \sigma> \rightarrow <G'', \sigma''> \\ <G' + G, \sigma> \rightarrow <G'', \sigma''> \end{array}}$$

Figure 1: `Scc` transition system

We are now in a position to define the operational semantics. Following the logic programming tradition, it specifies the final store of the successful computations.

It also indicates those stores associated with deadlock situations, namely situations corresponding either to the absence of suitable data on the store or to the absence of concurrent process(es) that would allow tell and ask primitives to proceed, i.e. to resume suspended tell's and ask's. Note that the two situations may be distinguished by a simple criterion: the existence of a store richer than the current one that would enable the computation to proceed. Note also that real failure, corresponding to the absence of suitable procedure declaration, does not occur since recursion is not treated here.

The following definition follows directly from this intuition. The symbols δ^+ and δ^s are used to indicate the computations ending by a success and a suspension, respectively.

Definition 2 *Define the operational semantics $\mathcal{O} : Sgoal \rightarrow \mathcal{P}(Sstore \times \{\delta^+, \delta^s\})$ as the following function: for any goal G,*

$$\begin{array}{rcll} \mathcal{O}(G) & = & \{ & <\tau, \delta^+> : \ <G, true> \rightarrow \cdots \rightarrow <\triangle, \tau> \quad \} \\ & & \cup \ \{ & <\tau, \delta^s> : \ <G, true> \rightarrow \cdots \rightarrow <G', \tau> \not\rightarrow \\ & & & \quad G' \neq \triangle \textit{ and there are } \sigma', G'', \sigma'' \\ & & & \quad \textit{such that } <G', \sigma'> \rightarrow <G'', \sigma''> \quad \}. \end{array}$$

4 Algebraic Semantics

Now we describe an axiomatisation in the style of process algebras for Scc. Part of the axioms is borrowed from traditional work on process algebras (see e.g. [2]) and from work already published on concurrent constraint programming (see e.g. [7]). Other axioms are specific to our work (see [4] for comparisson of our axioms with the axioms for asynchronous ccp given by deBoer and Palmidessi [7]).

Axioms from general process algebra

The first group (**A**) consists of axioms in Figure 2. They deal with the general requirements found in any process algebra of communicating systems. As usual, we use two auxiliary operators to axiomatise the parallel composition operator $\|$. They are $\|\!\!\lfloor$ for left merge and $|$ for communication merge. In the following $\alpha, \beta, \alpha(c)$ and $\beta(c)$ represent `ask`s or `tell`s on constraints, the constraint c when an axiom is parametric with respect to it. The system **A** axiomatises a notion of equivalence which is known as bisimulation (see e.g. [2]).

The next group of axioms (**T**) in Figure 3 permits the abstraction from silent steps τ. In the context of concurrent constraint programming τ corresponds to a `tell`(c) or `ask`(c) action where c is entailed by the current store, e.g. `tell`$(true)$ or `ask`$(true)$. Failure is axiomatised by axioms (**F**) given in Figure 3 (cf. [2], p.215).

A axioms							**General**
$(A1)$	$A+A$	$=$	A	$(A6)$	$\delta; A$	$=$	δ
$(A2)$	$A+B$	$=$	$B+A$	$(A7)$	$\delta + A$	$=$	A
$(A3)$	$A+(B+C)$	$=$	$(A+B)+C$	$(A8)$	$\Delta; A$	$=$	A
$(A4)$	$A;(B;C)$	$=$	$(A;B);C$	$(A9)$	$A; \Delta$	$=$	A
$(A5)$	$(A+B);C$	$=$	$A;C+B;C$				
$(A10)$	$A \parallel B$	$=$	$A \lfloor\!\lfloor B + B \lfloor\!\lfloor A + A \mid B$				
$(A11)$	$(A+B) \lfloor\!\lfloor C$	$=$	$(A \lfloor\!\lfloor C) + (B \lfloor\!\lfloor C)$				
$(A12)$	$(\alpha; A) \lfloor\!\lfloor B$	$=$	$\alpha; (A \parallel B)$				
$(A13)$	$(A+B) \mid C$	$=$	$A \mid B + A \mid C$	$(A16)$	$\Delta \lfloor\!\lfloor A$	$=$	δ
$(A14)$	$A \mid (B+C)$	$=$	$A \mid B + A \mid C$	$(A17)$	$\Delta \mid A$	$=$	δ
$(A15)$	$\alpha; A \mid \beta; B$	$=$	$(\alpha \mid \beta); (A \parallel B)$	$(A18)$	$A \mid \Delta$	$=$	δ

Figure 2: A-axioms

T and F axioms			**τ-abstraction and Failure**
$(T1)$	$A; \tau$	$=$	A
$(T2)$	$\tau; A + B$	$=$	$\tau; A + \tau; (A+B)$
$(F1)$	$\alpha; (\beta; A_1 + B_1) + \alpha; (\beta; A_2 + B_2)$	$=$	$\alpha; (\beta; A_1 + \beta; A_2 + B_1)$
			$+\alpha; (\beta; A_1 + \beta; A_2 + B_2)$
$(F2)$	$\alpha; (\beta + B_1) + \alpha; (\beta; A + B_2)$	$=$	$\alpha; (\beta + \beta; A + B_1) + \alpha; (\beta + \beta; A + B_2)$
$(F3)$	$\alpha; A + \alpha; (B+C)$	$=$	$\alpha; A + \alpha; (A+B) + \alpha; (B+C)$

Figure 3: T&F-axioms

Axioms from work on ccp

The group of axioms (**H**) in Figure 4 axiomatises hiding (quantification) in terms of the auxiliary operator $\exists_x^c$.

To axiomatise the communication merge $\mid$ we shall also need another auxiliary operator $\exists^c$. To that end we extend the notion of cylindric constraint system by adding the *identity* function $\exists : C \rightarrow C$. $\exists$ clearly satisfies the required conditions. Hence, we must add an extra transition rule to the rules from Figure 1 to cover $\exists^c$:

H axioms							**Hiding (Quantification)**
$(H1)$	$\exists x.A$	$=$	$\exists_x^{\mathrm{true}} A$	$(H4)$	$\exists_x^c.\delta$	$=$	δ
$(H2)$	$\exists_x^c.\Delta$	$=$	Δ	$(H5)$	$\exists_x^c.\alpha(d); A$	$=$	$\alpha(\forall_x(c \rightarrow d)); \exists_x^c.A$
$(H3)$	$\exists_x^c.(A+B)$	$=$	$\exists_x^c.A + \exists_x^c.B$	$(H6)$	$\exists^c.\mathtt{ask}(d)$	$=$	Δ if $c \vdash d$

Figure 4: H-axioms

L axioms			
$(L1)$	$\alpha(c); (\beta(d); A + B)$	$=$	$\alpha(c); (\beta(c \wedge d); A + B)$
$(L2)$	$\alpha(c); (\beta(c); A + B)$	$=$	$\alpha(c); (\tau; A + B)$
$(L3)$	$\mathtt{ask}(c); (\mathtt{ask}(d); A + B)$	$=$	$\mathtt{ask}(c); (\mathtt{ask}(d); A + B) + \mathtt{ask}(c \wedge d); A$
$(L4)$	$\mathtt{ask}(c) \mid \mathtt{tell}(d)$	$=$	$\mathtt{ask}(c) \mid \mathtt{tell}(d) + \mathtt{ask}(d) \mid \mathtt{tell}(d)$ if $d \vdash c$
$(L5)$	$\mathtt{ask}(c); A + \mathtt{ask}(c \wedge d); B$	$=$	$\mathtt{ask}(c); A + \mathtt{ask}(c); (A + \mathtt{ask}(d); B)$
$(L6)$	$(\mathtt{tell}(c) \mid \mathtt{ask}(d)); A + B$	$=$	$\exists^c(\mathtt{ask}(d)); (\mathtt{tell}(c) \mid \mathtt{ask}(c)); A + B$
$(L7)$	$\mathtt{tell}(c) \mid \mathtt{tell}(d) \mid \mathtt{ask}(e)$	$=$	$\mathtt{tell}(c \wedge d) \mid \mathtt{ask}(e)$ if $c \wedge d \vdash e, c \not\vdash, d \not\vdash e$
$(L8)$	$\mathtt{ask}(c) \mid \mathtt{ask}(d)$	$=$	$\mathtt{ask}(c \wedge d)$
$(L9)$	$\alpha(c); (\beta(d); A + B)$	$=$	$\alpha(c); (\beta(d); A + B) + \beta(d); \alpha(c); A$
$(L10)$	$\sum_i \alpha; \sum_j \mathtt{ask}(c_{i_j}); A_{i_j}$	$=$	$\sum_i \alpha; \sum_j \mathtt{ask}(c_{i_j}); A_{i_j} + \alpha; \sum_k \mathtt{ask}(c_k); A_k$
	if for all $f \in I \leftarrow J, k \in K \subseteq \{i_j \mid i \in I, j \in J\} : \mathtt{ask}(\wedge_i c_{i_{f(i)}}) \in \bigcup_{i,j}\{\mathtt{ask}(c_{i_j}); A_{i_j}\}$ whenever $\wedge_i c_{i_{f(i)}} \not\vdash c_k$		

Figure 5: L-axioms

$$\text{(R)} \qquad \frac{<tc[G], \sigma \wedge \rho> \;\rightarrow\; <tc[G'], \sigma'>}{<tc[\exists^{\rho}.G], \sigma> \;\rightarrow\; <tc[\exists^{\sigma'}.G'], \sigma \wedge \sigma'>}$$

The idea of hiding the empty set of variables is to allow a separate local computation. A local computation step proceeds from a store which consists of the *entire* global store seen "locally". The resulting global store of a local computation is the same as the one before starting the local computation.

Axioms specific to Scc

Now we present a group of axioms (**L**) which characterise the specific treatment of "lazy" tell in concurrent constraint programming. The axioms are presented in Figure 5.

The axiom (L1) expresses that once a constraint has been established by telling it, it remains in the store. The axiom (L2) expresses that asking or telling an entailed constraint results in a silent action τ.

The axiom (L3) permits the composition of ask actions, and the axiom (L4) permits the strengthening of an ask-guard in a suitable parallel context. The axiom (L5) allows the restricted decomposition of `ask` actions.

The justification of (L6) is as follows. Suppose that the current store together with c entails d. In this case the left-hand side process synchronizes, allowing both components to proceed, and the resulting store is enriched by c. The right-hand side process results in exactly the same store, namely it first "checks" whether the current store together with c implies d by performing a local computation of $\mathtt{ask}(d)$ and if this is true it then just "tells" c. Note that if this occurs in a parallel context, then telling c means "waiting" for a partner who asks for this information.

The axiom (L7) permits the restricted composition of `tell` actions and reflects our *minimality* condition (see rule (T) in Figure 1). The axiom (L8) permits the composition of `ask` actions.

The axiom (L9) is informally justified as follows. Suppose that the current store implies the constraint d. In this case the process represented by the right-hand side of the axiom can select the β branch, execute the α action and proceed with A. But this behaviour can be mimicked by the other branch, the order of the actions being unobservable. In the case the current store does not imply d, the only choice left is to execute the α branch.

5 Soundness and Completeness

This section discusses the soundness and completeness properties of our axiomatisation. Classically, algebraic theories identify computations which not only exhibit the same final results but also behave identically when they are placed in any context. We are thus lead to relate our algebraic semantics with a compositional semantics. However, the semantics $\mathcal{O}$ is not compositional, as shown by considering the goals $tell(c)$, $ask(c)$, $tell(c) \parallel tell(c)$, and $tell(c) \parallel ask(c)$: for any store σ such that $\sigma \not\vdash c$, the first three suspend whereas the last succeeds. Consequently, we first define a compositional operational semantics, prove that it is correct with respect to the semantics $\mathcal{O}$ and then relate it to the algebraic semantics just developed.

A Compositional Operational Semantics

Two problems need to be tackled in order to transform $\mathcal{O}$ into a compositional semantics. Firstly, the semantics should be modified in order to allow suspended goals to resume thanks to the store as computed by concurrent goals. This is subsequently achieved by reporting in the semantics not just the final results coupled to a status mark but sequences corresponding to the computation steps. Progress made by the concurrent goals is then indicated by steps of the form $\prec\sigma,\tau\succ^{e}$ indicating the update of the store σ in the store τ. Secondly, tell and ask primitives of goals should also be able to synchronise with primitives provided by concurrent processes. This is subsequently achieved by introducing in semantic sequences the steps of the form $\prec\sigma,\sigma'\succ^{(A,At,Rt)}$, where the (A, At, Rt) triple denotes the actors of the synchronisation, according to the three categories detailed in section 3. In fact each of these multi-sets is a pair composed of (the store argument of) the primitives of the considered goal and of (the store argument of) the primitives provided by its concurrent processes.

It is possible to extend the transition system of Figure 1 so as to reflect these extensions. The slight modifications to rules (T), (H), and (C) are given in Figure 6. They consist of adding labels describing the pair of initial and final values of the store for the steps under consideration. Note that rule (R) introduced in section 4 can likewise be modified straightforwardly.

(T) $$<tc[sp_1,\cdots,sp_m],\sigma> \xmapsto{\prec\sigma,\tau\succ} <tc[\triangle,\cdots,\triangle],\tau>$$

$$\mathit{if}\left\{\begin{array}{l} \{sp_1,\cdots,sp_m\} = \{\ \begin{array}[t]{l} ask(a_1),\cdots,ask(a_p), \\ tell(at_1),\cdots,tell(at_q), \\ tell(rt_1),\cdots,tell(rt_r)\quad \} \end{array} \\ \sigma\cup\{rt_1,\cdots,rt_r\}\vdash\{a_1,\cdots,a_p\}\cup\{at_1,\cdots,at_q\} \\ \text{there is no strict subset } S \text{ of } \{rt_1,\cdots,rt_r\} \\ \qquad \text{such that } \sigma\cup S\vdash\{a_1,\cdots,a_p\}\cup\{at_1,\cdots,at_q\} \\ \tau=\sigma\cup\{rt_1,\cdots,rt_r\} \\ m>0 \end{array}\right\}$$

(H) $$\frac{<tc[G[Y/X]],\sigma> \xmapsto{l} <tc[G'],\sigma'>}{<tc[\exists_X G],\sigma> \xmapsto{l} <tc[G'],\sigma'>} \quad \mathit{if}\ \{Y \text{ is a fresh variable }\}$$

(R) $$\frac{<tc[G],\sigma\wedge\rho> \xmapsto{l} <tc[G'],\sigma'>}{<tc[\exists^{\rho}.G],\sigma> \xmapsto{l} <tc[\exists^{\sigma'}.G'],\sigma\wedge\sigma'>}$$

(C) $$\frac{<G,\sigma> \xmapsto{l} <G'',\sigma''>}{\begin{array}{l} <G+G',\sigma> \xmapsto{l} <G'',\sigma''> \\ <G'+G,\sigma> \xmapsto{l} <G'',\sigma''> \end{array}}$$

Figure 6: Reformulation of the Scc transition system

Extensions required to achieve compositionality are given in Figure 7. Rule (E) deals with the first problem mentioned above. Rule (A) provides a solution to the second problem: a synchronisation similar to that of rule (T) is performed but by means of concurrent tell and ask primitives.

Rules (Sn) and (Ss) deal with suspension and success. The last one reflects the intuition already embodied in definition 2.

The treatment of suspension requires some care. Our solution, illustrated in rule (Sn), is to report, in a single mark, for each suspended configuration, the set of stores and the set of concurrent tell and ask primitives (split as before) that would allow a transition to take place. Suspension marks disappear when combining components in the case where the tell and ask primitives required by the components are mutually provided.

The intuition has now been provided for the following definitions.

Environment

(E) $$<G, \sigma> \xmapsto{\prec \sigma, \sigma' \succ^{e}} <G, \sigma'> \qquad \textit{if} \{\ \sigma' \vdash \sigma,\ \ G \neq \triangle\ \}$$

Reduction by means of auxiliary tell's and ask's

(A) $$<tc[sp_1, \cdots, sp_m], \sigma> \xmapsto{\prec \sigma, \tau \succ^{(A, At, Rt)}} <tc[\triangle, \cdots, \triangle], \tau>$$

$$\textit{if} \left\{ \begin{array}{l} \{sp_1, \cdots, sp_m\} = \ \{ \ \ ask(a_1), \cdots, ask(a_p), \\ \qquad\qquad\qquad\qquad tell(at_1), \cdots, tell(at_q), \\ \qquad\qquad\qquad\qquad tell(rt_1), \cdots, tell(rt_r) \quad \} \\ \sigma \cup \{rt_1, \cdots, rt_s\} \cup \{rrt_1, \cdots, rrt_i\} \\ \qquad \vdash \ \{a_1, \cdots, a_p\} \cup \{aa_1, \cdots, aa_k\} \cup \{at_1, \cdots, at_q\} \\ \qquad\quad \cup \{aat_1, \cdots, aat_l\} \\ \text{there is no strict subset } S \text{ of} \\ \qquad \{rt_1, \cdots, rt_s\} \cup \{rrt_1, \cdots, rrt_i\} \text{ such} \\ \qquad \text{that } \sigma \cup S \vdash \{a_1, \cdots, a_p\} \cup \{aa_1, \cdots, aa_k\} \cup \{at_1, \cdots, at_q\} \\ \qquad\qquad\qquad\quad \cup \{aat_1, \cdots, aat_l\} \\ A = (\{a_1, \cdots, a_p\}, \{aa_1, \cdots, aa_j\}) \\ At = (\{at_1, \cdots, at_q\}, \{aat_1, \cdots, aat_k\}) \\ Rt = (\{rt_1, \cdots, rt_r\}, \{rrt_1, \cdots, rrt_i\}) \\ \tau = \sigma \cup \{rt_1, \cdots, rt_r\} \cup \{rrt_1, \cdots, rrt_i\} \\ m > 0 \end{array} \right\}$$

Suspension

(Sn) $$<G, \sigma> \xmapsto{\prec \sigma, \delta^{s(\Omega, S)} \succ} <Susp, \sigma>$$

$$\textit{if} \left\{ \begin{array}{l} G \neq \triangle \\ <G, \sigma> \not\rightarrow \\ \text{there are } G'', \sigma', \sigma'' \text{ such that } \sigma' \vdash \sigma \text{ and } <G, \sigma'> \rightarrow <G'', \sigma''> \\ \Omega \text{ is the set of all such stores } \sigma' \\ S \text{ is the set of all triples } (A, At, Rt) \text{ fulfilling the conditions} \\ \qquad \text{of rule (A) together with some } sp_1, \ldots, sp_m \text{ such that} \\ \qquad G = tc[sp_1, \cdots, sp_m] \end{array} \right\}$$

Success

(Ss) $$<\triangle, \sigma> \xmapsto{\prec \sigma, \delta^{+} \succ} <Succ, \sigma>$$

Figure 7: Rules induced by the compositionality requirement

Definition 3

1) *The set* Sspair *of synchronisation pairs is defined as the set* $(Sstore \times Sstore) \times (Sstore \times Sstore) \times (Sstore \times Sstore) \times (Sstore \times Sstore) \times (Sstore \times Sstore)$.
2) *The set* Smark *of marks is defined as the set* $\{e\} \cup Sspair$. *Marks are typically denoted by the letter M, possibly subscripted.*
3) *The set* Sstep *of steps is defined as the set* $(Sstore \times Sstore) \cup (Sstore \times Sstore \times Smark)$. *Steps are typically denoted in an exponential fashion as* $\prec\sigma, \tau\succ^{[M]}$; *the notation* $[M]$ *indicating a possibly absent mark.*
4) *The set* Sterm *of terminators is defined as the set* $Sstore \times (\{\delta^+\} \cup \{\delta^{s(\Omega,S)} : \Omega \subseteq Sstore, S \subseteq Sspair\})$. *In the following, terminators are typically denoted as* $\prec\sigma, \delta^{\ddagger}\succ$, *Terminators not involving* δ^+ *are typically denoted as* $\prec\sigma, \delta^S\succ$, *when there is no need to specify further the mark S.*
5) *The set of semantic histories is defined as* $Shist = Sstep^{<\omega} \times Sterm$

Definition 4 *Let Susp and Succ be fresh names appearing in no set previously mentioned. Define the transition relation* $\mapsto$ *as the smallest relation of* $((Sgoal \cup \{Susp, Succ\}) \times Sstore) \times (Sstep \cup Sterm) \times ((Sgoal \cup \{Susp, Succ\}) \times Sstore)$ *satisfying the rules of Figures 6 and 7, where P is assumed to be given and the notation* $<G, \sigma> \xmapsto{l} <G', \sigma'>$ *is employed instead of* $(<G, \sigma>, l, <G', \sigma'>) \in \mapsto$.

The operational semantics $\mathcal{M}$ basically collects the labels appearing in the finitely ended computations. To obtain compositionality, those computations are allowed to start in any store.

Definition 5 *Define the operational semantics* $\mathcal{M} : Sgoal \to \mathcal{P}(Shist)$ *as the following function: for any goal G,*

$$\begin{array}{rcl} \mathcal{M}(G) & = & \{ \; l_1 \cdots l_m : \; <G, \sigma> \xmapsto{l_1} \cdots \xmapsto{l_m} C, \\ & & \qquad \sigma \in Sstore, C \in (\{Succ, Susp\}) \times Sstore \;\; \} \end{array}$$

The semantics $\mathcal{M}$ can be proved compositional. It is also correct with respect to the semantics $\mathcal{O}$ in the following sense. A few definitions are first in order.

Definition 6 *Let* $h = \prec\sigma_1, \tau_1\succ^{[M_1]} \ldots \prec\sigma_n, \tau_n\succ^{[M_n]}.\prec\sigma_{n+1}, \delta^{\ddagger}\succ$ *be a history,*$n \geq 0$.

1) *h is* continuous *iff there is no gap of stores between the steps ie iff* $\sigma_{i+1} = \tau_i$ *for* $i = 1, \ldots, n$.
2) *h* starts in σ *iff* $\sigma_1 = \sigma$.
3) *h is* non-hypothetical *iff it contains no step of* $Sstore \times Sstore \times Smark$.
4) $\prec\sigma_{n+1}, \delta^{\ddagger}\succ$ *is called the final step of h and h is said to end by this step.*

For any goal G, the semantics $\mathcal{O}(G)$ can be obtained from $\mathcal{M}(G)$ in the following way: (i)by retaining the histories of the latter which are continuous, non-hypothetical, and which start in *true*; (ii) by taking their final steps; (iii) by changing all the symbols δ^S by the symbol δ^s.

Proposition 7 *Define the function* $\alpha : Sstore \to \mathcal{P}(Shist) \to \mathcal{P}(Sstore \times \{\delta^+, \delta^s\})$ *as follows: for any store* $\sigma \in Sstore$*, any subset* $S \subseteq Shist$*,*

$$\begin{array}{rlll} \alpha(\sigma)(S) = & \{ \prec\tau, \delta^+\succ : & h \in S, h \textit{ continuous, non-hypothetical} & \\ & & h \textit{ starts in } \sigma, h \textit{ ends by } \prec\tau, \delta^+\succ & \} \\ & \cup \{ \prec\tau, \delta^s\succ : & h \in S, h \textit{ continuous, non-hypothetical} & \\ & & h \textit{ starts in } \sigma, h \textit{ ends in } \prec\tau, \delta^S\succ & \}. \end{array}$$

Then, for any program P *and any goal* G*,* $\mathcal{O}(G) = \alpha(true)(\mathcal{M}(G))$*.*

Soundness and completeness

We finally relate the algebraic semantics with the semantics $\mathcal{M}$. This is achieved following the classical lines, illustrated among others in [2].

The proof of soundness of our axiomatisation consists of a simple verification establishing $\mathcal{M}$(G) = $\mathcal{M}$(H), for any axiom $G = H$. As far as completeness is concerned, every process is first proved equal to a *basic process*, namely a process built up from the `ask`, `tell` and (`tell` | `ask`) constructs, and Δ, δ constants using sequencing and choice only. Completeness is then established for basic processes by inductively reasoning on their structure.

Summing up, the following theorem can be established.

Theorem 8 *For any processes* G *and* H

$$\vdash G = H \textit{ iff } \mathcal{M}(G) = \mathcal{M}(H)$$

An interesting relationship between the algebraic semantics and the semantics $\mathcal{O}$ can be derived therefrom and from proposition 7.

Proposition 9 *For any processes* G *and* H*, if* $\vdash G = H$ *then* $\mathcal{O}(G) = \mathcal{O}(H)$*.*

6 Conclusions

This paper has presented an algebraic semantics for a synchronous version of concurrent constraint programming. This version is based on new variants of the tell and ask primitives where the former behave as lazy producers of new pieces of information which are consumed by the latter at the same time they are produced. Stress on the novelty of the information permits tell and ask to proceed asynchronously in the case their corresponding constraint argument is entailed by the current contents of the store. This framework, called `Scc`, has been argued to be (in some aspects) different from traditional algebraic languages, exemplified by CCS, and from classical concurrent constraint programming, exemplified by the cc family of languages.

These differences call for semantic treatments, which have been proposed and compared with existing ones. Soundness and completeness of the proposed axiomatic system have been established by relating it to a compositional operational semantics, which has also been presented.

References

[1] Krzysztof Apt, editor. *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Washington, USA, 1992. The MIT Press.

[2] J.C.M. Baeten and W.P Weijland. *Process Algebra.* Cambridge University Press, 1990.

[3] L. Brim, J-M. Jacquet, D. Gilbert, and M. Křetínský. New Versions of Ask and Tell for Synchronous Communication in Concurrent Constraint Programming. Technical Report No. 1996/03. ISSN 1364-4009, Northampton Square, London EC1V 0HB, 1996.

[4] L. Brim, J-M. Jacquet, D. Gilbert, and M. Křetínský. A Process Algebra for Synchronous Concurrent Constraint Programming. Technical Report No. 1996/06. ISSN 1364-4009, Northampton Square, London EC1V 0HB, 1996.

[5] F. S de Boer, J. W. Klop, and C. Palamidessi. Asynchronous communication in process algebra. In *Proceedings, 7th Annual IEEE Symp. on Logic in Computer Science*, pages 137–147. IEEE Computer Society Press, 1992.

[6] 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/CAAP91*, Lecture Notes in Computer Science, pages 296–319. Springer-Verlag, 1991.

[7] F. S. de Boer and C. Palamidessi. A process algebra of concurrent constraint programming. In Apt [1], pages 463–477.

[8] M. Falaschi, G. Levi, and C. Palamidessi. A Synchronization Logic: Axiomatics and Formal Semantics of Generalized Horn Clauses. *Information and Control*, 60:36–69, 1994.

[9] J.-M. Jacquet and L. Monteiro. Communicating clauses: Towards synchronous communication in contextual logic programming. In Apt [1], pages 98–112.

[10] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[11] G. Plotkin. A structured approach to operational semantics. Technical report, DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[12] V. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[13] V. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of 17th POPL*, pages 232–245, 1990.

[14] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constrant programming. In *Proc. of 18th POPL.* ACM, 1991.