Electronic Notes in Theoretical Computer Science 62 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume62.html 14 pages

Comparative analysis of the expressiveness of shared dataspace coordination ⋆

A. Brogi[1] N. Busi[2] M. Gabbrielli[3] G. Zavattaro[2]

[1] *Dipartimento di Informatica, Univ. di Pisa*
[2] *Dipartimento di Scienze dell'Informazione, Univ. di Bologna*
[3] *Dipartimento di Matematica e Informatica, Univ. di Udine*

Abstract

We study the expressiveness of the most prominent representatives of the family of shared dataspace coordination languages, namely Linda, Gamma and Concurrent Constraint Programming.

The investigation is carried out by exploiting and integrating three different comparison techniques: weak and strong modular embedding and property–preserving encodings.

We obtain a hierarchy of coordination languages that provides useful insights for both the design and the use of coordination languages.

1 Introduction

Coordination languages are emerging as suitable architectures for making the programming of distributed applications easier. Most of the language proposals presented in the literature are based on the so-called shared dataspace model, where processes interact through the production, test and removal of data from a common repository. The languages Linda [11], Concurrent Constraint Programming [13], Gamma [1] are the most prominent representatives of this model of coordination.

The availability of a variety of coordination languages raises an interesting question concerned with the expressiveness of such languages. Simply stated, a natural question when in front of two different languages $\mathcal{L}$ and $\mathcal{L}'$ says: *Is* $\mathcal{L}$ *"more powerful" than* $\mathcal{L}'$*?* Some recent works by the authors [3,4,5,6,7,9,8] have been devoted to an investigation of the expressive power of coordination languages. The adopted approaches for language comparison can be classified into two main groups:

⋆ Work partially supported by Italian Ministry of University - MURST 40% - Progetto TOSCA.

- *Relative expressive power.* A natural way to compare the relative expressive power of two languages is to verify whether all programs written in one language can be "easily" and "equivalently" translated into the other one. This idea is formalised by the notion of *language embedding* introduced in [14] and refined by the notion of *modular embedding* defined in [2].
- *Property preserving encoding.* An alternative approach to comparing the expressive power of languages relies on computation theory. Informally the idea is to show that a behavioural property of programs (e.g., termination or divergence) is decidable in a language $\mathcal{L}$ while not in $\mathcal{L}'$, and hence there is no encoding of one language into the other that preserves the given property.

The aim of this paper is to exploit an integration of the above approaches to obtain a comparative analysis of the expressive power of the shared dataspace languages mentioned above, along with some relevant variants of them. Observe that, even if all these languages are based on the common idea i of shared dataspace, they exploit different formats of data (such as, e.g., tuples, constraints, etc.). We obtain a common framework for language comparison by considering unstructured data. We will establish equivalence and separation results for these languages by employing three different yard-sticks: Two forms of modular embedding (strong and weak) and termination-preserving encoding.

The overall result of the paper is a hierarchy of coordination languages that provides useful insights for both the theory and the practice of coordination-based approaches.

2 The calculi

In this section we introduce the syntax and semantics of the calculi that we will analyse.

Definition 2.1 Let *Data* be a denumerable set of data, and let $\mathcal{M}(Data)$ the set of the finite multisets on *Data*. The set *Prog* of programs is defined by the following grammar:

$$P ::= \textstyle\sum_{i \in I} \mu_i.P_i \mid P|P \mid K$$

$$\mu ::= out(a) \mid rd(a) \mid not(a) \mid in(a) \mid min(A)$$

with P, P_i programs, K a program constant, $a \in Data$, and $A \in \mathcal{M}(Data)$.

We assume that each index set I is finite and that each program constant is equipped with a single definition $K = P$ and, as usual, we admit guarded recursion only [12]. We adopt the following abbreviations: $0 = \sum_{i \in \emptyset} \mu_i.P_i$, $\mu_k.P_k = \sum_{i \in \{k\}} \mu_i.P_i$ and $\prod_{i \in I} P_i = P_1| \dots |P_n$ given $I = \{1, \dots, n\}$.

The operational semantics of the calculus is defined by the transition system of Table 1, where the state of a dataspace is modelled by a multiset of data (viz., an element of $\mathcal{M}(Data)$) and where $\oplus$ denotes multiset union.

(1)	$[out(a).P, DS] \longrightarrow [P, DS \oplus \{a\}]$	
(2)	$[rd(a).P, DS \oplus \{a\}] \longrightarrow [P, DS \oplus \{a\}]$	
(3)	$[not(a).P, DS] \longrightarrow [P, DS]$	$a \notin DS$
(4)	$[in(a).P, DS \oplus \{a\}] \longrightarrow [P, DS]$	
(5)	$[min(A).P, DS \oplus A] \longrightarrow [P, DS]$	
(6)	$\dfrac{[P_k, DS] \longrightarrow [P', DS']}{[\sum_{i \in I} P_i, DS] \longrightarrow [P', DS']}$	$k \in I$
(7)	$\dfrac{[P, DS] \longrightarrow [P', DS']}{[P \vert Q, DS] \longrightarrow [P' \vert Q, DS']}$	
(8)	$\dfrac{[P, DS] \longrightarrow [P', DS']}{[K, DS] \longrightarrow [P', DS']}$	if $K \equiv P$

Table 1
Operational semantics (the symmetric rule of (7) is omitted).

Each configuration is a pair denoting the active processes and the dataspace, i.e., $Conf = \{[P, DS] \mid P \in Prog, DS \in \mathcal{M}(Data)\}$.

The $out(a)$ primitive produces a new instance of datum a in the dataspace; $rd(a)$ and $not(a)$ test the status of the dataspace: $rd(a)$ succeeds if at least an instance of datum a is present, whereas $not(a)$ succeeds if the dataspace does not contain datum a. The $in(a)$ operation removes an instance of datum a, whereas the $min(A)$ operation removes the multiset A of data from the dataspace. Programs can be composed by means of guarded choice and parallel composition operators. Program constants permit to define recursive programs.

A configuration C is *terminated* (denoted by $C \not\longrightarrow$) if it has no outgoing transition, i.e., if and only if there exists no C' such that $C \longrightarrow C'$. A configuration C has a terminating computation (denoted by $C\downarrow$) if C can block after a finite amount of computation steps, i.e., there exists C' such that $C \longrightarrow^* C'$ and $C' \not\longrightarrow$. Given a sequence of programs $P_1, \ldots, P_n$, we denote with $n(P_1, \ldots, P_n)$ the set of data names occurring in $P_1, \ldots, P_n$.

In the following, we will consider different subcalculi of the calculus defined in Definition 2.1, which differ from one another for the set of communication primitives used. Syntactically, we will denote by $L[X]$ the calculus which uses only the set X of operations. For instance, $L[rd, out]$ is the calculus of Definition 2.1 where rd and out are the only communication operations considered.

We will focus on comparing the expressive power of five such subcalculi that represent well-known concurrent languages:

- *Linda* — the full calculus $L[rd, not, in, out]$ [11] where agents can add, delete and test the presence and absence of tuples in the dataspace;

- *coreLinda* — the subset $L[rd, in, out]$ of Linda without the *not* primitive for testing the absence of a tuple in the dataspace;
- *ccp* — the calculus $L[rd, out]$ is similar to concurrent constraint programming (ccp) [13], where agents can only add tokens to the dataspace and test their presence, and where the dataspace evolves monotonically;
- *nccp* — the calculus $L[rd, not, out]$ is similar to the timed ccp languages defined in [4,16] since the *not* primitive (for testing the absence of information) was introduced in [16] to model time-based notions such as time-outs and preemption;
- *Gamma* — the calculus $L[min, out]$ represents the language Gamma [1] which features multiset rewriting rules on a shared dataspace.

3 Modular embeddings

3.1 The notion of language embedding

A natural way to compare the expressive power of two languages is to verify whether each program written in one language can be translated into a program written in the other language while preserving the intended observable behaviour of the original program. This idea has been formalised by the notion of *embedding* as follows [14,2].

Consider two languages $\mathcal{L}$ and $\mathcal{L}'$ and let $\mathcal{P}_{\mathcal{L}}$ and $\mathcal{P}_{\mathcal{L}'}$ denote the set of the programs which can be written in $\mathcal{L}$ and in $\mathcal{L}'$, respectively. Assume that the meaning of programs is given by two functions (observables) $\mathcal{O} : \mathcal{P}_{\mathcal{L}} \to Obs$ and $\mathcal{O}' : \mathcal{P}_{\mathcal{L}'} \to Obs'$ which associate each program with the set of its observable properties (thus Obs and Obs' are assumed being some suitable power sets). Then we say that $\mathcal{L}$ is *more expressive than* $\mathcal{L}'$, or equivalently that $\mathcal{L}'$ *can be embedded into* $\mathcal{L}$, if there exists a mapping $\mathcal{C} : \mathcal{P}_{\mathcal{L}'} \to \mathcal{P}_{\mathcal{L}}$ (compiler) and a mapping $\mathcal{D} : Obs \to Obs'$ (decoder) such that, for each program P' in $\mathcal{P}_{\mathcal{L}'}$, the equality $\mathcal{D}(\mathcal{O}(\mathcal{C}(P'))) = \mathcal{O}'(P')$ holds.

$$\begin{tikzcd}
\mathcal{P}_{\mathcal{L}'} \arrow[r, "\mathcal{O}'"] \arrow[d, "\mathcal{C}"'] & Obs' \\
\mathcal{P}_{\mathcal{L}} \arrow[r, "\mathcal{O}"] & Obs \arrow[u, "\mathcal{D}"']
\end{tikzcd}$$

In other words, $\mathcal{L}$ can embed $\mathcal{L}'$ (written also as $\mathcal{L}' \leq \mathcal{L}$) if and only if given a program P' in $\mathcal{L}'$, its observables can be obtained by decoding the observables of the program $\mathcal{C}(P')$ resulting from the translation of P' into $\mathcal{L}$.

Clearly, as discussed in [2], in order to use the notion of embedding as a tool for language comparison some further restrictions should be imposed on the decoder and on the compiler, otherwise the previous equation would be satisfied by any Turing complete language (provided that we choose a powerful enough $\mathcal{O}$ for the target language). Usually these conditions indicate how easy is the translation process and how reasonable is the decoder. Also, note that the notion of embedding in general depends on the notion of observables, which should be expressive enough (considering a trivial $\mathcal{O}$ which associates the same element to any program, clearly we could embed a language into any other one).

The notion of embedding can be used to define a partial order over a family of languages and, in particular, it can be used to establish separation results ($\mathcal{L}' \leq \mathcal{L}$ and $\mathcal{L} \not\leq \mathcal{L}'$) and equivalence results ($\mathcal{L}' \leq \mathcal{L}$ and $\mathcal{L} \leq \mathcal{L}'$).

3.2 Modular embeddings

As already pointed out in the previous section, the basic notion of embedding is too weak since, for instance, the above equation is satisfied by any pair of Turing-complete languages. De Boer and Palamidessi hence proposed in [2] to add three constraints on the coder $\mathcal{C}$ and on the decoder $\mathcal{D}$ in order to obtain a notion of *modular* embedding suited for comparing concurrent languages:

(i) $\mathcal{D}$ should be defined in an element-wise way with respect to Obs, that is:

$$\forall X \in Obs: \ \mathcal{D}(X) = \{\mathcal{D}_{el}(x) \mid x \in X\}$$

for some appropriate mapping $\mathcal{D}_{el}$;

(ii) the coder $\mathcal{C}$ should be defined in a compositional way with respect to all the composition operators, for instance: $\mathcal{C}(A|B) = \mathcal{C}(A) \mid \mathcal{C}(B)$. [1]

(iii) the embedding should preserve the behaviour of the original processes with respect to deadlock, failure and success (*termination invariance*):

$$\forall X \in Obs, \forall x \in X: \ tm'(\mathcal{D}_{el}(x)) = tm(x)$$

where tm and tm' extract the information on termination from the observables of $\mathcal{L}$ and $\mathcal{L}'$, respectively.

An embedding is then called *modular* if it satisfies the above three properties.

The existence of a modular embedding from $\mathcal{L}'$ into $\mathcal{L}$ will be denoted by $\mathcal{L}' \leq \mathcal{L}$. It is easy to see that $\leq$ is a pre-order relation. Moreover if $\mathcal{L}' \subseteq \mathcal{L}$ then $\mathcal{L}' \leq \mathcal{L}$ that is, any language embeds all its sublanguages. This property descends immediately from the definition of embedding, by setting $\mathcal{C}$ and $\mathcal{D}$ equal to the identity function.

The notion of modular embedding has been employed in [5,6] to compare the relative expressive power of a family of Linda-like languages. The separation and equivalence results established in [5,6], restricted to the languages

[1] We assume that both languages contain the parallel composition operator $|$.

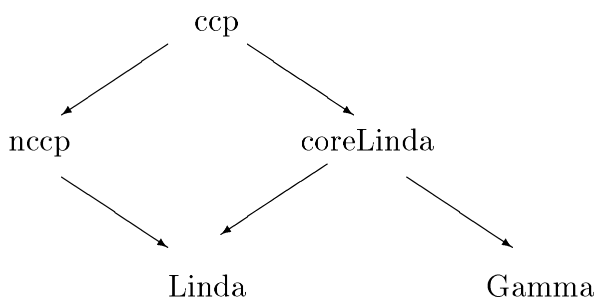

Fig. 1. The hierarchy defined by modular embedding.

described in Section 2, are summarised in Figure 1, where an arrow from a language $\mathcal{L}_1$ to a language $\mathcal{L}_2$ means that $\mathcal{L}_2$ embeds $\mathcal{L}_1$, that is $\mathcal{L}_1 \leq \mathcal{L}_2$. Notice that, thanks to the transitivity of embedding, the figure contains only a minimal amount of arrows. However, apart from these induced relations, no other relation holds. In particular, when there is one arrow from $\mathcal{L}_1$ to $\mathcal{L}_2$ but there is no arrow from $\mathcal{L}_2$ to $\mathcal{L}_1$, then $\mathcal{L}_1$ is strictly less expressive than $\mathcal{L}_2$. The observables considered in [5,6] are defined as follows:
$\mathcal{O}(P) = \{(\sigma, \delta^+) : [P, \emptyset] \longrightarrow^* [\prod_I 0, \sigma]\} \cup \{(\sigma, \delta^-) : [P, \emptyset] \longrightarrow^* [Q, \sigma] \not\longrightarrow, Q \neq \prod_I 0\}$
where δ^+ and δ^- are two fresh symbols denoting respectively success and (finite) failure.

The results illustrated in Figure 1 state that ccp is strictly less expressive of both nccp and coreLinda. Namely this means that both (the introduction of) the *not* primitive and (the introduction of) the *in* primitive strictly increases the expressive power of the basic calculus $L[rd, out]$. Moreover, both nccp and coreLinda are less expressive than the full Linda calculus, while they are not comparable one another. Finally, Gamma is strictly more expressive that coreLinda, while Gamma and full Linda are not comparable one another.

It is worth mentioning here two equivalence results that were established in [5]. Namely the languages $L[rd, in, out]$ and $L[in, out]$ have the same expressive power, that is, one can be modularly embedded in the other and vice-versa. The same hold for the languages $L[rd, not, in, out]$ and $L[not, in, out]$, which have the same expressive power. This means that the *rd* primitive is redundant both in coreLinda and in Linda, in the sense that its elimination does not affect the expressive power of the two languages.

3.3 Weak modular embedding

In this section we compare the languages ccp and nccp and their variants which use also the primitive *in*, by using a weaker notion of modular embedding. The results presented here are derived from the similar ones for (timed) ccp which appeared in [3].

We first define the following abstract notion of observables which distinguishes finite computations from infinite ones.

Definition 3.1 Let P be a process. We define

$$\begin{aligned}\mathcal{O}_\alpha(P) = \{\theta \mid \; & \text{there exists } DS \; s.t.[P, DS] \longrightarrow^* [Q, DS'] \not\rightarrow \\ & \text{and } \theta = \alpha(P, DS \cdots Q, DS') \;\}\end{aligned}$$

where α is any total (abstraction) function from the set of sequences of configurations to a suitable set.

Since our results are given w.r.t. $\mathcal{O}_\alpha$, they hold for any notion of observables which can be seen as an instance of $\mathcal{O}_\alpha$ (e.g. input/output pairs, finite traces etc.). In the following $\mathcal{O}_{ro} : L[out, rd] \rightarrow Obs_{ro}$ and $\mathcal{O}_{ron} : L[out, rd, not] \rightarrow Obs_{ron}$ denote the instances of $\mathcal{O}_\alpha$ representing the observables for the two languages considered in this Section.

As mentioned in Subsection 3.2, some restrictions on the decoder and the compiler are needed in order to use embedding as a tool for language comparison. It is natural to require that the decoder cannot extract any information from an empty set and, conversely, that it cannot cancel completely all the information which is present in a non empty set describing a computation. Therefore, denoting by Obs the observables of the target language, we require that

(i) $\forall O \in Obs$, $\mathcal{D}(O) = \emptyset$ iff $O = \emptyset$.

Furthermore, it is reasonable to require that the compiler $\mathcal{C}$ is a morphism w.r.t. the parallel operator, that is:

(ii) $\mathcal{C}(A|B) = \mathcal{C}(A)|\mathcal{C}(B)$.

These assumptions are weaker than those made in [2], where the decoder was assumed to be defined point-wise on the elements of any set of observables and it was assumed to preserve the (success, failure or deadlock) termination modes, while the compiler was assumed to be a morphism also w.r.t. the choice operator.

Obviously ccp can be embedded into nccp, being the former a sub-language of the latter, and analogously for the variant of these language which use also *in* either to replace *rd* or as a further primitive.

We now show that the presence of the *not* strictly augment the expressive power of the language, since nccp cannot be embedded into ccp.

We first observe that, if a ccp process $P|Q$ has a finite computation then both P and Q have a finite computation. This is the content of the following proposition whose proof is immediate.

Proposition 3.2 *Let P be a ccp process. If $\mathcal{O}_\alpha(P) = \emptyset$ then $\mathcal{O}_\alpha(P|Q) = \emptyset$ for any other ccp process Q.*

On the other hand, previous Proposition does not hold for nccp. In fact, the presence of the *not* construct enforces a kind of non-monotonic behaviour: Adding more information to the store can inhibit some computations, since the corresponding choice branches are discarded. Thus we have the following result

Theorem 3.3 *When considering any notion of observables which is an instance of $\mathcal{O}_\alpha$ the language nccp cannot be embedded into ccp while satisfying the conditions (i) and (ii).*

We have also the following.

Corollary 3.4 *When considering any notion of observables which is an instance of $\mathcal{O}_\alpha$*

- *the language Linda cannot be embedded into coreLinda and*
- *the language $L[out, not, in]$ cannot be embedded into $L[out, in]$*

while satisfying the conditions (i) and (ii).

4 Termination preserving encodings

An alternative approach to the study of the expressiveness of coordination languages (adopted, e.g., in [7]) consists in borrowing techniques from the theory of computation, that are used as a tool for languages comparison.

The key idea to provide a separation result between two languages consists in devising a behavioural property of programs (such as, e.g., the existence of a terminating computation or the existence of a divergent computation), that is decidable for one of the languages but turns out to be undecidable for the other one; hence, we can conclude that there exists no encoding of one language on the other one which preserves the given property.

In this section we show that there exists no termination-preserving encoding of Linda in Gamma, coreLinda and nccp. The results are a consequence of the following facts:

(i) There exists an implementation of Random Access Machines (RAMs) [17] in Linda which preservers the terminating behaviour. As RAMs are Turing equivalent, termination is not decidable for Linda.

(ii) There exists a termination-preserving encoding of Gamma on finite Place/Transition nets. As termination is decidable for this class of nets, the same holds for Gamma.

(iii) There exists a termination-preserving encoding of coreLinda in Gamma. As termination is decidable for Gamma, the same holds for coreLinda.

(iv) There exists a termination-preserving encoding of nccp in coreLinda. As termination is decidable for coreLinda, the same holds for nccp.

The result (iii) is a consequence of the existence of a modular embedding from coreLinda to Gamma; the proofs of the remaining results are sketched below.

4.1 Termination is undecidable for Linda

We show that (the rd-free fragment of) Linda is Turing equivalent by providing an encoding of Random Access Machines in Linda that preserves the existence of a terminating computation.

4.1.1 Random Access Machines

A Random Access Machine [17], simply RAM in the following, is a computational model composed of a finite set of registers $r_1 \dots r_n$, that can hold arbitrary large natural numbers, and a program $I_1 \dots I_k$, that is a sequence of simple numbered instructions.

The execution of the program begins with the first instruction and continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached.

The following two instructions are sufficient to model every recursive function:

- $Succ(r_j)$: adds 1 to the content of register r_j;
- $DecJump(r_j, s)$: if the content of register r_j is not zero, then decreases it by 1 and go to the next instruction, otherwise jumps to instruction s.

The (computation) state is represented by $(i, c_1, c_2, \dots, c_n)$, where i indicates the next instruction to execute and c_l is the content of the register r_l for each $l \in \{1, \dots, n\}$. Let R be a program $I_1 \dots I_k$, and $(i, c_1, c_2, \dots, c_n)$ be the corresponding state; we use the notation $(i, c_1, c_2, \dots, c_n) \longrightarrow_R (i', c'_1, c'_2, \dots, c'_n)$ to state that after the execution of the instruction I_i with contents of the registers $c_1, \dots, c_n$, the program counter points to the instruction $I_{i'}$, and the registers contain $c'_1, \dots, c'_n$. Moreover, we use $(i, c_1, c_2, \dots, c_n) \not\longrightarrow_R$ to indicate that $(i, c_1, c_2, \dots, c_n)$ is a terminal state, i.e., $i > k$.

In this section we recall an encoding of RAMs [7] in (the rd-free fragment of) Linda.

Consider the state $(i, c_1, c_2, \dots, c_n)$ with corresponding RAM program R. We represent the content of each register r_l by putting c_l occurrences of datum r_l in the dataspace. Suppose that the program R is composed of the sequence of instructions $I_1 \dots I_k$; we consider k programs $P_1 \dots P_k$, one for each instruction. The program P_i behaves as follows: if I_i is a $Succ$ instruction on register r_j, it simply emits an instance of datum r_j and then activates the program P_{i+1}; if it is an instruction $DecJump(r_j, s)$, the program P_i is a choice between consumption and test for absence on datum r_j. If an instance of r_j is present in the dataspace, the $in(r_j)$ operation is performed and the subsequent program is P_{i+1}; otherwise, the $not(r_j)$ operation is performed and the subsequent program is P_s. According to this approach we consider the following definitions for each $i \in \{1, \ldots, k\}$:

$$\begin{array}{ll} P_i = out(r_j).P_{i+1} & \text{if } I_i = Succ(r_j) \\ P_i = in(r_j).P_{i+1} + not(r_j).P_s & \text{if } I_i = DecJump(r_j, s) \end{array}$$

We also consider a definition $P_i = 0$ for each $i \notin \{1, \ldots, k\}$ which appears in one of the previous definitions. This is necessary in order to model the termination of the computation occurring when the next instruction to execute has an index outside the range $1, \ldots, k$.

The encoding is then defined as follows:

$$[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \quad = \quad [P_i, \bigoplus_{1 \leq l \leq n} \{\underbrace{r_l, \ldots, r_l}_{c_l \text{ times}}\}]$$

The correctness of the encoding is stated by the following theorem.

Theorem 4.1 *Given a RAM program R and a state $(i, c_1, c_2, \ldots, c_n)$, we have $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c'_1, c'_2, \ldots, c'_n)$ if and only if $[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \longrightarrow [\![(i', c'_1, c'_2, \ldots, c'_n)]\!]_R$.*

As a corollary of this theorem, we have that the encoding preserves termination.

Corollary 4.2 *Given a RAM program R, we have that R terminates if and only if $[\![(1, 0, 0, \ldots, 0)]\!]_R\!\downarrow$.*

4.2 Termination is decidable for Gamma

In order to show the impossibility to provide a termination-preserving encoding of Linda in Gamma, we prove that termination is decidable for Gamma. We resort to a semantics based on Place/Transition nets, a formalism for which termination is decidable[10,7]. Here, we report a definition of the formalism suitable for our purposes.

Definition 4.3 A P/T net is a triple $N = (S, T, m_0)$ where S is the set of *places*, T is the set of *transitions* (which are pairs $(c, p) \in \mathcal{M}(S) \times \mathcal{M}(S)$), and m_0 is a finite multiset of places. Finite multisets over the set S of places are called *markings*; m_0 is called *initial marking*. Given a marking m and a place s, $m(s)$ denotes the number of occurrences of s inside m and we say that the place s contains $m(s)$ *tokens*. A P/T net is finite if both S and T are finite.

A transition $t = (c, p)$ is usually written in the form $c \to p$. The marking c is called the *preset* of t and represents the tokens to be *consumed*. The marking p is called the *postset* of t and represents the tokens to be *produced*.

A transition $t = (c, p)$ is *enabled* at m if $c \subseteq m$. The execution of the transition produces the new marking m' such that $m'(s) = m(s) - c(s) + p(s)$.

$$dec([P, DS]) = dec(P) \oplus DS$$

$$dec(\sum_{i \in I} \mu_i.P_i) = \sum_{i \in I} \mu_i.P_i$$

$$dec(K) = dec(P) \qquad \text{if } K = P$$

$$dec(P|Q) = dec(P) \oplus dec(Q)$$

`min(A,Q,P)`	$min(A).Q + P \oplus A \rightarrow dec(Q)$
`out(a,Q,P)`	$out(a).Q + P \rightarrow a \oplus dec(Q)$

Table 2
Definition of the decomposition function *dec* and net transitions $\mathcal{T}$.

The basic idea underlying the definition of an operational net semantics for a process calculus is to decompose a term into a multiset of *sequential components*, which can be thought of as running in parallel. Each sequential component has a corresponding place in the net, and will be represented by a token in that place. Reductions are represented by transitions which consume and produce multisets of tokens.

In our particular case, sequential components are of the form $\sum_{i \in I} \mu_i.P_i$.

Any datum is represented by a token in a particular place a. Data production and consumption is represented as follows: $out(a)$ produces a new token in place a, while $min(A)$ removes $A(a)$ tokens from each place a.

The axioms in the first part of Table 2 describe the decomposition of programs and dataspaces in corresponding markings. The decomposition of a dataspace corresponds to the dataspace itself. A sequential component produces one token in the corresponding place; a program constant is treated as its corresponding program definition; the parallel composition is interpreted as multiset union.

The axioms in the second part of Table 2 define the possible transitions denoted by $\mathcal{T}$.

Definition 4.4 Let $C = [P, DS]$ be a configuration such that P has the following related program constant definitions: $K_1 = P_1, \ldots, K_n = P_n$. We define the triple $Net(C) = (S, T, m_0)$, where:

$$S = \{Q \mid Q \text{ is a sequential component of either } P,\ P_1,\ \ldots,\ P_n\} \cup$$
$$\{a \mid a \in n(P, P_1, \ldots, P_n) \cup DS\ \}$$

$$T = \{c \rightarrow p \in \mathcal{T} \mid \text{the sequential comp. and the data in } c \text{ are also in } S\}$$

$$m_0 = dec(C)$$

Note that, given a configuration C, the corresponding $Net(C)$ is a finite

$$[\![[P, DS]]\!] = [[\![P]\!], [\![DS]\!]]$$

$$[\![out(a).P]\!] = in(a_0).out(a_+).[\![P]\!] + in(a_+).out(a_+).[\![P]\!]$$
$$[\![rd(a).P]\!] = in(a_+).out(a_+).[\![P]\!]$$
$$[\![not(a).P]\!] = in(a_0).out(a_0).[\![P]\!]$$
$$[\![P|Q]\!] = [\![P]\!]|[\![Q]\!]$$
$$[\![\textstyle\sum_{i\in I} P_i]\!] = \textstyle\sum_{i\in I} [\![P_i]\!]$$
$$[\![K]\!] = K$$

$$[\![DS]\!] = \{a_+ \mid a \in DS\} \oplus \{a_0 \mid a \in n(P, P_1, \ldots, P_n) \wedge a \notin DS\}$$

Table 3
Encoding of nccp in coreLinda

P/T net.

The correctness of the semantics is stated by the following theorem:

Theorem 4.5 *Let D be a configuration of Gamma and $Net(D)$ the corresponding net. Let m be a marking of $Net(D)$ such that $m = dec(C)$ for some configuration C. We have that $C \longrightarrow C'$ iff $m \rightarrow dec(C')$ in $Net(D)$.*

As a corollary, we have that the net semantics preserves the existence of a terminating computation.

Corollary 4.6 *Given a configuration C of Gamma, we have that $C\downarrow$ if and only if $Net(C)$ has a terminating firing sequence.*

As the existence of a terminating firing sequence is decidable for P/T nets, we can conclude that termination is decidable for Gamma.

4.3 Encoding of nccp in coreLinda

To show that termination is decidable in nccp, we provide a termination-preserving encoding of nccp in (the rd free fragment of) coreLinda. The encoding is reported in Table 3. We also replace each constant definition $K = P$ by $K = [\![P]\!]$.

The presence of one or more instances of datum a is represented by a single datum a_+ in the dataspace, whereas the absence of datum a is represented by a datum a_0. The mapping of the $not(a)$ (resp. $rd(a)$) operations consists in testing the presence of datum a_0 (resp. a_+) in the dataspace. On the other hand, the mapping of the $out(a)$ operation replaces a_0 with a_+, or leaves the dataspace unchanged if a_+ was already contained in the dataspace.

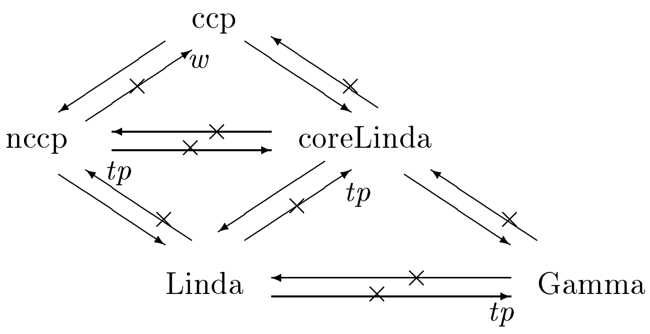

Fig. 2. Synthesis of all the results.

Theorem 4.7 *Let C be a configuration of nccp. If $C \longrightarrow C'$ then $[\![C]\!] \longrightarrow^+ [\![C']\!]$. If $[\![C]\!] \longrightarrow^+ D'$ then there exists C' s. t. $C \longrightarrow^+ C'$ and $D' \longrightarrow^* [\![C']\!]$.*

Corollary 4.8 *Let C be a configuration of nccp. Then $C\!\downarrow$ iff $[\![C]\!]\!\downarrow$.*

5 Concluding remarks

In the previous sections we have discussed the use of three different yard-sticks for measuring the expressive power of a family of Linda-like languages. In particular the five languages on which we focused (ccp, nccp, coreLinda, Linda and Gamma) have been compared both by means of modular embedding ($\leq$) and weak modular embedding ($\leq_w$) and by means of termination-preserving encoding ($\leq_{tp}$).

The results established in Sections 3 and 4 are synthesised in Figure 2, where only the strongest results are reported. Namely the existence of a modular embedding is a stronger result than the existence of a weak modular embedding which is in turn stronger than the existence of a termination-preserving encoding. The situation is obviously reversed for the case of negative results.

Figure 2 illustrates how the hierarchy established by means of modular embeddings (see Fig. 1) is enriched by the results established by weak modular embeddings and termination-preserving encodings. In particular, the separation result between ccp and nccp has been strengthened in the sense that nccp cannot be modularly embedded into ccp even if the constraints on the modularity of the coding are relaxed as shown in Sect. 3.3. The separation results between nccp and Linda, and between coreLinda and Linda have been further strengthened by showing that there is no termination-preserving encoding neither of Linda in nccp nor of Linda in coreLinda. The incomparability result of Linda and Gamma has also been strengthened by showing that there is no termination-preserving encoding of Linda in Gamma.

References

[1] J.P. Banatre and D. Le Métayer. Programming by Multiset Transformation. *Communication of the ACM*, 36(1) 98–111, 1993.

[2] F.S. de Boer and C. Palamidessi. Embedding as a tool for language comparison. *Information and Computation*, 108(1):128-157, 1991.

[3] F.S. de Boer, M. Gabbrielli, and M.C. Meo. Semantics and expressive power of a timed concurrent constraint language. In G. Smolka. editor, *Proc. Third Int'l Conf. on Principles and Practice of Constraint Programming (CP 97)*. LNCS, Springer-Verlag, 1997.

[4] F.S. de Boer, M. Gabbrielli and M.C. Meo. A Timed CCP Language. *Information and Computation*, 161, 2000.

[5] A. Brogi and J.M. Jacquet. On the expressiveness of Linda-like concurrent languages. *Electronic Notes in Theoretical Computer Science*, 16(2), 2000.

[6] A. Brogi and J.M. Jacquet. On the expressiveness of coordination via shared dataspaces. *Science of Computer Programming*. Forthcoming. (A short version appeared in P. Ciancarini and A. Wolf, editors, "Coordination: Languages and Models - Proceedings of Third International Conference,COORDINATION'99", LNCS 1594, pages 134–149. 1999. Springer-Verlag.)

[7] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156(1/2):90–121, 2000.

[8] N. Busi, R. Gorrieri, and G. Zavattaro. Temporary Data in Shared Dataspace Coordination Languages. In *Proc. of FOSSACS 2001*, LNCS 2030, pages 121–136. Springer-Verlag, Berlin, 2001.

[9] N. Busi and G. Zavattaro. On the Expressiveness of Event Notification in Data-driven Coordination Languages. In *Proc. of ESOP2000*, LNCS 1782, pages 41–55. Springer-Verlag, Berlin, 2000.

[10] A. Cheng, J. Esparza, and J. Palsberg. Complexity results for 1-safe nets. In *Theoretical Computer Science*, 147:117–136, 1995.

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

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

[13] V.A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proc Seventeenth ACM Symposium on Principles of Programming Languages*, 232–245, ACM Press, 1990.

[14] E. Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412-510, 1989.

[15] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundation of Concurrent Constraint Programming. In *Proc. Eighteenth ACM Symposium on Principles of Programming Languages*, pages 333-353. ACM Press, 1991.

[16] V.A. Saraswat, R. Jagadeesan, and V. Gupta Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5-6):475–520, 1996.

[17] J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.