From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics

F. Bueno*, M. García de la Banda*, M. Hermenegildo*
U. Montanari**, F. Rossi**

*Universidad Politécnica de Madrid (UPM), Facultad de Informática
28660 Boadilla del Monte, Madrid, Spain
E-mail: {bueno,maria,herme}@fi.upm.es
**Università di Pisa, Dipartimento di Informatica
Corso Italia 40, 56125 Pisa, Italy
E-mail: {ugo,rossi}@di.unipi.it.

Abstract. We present a concurrent semantics (i.e. a semantics where concurrency is explicitely represented) for CC programs with atomic tells. This allows to derive concurrency, dependency, and nondeterminism information for such languages. The ability to treat failure information puts CLP programs also in the range of applicability of our semantics: although such programs are not concurrent, the concurrency information derived in the semantics may be interpreted as possible parallelism, thus allowing to safely parallelize those goals (or subparts of goal executions) which appear to be concurrent in the net. Dually, the dependency information may also be interpreted as sequentialization, thus possibly exploiting it to schedule CC programs. The fact that the semantical structure contains dependency information suggests a new tell operation, which checks for consistency only the constraints it depends on, achieving a reasonable trade-off between efficiency and atomicity.

1 Introduction

A concurrent constraint (CC) program [Sar93, SR90, SRP91] consists of a set of agents interacting through a shared store, which is a set of constraints on some variables. The framework is parametric w.r.t. the kind of constraints that can be handled. The concurrent agents do not communicate with each other, but only with the shared store, by either checking if it entails a given constraint (ask operation) or adding a new constraint to it (tell operation). Therefore computations proceed by monotonically accumulating information (that is, constraints) into the shared store.

Usually the semantics of concurrent constraint programs is given operationally, following the SOS-style operational semantics [SR90, SRP91, BP91], and thus suffering from the typical pathologies of an interleaving semantics. On the other hand, the concurrent semantics approach introduced in [MR91], which is equipped with a non-monolithic model of the shared store and of its communication with the agents, allows to express uniformly the behavior of the store and that of the agents, and, as a consequence, to derive a semantical structure where it is possible and easy to see the maximal level of both concurrency and nondeterminism in a given program. Thus it can be much more useful than an interleaving semantics when exploiting semantic information for compile-time optimizations which require knowledge about any one of these two concepts. In fact, an interleaving semantics is not able to express such knowledge correctly, mainly due to the fact that concurrency is not directly expressible but is instead reduced to nondeterminism.

The concurrent semantics in [MR91], from which this paper starts from, is based on an operational semantics described via context-dependent rewrite rules. The evolution of each of the agents in a CC program, as well as the declarations of the program and its underlying constraint system, can all be expressed by sets of such rules. The concurrent semantical structure is then built from the rules by starting from the initial agent and unfolding it applying the rules in all possible ways. The result is a contextual net [MR93a], which is able to represent all the computations of a given CC program (as defined by its operational semantics) in a single structure, and for each of such computations to provide a partial order expressing the dependency pattern among the events of the computation.

There are two ways in which the basic tell operation of CC languages is usually interpreted: either *eventually*, which means that the constraint is added to the current store without any check, or *atomically*, which instead means that the constraint is added only if it is consistent with the current store. The concurrent semantics for CC program which we have just described (and which is described in [MR93b]) follows the eventual interpretation.

While the eventual interpretation of the tell operation allows for a completely uniform treatment of agents and constraints and thus a distributed representation of the constraint system, it suffers from the fact that possibly many computation steps of a failing computation are performed while not being needed. Therefore, the semantical structure presented in [MR93b] contained all such useless (and, most crucial, possibly infinite) parts of computations. Here we modify such semantics to allow for the atomic interpretation of the tell operation. This implies that now we must have the possibility of knowing immediately if a set of constraints is consistent or not. Thus it may seem that we have to go back to the usual notion of a constraint system as a black box which can answer yes/no questions in one step (which is what is used in all the semantics other than [MR91, MR93b]). However, this is not really true. In fact, the semantical structure still shows all the atomic entailment steps of the underlying constraint system, thus allowing to derive the correct dependencies among agents.

The new semantics can be obtained from the old one by defining an inconsistency relation on agents and constraints, and, based on that, by cutting all those parts of the semantical structure which depend on inconsistent "told" constraints. The basic idea is to derive the inconsistency relation from the constraint system, where we assume that an inconsistent set of constraints always entails the token *false*. Then, the inconsistency relation is propagated through the contextual net via the dependency relation. If, as a result of that, some items appear to be inconsistent with themselves, then it means that they could not appear in any computation without creating an inconsistent state of affairs. Therefore we prune such items and everything that depends on them. We also define the new semantics from scratch (instead of first deriving the semantical structure for eventual tells and then pruning it), by adopting a slightly more complicated inference rule.

Since our semantics introduces an explicit representation for failure (i.e. the attempt to add a constraint which is inconsistent with the current store), we can say that we achieve a faithful model for capturing backtracking. In fact, the ability of recognizing independence and/or nondeterminism in CLP programs is crucial when one is interested in parallelizing such programs while retaining their semantic meaning (in terms of input-output relation and time complexity). This is true also for the *dual* task, that of scheduling CC programs [KS92, KT91] (although for such task the treatment of failure is not necessary).

Both such tasks need some knowledge on dependencies (or independence) of goals, since in the first one we want to parallelize only goals which are not dependent on each other, and in the second one we want to schedule later goals which may be dependent on earlier scheduled goals. The attractive point of the proposed semantics is that the dependency relation is an integral part of the semantics and thus parallelization and scheduling decisions can be made by rather direct observations on the semantical structure. Furthermore, the level of granularity offered by the semantics allows scheduling or parallelizing tasks of a new nature and at a new level of detail. For example, it is possible to parallelize across the operations of the constraint solver and thus to create parallel tasks that include part of the solver operations all in the same semantic framework.

While the atomic interpretation of the tell operation allows to recognize, and thus stop, a failing computation possibly much earlier, it has the disadvantage that it can be extremely costly to achieve, especially in a distributed implementation of a CC language. For example, the store could be scattered over many locations, and thus checking its consistency with the new constraint to be told could require locking all the locations and thus all the other operations until the consistency check has been performed. For this reason, it would be reasonable to achieve a convenient trade-off between efficiency and atomicity, thus defining a new interpretation of the tell operation, which just checks some of the constraints in the current store, and not all of them. Our semantics gives a very natural hint on the definition and also the possible implementation of one such interpretation of the tell operation. In fact, being based on dependency information, it is natural to think of checking for consistency only the part of the current store on which the tell operation is dependent on. The interesting, and convenient, thing is that these are the constraints which are in some sense responsible for the presence of the tell agent, and therefore, in a distributed implementation, could be stored in a memory which is local to that agent. This means that they will be the most easily accessible and that thus the tell operation can be performed efficiently. For this locality reason we call this new operation a *locally atomic* tell.

2 Concurrent Constraint Programming

In the CC paradigm, the underlying constraint system can be described [SRP91] as a *partial information system* (derived from the *information system* introduced in [Sco82]) of the form $\langle D, \vdash \rangle$ where D is a set of *tokens* (or primitive constraints) and $\vdash \subseteq \wp(D) \times D$ is the entailment relation which states which tokens are entailed by which sets of other tokens. The relation $\vdash$ has to be reflexive and transitive. Note that there is no notion of consistency in a partial information system. This means that inconsistency has to be modelled through entailment. More precisely, the convention is that D contains a $false$ element, so that an inconsistent set of tokens is that one which entails $false$.

Given D, $\mid D \mid$ is the set of all subsets of D closed under entailment. Then, a constraint in a constraint system $\langle D, \vdash \rangle$ is simply an element of $\mid D \mid$, that is, a set of tokens, closed under entailment. In the rest of the paper we will consider a constraint as simply a set of tokens.

Consider the class of programs P, the class of sequences of procedure declarations F, and the class of agents A. Let c range over constraints, and $\mathbf{x}$ denote a tuple of variables. The following grammar describes the CC language we consider:

$$P ::= F.A \qquad\qquad F ::= p(\mathbf{x}) :: A \mid F.F$$

$$A ::= success \mid failure \mid tell(c) \rightarrow A \mid \sum_{i=1,\ldots,n} ask(c_i) \rightarrow A_i \mid A \parallel A \mid \exists \mathbf{x}.A \mid p(\mathbf{x})$$

Each procedure is defined once, thus nondeterminism is expressed via the $+$ combinator only (which is here denoted by $\sum$). We also assume that, in $p(\mathbf{x}) :: A$, $vars(A) \subseteq \mathbf{x}$, where $vars(A)$ is the set of all variables occurring free in agent A. In a program $P = F.A$, A is called initial agent, to be executed in the context of the set of declarations F.

Agent "$\sum_{i=1,\ldots,n} ask(c_i) \rightarrow A_i$" behaves as a set of guarded agents A_i, where the success of the guard $ask(c_i)$ coincides with the entailment of the constraint c_i by the current store. If instead c_i is inconsistent with the current store, then the guard fails. Lastly, if c_i is not entailed but it is consistent with the current store, then the guarded agent gets suspended. No particular order of selection of the guarded agents is assumed, and only one of the choices is taken. In an "atomic" interpretation of the tell operation, agent "tell(c) $\rightarrow$ A" adds constraint c to the current store and then, if the resulting store is consistent, behaves like A, otherwise it fails; in an "eventual" interpretation of the tell, this same agent adds c to the store (without any consistency check) and then behaves like A (if the resulting store is inconsistent this will result in an uncontrolled behaviour of the system, since from now on all ask operations will succeed).

Given a program P, in the following we will refer to $Ag(P)$ as the set of all agents (and subagents) occurring in P, i.e. all the elements of type A occurring in a derivation of P according to the above grammar.

The CC language we consider in this paper does not use the notion of *cylindric* constraint system, as defined for example in [SRP91]. Therefore constraints cannot be projected over some of their variables. However, we strongly believe that our whole framework and results could be extended also to this more general case. Another extension could be the presence of tell agents in the guards of an indeterministic agent: this would certainly not cause any problem to our approach. We made a less general choice here just for reasons of space.

3 The Operational Semantics

Each state of a CC computation consists of a multiset of (active) agents and of (already generated) tokens. Each computation step models either the evolution of a single agent, or the entailment of a new token through the $\vdash$ relation. Such a change in the state of the computation is performed via the application of a rewrite rule. There are as many rewrite rules as the number of agents and declarations in a program (which is finite), plus the number of pairs of the entailment relation (which can be infinite).

Definition 1 ([computation state]) *Given a program $P = F.A$ with a constraint system $\langle D, \vdash \rangle$, a state is a multiset of elements of $Ag(P) \cup D$.* $\Box$

Definition 2 ([rewrite rules]) *Have the form $r : L(r)(\mathbf{x}) \overset{c(r)(\mathbf{x})}{\rightsquigarrow} R(r)(\mathbf{xy})$ where $L(r)$ is an agent, $c(r)$ is a constraint, and $R(r)$ is a state. Also, $\mathbf{x}$ is the tuple of variables appearing in both $L(r) \cup c(r)$ and in $R(r)$, while $\mathbf{y}$ is the tuple of variables appearing only in $R(r)$ and/or existentially quantified in $L(r)$.* $\Box$

The intuitive meaning of a rule is that $L(r)$,[1] which is called the left hand side of the rule, is rewritten into (or replaced by) $R(r)$, i.e. the right hand side, if $c(r)$ is present in the current state. $R(r)$ could contain some variables not appearing in $L(r)$ nor in $c(r)$ (i.e. the tuple $\mathbf{y}$). The application of r would then rename such variables to constants which are different from all the others already in use. The items in $c(r)$ have to be interpreted as a context, since it is necessary for the application of the rule but it is not affected by such application. In the CC framework, such context is used to represent asked constraints.

Definition 3 ([from programs to rules]) *The rules corresponding to agents, declarations, and entailment pairs are given as follows:*

1. $(tell(c) \rightarrow A) \rightsquigarrow c, A$	4. $(\sum_{i=1,\ldots,n} ask(c_i) \rightarrow A_i) \overset{c_i}{\rightsquigarrow} A_i \forall i = 1, \ldots, n$
2. $A_1 \parallel A_2 \rightsquigarrow A_1, A_2$	5. $p(\mathbf{x}) \rightsquigarrow A$ *for all* $p(\mathbf{x}) :: A$
3. $\exists \mathbf{x}.A \rightsquigarrow A$	6. $\overset{S}{\rightsquigarrow} t$ *for all* $S \vdash t$

Given a CC program $P = F.A$ and its underlying constraint system $\langle D, \vdash \rangle$, we will call $RR(P)$ the set of rewrite rules associated to P, which consists of the rules corresponding to all agents in $Ag(P)$, plus the rules representing the declarations in F, plus those rules representing the pairs of the entailment relation. $\Box$

[1] Note that in a slight abuse of notation we consider $L(r)$ as a set, which will be either a singleton or the empty set.

In an eventual CC language, a rule r can be applied to a state S_1 if both the left hand side of r and its context can be found (via a suitable substitution) in S_1. The application of r removes its left and adds its right hand side to S_1.

Definition 4 ([eventual computation steps]) *Let a computation state* $S_1(\mathbf{a})$ *and a rule* $r : L(r)(\mathbf{x}) \overset{c(r)(\mathbf{x})}{\leadsto} R(r)(\mathbf{xy})$, *such that* $(L(r) \cup c(r))[\mathbf{a}/\mathbf{x}] \subseteq S_1(\mathbf{a})$. *The application of* r *to* S_1 *is an eventual computation step which yields a new computation state* $S_2 = (S_1 \setminus L(r)[\mathbf{a}/\mathbf{x}]) \cup R(r)[\mathbf{a}/\mathbf{x}][\mathbf{b}/\mathbf{y}]$, *where the constants in* $\mathbf{b}$ *are fresh, i.e. they do not appear in* S_1. *We will write* $S_1 \overset{r[\mathbf{a}/\mathbf{x}][\mathbf{b}/\mathbf{y}]}{\Longrightarrow} S_2$. $\square$

Instead, in an atomic CC language, not only the left hand side and the context of a rule have to match some elements in the current state, but also, if the rule implements a tell agent, a check has to be done for the constraints that such tell wants to add to be consistent with the current store.

Definition 5 ([atomic computation steps]) *Let a computation state* $S_1(\mathbf{a})$ *and a rule* $r : L(r)(\mathbf{x}) \overset{c(r)(\mathbf{x})}{\leadsto} R(r)(\mathbf{xy})$, *such that*

- $(L(r) \cup c(r))[\mathbf{a}/\mathbf{x}] \subseteq S_1(\mathbf{a})$ *and*
- *if* $r = ((tell(c) \to A) \leadsto c, A)$, *then* $c \cup cons(S_1) \not\vdash false$ *(where* $cons(S)$ *is the set of constraints in state* S*)*

The application of r *to* S_1 *is an atomic computation step which yields a new computation state* $S_2 = (S_1 \setminus L(r)[\mathbf{a}/\mathbf{x}]) \cup R(r)[\mathbf{a}/\mathbf{x}][\mathbf{b}/\mathbf{y}]$, *where the constants in* $\mathbf{b}$ *are fresh, i.e. they do not appear in* S_1. *We will write* $S_1 \overset{r[\mathbf{a}/\mathbf{x}][\mathbf{b}/\mathbf{y}]}{\Longrightarrow} S_2$. $\square$

Definition 6 ([computations]) *Given a CC program* $P = F.A$, *an eventual (resp. atomic) computation segment for* P *is any sequence of eventual (resp. atomic) computation steps* $S_1 \overset{r_1[\mathbf{a_1}/\mathbf{x_1}]}{\Longrightarrow} S_2 \overset{r_2[\mathbf{a_2}/\mathbf{x_2}]}{\Longrightarrow} S_3 \ldots$ *such that* $S_1 = \{A\}$ *and* $r_i \in RR(P)$, $i = 1,2, \ldots$. *Two eventual (resp. atomic) computation segments which are the same except that different fresh constants are employed in the various steps, are called* α*-equivalent. An eventual (resp. atomic) computation is an eventual (resp. atomic) computation segment, say* CS, *such that for each eventual (resp. atomic) computation segment, say* CS', *of which* CS *is a prefix,* CS' *adds to* CS *only steps applying rules for the entailment relation.* $\square$

Definition 7 ([successful, suspended, and failing computations]) *A successful computation is a finite computation where the last state contains only a set of constraints, say* S, *and* $S \not\vdash false$. *A suspended computation is a finite computation where the last state does not contain tell agents but contains ask agents, and its set of constraints* S *is such that* $S \not\vdash false$. *A failing computation is a computation which is neither successful nor suspended.* $\square$

Notice that a computation has been defined as a sequence of computation steps which is maximal w.r.t. the evolution of the agents. This means that there could be some subsequent step due to the entailment relation, but no step due to the agents. The reason for this is that, after all the agents have evolved, there could be an infinite number of entailment steps, and still we do not want to consider such a computation failing just because of that. A consequence of this is that to recognize a successful computation we have to ask the constraint system for a consistency test even in an eventual environment. Thus, the difference between atomic and eventual tell is just *when* such a check is asked for.

In the following we will only consider either finite computations or infinite computations which are fair. Here fairness means, informally, that if a rule can continuously be applied from some point onwards, then it will eventually be applied. This implies that both goal selection (among several goals in the current state) and rule selection (among several rules applicable to a goal) are fair.

Definition 8 ([eventual and atomic operational semantics]) *Given a CC program $P = F.A$, its eventual operational semantics, say $EO(P)$, is the set of all its eventual computations, and its atomic operational semantics, say $AO(P)$, is the set of all its atomic computations.* □

4 Contextual Nets and Consistent Contextual Nets

In the following, we assume the reader to be familiar with the classical notions of nets. For the formal definitions missing here we refer to [Rei85] and [MR93a].

4.1 Contextual Nets

The formal technique which we use to introduce contexts consists in adding a new relation, besides the usual flow relation, which we call the *context relation.*

Definition 9 ([contextual nets]) *Are quadruples $(B, E; F_1, F_2)$ where elements of B are called conditions and those of E events; $F_1 \subseteq (B \times E) \cup (E \times B)$ and it is called the flow relation; $F_2 \subseteq (B \times E)$ and it is called the context relation; and it holds that $B \cap E = \emptyset$ and $(F_1 \cup F_1^{-1}) \cap F_2 = \emptyset$.* □

Definition 10 ([pre-set, post-set, and context]) *Given a contextual net $N = (B, E;\ F_1, F_2)$ and an element $x \in B \cup E$, the pre-set of x is the set ${}^\bullet x = \{y \mid yF_1x)\}$; the post-set of x is the set $x^\bullet = \{y \mid xF_1y)\}$; the context of x is defined if $x \in E$ and it is the set $\widehat{x} = \{y \mid yF_2x)\}$.* □

In our concurrent semantics the underlying notion is that of a contextual process, which is a contextual occurrence net together with a suitable mapping of the elements of the net to the syntactic objects of the program execution. Through the mapping, each condition of the contextual net represents an agent or a constraint, and each event represents a rule application. Informally, a contextual occurrence net is just an acyclic contextual net, where acyclicity referes to the dependency relation induced by F_1 and F_2.

Contextual nets will be graphically represented in the same way as nets. Thus, conditions are circles, events are boxes, the flow relation is represented by directed arcs from circles to boxes or viceversa, and the context relation by undirected arcs. An example of a contextual net can be seen in Figure 1. In this figure we see four events, of which two of them share a context.

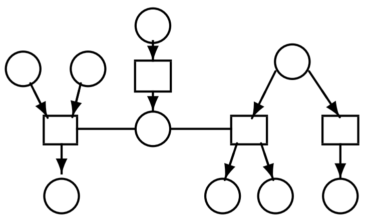
Fig. 1. A contextual net.

Definition 11 ([dependency]) *Consider a contextual net $N = (B, E; F_1, F_2)$. Then we define a corresponding structure $(B \cup E, \leq_N)$, where the dependency relation $\leq_N$ is the minimal relation which is reflexive, transitive, and which satisfies the following conditions: xF_1y implies $x \leq_N y$; e_1F_1b and bF_2e_2 implies $e_1 \leq_N e_2$; bF_2e_1 and bF_1e_2 implies $e_1 \leq_N e_2$.* $\square$

Therefore in the following we will say that x depends on y whenever $y \leq_N x$. However, a contextual net gives information not only about dependency of events and conditions, but also about concurrency and mutual exclusion (or conflict).

Definition 12 ([mutual exclusion and concurrency]) *Consider a contextual net $N = (B, E; F_1, F_2)$ and the associated dependency relation $\leq_N$. Assume that $\leq_N$ is antisymmetric, and let $\leq\geq \in (B \cup E) \times (B \cup E)$ be defined as $\leq\geq = \{\langle x, y\rangle \mid x \leq_N y \text{ or } y \leq_N x\}$.*

- *The mutual exclusion relation $\#_N \subseteq ((B \cup E) \times (B \cup E))$ is defined as follows. First we define $x\#'y$ iff $x, y \in E$ and $\exists z \in B$ such that zF_1x and zF_1y. Then, $\#_N$ is the minimal relation which includes $\#'$ and which is symmetric and hereditary (i.e. if $x\#_Ny$ and $x \leq z$, then $z\#_Ny$).*
- *The concurrency relation co_N is just $((B \cup E) \times (B \cup E)) \setminus (\leq\geq \cup \#_N)$.* $\square$

In words, the mutual exclusion is originated by the existence of conditions which cause more than one event, and then it is propagated downwards through the dependency relation. Instead, two items are concurrent if they are not dependent on each other nor mutually exclusive.

Definition 13 ([contextual occurrence net]) *A contextual occurrence net is a contextual net N, where $N = (B, E; F_1, F_2)$ and: $\leq_N$ is antisymmetric; $b \in B$ implies $\mid {}^\bullet b \mid \leq 1$; $\#_N$ is irreflexive.* $\square$

A useful special case of a contextual occurrence net occurs when the mutual exclusion relation is empty. This means that, taken any two items in the net, they are either concurrent or dependent. Since no conflict is expressed in such nets, they represent a completely deterministic behaviour.

Definition 14 ([deterministic contextual occurrence net]) *A deterministic contextual occurrence net is a quadruple $N = (B, E; F_1, F_2)$ such that N is a contextual occurrence net with $\#_N = \emptyset$.* $\square$

Given a (nondeterministic) contextual occurrence net, it is easy to derive the set of all its subnets which are deterministic. For this we use restrictions defined as $F_{|S} = \{x \in F | x \in S\}$ (set intersection).

Definition 15 ([from contextual to deterministic contextual occ. nets]) *Given a contextual occurrence net $N = (B, E; F_1, F_2)$ and the associated relations $\leq_N$, $\#_N$, and co_N, a deterministic contextual occurrence net of N is a deterministic contextual occurrence net $N' = (B', E'; F_1', F_2')$ where*

- $B' \subseteq B$ *and* $E' \subseteq E$;
- $x \in (B' \cup E')$ *and* $y \in (B \cup E)$ *such that* $y \leq_N x$ *implies that* $y \in (B' \cup E')$;
- $F_1' = F_{1|(B' \times E') \cup (E' \times B')}$ *and* $F_2' = F_{2|(B' \times E')}$. $\square$

We are now ready to define the contextual processes, which, as anticipated above, will be used to give a concurrent semantics to CC programs. We recall that, informally, a contextual process is just a contextual occurrence net plus a suitable mapping from the items of the net (i.e. conditions and events) to the agents of the CC program and the rules representing it.

Definition 16 ([contextual process]) *Given a CC program P with initial agent A, and the associated sets of rewrite rules $RR(P)$, agents $Ag(P)$, and tokens D, consider the sets $RB = \{b\theta\}$ and $RE = \{r\theta\}$, with $b \in (Ag(P) \cup D)$, $r \in RR(P)$ and θ any substitution. Then a contextual process is a pair $\langle N, \pi \rangle$, where*

- $N = (B, E; F_1, F_2)$ *is a (nondeterministic) contextual occurrence net;*
- $\pi : (B \cup E) \to (RB \cup RE)$ *is a mapping where*
  - $\forall b \in B,\ \pi(b) \in RB$ *and* $\forall e \in E,\ \pi(e) \in RE$;
  - $\forall x \in B$ *such that* $\nexists y \in (B \cup E),\ y \leq_N x,\ \pi(x) = A$;
  - *let* $\pi(e) = r\theta$, *with* $r = L \overset{c}{\rightsquigarrow} R$, *then* $\{\pi(x) | x \in {}^\bullet e\} = L\theta$, $\{\pi(x) | x \in \widehat{e}\} = c\theta$, $\{\pi(x) | x \in e^\bullet\} = R\theta$. $\square$

4.2 Consistent Contextual Nets

A consistent contextual net is just a contextual net with an additional relation, called in the following the mutual inconsistency relation, which defines, together with the mutual exclusion relation, which items of the net cannot be present in the same computation. In the same way as mutual exclusion, dependency, and concurreny are defined in contextual nets starting from the basic relations F_1 and F_2, the mutual inconsistency relation is defined starting from them and a new basic relation F_3. The addition of such relation has however some heavy consequences, among which the fact that the concurrency relation is not binary any more.

Definition 17 ([consistent contextual nets]) *A consistent contextual net is a quintuple $(B, E; F_1, F_2, F_3)$ where $N = (B, E; F_1, F_2)$ is a contextual net, and $F_3 \subseteq \wp(E)$ such that $F_3(S)$ implies $\forall e_1, e_2 \in S$, $e_1\ co_N\ e_2$ and $\neg F_3(S')$ for all $S' \subset S$.* $\square$

Pre-set, post-set, and context are defined as for contextual nets. The same holds also for the dependency ($\leq$ from now on) and the mutual exclusion ($\#$) relation. However, now we have to define the new *mutual inconsistency* relation ($@$), starting from F_3, and we have to redefine the concurrency relation (co).

Definition 18 ([mutual inconsistency and concurrency]) *Consider a consistent contextual net, given as* $(B, E; F_1, F_2, F_3)$*, and its dependency and mutual exclusion relations* $\leq$ *and* $\#$*.*

- *The mutual inconsistency relation* $@ \subseteq \wp(B \cup E)$ *is defined as follows:*
  - $F_3(S)$ *implies* $@(S)$*, and*
  - $@(S \cup \{t\})$ *and* $t \leq t'$ *implies* $@(S \cup \{t'\})$*.*
- *The concurrency relation* $co \in \wp(B \cup E)$ *is defined as follows:* $co(S)$ *if there is no subset* $S' \subset S$ *s.t.* $@(S')$ *and no* $s_1, s_2 \in S$ *s.t.* $s_1 \# s_2$ *or* $s_1 \leq s_2$*.* $\square$

In words, the mutual inconsistency relation includes the F_3 relation and it is hereditary. Instead, the concurrency relation is as usually defined by taking what is forbidden by the other relations. However, while usually such relation is binary, now it becomes n-ary, due to the fact that the new mutually inconsistency relation may be n-ary in general.

Since the mutual inconsistency relation is hereditary, there could be items inconsistent with themselves (which will be called *self-inconsistent* in the following). This informally means that they cannot appear in any computation, since they are inconsistent with their parents. We call a net *admissible* if it does not contain any of such items, and from now on we will only consider admissible consistent contextual nets.

Definition 19 ([admissible consistent nets]) *A consistent contextual net* $N = (B, E; F_1, F_2, F_3)$ *is admissible whenever* $\nexists e \in E$ *such that* $@(\{e\})$*.* $\square$

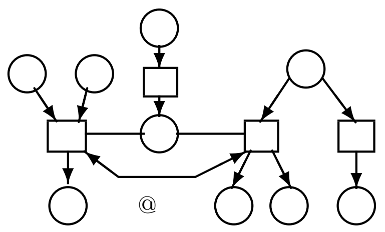

Fig. 2. A consistent contextual net.

An admissible consistent contextual net can be seen in Figure 2. Notice that we choose to represent the mutual exclusion relation by (hyper)arcs which have arrows on all their endings.

As in the previous section, we now define deterministic and occurrence nets for the class of consistent contextual nets. The only difference is that now we define a net to be deterministic whenever both the mutual exclusion and the mutual inconsistency relations are empty.

Definition 20 ([(deterministic) consistent contextual occurrence nets]) *A consistent contextual occurrence net is a consistent contextual net $(B, E; F_1, F_2, F_3)$ such that $(B, E; F_1, F_2)$ is a contextual occurrence net. A consistent contextual occurrence net $(B, E; F_1, F_2, F_3)$ is deterministic when $F_3 = \# = \emptyset$.* $\Box$

Notice that a deterministic consistent contextual occurrence net is just a (deterministic) contextual occurrence net, since $F_3 = \emptyset$. Therefore the way to obtain the deterministic consistent contextual occurrence nets of a given consistent contextual net is the same as in Definition 15.

If instead we just require the absence of mutually exclusive elements, just as in classical and contextual nets, then we still get subnets which have a meaning. In fact, we will see that they will be used to model the *locally atomical* interpretation for the tell operation, in which a computation step just checks the consistency of the constraint told within a local store.

Definition 21 ([(deterministic) locally consistent contextual occ. nets]) *A deterministic locally consistent contextual occurrence net $(B, E; F_1, F_2, F_3)$ is a consistent contextual occurrence net with $\# = \emptyset$.* $\Box$

Finally, we will relate consistent occurrence nets to CC programs by means of consistent contextual processes, whose definition is straightforward.

Definition 22 ([consistent contextual process]) *A consistent contextual process is a pair $\langle N, \pi\rangle$ such that $N = (B, E; F_1, F_2, F_3)$ is a consistent contextual occurrence net, and $\langle (B, E; F_1, F_2), \pi\rangle$ is a contextual process.*$\Box$

5 Concurrent Semantics for CC with Eventual Tell

The key idea in the semantics is to take the set of rewrite rules $RR(P)$ associated to a given CC program P and to incrementally construct a corresponding contextual process. A longer description of this semantics is contained in [MR93b].

Definition 23 ([from rewrite rules to a contextual process]) *Given a CC program P, the pair $CP(P) = \langle (B, E; F_1, F_2), \pi\rangle$ is constructed by means of the following two inference rules:*

- if $A(\mathbf{a})$ *initial agent of* P then $\langle A(\mathbf{a}), \emptyset, 1\rangle \in B$*;*
- if $\exists r \in RR(P)$ *such that* $L(r) \cup c(r) = \{B_1(\mathbf{x}_1), \ldots, B_n(\mathbf{x}_n)\}$*, and*
  - $\exists \{s_1, \ldots, s_n\} \subseteq B$ *such that* $\forall i, j = 1, \ldots, n,\ s_i\ co_N\ s_j$*, and*
  - $\forall i = 1, \ldots, n,\ s_i = \langle e_i, B_i(\mathbf{a_i}), k_i\rangle$*, and for some* $\mathbf{a}$*,* $B_i(\mathbf{x}_i)[\mathbf{a}/\mathbf{x}] = B_i(\mathbf{a}_i)$

  then
  - $e = \langle r[\mathbf{a}/\mathbf{x}], \{s_1, \ldots, s_n\}, 1\rangle \in E$*,*
  - $s_i F_1 e$ *for all* $s_i = \langle e_i, B_i(\mathbf{a_i}), k_i\rangle$ *such that for some* $\mathbf{a}$*,* $B_i(\mathbf{x}_i)[\mathbf{a}/\mathbf{x}] = B_i(\mathbf{a}_i)$ *and* $B_i(\mathbf{x}_i) \in L(r)$
  - $s_i F_2 e$ *for all* $s_i = \langle e_i, B_i(\mathbf{a_i}), k_i\rangle$ *such that for some* $\mathbf{a}$*,* $B_i(\mathbf{x}_i)[\mathbf{a}/\mathbf{x}] = B_i(\mathbf{a}_i)$ *and* $B_i(\mathbf{x}_i) \in c(r)$

- *let h be the multiplicity of $B(\mathbf{x}, y_1, \ldots, y_m) \in R(r)$, then for all $l = 1, \ldots, h$,*
  $b_l = \langle B[\mathbf{a}/\mathbf{x}][\langle e, y_1\rangle/y_1] \ldots [\langle e, y_m\rangle/y_m], e, l\rangle \in B$, *and* eF_1b_l.

Moreover, for any item $x = \langle x_1, x_2, x_3\rangle \in (B \cup E)$, $\pi(x) = x_1$. $\Box$

The elements of the net in the contextual process are built in such a way that elements generated by using different sequences of rules are indeed different. In fact, each element is a term consisting of a triple, of which the first element is the *type* of the term, and represents the rule or agent or constraint the term corresponds to, the second element is its *history*, and this is what makes different terms which are generated in different ways, and the third element is its *multiplicity*, and takes care of different copies of the same element in the same computation state.

Each time the inference rule is applied, a rule in RR(P) is chosen whose left hand side and context are *matched* by some elements already present in the partially built process. Such elements have to be concurrent, otherwise it would mean that they cannot be together in a state. Then, a new element representing the rule application is added (as an event), as well as new elements representing the right hand side of the rule (as conditions).

Theorem 24 [$CP(P)$ **is a contextual process**]**.** *Given a CC program P, the structure $CP(P)$ of Definition 23 is a contextual process.* $\Box$

Theorem 25 [**soundness and completeness of** $CP(P)$ **w.r.t.** $EO(P)$]**.** *Let P be a CC program and let $CP(P) = \langle N, \pi\rangle$ be the corresponding contextual process.*

- *For a given computation in $EO(P)$ there are (1) an α-equivalent computation $S_1 \overset{r_1[\mathbf{a}_1/\mathbf{x}_1]}{\Longrightarrow} S_2 \overset{r_2[\mathbf{a}_2/\mathbf{x}_2]}{\Longrightarrow} S_3 \ldots$, and (2) one linearization (restricted to events), say $e_1e_2, \ldots$, of the partial order associated to a maximal deterministic contextual occurrence net of N, such that $\pi(e_i) = r_i[\mathbf{a}_i/\mathbf{x}_i]$ for all $i = 1, 2, \ldots$*
- *For any linearization $e_1e_2 \ldots$ of the partial order associated to a deterministic contextual occurrence net of N, there is a computation in $EO(P)$, say $S_1 \overset{r_1[\mathbf{a}_1/\mathbf{x}_1]}{\Longrightarrow} S_2 \overset{r_2[\mathbf{a}_2/\mathbf{x}_2]}{\Longrightarrow} S_3 \ldots$, such that, if $e_i = \langle e_{i1}, e_{i2}, e_{i3}\rangle$ and $\pi(e_i) = r$, then $r_i[\mathbf{a}_i/\mathbf{x}_i] = r$ for all $i = 1, \ldots$* $\Box$

As just shown by the above theorem, the concurrent semantics defined in this section considers the eventual interpretation of the tell operation: constraints are added to the store without checking their consistency with the current set of constraints already in it. Therefore there may be parts of the net which represent computation sequences which would not happen if taking the atomic interpretation of the tell operation. In the following section we show how to recognize and then delete such parts, obtaining a (possibly much) smaller process. We will also give a new inference rule which allows to not even generate those parts.

6 Concurrent Semantics for CC with Atomic Tell

In order to correctly treat atomic tell, we need to know when a set of constraints is inconsistent. This can be done by just looking at the constraint system, since we assumed that a set of inconsistent constraints entails the token *false*.

Definition 26 ([inconsistent constraints]) *Given a constraint system $\langle D, \vdash \rangle$, we say that $u \in \wp(D)$ is inconsistent, and we write $inc(u)$, whenever $u \vdash false$. Moreover, we write $inc_0(u)$ whenever $inc(u)$ holds and also $\not\exists v \in \wp(D)$ such that $v \subset u$ and $v \vdash false$.* $\Box$

From the inconsistency of a set of tokens we can then derive the mutual inconsistency of a set of conditions and/or events in the contextual process. Mutual inconsistency means impossibility of appearing in the same computation without creating an inconsistent store.

Definition 27 ([mutual inconsistency]) *Given a CC program P, a constraint system $\langle D, \vdash \rangle$, and the contextual process $CP(P) = \langle (B, E; F_1, F_2), \pi \rangle$, we define a mutual inconsistency relation $@ \subseteq \wp(B \cup E)$ (and $@'$) as follows:*

- *if $\{b_1, \ldots, b_n\} \in B$ and $\forall i = 1, \ldots, n, \pi(b_i) \in D$ and $inc_0(\{\pi(b_1), \ldots, \pi(b_n)\})$ and $\not\exists i, j = 1, \ldots, n$ such that $b_i \# b_j$, then $@'(\{b_1, \ldots, b_n\})$;*
- *if $@'(\{b_1, \ldots, b_n\})$ and $\forall i = 1, \ldots, n$, $\exists e_i \in E$ s.t. $e_i F_1 b_i$, then $@'(\{e_1, \ldots, e_n\})$;*
- *$@$ is the minimal relation which includes $@'$ and which is hereditary (i.e. if $@(S \cup \{s\})$ and $s \leq s'$, then $@(S \cup \{s'\})$).* $\Box$

In particular, the elements of the process which are self-inconsistent cannot appear in any computation. Therefore, one step which allows us to change the semantical structure which represents the eventual operational semantics of a CC program and get closer to that which represents the atomic operational semantics of the same program consists in deleting everything that depends on them. In fact, such steps are exactly those tell operations which could be done only because it was not performed any consistency check.

Definition 28 ([net pruning]) *Given a CC program P, a constraint system $\langle D, \vdash \rangle$, the contextual process $CP(P) = \langle (B, E; F_1, F_2), \pi \rangle$, and the relation $@$ of Definition 27, the new process is $CP'(P) = \langle (B', E'; F_1', F_2'), \pi' \rangle$, where*

- *$B' = B \setminus \{b \mid \exists e \in E$ s.t. $@(\{e\})$ and $e \leq b\}$,*
- *$E' = E \setminus \{e \mid \exists e' \in E$ s.t. $@(\{e'\})$ and $e' \leq e\}$,*
- *$F_1' = F_{1|B' \times E' \cup E' \times B'}$ and $F_2' = F_{2|B' \times E'}$, and*
- *π' is the restriction of π to $B' \cup E'$.* $\Box$

Theorem 29 [**$CP'(P)$ is a consistent contextual process**]. *Consider the process $CP'(P) = \langle (B', E'; F_1', F_2'), \pi' \rangle$ of Definition 28 and the relation $@$ of Definition 27. Then $\langle (B', E'; F_1', F_2', @'_{|\wp(E')}), \pi' \rangle$ is a consistent contextual process.* $\Box$

Theorem 30 [**soundness and completeness of $CP'(P)$ w.r.t. $AO(P)$**]. *Let P be a CC program and $CP'(P) = \langle N, \pi \rangle$ its consistent contextual process.*

– *For any computation in $AO(P)$, there are (1) an α-equivalent computation $S_1 \overset{r_1[\mathbf{a}_1/\mathbf{x}_1]}{\Longrightarrow} S_2 \overset{r_2[\mathbf{a}_2/\mathbf{x}_2]}{\Longrightarrow} S_3 \ldots$, and (2) one linearization (restricted to events), $e_1, e_2, \ldots$, of the partial order associated to a maximal deterministic consistent contextual occurrence net of N, s.t. $\forall i = 1, 2, \ldots, \pi(e_i) = r_i[\mathbf{a}_i/\mathbf{x}_i]$*
– *For any linearization $e_1 e_2 \ldots$ of the partial order associated to a deterministic consistent contextual occurrence net of N, there is a computation in $AO(P)$, say $S_1 \overset{r_1[\mathbf{a}_1/\mathbf{x}_1]}{\Longrightarrow} S_2 \overset{r_2[\mathbf{a}_2/\mathbf{x}_2]}{\Longrightarrow} S_3 \ldots$, such that, if $e_i = \langle e_{i1}, e_{i2}, e_{i3}\rangle$ and $\pi(e_i) = r$, then $r_i[\mathbf{a}_i/\mathbf{x}_i] = r$ for all $i = 1, \ldots$* □

It is also possible to characterize failing, successful, and suspended computations directly in the concurrent semantics, instead of having to map them back to the corresponding computations in the operational semantics.

Definition 31 ([successful, failing, and suspended nets]) *Given a CC program P and a constraint system $\langle D, \vdash\rangle$, consider the corresponding consistent contextual process $CP(P) = \langle (B, E; F_1, F_2, F_3), \pi\rangle$. Consider also any maximal deterministic consistent contextual net of $(B, E; F_1, F_2, F_3)$, $DN = (B', E'; F_1', F_2', \emptyset)$, and the elements $DN^\circ = \{b \mid b \in B' \text{ and } \nexists b' \in B',\ b \le b'\}$. Then DN is:*

– *successful if the set of events representing agent rules is finite, and $\forall b \in DN^\circ,\ \pi(b) \in (D \setminus \{false\})$;*
– *suspended if the set of events representing agent rules is finite, and $\forall b \in DN^\circ$ such that $\pi(b) \in Ag(P)$, $\pi(b)$ is an ask agent;*
– *failing otherwise.* □

Theorem 32 [characterization of success, failure, and suspension]. *Let P be a CC program and $CP(P) = \langle (B, E; F_1, F_2, F_3), \pi\rangle$ its corresponding consistent contextual process. Consider any maximal deterministic consistent contextual net of $(B, E; F_1, F_2, F_3)$, say $DN = (B', E'; F_1', F_2', \emptyset)$. If DN is successful (resp., suspended, failing) then all the computations in $AO(P)$ corresponding to DN according to Theorem 30 are successful (resp., suspended, failing).* □

Now we will obtain the same consistent contextual process by means of a new inference rule, instead of first producing the contextual process as in Definition 23 and then pruning it. The advantage consists in a possibly much smaller resulting process. However, the drawback is a much more costly condition to check during the generation, each time the inference rule is applied.

Definition 33 ([from rewrite rules to a consistent contextual process]) *Let P be a CC program, $CCP(P) = \langle (B, E; F_1, F_2, F_3), \pi\rangle$ is constructed by means of the following two inference rules:*

– if $A(\mathbf{a})$ *initial agent of P* then $\langle A(\mathbf{a}), \emptyset, 1\rangle \in B$;
– if $\exists r \in RR(P)$ *such that* $L(r) \cup c(r) = \{B_1(\mathbf{x}_1), \ldots, B_n(\mathbf{x}_n)\}$, *and*
  - $\exists \{s_1, \ldots, s_n\} \subseteq B$ *such that* $co(\{s_1, \ldots, s_n\})$, *and*
  - $\forall i = 1, \ldots, n,\ s_i = \langle e_i, B_i(\mathbf{a_i}), k_i\rangle$, *and for some* $\mathbf{a}$, $B_i(\mathbf{x}_i)[\mathbf{a}/\mathbf{x}] = B_i(\mathbf{a}_i)$

- *$\neg inc(ct(\{e\}))$, for $e = \langle r[\mathbf{a}/\mathbf{x}], \{s_1, \ldots, s_n\}, 1\rangle$, where $ct : \wp(B \cup E) \to \wp(D)$ is defined as follows: $\forall \langle t_1, t_2, t_3\rangle \in (B \cup E)$,*

  $$ct(S \cup \{\langle t_1, t_2, t_3\rangle\}) = \begin{cases} ct(S \cup t_2) \cup (R(r)[\mathbf{a}/\mathbf{x}] \cap D) & \text{if } t_1 = r[\mathbf{a}/\mathbf{x}] \text{ and } r \text{ is a rule for a tell agent} \\ ct(S \cup t_2) & \text{otherwise} \end{cases}$$

  $ct(\emptyset) = \emptyset$

then

- *$e \in E$,*
- *$s_i F_1 e$ for all $s_i = \langle e_i, B_i(\mathbf{a_i}), k_i\rangle$ such that for some $\mathbf{a}$, $B_i(\mathbf{x}_i)[\mathbf{a}/\mathbf{x}] = B_i(\mathbf{a}_i)$ and $B_i(\mathbf{x}_i) \in L(r)$*
- *$s_i F_2 e$ for all $s_i = \langle e_i, B_i(\mathbf{a_i}), k_i\rangle$ such that for some $\mathbf{a}$, $B_i(\mathbf{x}_i)[\mathbf{a}/\mathbf{x}] = B_i(\mathbf{a}_i)$ and $B_i(\mathbf{x}_i) \in c(r)$*
- *let h be the multiplicity of $B(\mathbf{x}, y_1, \ldots, y_m) \in R(r)$, for all $l = 1, \ldots, h$, $b_l = \langle B[\mathbf{a}/\mathbf{x}][\langle e, y_1\rangle/y_1] \ldots [\langle e, y_m\rangle/y_m], \{e\}, l\rangle \in B$, and $eF_1 b_l$.*
- *$F_3(S \cup \{e\})$ for all $S = \{e_1, \ldots, e_n\} \subseteq E$ such that $co(S \cup \{s_1, \ldots, s_n\})$, and $inc(ct(\{e\} \cup S))$, and $\not\exists S' \subseteq E$ such that $S' \cup \{s_1, \ldots, s_n\} \in co$, and $inc(ct(\{e\} \cup S'))$, and $\forall e' \in S'\ \exists e \in S$ such that $e' \leq e$.*

Moreover, for any item $x = \langle x_1, x_2, x_3\rangle \in (B \cup E)$, $\pi(x) = x_1$. $\square$

The main difference of the above definition w.r.t. Definition 23 is the condition which has to be checked for applying the second inference rule. It is not enough to check that there are conditions which are concurrent and which match the left hand side and the context of a rule. It is also necessary to check that the constraints which would be added to the process because of the application of the chosen rule are consistent with those which are in the history of the rule itself. In fact, such constraints would be in any store where that rule is applied, no matter which linearization one chooses. Such constraints are retrieved by function ct, which traverses a term and gets all the constraints in its history.

Another difference concerns the creation of relation F_3. Inconsistency of the new event e with a set S of events, already in the process, is derived if e and the constraints generated in the history of S are inconsistent. This is done only if e is concurrent with them (checked by looking at the pre-conditions of e, $s_1, \ldots, s_n$, since e is not formally in the process yet). This would create an F_3 relation which is already hereditary. However, we prefer to have F_3 as the base relation, and then to close it by inheritance as by Definition 18 to get the mutual inconsistency relation. This is the reason why we also check that there is no other set S' of events which has the same relation as S with e but on which S depends.

Theorem 34 [equivalence of $CP'(P)$ and $CCP(P)$]. *Given a CC program P, its pruned contextual process $CP'(P)$ and its consistent contextual process $CCP(P)$, then $CP'(P) = CCP(P)$.* $\square$

Part of the complexity of this approach to the construction of the consistent contextual process for a given CC program lies in the fact that we desire to employ a standard way of selecting the subnets corresponding to (equivalence classes of) computations. In fact, assuming that mutual inconsistency is just another aspect of mutual exclusion (that is, just another reason for certain items not to be in the same computation), then the desired subnets are, as usual, those which are maximal, left-closed, and without mutual exclusion. Simpler approaches could be taken; however, they would require ad hoc subnet selection procedures.

7 Locally Atomic Tell

Let us consider now a *locally atomic* tell operation, where a constraint is added to the store if it is consistent with the set of constraints it depends on. Then, it is easy to see that such operation, and the corresponding resulting computations, are very easily expressed by the same process. It is just a matter of selecting different subnets of the process: the (deterministic) locally consistent contextual occurrence nets instead of the deterministic contextual occurrence nets. Recall that the only difference between these two classes of nets is that in the former only the mutual exclusion relation is empty, while in the latter also the mutual inconsistency relation is so. In fact, if in a computation we allow steps which are mutually inconsistent between them, while still not allowing any self-inconsistent step, it means that the only way a computation can finitely fail is that a self-inconsistent step is tried. But we know that such steps represent tell operations which attempt to add a constraint which is inconsistent with the constraints in their history. Therefore, these subnets only have those computation steps which are allowed by the locally atomic interpretation of the tell operation.

```
p(X) :: tell(X=a), tell(X=b).     p(X) :: tell(X=a) -> tell(X=b).
```

Fig. 3. Simple CC programs: query is `p(X)`.

Consider for example the very simple CC program of Figure 3left, where the comma represents the parallel composition operator $\|$, and the absence of "$\rightarrow A$" after a tell operation means that $A = \mathit{success}$. The contextual process corresponding to this program can be seen in Figure 4a, while its consistent contextual process is that of Figure 4b. Also, the set of subnets corresponding to classes of computations which differ only for the scheduling order is, in the case of eventual tell, a singleton set containing the whole contextual process, and in the case of atomic tell a set of two processes whose nets can be seen in Figure 5. In fact, in the eventual tell interpretation, we just have two computations (depending on the order of execution of the two tell operations), both of them failing. Instead, in the atomic tell interpretation, we have two computations, each one performing just one of the tell operations, and both of them failing (which can be seen from the fact that some tell agent is not "expanded"). Consider now the locally atomic tell operation. In this case there is only one subnet, which incidentally coincides with the contextual process. In fact, with this interpretation, both tells are performed, since there is no constraint they depend on (and thus the *incomplete* consistency check for such tells succeeds).

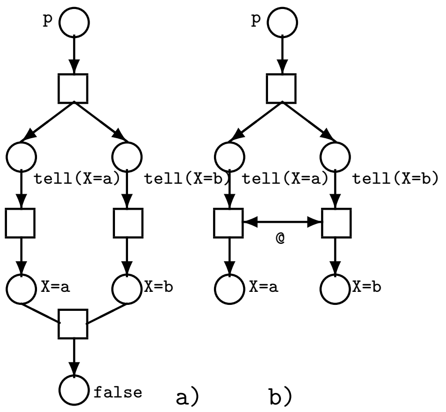

Fig. 4. Contextual and consistent contextual process.

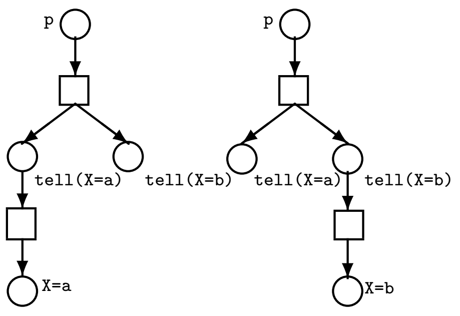

Fig. 5. Consistent contextual nets.

Consider now the CC program of Figure 3right. With the eventual tell interpretation, we obtain the process in Figure 6a, while with the atomic tell interpretation we obtain the consistent contextual process in Figure 6b. Indeed, the second tell operation is self-inconsistent and thus it is not present in the atomic semantics. The locally atomic semantics and the atomic semantics coincide, since no tell attempts to add a constraint which is inconsistent with the current store but not with the current local store. With the eventual tell, there is only one failing computation, which performs both tells and generates an inconsistent store. Instead, with the (locally) atomic tell there is one computation as well, which however performs just one tell operation and then stops.

8 Applications: CLP parallelization and CC scheduling

Being able to explicitely express concurrency and dependency, our semantics can be exploited in several tasks which need such kind of information. One such task is the (compile-time) scheduling of CC programs, or schedule analysis [KS92].

The goal of schedule analysis is to find maximal linearizations of the program processes (agents in our case) where the efficient compilation techniques of sequential implementations can be applied. The best case would be to obtain a complete total order, but in general we may instead obtain a set of total orders, which specify *threads* of sequential execution which, because of the interdependencies in the program, cannot be sequentialized among them [KS92]. Moreover, in each single thread, one would like to schedule the producer(s) before the corresponding consumer(s), so that the consumers do not need to be suspended

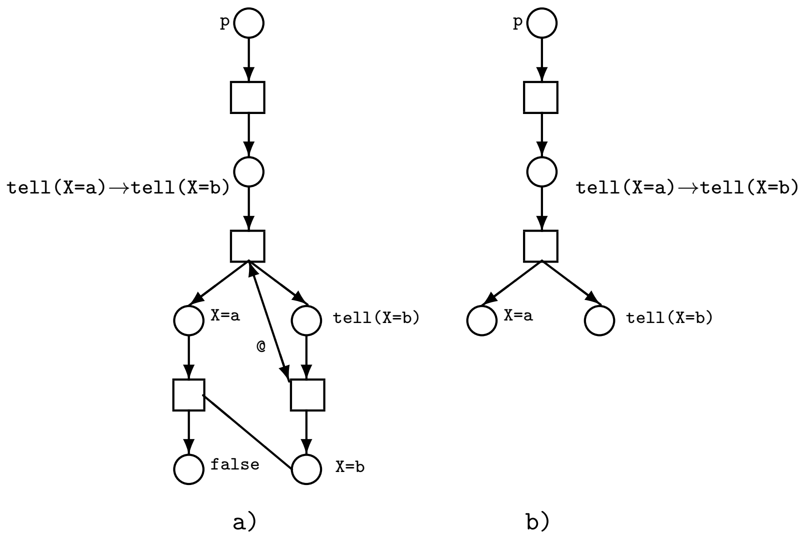

Fig. 6. A contextual process and a consistent contextual process.

and then woken up later. In the specific case of CC programs, the producers are the tell operations and the consumers are the ask operations, so this desirable property of each thread here means that some ask operations could be deleted, if we can be sure that when they will be scheduled the asked constraint has already been told. In [KS92] a framework for this analysis is defined, which is safe w.r.t. the termination properties of the program, and which is based on an input *data-dependency* relation among atoms in the clauses of the program. It is easy to show that in our approach the dependency relation of the contextual process of a program can provide such an input. In fact, it is intuitive to see that the order between two goals in the body of a clause can be easily decided by looking at the contextual net describing the behaviour of the original CC program: if the subnets rooted at these two goals are linked by dependency links which all go in the same direction (from one subnet to the other one), then this direction is the order to be taken for the scheduling; if instead the dependency links go in both directions, then the two goals must belong to two different threads; otherwise (that is, if there are no dependency links between the two subnets), we can order them in any way. Once the order has been chosen, each ask operation which is scheduled later than all the items of the net on which it depends on can safely be deleted. Of course finding the best scheduling is an NP-complete problem. Therefore the optimal solution would require a global analysis of the relationship among the subnets corresponding to all the goals in the body of the considered clause.

Another interesting application is the parallelization of CLP programs. In this task, the problem consists in parallelizing the executions of some of the goals if we are sure that doing that will not change the input-output semantics of the program, nor increase the execution time. What is usually said is that we can parallelize two (or more) goals if we can recognize that they are in some sense "independent," meaning that their executions do not interfere with each other. Instead, for all the goals which do not meet this independence criteria, we resort to the usual left-to-right order. However, the traditional concepts of independence in logic programming [HR93] do not carry over trivially to CLP. In fact, the generalization of the conditions for search space preservation is no longer sufficient for ensuring the efficiency of several optimizations when arbitrary CLP languages are taken into account, and the definition of *constraint independence* in the CLP framework is not trivial [dlBHM93]. Following constraint independence notions, we argue that an efficient parallelization scheme for CLP programs can be developed from the mutual inconsistency relation between events in the consistent contextual processes of the programs. Current work is being devoted towards making this explicit in the (consistent) contextual nets by the new notion of *local independence* [BBHRM93]. In particular, by using our concurrent semantics, we are able to apply the notion of goal independence at a granularity level which, to our knowledge, allows more goals to be safely run in parallel than any other approach. Note that local independence is in general different from concurrency: the idea is that only items which are concurrent (as defined previously in this paper) and which are not dependent because of left-to-right links, are locally independent.

References

[dlBHM93] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *Proc. ILPS*. MIT Press, 1993.

[BBHRM93] F. Bueno, M. García de la Banda, M. Hermenegildo, F. Rossi, and U. Montanari. Towards true concurrency semantics based transformation between CLP and CC. TR CLIP2/93.1, UPM, 1993.

[BP91] F.S. De Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In *Proc. CAAP*. Springer-Verlag, 1991.

[HR93] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1993. To appear.

[JL87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proc. POPL*. ACM, 1987.

[KS92] A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Proc. JICSLP*, pages 478–492, MIT Press, 1992.

[KT91] M. Koorsloot and E. Tick. Sequentializing parallel programs. In *Phoenix Seminar and Workshop on Declarative Programming*. Hohritt, Sasbachwalden, Germany, Springer-Verlag, 1991.

[MR91] U. Montanari and F. Rossi. True concurrency in concurrent constraint programming. In *Proc. ILPS*. MIT Press, 1991.

[MR93a] U. Montanari and F. Rossi. Contextual nets. Technical Report TR-4/93, CS Department, University of Pisa, Italy, 1993.

[MR93b] U. Montanari and F. Rossi. Contextual occurrence nets and concurrent constraint programming. In *Proc. Dagstuhl Seminar on Graph Transformations in Computer Science*. Springer-Verlag, LNCS, 1993.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1985.

[Sar93] V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[SR90] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. POPL*. ACM, 1990.

[SRP91] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. POPL*. ACM, 1991.

[Sco82] D. S. Scott. Domains for denotational semantics. In *Proc. ICALP*. Springer-Verlag, 1982.