Partial Order and Contextual Net Semantics for Atomic and Locally Atomic CC Programs

F. Bueno [a], M. Hermenegildo [a], U. Montanari [b], F. Rossi [b],

[a] *Universidad Politécnica de Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain. E-mail: {bueno,herme}@fi.upm.es*

[b] *Università di Pisa, Dipartimento di Informatica, Corso Italia 40, 56125 Pisa, Italy. E-mail: {ugo,rossi}@di.unipi.it*

Abstract

We present two concurrent semantics (i.e. semantics where concurrency is explicitely represented) for CC programs with atomic tells. One is based on simple *partial orders* of computation steps, while the other one is based on *contextual nets* and it is an extension of a previous one for eventual CC programs. Both such semantics allow us to derive concurrency, dependency, and nondeterminism information for the considered languages. We prove some properties about the relation between the two semantics, and also about the relation between them and the operational semantics. Moreover, we discuss how to use the contextual net semantics in the context of CLP programs. More precisely, by interpreting concurrency as possible parallelism, our semantics can be useful for a safe parallelization of some CLP computation steps. Dually, the dependency information may also be interpreted as necessary sequentialization, thus possibly exploiting it for the task of scheduling CC programs. Moreover, our semantics is also suitable for CC programs with a new kind of atomic tell (called *locally atomic* tell), which checks for consistency only the constraints it depends on. Such a tell achieves a reasonable trade-off between efficiency and atomicity, since the checked constraints can be stored in a local memory and are thus easily accessible even in a distributed implementation.

1 Introduction

The concurrent constraint programming paradigm [15] has its roots both in the constraint logic programming scheme [7] and in concurrent logic programming languages [17]. A concurrent constraint (CC) program [15,18,19] 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 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 store.

The semantics of CC programs is usually given following the SOS-style operational semantics [18,19,3], and thus it suffers from the typical pathologies of an interleaving semantics. On the other hand, the concurrent semantics approach introduced in [11], 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 semantic 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 [11] is based on an operational semantics described via context-dependent rewrite rules, i.e. rules which have a left hand side, a right hand side, and a context. Each rule is applicable if both its left hand side and its context are present in the current state of the computation. A rule application removes the left hand side (but not the context) and adds the right hand side. In particular, the context is crucial in faithfully representing ask constraints, which are checked for presence but not affected by the computation. 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. In this way each computation step (i.e. the application of one of such rules), represents either the evolution of an agent, or the expansion of a declaration, or the entailment of some new constraint.

The concurrent semantic 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 [10], which is just an acyclic Petri net [13] where the presence of context conditions, besides pre- and post-conditions, is allowed. Furthermore, such net is labelled, so that for each element we know the agent or constraint it corresponds to. This contextual net is able to represent all the computations of a given CC program (as defined by its operational semantics), and for each of such computations it provides a partial order expressing the dependency pattern among the computation steps. As a result, all such computations are represented in a unique structure, where it is possible to see the maximal degree of both concurrency (via the concurrency relation) and indeterminism (via the mutual exclusion relation) available both at the program level and at the underlying constraint system.

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 consistency 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 programs which we have just described (and which is defined in detail in [12]) 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. In fact, if a constraint is added to the store in any case (that is, without performing any consistency check), then it may be used by other (ask) agents, and maybe only much later it is recognized that some previous tell added a constraint inconsistent with the current store. Therefore, the semantic structure presented in [12] 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: constraints are added only if they are consistent with the current store. 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 [11,12]). However, this is not true: the semantic structure that we obtain still shows all the atomic entailment steps, thus allowing us 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 then cutting all those parts of the semantic structure which depend on inconsistently "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 are 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 show how to derive the new semantics from scratch (instead of first deriving the semantic structure for eventual tells and then pruning it), by adopting a slightly more complicated inference rule.

In this paper we also present a different semantics, which associates a partial order (of computation steps) to each computation, and we relate it to the semantics based on contextual nets. In particular, we show that, taken a program, the partial orders associated to its computations by this semantics, and the net associated to the program by the net semantics discussed above, every partial order can be derived from the net, and all the computations represented by the same partial order are represented, in the net, by the same deterministic subnet. This additional semantics is basically a trade-off between the scarse expressive power (in terms of concurrency) of the operational semantics, which just shows a sequence of steps, and that of the net, which shows the whole history of all the computations (and thus all possible concurrency and nondeterminism). Moreover, it is worth noting that the partial order semantics and the net semantics are generated in completely different way: the former one by extracting information from already generated computations, and the latter one by generating from scratch a sort of decorated computations (where the decoration is the history).

Since our net 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, since failing branches are also captured, we are allowed to make a step towards exchanging nondeterminism for indeterminism. Thus our semantics, originally thought for indeterministic CC programs, can also be used for nondeterministic programs, and, most important, for CLP programs [2]. The only difference is the interpretation of the mutual exclusion relation, which expresses indeterminism when applied to CC programs, and nondeterminism when applied to CLP programs. 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 [8,9] (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 semantic structure. Furthermore, the level of granularity offered by the semantics allows for scheduling or parallelization 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. 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. From a formal point of view, the semantic structure corresponding to the locally atomic tell interpretation is the minimal one that still is complete, since it does not contain any step which is inconsistent with itself.

In the following, we will first introduce CC programming (Section 2) and its operational semantics (Section 3). We provide CC programs with a partial order semantics (Section 4) and then introduce the required definitions for contextual nets in Section 5. In Section 6 we present the concurrent semantics for CC with eventual tell and in Section 7 that for atomic tell, relating them to the partial order semantics in Section 8. We discuss the *locally atomic* interpretation of the tell operation in Section 9, provide hints to possible applications of our semantics in Section 10, and conclude with Section 11.

This paper is a revised and extended version of [1]. In particular, the extension concerns mainly the partial order semantics given in Section 4, and the theorems concerning its relation to the net semantics.

2 Concurrent Constraint Programming

In the CC paradigm, the underlying constraint system can be described [19] as a *partial information system* (derived from the *information system* introduced in [16]) 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 $false$. Then, a constraint in a constraint system $\langle D, \vdash \rangle$ is simply a set of tokens[1].

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 $\vec{x}$ denote a tuple of variables. The following grammar describes the CC language we consider:
$P ::= F.A$
$F ::= p(\vec{x}) :: A \mid F.F$
$A ::= succ \mid fail \mid tell(c) \rightarrow A \mid \sum_{i=1,\ldots,n} ask(c_i) \rightarrow A_i \mid A \parallel A \mid \exists \vec{x}.A \mid p(\vec{x})$

Each procedure is defined once, thus nondeterminism is expressed via the $+$ combinator only (which is here denoted by $\sum$). We also assume that, in $p(\vec{x}) :: A$, $vars(A) \subseteq \vec{x}$, 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 suspends. 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.

Given a program P, in the following we will refer to $Ag(P)$ as the set containing 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. Also, consider the set V of all free variables appearing in P. Then, let us define the set of *substituted agents*, $\overline{Ag}(P)$, as the set obtained by taking every agent in $Ag(P)$ and substituting each free variable with another variable in V, in all possible ways.

[1] Note that this approach is different from that in [19], where constraints are instead sets of tokens closed under entailment. The reason why we choose not to close sets of tokens under entailment is that we need to distinguish different tokens, and their possibly different causes, in order to give a faithful description of the concurrency present in a program execution.

The CC language we consider in this paper does not use the notion of *cylindric* constraint system, as defined for example in [19]. Therefore, we cannot use that machinery to project constraints over some of their variables. This does not mean that constraints cannot be renamed. In fact, if a constraint appears within an agent which has an existentially quantified variable, and refers to that variables, like in $\exists x.tell(x = 1)$, then the variable in such a constraint is in fact renamed during execution (see next section for details). However, we believe that our whole framework, and corresponding results, can be extended to deal also with cylindrification operators. 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 have made a less general choice here for simplicity reasons, and also because the classical CC framework does not allow tells in guards.

3 The Operational Semantics

Each state of a CC computation consists of a set of elements, labelled over (active) agents and (already generated) tokens. The reason we use a labelled set instead of a set is that we need to have a precise representation of a multiset where different occurrences of the same object can be distinguished. In fact, in general the same agent (and also the same token) may occur in a state with multiplicity higher than one (just think of the computations of $A \parallel A$), and we need to recognize these situations and distinguish among the different occurrences. Both agents and tokens will have associated the free variables they involve.

Each computation step models either the evolution of (an occurrence 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 labelled set described as $S = \langle O, l \rangle$, where O is a finite set of objects (denoted by $Obj(S)$), and $l : O \to (\overline{Ag}(P) \cup D)$. Two states are isomorphic if there is a bijection between their object sets which preserves the labelling.*$\square$

Note that a state $S = \langle O, l \rangle$ can contains two (or more) objects, say $o_1, o_2 \in O$, such that $l(o_1) = l(o_2)$. This means that o_1 and o_2 are different occurrences of the same agent or constraint.

In the following, states will be mostly considered up to isomorphism. This basically means that the identity of the objects in a state will not be significant. For example, $\langle \{o_1, o_2\}, l \rangle$ and $\langle \{o_3, o_4\}, l' \rangle$, such that $l(o_1) = l'(o_3)$ and $l(o_2) = l'(o_4)$, belong to the same isomorphism class and thus will be considered as the same state up to isomorphism. Also, we will refer to l to mean the labelling function of any state, whenever it is clear from the context to which state it refers to. Moreover, sometimes we will write $l(S)$ to mean the whole range of the labelling function defined over the elements of S. Finally, consider the state S with free variables $\vec{x}$, and consider also the vector $\vec{y}$ of other variables from V (of the same lenght as $\vec{x}$); whenever we write $S[\vec{y}/\vec{x}]$ we will mean the state obtained from S by replacing each occurrence of a variable in $\vec{x}$ with the corresponding variable in $\vec{y}$ in all agents and constraints in $l(S)$. Note that by passing from S to $S[\vec{y}/\vec{x}]$ we do not change the set of objects.

Definition 2 (rewrite rules) *Given a program $P = F.A$ with a constraint system $\langle D, \vdash \rangle$, a rewrite rule has the form $r : L(\vec{x}) \overset{c(\vec{x})}{\leadsto} R(\vec{x}\vec{y})$ where L is an agent, c is a constraint, and R is any state. Also, $\vec{x}$ is the tuple of free variables appearing in both $L \cup c$ and in R, while $\vec{y}$ is the tuple of free variables appearing only in R. The state R is always intended up to isomorphism.* □

The intuitive meaning of a rule is that L, which is called the left hand side of the rule, is rewritten into (or replaced by) R, i.e. the right hand side, if c is present in the current state. That is, the items in c have to be interpreted as a context, since they are necessary for the application of the rule but are not affected by such application. In the CC framework, such context is used to represent in a faithful way asked constraints.

Note that the left hand side L and the context c of a rule are elements of $(\overline{Ag}(P) \cup D)$, while the right hand side R is a state, that is, a set labelled over $(\overline{Ag}(P) \cup D)$.

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

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

where the comma in the right hand side has to be interpreted as union of labelled sets.

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 $\overline{Ag}(P)$, plus the rules representing the declarations in F, plus those rules representing the pairs of the entailment relation.* $\square$

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 matching) in S_1. The application of r removes its left hand side and adds its right hand side to S_1.

Definition 4 (eventual computation steps) *Given*

- *a computation state $S_1(\vec{a})$,*
- *a rule $r : L(\vec{x}) \overset{c(\vec{x})}{\leadsto} R(\vec{x}\vec{y})$, and*
- *an injective function $g : (L \cup c) \to Obj(S_1)$ such that there is a binding $[\vec{a}/\vec{x}]$ with $L[\vec{a}/\vec{x}] = l(g(L))$ and $c[\vec{a}/\vec{x}] = l(g(c))$,*

the application of r to S_1 is an eventual computation step which yields a new computation state $S_2 = (S_1 \setminus g(L)) \cup R'$, where

- *R' is a new labelled set of objects such that $R' \cap Obj(S_1) = \emptyset$ and such that there is a bijection between $Obj(R)$ and $Obj(R')$;*
- *the labelling of objects in S_2 is augmented w.r.t. that of S_1 by a labelling of the objects of R', such that $l : Obj(R') \to R[\vec{a}/\vec{x}][\vec{b}/\vec{y}]$;*
- *the variables in $\vec{b}$ are fresh, i.e. they do not appear in S_1.*

We will write $S_1 \overset{r[\vec{a}/\vec{x}][\vec{b}/\vec{y}],g}{\Longrightarrow} S_2$. $\square$

In the above definition, it is worthwhile to point out the different role played by the variables in vectors $\vec{a}$ and $\vec{b}$, and by those in vectors $\vec{x}$ and $\vec{y}$. In fact, computation proceeds by substituting variables in the rules (i.e., $\vec{x}$ and $\vec{y}$) by variables in the states (i.e., $\vec{a}$) and new variables (i.e., $\vec{b}$). Therefore, the variables in $\vec{a}$ and $\vec{b}$ are never substituted by other variables during any computation. On the contrary, vectors $\vec{x}$ and $\vec{y}$ are made of variables which will be bound to the variables in vectors $\vec{a}$ and $\vec{b}$ during a rule application.

Note also that it is the use of the renaming $[\vec{b}/\vec{y}]$ for the free variables ($\vec{y}$) present in the right-hand side but not in the left-hand side of a rule that allows us to treat existential variables in the correct way. This occurs in the application of rule 2 in Definition 3, as illustrated in the example below.

Finally, let us observe that the application of a rule depends not only on the rule and on the current state, but also on the function g, since a rule may be applicable to the same state via different such functions. This accounts, for example, for the treatment of multiple agents in a state.

Example: Consider the simple agent $\exists x.A(x) \mid\mid \exists x.A(x)$, which is the parallel composition of two occurrences of the same agent A, where each occurrence refers to a variable which is existentially quantified. By applying rule 2 to the state containg only that agent, we get the state $\{o_1, o_2\}$, with $l(o_1) = l(o_2) = \exists x.A(x)$. Now we can apply rule 3 (which in this case is $\exists x.A(x) \leadsto A(x)$) either with $g(\exists x.A(x)) = o_1$ or $g(\exists x.A(x)) = o_2$. By using the first one, we get $\{o'_1, o_2\}$, with $l(o'_1) = A(b_1)$. In fact, variable x is free in the right-hand side of the rule, and thus it is bound to a fresh variable (b_1) by definition of rule application. Then we apply rule 3 again to the other agent and we get the state $\{o'_1, o'_2\}$, with $l(o'_2) = A(b_2)$, where b_2 is another fresh variable. Thus the final state is $\{o'_1, o'_2\}$, with $l(o'_1) = A(b_1)$ and $l(o'_2) = A(b_2)$. □

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) *Consider an eventual computation step* $S_1 \overset{r[\vec{a}/\vec{x}][\vec{b}/\vec{y}],g}{\Longrightarrow} S_2$. *This is an atomic computation step if, whenever* $r = ((tell(c) \rightarrow A) \leadsto c, A)$, *then* $c \cup cons(S_1) \not\vdash false$ *(where* $cons(S)$ *is the set of constraints in state* S*).*□

Definition 6 (computations) *Given a CC program* $P = F.A$, *an eventual (resp. atomic) computation segment for* P *is any (finite or infinite) sequence of eventual (resp. atomic) computation steps* $S_1 \overset{r_1[\vec{a_1}/\vec{x_1}][\vec{b_1}/\vec{y_1}],g_1}{\Longrightarrow} S_2 \overset{r_2[\vec{a_2}/\vec{x_2}][\vec{b_2}/\vec{y_2}],g_2}{\Longrightarrow} S_3 \ldots$ *such that* $S_1 = \{A[\vec{a_0}/\vec{x_0}]\}$ *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* CS *such that for each eventual (resp. atomic) computation segment* CS', *of which* CS *is a prefix,* CS' *adds to* CS *only steps applying rules for the entailment relation.* □

Definition 7 (successful, suspended, and failing computations) *Given a CC program and one of its computations (either eventual or atomic), we will say that such computation is:*

- *successful, if it is a finite computation where the last state contains only a set of constraints, say* S, *and* $S \not\vdash false$;
- *suspended, if it 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* $S \not\vdash false$;
- *failing, if it is an infinite computation, or a finite computation which is neither successful nor suspended.* □

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 (either at the moment of the tell or sometime later).

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 to some (sub)state from some point onwards, then it will eventually be applied to that (sub)state[2].

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.* $\square$

4 A Partial Order Semantics

We will now provide CC programs with a partial order semantics, that is, a semantics which associates a partial order to each computation. Each partial order, however, will not be representing only one computation, but an entire class of computations, which differ just in the order in which independent steps are executed (where by *independent*, or *concurrent*, steps, we mean those steps that can be executed in any order). The idea is to take a computation, and build the associated partial order piecewise, by considering one computation step after the other one. Each computation step will help us build a part of the partial order (that is, some of its elements and some pairs of the partial order relation).

Definition 9 (from computations to partial orders) *Given a finite*[3] *computation*

$$C = S_1 \overset{r_1[\vec{a_1}/\vec{x_1}][\vec{b_1}/\vec{y_1}],g_1}{\Longrightarrow} S_2 \ldots S_n \overset{r_n[\vec{a_n}/\vec{x_n}][\vec{b_n}/\vec{y_n}],g_n}{\Longrightarrow} S_{n+1}$$

[2] In logic programming terms, this can be phrased as the fact that both goal selection (among several goals in the current state) and rule selection (among several rules applicable to a goal) are fair.

[3] The definition can also be extended to infinite computations without problems.

where $r_i = L_i \overset{c_i}{\rightsquigarrow} R_i$ for $i = 1, \ldots, n$, let us set $E = \{e_1, \ldots, e_n\}$. Then, let us define the relation F by using the following inference rules (where $i = 1, \ldots, n$):

- *$x \in L_i$ or $x \in c_i$ implies $g_i(x) F e_i$;*
- *$x \in R'_i$ implies $e_i F x$.*

The partial order associated to the computation C is then $PO(C) = (E, F^{+}_{|E})$. □

In words, the above definition just says that, for each step of the computation, we add one element e_i to the partial order, and we relate it to the other elements representing items in the left-hand side, context and right-hand side of the applied rule r_i. If aFb, we mean that a causes b, or that b depends on a. Thus the partial order construction is such that the event representing the application of a rule, say r, depends on the items in the left-hand side and the context of r, and has to cause the items in the right-hand side of r.

In the end (that is, after examining all the n computation steps), we get a partial order with n elements (the events), which shows the dependency pattern among the steps of the considered computation. In fact, events not depending on each other are concurrent, that is, they represent computation steps which do not need each other to be performed (and therefore their execution order can be exchanged). Instead, events which depend on each other represent computation steps where one of the steps need some element generated by the other one, and thus their execution order cannot be exchanged. Note that, because of these properties, the same partial order can be obtained from different computations: all those that differ only in the order in which the concurrent steps are executed.

Example: Consider the agent

$$A = \mathit{tell}(x = a) \parallel \mathit{ask}(x = a) \rightarrow \mathit{tell}(y = b) \parallel \mathit{tell}(x = c),$$

and the eventual computation that executes first the leftmost tell, then the ask, and then the rightmost tell. Then the resulting partial order can be seen in Figure 1 a), where for simplicity only the events corresponding to ask or tell agents are visible, and are decorated with the corresponding constraint generated by the agent. Had we used an atomic tell, event e_4 would not have been present if $x = a$ and $x = b$ are assumed to be inconsistent in the chosen constraint system. Note that the partial order in Figure 1 a) represents also the eventual computation which executes first the rightmost tell, then the leftmost one, and then the ask. Consider now the agent

$$\mathit{tell}(c_1) \parallel \mathit{tell}(c_2).$$

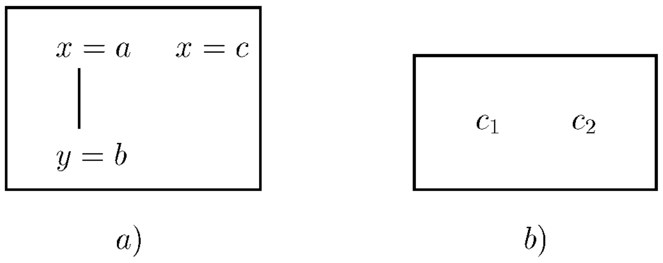

Fig. 1. Partial orders.

The partial order corresponding to all its eventual computations can be seen in Figure 1 b). Assuming that the constraint $\{c_1, c_2\}$ is consistent, the same partial order represents also all its atomic computations. If instead it is inconsistent, then there would be two partial orders representing the (two) atomic computations, one which contains only the event decorated with c_1, and the other one only the event decorated with c_2.□

Definition 10 (eventual and atomic partial order semantics) *Given a CC program P, its eventual partial order semantics is $EPO(P) = \{PO(C)|$ C is an eventual computation of $P\}$, and its atomic partial order semantics is $APO(P) = \{PO(C)|$ C is an atomic computation of $P\}$.* □

5 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 [13] and [10].

In classical nets, as defined for example in [13], each element of the set of conditions can be a pre-condition (if it belongs to the pre-set of an event) or a post-condition (if it belongs to the post-set of an event). In contextual nets a condition can also be a *context* for an event. Informally, a context is something which is necessary for the event to be enabled, but which is not affected by the firing of that event. Still, the usual three relations which are defined on classical nets, that is, dependency, mutual exclusion, and concurrency, can be defined for contextual nets as well, and similar properties hold.

In consistent contextual nets, instead, we assume given also a *mutual inconsistency* relation, which, together with the usual mutual exclusion relation, helps defining those sets of events and/or conditions which cannot appear in the same computation. As a result, four relations are needed instead of three. In the special case of contextual nets used to model CC programs, this additional relation is strongly related to the constraint system, since it is derived from the notion of inconsistency of sets of constraints, and is then propagated to other objects (agents and events) besides constraints.

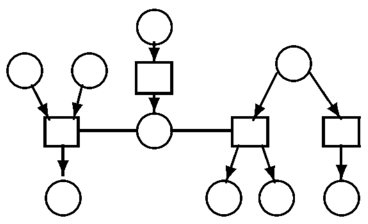
Fig. 2. A contextual net.

5.1 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.* Such relations state which conditions are to be considered as a context for which event. Nets with such contexts will be called *contextual nets.*

Definition 11 (contextual net) *A contextual net is a quadruple* $(B, E; F_1, F_2)$ *where*

- *elements of* B *are called conditions and those of* E *events;*
- $F_1 \subseteq (B \times E) \cup (E \times B)$ *is called the flow relation;*
- $F_2 \subseteq (B \times E)$ *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 12 (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* $\widehat{x} = \{y \mid yF_2x)\}$. □

Context-dependent nets will be graphically represented in the same way as nets. Thus, conditions are circles, events are boxes, and the flow relation is represented by directed arcs from circles to boxes or viceversa. We choose to represent the context relation by undirected arcs because the direction of such relation is unambiguous, i.e. from elements of B to elements of E. An example of a contextual net can be seen in Figure 2. In this figure we see four events, of which two of them share a context.

Here we are not interested in how a contextual net works, i.e. how and when events may be fired. We just need to know that an event can happen whenever its pre-set and context are present, and as a result the pre-set is consumed and the post-set is generated. For more formal definitions, we refer to [10].

In our concurrent semantics the crucial 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 refers to the dependency relation induced by F_1 and F_2.

Definition 13 (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 $e_1 \leq_N e_2$.* $\Box$

Therefore in the following we will say that x depends on y whenever $y \leq_N x$. Note that the dependency relation provides nets with a partial order [14]. In particular, and when restricted to events, the partial order relates two events e_1 and e_2, in the sense that e_2 depends on e_1, whenever there is a postcondition for e_1 which is a context or a precondition for e_2.

However, a contextual net gives information not only about dependency of events and conditions, but also about concurrency and mutual exclusion (or conflict).

Definition 14 (mutual exclusion and concurrency) *Let 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 $\leq\geq \subseteq ((B \cup E) \times (B \cup E))$ be defined as $\leq\geq = \{\langle x, y\rangle \mid x \leq_N y \text{ or } y \leq_N x\}$. Then*

- *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 $x \leq_N z$, then $z\#_Ny$);*
- *the concurrency relation co_N is just $((B \cup E) \times (B \cup E)) \setminus (\leq\geq \cup \#_N)$.* $\Box$

In other 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 15 (contextual occurrence net) *A contextual occurrence net is a contextual net $N = (B, E; F_1, F_2)$ s.t.:*

- *$\leq_N$ 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. For this reason they are called deterministic occurrence nets.

Definition 16 (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 just set intersection, $F_{|S} = F \cap S$.

Definition 17 (from contextual to deterministic contextual occ. nets) *Let 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 $E' \subseteq E$ and*

- *$x \in (B' \cup E')$ and $y \in (B \cup E)$ s.t. $y \leq_N x$ 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 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 18 (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;*
- *$\pi : (B \cup E) \rightarrow (RB \cup RE)$ 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 $r = L \overset{c}{\leadsto} R$, then $\{\pi(x) | x \in {}^\bullet e\} = L\theta$, $\{\pi(x) | x \in \hat{e}\} = c\theta$, $\{\pi(x) | x \in e^\bullet\} = R\theta$.* $\square$

A consistent contextual net is just a contextual net with an additional relation, called 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 19 (consistent contextual net) *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 $F_3 \subseteq \wp(E)$ s.t. $F_3(S)$ implies $\forall e_1, e_2 \in S$, e_1 co_N e_2 and $\forall S' \subset S$, $\neg F_3(S')$.* $\square$

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 (written as @), starting from F_3, and we have to redefine the concurrency relation (co).

Definition 20 (mutual inconsistency and concurrency) *Let $(B, E; F_1, F_2, F_3)$ be a consistent contextual net, 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 which turn out to be 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 21 (admissible consistent net) *A consistent contextual net* $N = (B, E;\ F_1,\ F_2, F_3)$ *is admissible whenever there is no* $e \in E$ *such that* $@(\{e\})$. □

Example: An admissible consistent contextual net can be seen in Figure 3. Notice that we choose to represent the mutual exclusion relation by (hyper)arcs which have arrows on all their endings. In this figure, suppose that the inconsistency link was between the event on the left and the one generating its context. Because of inheritance, the leftmost event would then be inconsistent with itself. Therefore, the net would not be admissible.□

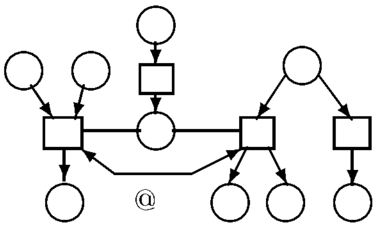

Fig. 3. A consistent contextual net.

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 22 ((deterministic) consistent contextual occ. net) *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* $F_3 = \# = \emptyset$. □

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 17.

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 23 ((deterministic) locally consistent contextual occ. net) *A deterministic locally consistent contextual occurrence net* $(B, E; F_1, F_2, F_3)$ *is a consistent contextual occurrence net with* $\# = \emptyset$. □

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

Definition 24 (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.* $\Box$

6 Concurrent Semantics for Eventual CC

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. Such process is able to represent all possible computations of the CC program P in a unique structure. A longer description of this semantics is contained in [12].

Definition 25 (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 $A(\vec{a})$ initial agent of P then $\langle A(\vec{a}), \emptyset, 1 \rangle \in B$;*
- *if $\exists r \in RR(P)$ such that $L(r) \cup c(r) = \{B_1(\vec{x}_1), \dots, B_n(\vec{x}_n)\}$, and*
  - *$\exists \{s_1, \dots, s_n\} \subseteq B$ such that $\forall i, j = 1, \dots, n$, $s_i \; co_N \; s_j$, and*
  - *$\forall i = 1, \dots, n$, $s_i = \langle e_i, B_i(\vec{a_i}), k_i \rangle$, and for some $\vec{a}$, $B_i(\vec{x_i})[\vec{a}/\vec{x}] = B_i(\vec{a_i})$ then*
  - *$e = \langle r[\vec{a}/\vec{x}], \{s_1, \dots, s_n\}, 1 \rangle \in E$,*
  - *$s_i F_1 e$ for all $s_i = \langle e_i, B_i(\vec{a_i}), k_i \rangle$ such that $B_i(\vec{x_i}) \in L(r)$*
  - *$s_i F_2 e$ for all $s_i = \langle e_i, B_i(\vec{a_i}), k_i \rangle$ such that $B_i(\vec{x_i}) \in c(r)$*
  - *let h be the multiplicity of $B(\vec{x}, y_1, \dots, y_m) \in R(r)$, then $\forall l = 1, \dots, h$, $b_l = \langle B[\vec{a}/\vec{x}][\langle e, y_1 \rangle / y_1] \dots [\langle e, y_m \rangle / y_m], e, l \rangle \in B$, and $e F_1 b_l$.*

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 26 **($CP(P)$ is a contextual process)** *Given a CC program P, its corresponding structure $CP(P)$ built according to Definition 25 is a contextual process.*

PROOF. Given a CC program P, consider the structure $CP(P) = \langle (B, E; F_1, F_2), \pi \rangle$ as defined in Definition 25. To show that it is a contextual process, we need to prove that $N = (B, E; F_1, F_2)$ is a contextual occurrence net, and that π is a mapping with the required properties. We will prove it by induction on the number of applications of the inference rule. The base case is easy, since it just contains one condition, thus all properties in Definition 15 are satisfied. Consider now an intermediate step where the inference rule has been applied already n times, and assume the properties hold for the structure already generated.

- Consider the dependency relation $\leq_N$. The n+1-th application of the inference rule adds new conditions and one new event, and pairs in F_1 and F_2 which relate only such new items. Since in the structure already generated $\leq_N$ is antisymmetric, and there is no pair relating the new items to an old item, $\leq_N$ remains antisymmetric.
- By the induction hypothesis, each condition already in the structure is generated by only one event. This is also preserved by the new application of the inference rule, since it only adds conditions b for all $B \in R(r)$, and pairs eF_1b for all such $b's$. Therefore, for all new $b's$, ${}^\bullet b = \{e\}$, and thus $|^\bullet\ b\ | = 1$.
- Consider the mutual exclusion relation $\#_N$. It is irreflexive in the structure already generated. This means that it does not hold that $s\#_N s$ for any s precondition or context of the newly added event e. Since we have proved that $|^\bullet\ b\ | = 1$ for every b postcondition of e, then it cannot be $b\#'b$. The only other way that $b\#_N b$ or $e\#_N e$ (Definition 14) is that there are x and y in the structure s.t. $x\#_N y$, and $x \leq_N e$ and $y \leq_N e$. But this will mean that there is a precondition or context of e, say s, for which $x \leq_N s$ and $y \leq_N s$. And in this case, $s\#_N s$, which cannot be by inductive hypothesis. Thus $\#_N$ remains irreflexive.

As a result, N is a contextual occurrence net. Consider now the mapping π. By Definition 25, it always maps an element $x = \langle x_1, x_2, x_3 \rangle$ of the net N to x_1. ¿From the way such items are built, x_1 is always an instance of a rewrite rule if x is an event, and an instance of an agent or a constraints if x is a condition. In fact, this is true after the first application of the inference rule (when there is only one condition, mapped onto the initial agent), and subsequent applications trivially preserve this property. Also, all the conditions that the inference rule generates (apart from the initial one) always have a singleton pre-set. Thus, there is only one condition with an empty pre-set (and therefore, *minimal* in the partial order of $\leq_N$), and it is mapped onto the initial agent. Finally, the "enviroment-preserving" condition that requires that the mapping of the preconditions (resp., context conditions, postconditions) of an event are the left hand side (resp., context, right hand side) of the rule the event is mapped to, is trivially satisfied since the inference rule in Definition 25 works exactly in this way. That is, it chooses a set of concurrent conditions that match the left hand side L and the context c of a rule r, maps them to L and c, then generates an event e and maps it to r, and finally generates a set of postconditions for e and maps them to the right hand side of r.

Thus $CP(P)$ is a contextual process. $\square$

Theorem 27 (soundness and completeness of $CP(P)$ w.r.t. $EO(P)$) *Given a CC program P and its corresponding contextual process $CP(P) = \langle N, \pi \rangle$, we have the following.*

- *For a given computation in $EO(P)$ there are (1) an α-equivalent computation $S_1 \overset{r_1[\vec{a}_1/\vec{x}_1],g_1}{\Longrightarrow} S_2 \overset{r_2[\vec{a}_2/\vec{x}_2],g_2}{\Longrightarrow} S_3 \ldots$, and (2) one linearization (restricted to events), say $e_1 e_2 \ldots$, of the partial order associated to a maximal deterministic contextual occurrence net of N, s.t. $\forall i = 1, 2, \ldots,\ \pi(e_i) = r_i[\vec{a}_i/\vec{x}_i]$*
- *For any linearization $e_1 e_2 \ldots$ of the partial order associated to a deterministic contextual occurrence net of N, there is a computation in $EO(P)$, say $S_1 \overset{r_1[\vec{a}_1/\vec{x}_1],g_1}{\Longrightarrow} S_2 \overset{r_2[\vec{a}_2/\vec{x}_2],g_2}{\Longrightarrow} S_3 \ldots$, such that $\pi(e_i) = r_i[\vec{a}_i/\vec{x}_i]$ for all $i = 1, \ldots$*

PROOF. We will prove it by induction on the length of the computation segment. If a computation segment has only one step, then of course it is possible to find the corresponding event in the process, since the existence of such computation segment means that the left hand side and the context of the rule applied in the step are present in the initial state, which is the requirement to add the event to the net in the inference rule in Definition 25. The converse also holds: the presence of a minimal event in the net means that the left-hand side and the context of the corresponding rule are present in the initial state, thus there must exist a computation segment of one step which applies such rule. Assume now that the statement of the theorem holds for a computation segment of length n, and consider a computation segment of length $n + 1$. By the inductive hypothesis, one can find a linearization of the net with n events, which correspond to the n rule applications of the first n computation steps of the considered segment. Now, the presence of the $n + 1$-th step means that the left-hand side and the context of the rule applied in such step is present in the state obtained after the first n steps. Such a state appears in the net also, as a set of concurrent conditions. Thus the inference rule of Definition 25 can add an event corresponding to such rule application, and such event will be either independent from all the first n events, or dependent on on at least one of them, thus it can be included in the partial order, and in the linearization with $n+1$ events. On the other hand, given a linearization with $n+1$ events, by inductive hypothesis there is a computation of length n which corresponds to the first n steps. Again, the presence of the $n+1$-th event in the linearization implies that the left hand side and the context of the rule corresponding to such event are present in the net obtained after the first n events. Thus they are also contained in the state obtained after the computation segment of length n. Therefore the rule can be applied in such state, yielding a computation segment of length $n+1$ matching the given linearization of $n+1$ events. $\square$

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.

7 Concurrent Semantics for Atomic CC

In order to treat in a correct way atomic tells, 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 28 (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 $\nexists v \in \wp(D)$ such that $v \subset u$ and $v \vdash false$.* $\square$

¿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 29 (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 de-*

fine a mutual inconsistency relation $@ \subseteq \wp(B \cup E)$ *(and* $@'$*) as follows:*

- (from constraints to conditions) *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 ther are no* $i, j \in \{1, \ldots, n\}$ *such that* $b_i \# b_j$*, then* $@'(\{b_1, \ldots, b_n\})$*;*
- (from conditions to events) *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* $@(S \cup \{s'\})$*).* $\square$

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 semantic 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 of 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 30 (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 29, the new process is* $CP'(P) = \langle (B', E'; F_1', F_2'), \pi' \rangle$*, where*

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

Theorem 31 ($CP'(P)$ is a consistent contextual process) *Consider the process* $CP'(P) = \langle (B', E'; F_1', F_2'), \pi' \rangle$ *of Definition 30 and the relation* $@$ *of Definition 29. Then the corresponding net* $\langle (B', E'; F_1', F_2', @'_{|\wp(E')}), \pi' \rangle$ *is a consistent contextual process.*

PROOF. It is easy to see that $(B', E'; F_1', F_2')$ is a contextual occurrence net. In fact, $(B, E; F_1, F_2)$ is so (by Theorem 26), and $(B', E'; F_1', F_2')$ is obtained from it by just removing items and links. Thus all properties required by Definition 15 still hold.

We now have to prove that relation $F_3 = @'_{|\wp(E')}$ satisfies the following: $F_3(S)$ implies $\forall e_1, e_2 \in S$, $e_1 \; co_N \; e_2$ and $\forall S' \subset S$, $\neg F_3(S')$.

The first part of the statement ($\forall e_1, e_2 \in S$, $e_1 \; co_N \; e_2$) can be proved by looking at Definition 29. Since $@'(S)$ holds, then it must be $@'({}^\bullet S)$. Take $b_1, b_2 \in {}^\bullet S$, preconditions of e_1 and e_2, respectively. From Definition 29, it cannot be that $b_1 \# b_2$, and thus, by inheritance, neither that $e_1 \# e_2$. Consider now $b_1 \leq b_2$, and assume that $e_1 \leq e_2$. Since $b_1 \leq b_2$, there must be an event

e such that $b_1 \leq e \leq b_2$. Thus, since $b_1 \leq e_1$, $e \# e_1$. Also, since $b_2 \leq e_2$, we have $e \leq e_2$. Furthermore, we assumed $e_1 \leq e_2$. Thus, by inheritance, we get $e_2 \# e_2$. But this cannot be (Definition 30). By contradiction, $e_1 \leq e_2$ cannot hold, and therefore $e_1 co_N e_2$ holds.

The second part of the statement ($\forall S' \subset S$, $\neg F_3(S')$) can be proved by contradiction considering that relation inc_0 is minimal and reasoning on the preconditions of S and S'. From Definition 29, the only way that $F_3(S')$ could hold is that both $inc_0({}^\bullet S)$ and $inc_0({}^\bullet S')$, which is impossible from the minimality of inc_0.

Thus we have proved that $(B', E'; F_1', F_2', @'_{|\wp(E')})$ is a consistent contextual occurrence net. Now we have to prove that π' satisfies the required properties. But this follows from Theorem 26, from the fact that π' is obtained by π by just restricting it to a subset of the elements of the net, and considering that the pruning does not create any other minimal element (since if an element is pruned, then also all the elements depending on it are pruned as well). Thus $\langle (B', E'; F_1', F_2', @'_{|\wp(E')}), \pi' \rangle$ is a consistent contextual process. $\square$

Theorem 32 (soundness and completeness of $CP'(P)$ w.r.t. $AO(P)$) *Given a CC program P and its consistent contextual process $CP'(P) = \langle N, \pi \rangle$, we have the following.*

- *For any computation in $AO(P)$, there are (1) an α-equivalent computation $S_1 \overset{r_1[\vec{a}_1/\vec{x}_1]}{\Longrightarrow} S_2 \overset{r_2[\vec{a}_2/\vec{x}_2]}{\Longrightarrow} S_3 \ldots$, 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. $\forall i = 1, 2, \ldots, \pi(e_i) = r_i[\vec{a}_i/\vec{x}_i]$*
- *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 $S_1 \overset{r_1[\vec{a}_1/\vec{x}_1]}{\Longrightarrow} S_2 \overset{r_2[\vec{a}_2/\vec{x}_2]}{\Longrightarrow} S_3 \ldots$, such that $\pi(e_i) = r_i[\vec{a}_i/\vec{x}_i]$ for all $i = 1, \ldots$*

PROOF. In the atomic operational semantics, a tell step is possible only if the constraint to be added to the current state is consistent with it. Thus, in order to prove the theorem, we have to prove that such forbidden steps are exactly those events that are pruned while going from $CP(P)$ to $CP'(P)$. Now, the pruned elements are those that are inconsistent with themselves, plus all those depending on them. By definition, an event e is inconsistent with itself if one of its postconditions, together with the postcondition of some other event e' it depends on, create an inconsistency. In fact, in this case the mutual inconsistency relation, which holds between e and e', is inherited via the dependency relation onto the event e itself. But this is exactly the case in which the event e represents a tell operation which adds a constraint inconsistent with some other constraint in the current state. Thus e represents a computation step that is not allowed in the atomic operational semantics. Therefore, the steps which are forbidden in the atomic operational semantics are indeed not present in the process $CP'(P)$. Thus, with a reasoning similar to that of the proof of Theorem 27, we can conclude the statement of the theorem. □

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 33 (successful, failing, and suspended nets) *Given a CC program P and a constraint system $\langle D, \vdash \rangle$, let $CP'(P) = \langle (B, E; F_1, F_2, F_3), \pi \rangle$ be the corresponding consistent contextual process. Consider any maximal deterministic consistent contextual net of $(B, E; F_1, F_2, F_3)$, say $DN = (B', E'; F_1', F_2', \emptyset)$, and $DN^\circ = \{b \mid b \in B' \text{ and } \nexists b' \in B',\ b \leq b'\}$. 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 34 (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 32 are successful (resp., suspended, failing).*

PROOF. Assume DN is successful. Then, by Definition 33, the set of events of DN representing agent evolutions is finite, and no maximal element denotes the constraint *false* (meaning that there is no inconsistency). Consider now any linearization of DN and the corresponding computation of the atomic operational semantics via Theorem 32. Such computation is finite, since its computation steps representing agent evolutions are in correspondence with the events of the linearization, which by assumption are in a finite number. Also, no computation step can produce the constraint *false*, otherwise by Theorem 32 there would be an event in the linearization one postcondition of which would represent the constraint *false*, which we assumed it is not the case. Thus all computations corresponding to linearizations of DN are successful. A similar reasoning can be used also for subnets which are suspended and 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 25 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 35 (from rewrite rules to a consistent contextual process) *Let P be a CC program. Then its consistent contextual process $CCP(P) = \langle (B, E; F_1, F_2, F_3), \pi \rangle$ is constructed by means of the following two inference rules:*

- *if $A(\vec{a})$ initial agent of P then $\langle A(\vec{a}), \emptyset, 1\rangle \in B$;*
- *if $\exists r \in RR(P)$ such that $L(r) \cup c(r) = \{B_1(\vec{x}_1), \ldots, B_n(\vec{x}_n)\}$, and*
  - *$\exists\{s_1, \ldots, s_n\} \subseteq B$ such that $co(\{s_1, \ldots, s_n\})$, and*
  - *$\forall i = 1, \ldots, n$, $s_i = \langle e_i, B_i(\vec{a}_i), k_i\rangle$, and for some $\vec{a}$, $B_i(\vec{x}_i)[\vec{a}/\vec{x}] = B_i(\vec{a}_i)$*
  - *$\neg inc(ct(\{e\}))$, for $e = \langle r[\vec{a}/\vec{x}], \{s_1, \ldots, s_n\}, 1\rangle$, where $ct : \wp(B \cup E) \to \wp(D)$ is defined as follows: $\forall \langle t_1, t_2, t_3\rangle \in (B \cup E)$,*

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

  $ct(\emptyset) = \emptyset$
  then
  - *$e \in E$,*
  - *$s_i F_1 e$ for all $s_i = \langle e_i, B_i(\vec{a}_i), k_i\rangle$ such that $B_i(\vec{x}_i) \in L(r)$*
  - *$s_i F_2 e$ for all $s_i = \langle e_i, B_i(\vec{a}_i), k_i\rangle$ such that $B_i(\vec{x}_i) \in c(r)$*
  - *let h be the multiplicity of $B(\vec{x}, y_1, \ldots, y_m) \in R(r)$, then $\forall l = 1, \ldots, h$, $b_l = \langle B[\vec{a}/\vec{x}][\langle e, y_1\rangle/y_1] \ldots [\langle e, y_m\rangle/y_m], \{e\}, l\rangle \in B$, and eF_1b_l.*
  - *$F_3(S \cup \{e\})$ 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 $\not\exists S' \subseteq E$ for which $(\forall e' \in S'\ \exists e \in S, e' \leq e)$ and $co(S' \cup \{s_1, \ldots, s_n\})$ and $inc(ct(\{e\} \cup S'))$.*

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 25 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 preconditions 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 20 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 36 (equivalence of $CP'(P)$ and $CCP(P)$) *Given a CC program P, its corresponding pruned contextual process $CP'(P)$ and consistent contextual process $CCP(P)$, then $CP'(P) = CCP(P)$.*

PROOF. If an event appears in the process $CCP(P)$, then it also appears in $CP(P)$ since the inference rule in Definition 35 has a stronger applicability condition than that of Definition 25. Also, such event cannot be inconsistent with itself, since the only way this could happen is if some of its postconditions are inconsistent with postconditions of events on which it depends, but this is not allowed by the inference rule, which in this case would not be applicable. Thus this event also appears in $CP'(P)$, since $CP'(P)$ is obtained from $CP(P)$ by pruning only the elements which are inconsistent with themselves. In reality, the pruning involves also those elements that depend on the self-inconsistent events, but it is easy to see that such elements cannot appear in $CCP(P)$, since there would not be the necessary preconditions or context conditions for their generation. Thus all events in $CCP(P)$ are also in $CP'(P)$. Consider now any element in $CP'(P)$. Such element is consistent with itself, thus it does not add any constraint which is inconsistent with some other constraint generated by events on which it depends. Therefore the applicability condition of the inference rule in Definition 35 is satisfied, which means that the event is also present in $CCP(P)$. □

Part of the complexity of this approach to the construction of the consistent contextual process for a given CC program comes from our aim of employing 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.

8 Contextual Net Semantics and Partial Order Semantics

We will now show that there is a strong relationship between the semantics based on contextual nets (or on consistent contextual nets) described in the previous section and the partial order semantics defined in Section 4. In fact, it is possible to show that one can derive all the partial orders from the (consistent) contextual net. An even stronger result, which is the one we will prove here, is that each partial order corresponds to one deterministic subnet of the given (consistent) contextual net.

Theorem 37 (deterministic subnets and partial orders) *Given a CC program P, we have the following:*

(1) Consider its contextual process $CP(P) = \langle N, \pi \rangle$ and its eventual partial order semantics $EPO(P)$. Consider also any finite maximal deterministic contextual occurrence net of N, say $ON = \langle B, E, F_1, F_2 \rangle$, and let $\leq$ its dependency relation. Then there is a partial order in $EPO(P)$, say PO, such that $\langle E, \leq_{|E} \rangle$ and PO are isomorphic.

(2) Consider the consistent contextual process $CCP(P) = \langle N', \pi' \rangle$ and the atomic partial order semantics $APO(P)$. Consider also any finite maximal deterministic consistent contextual occurrence net of N, say $ON' = \langle B', E', F_1', F_2', F_3' \rangle$, and let $\leq'$ its dependency relation. Then there is a partial order in $EPO(P)$, say PO', such that $\langle E', \leq'_{|E'} \rangle$ and PO' are isomorphic.

PROOF.

(1) Take any finite computation of P, say

$$S_1 \overset{r_1[\vec{a_1}/\vec{x_1}][\vec{b_1}/\vec{y_1}]}{\Longrightarrow} \dots \overset{r_i[\vec{a_i}/\vec{x_i}][\vec{b_i}/\vec{y_i}]}{\Longrightarrow} \dots \overset{r_j[\vec{a_j}/\vec{x_j}][\vec{b_j}/\vec{y_j}]}{\Longrightarrow} \dots \overset{r_n[\vec{a_n}/\vec{x_n}][\vec{b_n}/\vec{y_n}]}{\Longrightarrow} S_{n+1}.$$

Such computation corresponds, by Theorem 27, to a deterministic subnet of N, say $ON = \langle B, E, F_1, F_2 \rangle$. Consider now the dependency relation $\leq$ of ON, and the partial order $PO = \langle E, \leq_{|E} \rangle$. Take now the partial order associated to the considered computation via Definition 9, say $PO' = \langle E', \leq' \rangle$. We will prove that PO and PO' are isomorphic[4].

It is easy to see that E and E' have the same cardinality, since they represent the same computation. Thus we only need to prove that, for any two events e_1 and e_2 in E such that $e_1 \leq e_2$, there are two corresponding events (via a isomorphism) e_1' and e_2' in E' such that $e_1' \leq' e_2'$.

¿From Theorem 27, it is the case that $\pi(e_1) = r_1[\vec{a}_1/\vec{x}_1]$ and $\pi(e_2) = r_2[\vec{a}_2/\vec{x}_2]$. Let us now consider the computation steps which involve such

[4] For simplicity, let us consider just the Hasse diagram of such partial orders.

rule applications, say s_1 and s_2, and the corresponding events in PO' via Definition 9, say e'_1 and e'_2.

Since we assumed that $e_1 \leq e_2$, from Definition 13 it must be that $\exists b \in B$ such that $e_1 F_1 b$ and $(bF_2 e_2 \vee bF_1 e_2)$. Also, from Theorem 27, we have that $\pi(b) = s$, with $s\theta_i \in R(r_i)$ and $s\theta_j \in (L(r_j) \cup c(r_j))$. Thus, by Definition 9, we must also have $e'_1 \leq' e'_2$. Thus the isomorphism which maps e_1 to e'_1 and e_2 to e'_2 make the statement of the first part of the theorem hold.

(2) A similar reasoning as above, but applying Theorem 32 instead of Theorem 27, allows one to prove also this case. □

9 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. 4. Simple CC programs: query is `p(X)`.

Consider the very simple CC programs of Figure 4, where the comma represents the parallel composition operator $||$, and the absence of "$\rightarrow A$" after a tell operation means that $A = succ$.

The contextual process corresponding to the program on the left in Figure 4 can be seen in Figure 5a, while its consistent contextual process is that of Figure 5b. 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 6. In fact, in the

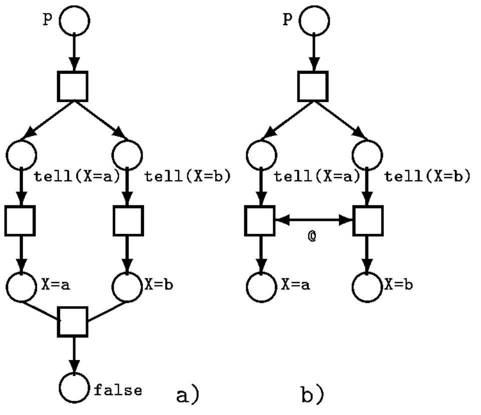

Fig. 5. Contextual and consistent contextual process.

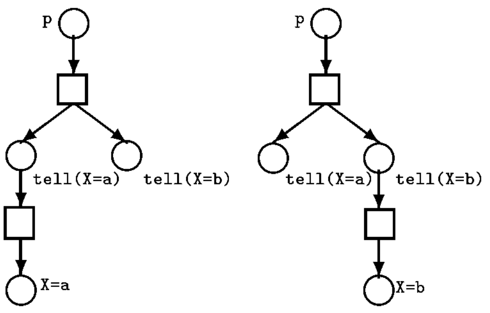

Fig. 6. Consistent contextual nets.

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).

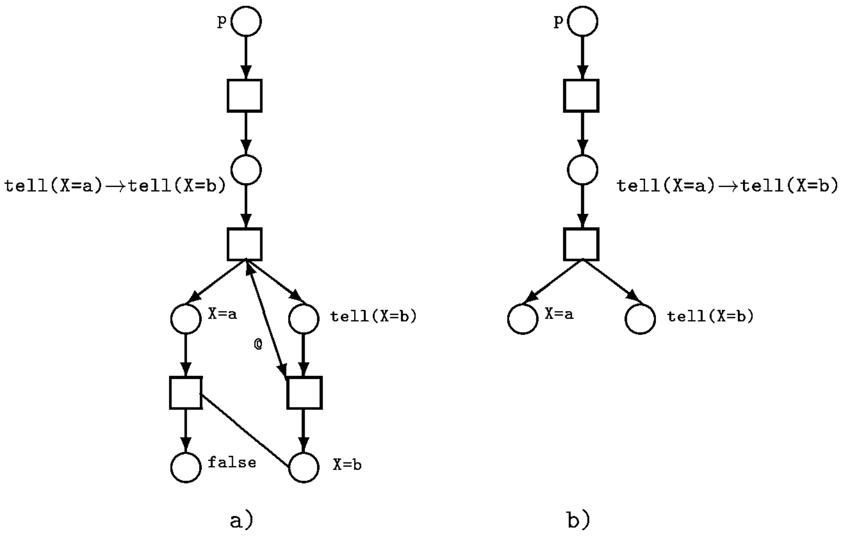

Fig. 7. A contextual process and a consistent contextual process.

Consider now the CC program on the right in Figure 4. With the eventual tell interpretation, we obtain the process in Figure 7a, while with the atomic tell interpretation we obtain the consistent contextual process in Figure 7b. 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.

Notice that it does not make sense to define a locally atomical operational semantics, since the operational semantics, as defined in Section 3 and also in other papers, is not able to express the dependency information needed to define the locally atomical tell operation. However, we feel that a suitable distributed implementation, which uses our concurrent semantics as a basis and which distributes newly added constraints to different locations accordingly to their interdependencies, could easily be developed.

10 Applications

In extending the semantics of Section 6 to that of Section 7 we have basically introduced the ability to handle failure, in the sense of detecting inconsistencies generated by tell operations. Having introduced an explicit representation for failure in the semantics it is also possible to model CLP computations: since failing branches are also captured, we are allowed to make a step further towards exchanging nondeterminism for indeterminism. The atomic contextual processes we have defined for CC programs can also be used to represent the computations of a CLP program, just by interpreting the mutual exclusion relation as nondeterminism (i.e. backtracking) instead of indeterminism (i.e. commited-choice). A feature of such processes representing CLP programs is that, since CLP does not have ask operations, the context relation (F_2) is empty. Therefore the net for a CLP program is actually a tree.

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 [8]. Another such task, in view that our semantics can also handle CLP programs, is the (compile-time) parallelization of these programs.

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 [8]. 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 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 [8] 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 [4]. 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 [6] 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 [5]. 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* [2]. 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 inconsistency, are locally independent. Only these items may be worth running in parallel.

11 Conclusions

We have presented a concurrent semantics for CC programs which models the atomic interpretation of the tell operation. This semantics extends a previous one for CC programs with eventual tell [12], but the extension is not straightforward. In fact, a new semantic structure (consistent contextual processes) is needed for this extension, and new technical machinery to allow for realistically modelling inconsistency. We have shown how the new semantics can be obtained from the previous one by either pruning some parts of the original semantic structure, or right from scratch with a new inference rule.

We have also introduced a more abstract semantics which associates to each computation a partial order of events, and we have related the semantics based on contextual nets and this partial order semantics.

Finally, we have proposed a new interpretation for the tell operation which allows for local consistency checks on the store. The locally atomic interpretation of the tell operation is easily captured by our (extended) semantics based on contextual nets. Such interpretation corresponds to checking consistency only against the part of the current store on which the tell operation is dependent on, and thus will represent a reasonable trade-off between efficiency and atomicity in a distributed implementation.

All the semantics presented are "truly" concurrent, in the sense that they explicitely show the concurrency (in the form of a partial order of dependency links) present not only at the program level but also at that of the underlying constraint system. Moreover, the semantics based on nets is also able to represent all the computations of a given CC program in a unique structure, where it is possible to see the maximal degree of both concurrency and indeterminism. Not only this, but also inconsistency (or failure) is captured in the semantics at different levels of atomicity.

Being able to handle failure, our semantic structures can be used to reason about the behaviour of both CC and CLP programs. In particular, we have discussed how compile-time scheduling of CC programs and parallelization of CLP programs can be performed from analyses over the concurrent nets. For the applications to be practical, we propose to perform a finite approximation of the executions of the program at compile-time using the technique of abstract interpretation. Current work is devoted to defining an abstract contextual process, which finitely represents the possibly infinite set of possibly infinite concrete structures which can be obtained for a given abstract "query mode."

Notice that, while the CC schedule analysis can be performed both on eventual and on atomic CC programs (and the corresponding semantic structures), the analysis needed for the CLP parallelization task is only possible on the semantics for atomic CC programs, since this is the only one where nondeterminism can be exchanged for indeterminism, due to the presence of the inconsistency relation. Therefore the main result of this paper, that is, a concurrent semantics for atomic CC programs, is the necessary starting point for exploiting our semantic approach towards the CLP parallelization goal.

References

[1] F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. From eventual to atomic and locally atomic cc programs: A concurrent semantics. In *Proc. Int. Conference on Algebraic and Logic Programming (ALP94)*. Springer-Verlag, LNCS 850, 1994.

[2] F. Bueno, M. García de la Banda, M. Hermenegildo, F. Rossi, and U. Montanari. Towards true concurrency semantics based transformation between clp and cc. In *Proc. second Int. Workshop on Principles and Practice of Constraint Programming (PPCP94)*. Springer-Verlag, LNCS 874, 1994.

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

[4] F. Bueno. *Automatic Optimisation and Parallelisation of Logic Programs through Program Transformation.* PhD Thesis, Facultad de Informática, Universidad Politécnica de Madrid, 1994.

[5] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in constraint logic programs. In *Proc. ILPS*. MIT Press, 1993.

[6] 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*, 22(1), North Holland, 1995.

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

[8] A. King and P. Soper. Schedule analysis of concurrent logic languages. In *Proceedings IJCSLP*. MIT Press, 1992.

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

[10] U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, vol.32, 1995.

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

[12] 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 776, 1993.

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

[14] F. Rossi. *Constraints and Concurrency.* PhD Thesis, Dipartimento di Informatica, Università di Pisa, TD 14-93, 1993.

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

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

[17] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Survey*, 21(3), 1989.

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

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